The Personal Shopper's Dilemma: Shopping Time vs. Shopping Cost

by

Samiul Anwar

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Consider a customer who needs to fulfill a shopping list, and also a personal shopper who is willing to buy and resell to customers the goods in their shopping lists. It is in the personal shopper's best interest to find (shopping) routes that (i) minimize the time serving a customer, in order to be able to serve more customers, and (ii) minimize the price paid for the goods, in order to maximize his/her potential profit when reselling them. Those are typically competing criteria leading to what we refer to as the *Personal Shopper's Dilemma* query, i.e., to determine where to buy each of the required goods while attempting to optimize both criteria at the same time. Given the query's NP-hardness we propose a heuristic approach to determine a subset of the sub-optimal routes under any linear combination of the aforementioned criteria, i.e., the query's approximate linear skyline set. In order to measure the effectiveness of our approach we also introduce two new metrics, optimality and coverage gaps w.r.t. an optimal, but computationally expensive, baseline solution. Our experiments, using realistic city-scale datasets, show that our proposed approach is two orders of magnitude faster than the baseline and yields low values for the optimality and coverage gaps.

Preface

This thesis is an original work by Samiul Anwar supervised by Professor Mario Nascimento. No part of this thesis has been previously published but has been submitted to SIGSPATIAL-2020 as a research paper.

To my Parents

For giving me the gift of literacy.

Acknowledgements

I am greatly indebted to Professor Mario Nascimento for his supervision and guidance throughout the thesis. It would not be possible to complete a graduate-level research in Spatial-Database without his help.

I would also like to thank Dr. Francesco Lettich for his continuous guidance and immense support throughout the thesis. This thesis would not have come together as it is without his help.

Contents

List of Tables

List of Figures

1 Introduction

Let us consider a *customer* who needs to acquire some products available in *stores* within a geographical area, e.g., his/her city's road network. To accomplish this goal the customer submits his/her request to a *personal shopper* in the form of a *shopping list* of products, along with their respective quantities. The personal shopper's task is to serve customers by fulfilling their shopping list. Serving a customer's request entails two different and *competing* criteria. The first criterion is the *(shopping) time* needed to serve the customer, i.e., the time needed to visit a sequence of stores in order to acquire all products in the customer's shopping list plus the time needed for delivering them to said customer. The second one is the *(shopping) cost* of the shopping list, i.e., the sum of the costs of all the products in the shopping list at the stores where they were acquired. Clearly, the shopper's main goal should be to minimize *concurrently* both criteria; the faster he/she can deliver the goods the more customers he/she can serve, and the smaller the actual cost of the goods the higher the profit margins he/she can enjoy. The main question the shopper needs to answer then becomes: *"Which sequence of stores to visit and which products to acquire in those stores in order to fulfill the customer's shopping list, while, at the same time, minimizing both the shopping time and the shopping list's cost?"* Unfortunately, from a practical perspective, it is seldom possible to find a single set of stores that satisfies both criteria. We call this new and practical problem as the *Personal Shopper's Dilemma* (PSD) query.

At this point one could be tempted to argue that the two considered criteria could be linearly combined into a single one, thus casting PSD into a cost function minimization problem. For instance, though not always desirable, one could express shopping time in terms of monetary cost and then assign appropriate weights to combine both criteria into a single one. While we use shopping time and shopping cost as the two main criteria one could use, other criteria that could not be expressed in terms of a common unit (e.g., maximize the number of organic and/not locally produced goods, and minimize the customer's waiting time). There is another, more subtle, limitation to be considered here. Even if a linear combination of the criteria were to be established, appropriate weights would need to be predetermined by the shopper *a priori*. Thus, any – potentially more interesting – solution for slightly different combinations of weights would not be returned to the shopper *by design*. To overcome such limitations we let the shopper decide by him/herself how to (linearly) combine both criteria *a posteriori*. That is, we aim at computing the set of *all* optimal sequence of stores for *any* linear combination of shopping cost and shopping time, also known as *linear skyline set* (discussed further shortly). Notably, the shopper's decisions can be made after running *one single* PSD query, instead of running one query for each linear combination of the two criteria. Thus, in hectic periods of the day the shopper may prefer to prioritize more short trips, each with smaller profit, while calmer parts of the day (or changes in traffic patterns) may push the shopper towards longer trips yielding larger profits, or anything in between.

In order to illustrate the PSD query, let us introduce a simple instance. Let us suppose that a customer wishes to buy products A, B, C and D and issues the corresponding shopping list to the shopper[1]. Now, let us suppose that the stores available are those shown in Figure 1.1 and they are embedded in a road network not shown for the sake of simplicity. Here, stores are depicted by orange squares (and denoted by s_x), while the customer's delivery location

[1]For the sake of simplicity and, in the scope of this example only, let us consider that only one unit of each product is required in the shopping list.

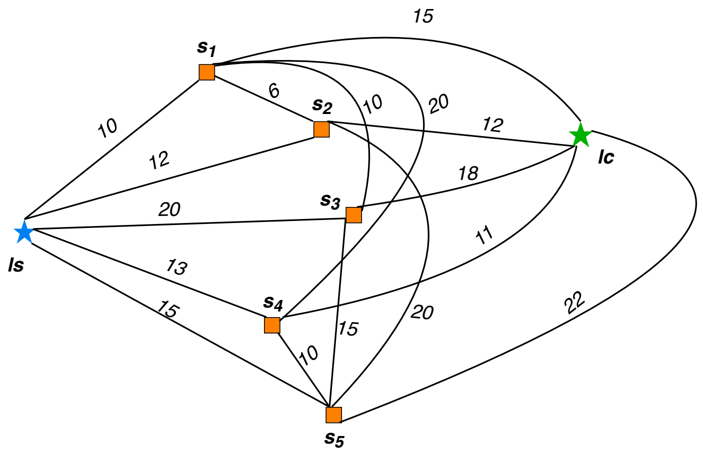

Figure 1.1: Sample network.

is denoted by lc and the shopper's current location is denoted by ls. Edges between locations represent the fastest paths in the road network connecting them, with labels denoting the associated travel time. Finally, Table 1.1 shows the products available in each store, along with their unit prices.

Store	Products
s_1	(A, \$7), (B, \$8), (F, \$10)
s_2	(C, \$10), (D, \$8), (E, \$10)
s_3	(C, \$5), (D, \$4), (F, \$6)
s_4	(C, \$8), (D, \$7), (F, \$12)
s_5	(A, \$6), (B, \$7), (E, \$8)

Table 1.1: List of products available in the stores depicted in Figure 1.1. Each pair represents a specific product and the associated unit cost.

From the above scenario we see that there are multiple solutions that may be of interest for the shopper (Table 1.2). On the one end of the spectrum route R_1 represents the best solution in terms of shopping time, yet its shopping cost is the largest. On the other hand, R_5 offers the lowest shopping cost, but it requires to traverse an expensive route in terms of shopping time. Between these extremes there are several solutions that may *interest* the shopper, where the notion of "interestingness" depends on the shopper's particular preferences or needs at query time.

Route	Shopping Time	Shopping Cost
$R_1 = \langle ls, s_1, s_2, lc \rangle$	28	33$
$R_2 = \langle ls, s_1, s_3, lc \rangle$	38	23$
$R_3 = \langle ls, s_1, s_4, lc \rangle$	41	30$
$R_4 = \langle ls, s_5, s_2, lc \rangle$	47	31$
$R_5 = \langle ls, s_5, s_3, lc \rangle$	48	21$
$R_6 = \langle ls, s_5, s_4, lc \rangle$	36	28$

Table 1.2: Set of routes from Figure 1.1 that can satisfy the shopping list $\{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$.

When dealing with multiple cost criteria and the problem of determining a set of results that are optimal under any arbitrary combination thereof, a well-known and extensively used tool in the literature are *skyline queries* [5]. In general terms, given a set of cost criteria and a pair of objects o_i and o_j, we say that o_j *dominates* o_i if (i) for each cost criterion the cost of o_j is *smaller or equal* than that of o_i and (ii) there is at least one criterion for which the cost of o_j is *strictly smaller* than that of o_i. In turn, the set of objects that are non-dominated by any other object defines the notion of *skyline*, and represents the desired solution. If we represent a skyline on a Cartesian plane, then the skyline represents a frontier beyond which all objects are dominated, i.e., they are not better than those in the frontier.

Computing skyline queries is a computationally intensive task that typically returns many very similar solutions, thus possibly making the choice of a specific solution rather difficult for users. To tackle these issues, Shekelyan et al. [21] introduced the notion of *linear skylines*. In general terms, a linear skyline is the subset of the objects defining the *convex hull* of the (conventional) skyline. Objects belonging to a linear skyline are required to be optimal under *any linear combination* of the competing cost criteria. From that, it follows that any solution that is conventionally dominated is also linearly dominated. Thanks to this stricter, though still practical, requirement, linear skylines typically contain much less objects than conventional skylines and are thus easier to interpret. Considering the above example, the linear skyline is represented

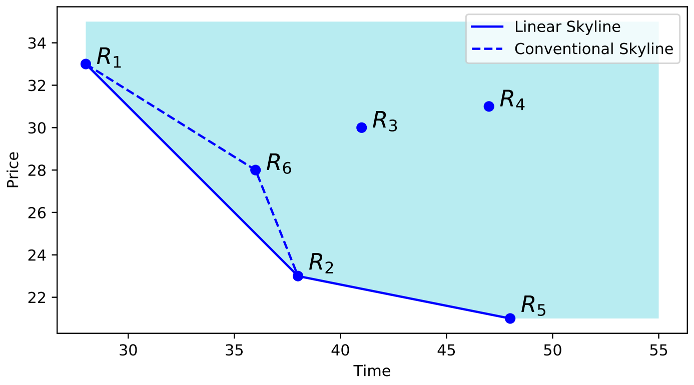

Figure 1.2: Comparison between *linear* skyline (continuous curve) vs *conventional* skyline (dashed curve). Shaded area represents the space dominated by the linear skyline.

by the set of routes $LS = \{R_1, R_2, R_5\}$, depicted with a solid line in Figure 1.1, whereas the dashed line represents the conventional skyline.

Now we can easily observe the shopper's dilemma. None of the routes R_1, R_2 or R_5 is strictly better than the others, in that each solution creates a different trade-off between shopping time and shopping cost, and that each can be interesting under different circumstances. Therefore we target the problem of computing a set of *interesting*, *meaningful*, and *diverse* shopping routes.

Unfortunately, as formally shown in Section 3.1, even a simplified version of the PSD query, namely one where each store sells a single product and each product is sold at the same price in every store where it is available, is NP-hard. Therefore we propose as our main contribution (in Section 4) a heuristic solution that relies on a provably correct *pruning framework* in order to efficiently retrieve a sub-optimal linear skyline. We also develop a framework to compare the solutions obtained w.r.t. (costly) optimal ones which we use in our experiments using city-scale realistic datasets (Section 5. Overall we found that our proposed heuristic is robust and able to offer solutions of good quality much faster than the optimal approach.

2

Literature Review

Spatial databases have been extensively studied in the last few decades, leading to developments in query processing techniques, data structures, memory management techniques, and real-time application services. In the early days, researchers only considered Cartesian spaces in terms of calculating distances between objects while processing queries, e.g., [6], [11], [16]. However, in reality, objects can only move along some predefined trajectories defined by networks. Papadias et al. [15] introduced a new data architecture where they separate the spatial entities from the underlying network by indexing every entity dataset using R-trees [10]. Furthermore, they provide algorithms for four common queries in the spatial network database. The use of networks in spatial queries revolutionised the field and led to various types of trip planning queries meant to provide solutions to the users in real-time mostly using spatial network databases.

A typical planning query consists of a source, a destination, and some constraints. The solution would be a route or routes depending on the query that fulfills the constraints as well as the objectives. In recent years, there have been numerous works on solving various types of trip planning queries, e.g., [8], [9], [14], [19], [21].

However, in this chapter, we only discuss the ones that are relevant to the problem we address. In the next sections, we categorize such works based on their constraints and objectives as well as discuss and contrast them against the PSD problem.

2.1 Trip Planning Queries

Li et al. [14] proposed a new type of query called Trip Planning Query (TPQ). Given a set of categories C each having several points of interest (POIs), a source s and destination d, TPQ finds the "best" possible trip that starts from s then visits at least one POI from each category in C and finally reaches d. The quality of a trip can be measured based on distance, travel time, traffic, road condition, or any other quantifiable property of the trip. Moreover, TPQ can be viewed as a generalization of the Travelling Salesman Problem (TSP) [3], [4]. The authors show that TSP can be reduced to TPQ which proves that TPQ is an NP-hard problem. Therefore, they propose four approximation algorithms to answer TPQ queries. Two of these algorithms have approximations ratios based on the total number categories. The first algorithm is called the Nearest Neighbor Algorithm ($\mathcal{A}_{NN}$). $\mathcal{A}_{NN}$ follows a greedy approach where it starts from s and finds the nearest POI p. It then removes the category corresponding to p from C and searches for the nearest POI from the location of p. $\mathcal{A}_{NN}$ expands the trip until all of the categories have been visited and then it adds d to the trip. The authors prove that $\mathcal{A}_{NN}$ gives a $(2^{m+1} - 1)$-approximation w.r.t the optimal solution where m is the total number of categories. Another interesting algorithm the authors propose is the Minimum Distance Algorithm ($\mathcal{A}_{MD}$) which gives m-approximation or $(m + 1)$-approximation solution when m is odd or even respectively. $\mathcal{A}_{MD}$ chooses one representative from each category based on the minimum cost achieved. A POI p is the representative of its category if the sum of the costs between (s, p) and (p, d) is minimum among all other POIs from that category. $\mathcal{A}_{MD}$ then generates the trip from s to d by visiting each of the representative POIs. It adds the nearest representative POI of the last added vertex (a POI or s) starting with s. Next, the authors show that there is a polynomial-time algorithm based on Integer Linear Programming for their TPQ problem. It can generate $\frac{9}{2}\rho-$approximation solution where ρ is the maximum cardinality of a category. They reformulate TPQ to Loop Trip Planning Query (LTPQ) by adding a constraint that $s = d$. They show that a β-approximation algorithm for LTPQ would imply a 3β-approximation algorithm for TPQ which eventually leads them to the proof of the polynomial-time algorithm. Finally, the authors introduce an approximation algorithm in terms of both m and ρ using the Generalized Minimum Spanning Tree (GMST). Moreover, they show that by using a β-approximation algorithm for GMST their algorithm can generate 2β-approximation solution for the TPQ query. Subsequently, the authors show how $\mathcal{A}_{NN}$ and A_{MD} can be implemented in both Euclidean space as well as in road networks. We can easily observe the similarity between PSD and TPQ in terms of the absence of a specific sequence. However, PSD does not have any notion of categories. Besides, TPQ does not generate any linear skyline whereas PSD's final solution is a linear skyline. Nonetheless, we can reduce PSD to a special case and map it to the TPQ query – we discuss this later when we prove that PSD is an NP-hard problem.

In [17], Salgado et al. solve a variant of the TPQ query called the Category Aware Multi-criteria route planning query (CAM). CAM provides an optimal route where the source and the target are located in an indoor venue. Like TPQ, CAM has a list of categories and it returns the optimal route that visits at least one indoor point from each category. However, instead of only minimizing the one criterion (e.g., travel distance) like TPQ, CAM minimizes other relevant attributes. The authors show that, indeed, CAM is NP-hard like TPQ. Therefore, they propose an approximation algorithm called the Global Category Nearest Neighbor (GCNN) algorithm. GCNN uses a greedy technique to extend a candidate partial route and minimizes the route cost. The route cost is a linear combination of the travel cost and static costs. Moreover, the weights of the travel and static costs are user-defined. Similar to TPQ, CAM is different from our PSD query. Even though CAM aims to minimize competing criteria like the PSD query, it is missing the constraint of fulfilling a shopping list. This constraint makes generating a route harder for the PSD query. Moreover, the PSD aims to find a linear skyline where each route is non-dominant under any linear combination whereas the GCNN algorithm uses a user-defined linear combination which generates only one route. Thus, we can not adapt our PSD to CAM and use its solution to solve the PSD query.

Sharifzadeh et al. [19] introduce a new type of query, Optimal Sequenced Route (OSR) query. The OSR query provides the user's starting point and a *sequence* of categories of interest (COIs) the user intends to visit. The result is the route that, departing from the user's starting point, (1) stops by the COIs exactly in the order provided in the sequence (one POI per COI), and that (2) minimizes the cost criterion being considered (e.g., travel distance). Evidently, the OSR query represents a *particular instance* of the TPQ, as its solution must stop by the COIs *exactly* in the *order* (rather than any order) dictated by the sequence. The authors propose three efficient approaches to solve the OSR problem. The first algorithm the authors propose is called Light Optimal Route Discoverer (LORD). LORD is developed to work on Euclidean space. Next, the authors propose an improvement on LORD which uses an R-tree data structure to store the POIs and they call this algorithm R-tree based LORD (R-LORD). Although LORD and R-LORD work well for Euclidean space, it fails to perform in the real road network. Therefore, the authors propose another algorithm called Progressive Neighbor Exploration (PNE). PNE follows a top-down approach where it starts to build a partial route from the first category of points and gradually progresses towards the final category of POIs in the sequence. Interestingly, in the paper [20] Sharifzadeh et al. use a different approach to solve the OSR query. They use a variant of Voronoi diagrams called AW-Voronoi diagrams to solve the query in both vector and metric space. Although this approach is faster in terms of real-time usage, it requires large offline computations as a trade-off. The OSR query is different from our PSD query as the PSD does not have any notion of a sequence. Besides, the OSR query computes only one solution where the PSD requires a linear skyline. Furthermore, in the OSR query a POI can not be a part of multiple categories whereas, in the PSD query, a store can sell multiple products. As a result, we can not map nor compare the PSD query to the

OSR query.

2.2 Skyline Queries

Kriegel et al. [12] propose two algorithms to compute skyline routes using multiple attributes. They argue that in real life multiple attributes need to be taken into account while computing an optimal path. Both their algorithms use a multi-attributed network graph (MAG). In a MAG each edge has an attribute vector that contains the value of each attribute. Furthermore, they define the cost of a path as the linear sum of all the attribute vectors of the edges from that path. The first algorithm the authors propose is called Basic Route Skyline Computation (BRSC). BRSC uses a priority queue to hold the partial routes. It also uses a list to contain the skyline. The priority queue uses a preference function to determine the priority of a partial route. Furthermore, it evaluates a partial route for early pruning using forward estimation and domination check. Nevertheless, BRSC expands the remaining partial routes in every direction by one hop. Later, it adds the new partial routes to the queue. Eventually, BRSC finds a full route and adds it to the skyline if eligible. Finally, BRSC terminates when the queue is empty. The second algorithm the authors introduce is called Advanced Route Skyline Computation (ARSC). In contrast to BRSC, ARSC uses a priority queue of lists. Each list corresponds to a node and contains only routes that end with the corresponding node. Likewise, ARSC uses a list to contain the skyline. Besides, it uses a local pruning technique where a dominated partial route is not further extended. It too terminates when the priority queue is empty. Other than these differences ARSC is analogous to BRSC. Subsequently, the authors show that ARSC performs better then BRSC in large networks. Although the authors are using the skyline paradigm to find the optimal solution, the problems they discussed has different constraints from the PSD query. The PSD requires to fulfill a shopping list for a route to be considered as a full route. Besides, each store in the PSD contains multiple products which makes it computationally expensive to decide which product to buy from which store. Therefore, we can not adapt the PSD query so that we can use either of BRSC and ARSC to solve it.

In [21] Shekelyan et al. proposed Linear Skylines as an alternative to the conventional skyline operator [5]. A linear skyline is guaranteed to be optimal under *all linear combinations* of the considered attributes, and at the same time provides a more concise and diverse answer set. (We discuss the notion of conventional and linear skylines in more detail in Section 4.) To compute a linear path skyline, the authors introduce an algorithm called "BLRSC". It uses a priority queue based on one attribute to visit neighboring nodes in the network starting from the source. It gradually expands its network and generates partial paths until it exhausts every node from the priority queue. Furthermore, the authors apply some pruning techniques to speed up BLRSC. They check the partial paths for linear domination. They show that if a partial path is already dominated by some other partial path or path, then it will surely be dominated by some path in the linear skyline. Consequently, BLRSC does not expand any partial path that is already being dominated which reduces the total computation time. Once a full path is generated, BLRSC tries to insert it into the linear skyline. Besides, if the addition of the new path makes its immediate previous or next paths in the linear skyline (sorted based on only one attribute) dominated, then there can exist paths both to the left or right of the new path in the linear skyline that are now dominated. Thus BLRSC removes such paths from the linear skyline. Subsequently, the authors successfully demonstrated that the linear skyline creates a more concise impression of the conventional skyline. Even though PSD uses a linear skyline as the final solution, it is only a part of our problem. We cannot directly use BLRSC since we have to fulfill a shopping list on the top of computing the linear skyline.

Ahmadi et al. [1] introduce an interesting query called best compromise in-route nearest neighbor queries (BC-IRNN). The BC-IRNN query provides a linear skyline of routes containing a detour from a preferred path. The linear skyline in this query uses the total distance and the total detour in particular. The authors propose one efficient algorithm to solve the BC-IRNN. It uses two upper bounds so that it can prune infeasible solutions. First, it generates an upper bound on the total route distance. Interestingly, it computes such upper bound by generating a route with the minimum detour. For this purpose, it uses the group nearest neighbor query (GNN). Likewise, the second upper bound the algorithm uses is the upper bound on the total detour. In contrast, it computes this upper bound by generating a route with the minimum network distance. It generates such a route by using the centroid between the source and the destination. Once it computes both the upper bounds, it starts generating routes for the query. For instance, it generates routes by using a priority queue based on the optimistic traveling distance of a partial route. It computes the optimistic traveling distance of a partial route using an A*-search that uses an admissible heuristic function. Besides, it checks every dequeued partial route against the upper bounds for early pruning. Subsequently, it expands any remaining routes in every direction avoiding any cycle. Moreover, the algorithm updates the linear skyline when it generates a valid complete route. Finally, it terminates when the queue is empty or the optimistic traveling distance of a partial route exceeds the upper bound of the total route distance. Furthermore, the authors provide a baseline approach to solve the BC-IRNN query. The baseline approach uses TPQ [14] query between every pair of vertices in the preferred path. Subsequently, it generates a linear skyline using the routes with minimum travel distance from every pair of vertices. Although both BC-IRNN and PSD queries produce linear skylines as a solution, both of the problems are fundamentally different. The constraint of completing a shopping list to create a full route makes PSD a harder problem to solve. Thus we can not use the solution of BC-IRNN to solve the PSD query.

Costa et al. [7] introduce a heuristic approach that provides a solution to "In-route task selection in crowdsourcing" (IRTS) problem in real-time. First of all, IRTS tries to minimize detour from a preferred path and maximize the reward for a worker. For instance, a worker will have a preferred path and a budget for traveling maximum distance. Meanwhile, there will be tasks that are available and need someone's presence at the site. Each task will have a reward associated with it. Given the layout, the IRTS problem requires a set of non dominated paths where each path will contain at least one task. Besides, along with the heuristic approach, the authors also propose an approach to compute the exact solution. The exact approach uses a queue to store paths. Each time a path is dequeued, it updates the linear skyline if an update is required. Otherwise, if the path is not eligible for early pruning, it extends the path in increasing order of detour distance. Besides, if an extended path is eligible for early pruning it will not be stored in the queue will be discarded. Once it exhausts all the combination, it provides the exact linear skyline for the IRTS problem. Nevertheless, the exact approach can not provide the solution in real-time since the problem is NP-hard. Therefore, the authors propose a heuristic approach that would provide a solution in real-time. It approximates the exact linear skyline by prioritizing the paths with lower detour distance for a given sequence of tasks. First of all, it uses a task graph TG that includes the source, destination, and a subset of the tasks available that are feasible. Furthermore, it uses the heuristic called detour oriented heuristic (DOH) to find non-dominated paths in increasing order of detour. Meanwhile, the non-dominated paths are included in the linear skyline which in the end is the solution. The IRTS problem is different from the PSD problem since PSD requires routes that fulfill a given shopping list. However, the NP-hardness of both problems makes them similar in the sense that both of them require a heuristic approach to find a solution in real-time. Nevertheless, we can not use the proposed heuristic approach by the authors to solve the PSD due to the difference in the objectives of both problems.

Lettich et al. [13] introduce a new variant of the OSR query called Trade-Off Aware Sequenced Routing (TASeR). TASeR query adds two new constraints on the OSR query such as (1) each POI will have a non-null cost and (2) the user aims to minimize both the travel distance as well as total travel cost. Since the total travel distance and the travel cost can be two competing criteria, the authors use the linear skyline paradigm to find every optimal solution under any linear combination. Their proposed algorithm LS-TASeR uses two upper bounds on the travel distance and the travel cost to prune out infeasible partial routes. It computes the upper bound of the total distance by finding the route with the minimum travel cost. Conversely, it computes the upper bound of the travel cost by finding the route with the minimum travel distance. Moreover, LS-TASeR uses a priority queue to store partial routes based on the travel distance. Also, it prunes out any partial route that is infeasible considering on the upper bounds. In case it dequeues a complete route, it updates the linear skyline accordingly. Otherwise, it extends the partial route maintaining the sequence as well as the ascending order of travel distance. Finally, LS-TASeR terminates if it generates any complete route that has the travel cost equal to the minimum travel cost possible. As we can see, the TASeR query is fairly similar to the PSD query. However, the absence of a sequence makes the PSD a harder problem to solve. As a result, we can not use LS-TASeR to solve the PSD query.

In conclusion, even though the above discussions show that much work has been done w.r.t. routing queries in road networks and the use of the skyline paradigm in those, PSD is unique mainly due to the notion of a shopping list and of stores as POIs that are no distinct in nature from each other, i.e., all can be considered in the same COI, but are distinct in the sense that they can contribute differently to the solution."

3

Preliminaries

We assume as underlying framework for the PSD query a city's road network modeled by an undirected graph $G(V, E, W)$, where V is a set of vertices that represent road intersections and end-points, E is the set of edges containing all road segments, and W indicates the weight of edges in E. The weight of an edge connecting two vertices v_i and v_j, denoted by $w(v_i, v_j) \in W$, is given by the *time* needed to traverse the associated road network segment.

There are four main entities within the PSD's model: stores, customers, shopping lists, and personal shoppers.

- A *store* τ_j is located at a vertex $v_{\tau_j} \in V$, and the set of all stores forms a set T. Each store sells a specific selection of products. A product has a positive cost, which may differ between stores, and we denote by $c(i, \tau_j)$ the cost of a product i at a store τ_j. For simplicity we assume that all stores have an arbitrarily large inventory of all products it sells.

- A *customer* σ wants to buy a set of products. To this end it issues a request to a personal shopper in the form of a shopping list (described next). We assume the products need to be delivered at the customer's location, which is a vertex in V denoted by v_σ.

- λ denotes the *shopping list* issued by σ and we represent it as a set of pairs $\langle i, q \rangle$ where each i is a product identifier and q represents the number of required units of such product.

- Finally, the *personal shopper* ω is in charge of satisfying a customer's request, i.e., fulfil and deliver the shopping list, and we denote the shopper's location by a vertex v_ω in V.

The answer of a PSD query is a set of "shopping routes", each representing a sequence of stores to be visited and yielding a shopping cost (i.e., the monetary cost of acquiring all required products in the visited stores) as well as shopping time (i.e., the time needed to do all needed shopping and deliver the products to the customer). Next, we define such concepts formally.

Definition 1 (Shopping route and its feasibility). *A shopping route θ_i over G is a sequence of stores $\langle \tau_1^i, \ldots, \tau_n^i \rangle$. Furthermore, given a shopping list λ we say that θ_i is feasible w.r.t. λ if all products in λ are sold in at least one store in θ_i.*

Definition 2 (Shopping time). *Let θ_i represent a feasible shopping route w.r.t. a customer's shopping list λ. We define the shopping time associated with θ_i, denoted as $ST(\theta_i)$, as the time needed by the shopper to traverse the path that departs from v_ω, visits the stores according to the order defined by θ_i, and finally ends at v_σ, i.e.:*

$$ST(\theta_i) = mTT(v_\omega, \tau_1) + \sum_{i=1}^{n-1} mTT(\tau_i, \tau_{i+1}) + mTT(\tau_n, v_\sigma)$$

where $mTT(v_i, v_j)$ is the time required by the fastest path connecting vertices v_i and v_j (though other notions of travel cost could also be used).

Definition 3 (Shopping cost). *Let λ be the shopping list issued by a customer σ and let θ_i be a feasible shopping route w.r.t. λ. Then, we define the shopping cost of θ_i as follows:*

$$SC(\theta_i) = \sum_{\langle j,q \rangle \in \lambda} (c(j, ss(\theta_i, j)) \times q)$$

where $ss(\theta_i, j)$ is a function that returns the store in θ_i from which the product j in λ is to be bought [1].

Note that due to the assumption of arbitrarily large inventories at each store, it is safe to assume that all units of a given product can be acquired at a single store. The case where stores' inventories are limited, and more than one store may be required to fulfil the need for a given product, is left as future work for the time being.

Clearly, there may be different combinations of stores that could interest the shopper, each with its own trade-off between shopping cost and shopping time. Thus, we propose an approach where the shopper can do his/her own evaluation to choose the one combination that best fits his/her immediate needs/goals. We model such notion of "interestingness" by leveraging the concept of *linear skyline*, discussed next.

A linear skyline always represents the subset of some *conventional* skyline; to distinguish between the two, we start by providing their definitions. Let θ_i be a feasible shopping route. The two criteria that are being optimized are $SC(\theta_i)$ and $ST(\theta_i)$, i.e., the cost vector that is being optimized is $CV(\theta_i) = \langle SC(\theta_i), ST(\theta_i)\rangle$.

Definition 4 (Conventional domination)**.** *Let θ_i and θ_j be two shopping routes. Then, we say that θ_i conventionally dominates θ_j, denoting by $\theta_i \prec \theta_j$, if:*

$$\Big(\big(SC(\theta_i) < SC(\theta_j)\big) \wedge \big(ST(\theta_i) \leq ST(\theta_j)\big)\Big) \vee \Big(\big(SC(\theta_i) \leq SC(\theta_j)\big) \wedge \big(ST(\theta_i) < ST(\theta_j)\big)\Big)$$

From this, it follows the definition of *conventional skyline.*

Definition 5 (Conventional skyline)**.** *Let Θ be a set of shopping routes. Then, we define the conventional skyline of Θ to be the set of shopping routes that are not conventionally dominated, i.e.,* $\{\theta_i \in \Theta| \;\nexists \theta_j \in \Theta : \theta_j \prec \theta_i\}$.

[1] As discussed in the following Section, such store-product assignment is determined as the solution is obtained.

A linear skyline consists of the subset of a conventional skyline that is optimal under *all linear combinations* of the competing cost criteria [21]. Hence, in the scenario considered in this work a linear skyline is composed of combinations of stores that minimize the linear combination $\mathcal{F} = \delta_1 SC(\theta_i) + \delta_2 ST(\theta_i)$, with $\delta = (\delta_1, \delta_2)$ being a weight vector in $\mathbb{R}^2_{>0}$. Note that we find the optimal solution for all such δ, i.e., we do not require any weight vector to be provided beforehand. In the following, we remind the definition of δ-dominance [21], which determines when a combination of stores linearly dominates another one provided a particular δ.

Definition 6 (Linear dominance)**.** *A shopping route θ_i is said to δ-dominate another θ_j if and only if $\delta^T CV(\theta_i) < \delta^T CV(\theta_j)$.*

From the definition of linear dominance follows the definition of *linear skyline* [21].

Definition 7 (Linear skyline)**.** *Let Θ be a set of shopping routes. Let also $\Theta' = \{\theta_1, \ldots, \theta_K\} \subseteq \Theta$. Then, we say that Θ' linearly dominates a shopping route $\theta \in \Theta$ if and only if:*

$$\big(\exists \theta' \in \Theta' : \theta' \prec \theta\big) \vee \big(\forall \delta \in \mathbb{R}^2_{>0}\ \exists \theta' \in \Theta' : \delta^T CV(\theta') < \delta^T CV(\theta)\big)$$

The maximal set of linearly non-dominated combinations of stores is referred to as linear skyline and it can be seen as an ordered set w.r.t. the first cost criteria.

Testing the condition on the right hand side of Definition 7 would require to try out every possible vector $\delta \in R^2_{>0}$, which would be computationally impractical. As such, in [21] the authors consider the problem from a different perspective, giving it an intuitive geometrical interpretation. More precisely, the authors observe that a route θ is linearly dominated only if it lies *above* the segment connecting any pair of shopping routes $\{\theta_q, \theta_j\}$ belonging to the linear skyline – this fact is formally denoted by $\{\theta_q, \theta_j\} \prec_L \Theta$. Then they show that it is possible to quickly test whether a route is linearly dominated or not (and thus determine if it can be added to the linear skyline).

Table 3.1 summarizes the main notation used throughout the rest of the thesis.

Symbol	**Semantics**
$\tau \in T$	A store in the set of stores T
$\theta_i = \langle \tau_1^i, \ldots, \tau_n^i \rangle$	Shopping route
$\tau_j^i \in R_i$	The j-th store visited by θ_i
v_ω	Shopper's location
v_σ	Customer's delivery location
λ	Shopping list
$ST(\theta_i)$	Shopping time of route θ_i
$SC(\theta_i)$	Shopping cost of route θ_i
$\theta_i \prec \theta_j$	θ_i conventionally dominates θ_i
$\{\theta_i, \theta_j\} \prec_L \theta_q$	θ_q linearly dominated by $\{\theta_i, \theta_j\}$
θ^{SC}	Comb. of stores with minimum shopping cost
$ST^U = ST(\theta^{SC})$	Shopping time upper bound

Table 3.1: Main notation used throughout the thesis.

3.1 PSD query's NP-hardness

The PSD query can be demonstrated to be NP-hard by showing that any instance of a Trip Planning Query (TPQ), a problem known to be NP-hard [14], can be reduced to a PSD one.

Let us suppose to have a road network G, a set C of categories of interest (COIs), and a set of points of interest (POIs) P (also vertices in G), each belonging to some COI $c \in C$. Given a starting location v_s, an ending location v_d, and a subset of COIs $C' \subseteq C$ provided by the user, a TPQ requires to compute the route with minimum cost (e.g., travel time) from v_s to v_d that visits exactly one POI from each COI in C'.

In the following we show how we can reduce any TPQ instance to a PSD one. v_s can be trivially mapped to the shopper's location v_ω, while v_d can be trivially mapped to the customer's location v_σ. Next, we map the notion of subset of COIs to visit $C' \subseteq C$ with that of shopping list λ. Each COI $c \in C$ in TPQ can be mapped to the notion of a *product* in PSD, with the POIs within that category representing the *stores* selling that product. Then, any subset

$C' \subseteq C$ of COIs to visit can be expressed as a shopping list λ, where we require to buy *exactly one unit* of each such fictitious products in λ. Finally, recall that TPQ requires to compute the route minimizing the considered cost criterion (e.g., travel time), while the PSD query requires to find the set of linearly non-dominated routes w.r.t. shopping cost and shopping time. Considering that in TPQ there are no costs associated to POIs, we are free to impose that all the products have the same cost across all the stores in which they are sold, hence all the shopping routes satisfying λ will have the same shopping cost. Therefore, the linear skyline will be a singleton containing the route with minimum shopping time, i.e., the optimal solution required by TPQ.

3.1.1 Linear skylines, total order, and efficient insertions

Let ST be the cost criterion used to order the evaluation of shopping routes, let θ^{ST} be the route yielding minimum shopping time, let LS be the skyline under construction – initially $LS = \{\theta^{ST}\}$ –, and let θ be a shopping route considered for insertion. Shekelyan et al. [21] demonstrate that it suffices to verify whether θ's left *neighbor* in the skyline, LS_K, i.e., the shopping route being the closest to θ w.r.t. to ST, with $ST(LS_K) \leq ST(\theta)$[2], *conventionally dominates* θ. In other words, it suffices to verify whether $ST(LS_K) \leq ST(\theta)$. If $ST(LS_K) > ST(\theta)$ then θ qualifies for insertion and it is necessary to verify if any route in the skyline becomes dominated due to the insertion of θ. First, it is necessary to verify whether $ST(LS_K) = ST(\theta)$: if that's the case, LS_K is removed from the skyline as it is conventionally dominated by θ. Subsequently, thanks to the order in which linearly non-dominated shopping routes are discovered, it is sufficient to verify whether θ's left neighbor, LS_K, is linearly dominated by LS_{K-1} (LS_K's left neighbor) and θ, i.e., verify if $\{LS_{K-1}, \theta\} \prec_L LS_K$ is true, and remove LS_K if this is the case. The procedure is iterated until θ's current left neighbor cannot be removed from the skyline or it represents the first route in the skyline, i.e., the shopping

[2]This corresponds to the last shopping route inserted into the skyline, or θ^{ST} in case no route was previously inserted.

route with shopping time. Considering that linear skylines typically contain few elements, the overall cost of an insertion check can be assumed to be, on average, *constant*.

4

Computing Solution for Personal Shopper's Dilemma Query

In this chapter, we propose two approaches to solve the PSD query. The first one, BSL-PSD, is a baseline capable of computing *optimal* linear skylines. Given some shopping list, the strategy employed by BSL-PSD evaluates in strict increasing order of shopping time shopping routes that fulfill the list, and orchestrate the construction of the skyline accordingly. As shown in Section 3.1, computing PSD queries is NP-hard, and thus computing optimal linear skylines becomes unfeasible when the number of stores in a road network or the shopping list size becomes large. We thus propose a second approach, APX-PSD (Section 4.2). The idea behind APX-PSD is to cluster stores spatially and then generate and combine shopping routes from such clusters, rather than from the entire set of stores, to reduce the number of shopping routes to possibly evaluate. To this end APX-PSD first superimposes a quad-tree over the stores within the considered geographical area. Then, given a PSD query APX-PSD performs a depth-first search (DFS) of the quad-tree that is *driven* by a *scoring function.* Such function directs the search towards the tree leaves that are deemed the most "*promising*" in terms of shopping time and shopping cost. The generation and expansion of shopping routes are thus conducted within single partitions, rather than on the entire set of stores, to considerably reduce the candidates' search space.

4.1 A baseline approach: BSL-PSD

The first approach we propose to solve the PSD query is BSL-PSD, a baseline capable of computing *optimal* linear skylines. Before delving into its presentation, two important points of discussion concerning the strategy employed by BSL-PSD to compute the linear skyline are: (i) our choice of *imposing* a *total order* over the evaluation of shopping routes – from here on shopping routes under evaluation will be called *candidate* routes – and, (ii) the *cost criterion* used to define such order. For what concerns (i), [21] shows that having a total order over the candidates allows efficient insertions while constructing a linear skyline. Indeed, in order to determine if the candidate qualifies for insertion it suffices to verify whether a candidate is *conventionally dominated* by the last shopping route inserted into the skyline.

Observation 1. *Let us choose one of the two cost criteria as the primary cost. Let us also suppose that it is possible to compute the shopping route yielding the minimum possible value for the primary cost, which in turn establishes an upper bound for the second cost criterion, i.e., shopping routes having the second cost criterion larger than the upper bound are surely dominated by the associated route. Let us finally assume that linearly non-dominated shopping routes are evaluated in increasing order w.r.t. the primary cost criterion. Then, it is sufficient to perform a single conventional domination check to determine if a route qualifies for insertion into the linear skyline.*

For instance, Let ST be the cost criterion used to order the evaluation of shopping routes, let θ^{ST} be the route yielding minimum shopping time, let LS be the skyline under construction – initially $LS = \{\theta^{ST}\}$ – and let θ be a shopping route considered for insertion. Shekelyan et al. [21] demonstrate that it suffices to verify whether θ's left *neighbor* in the skyline, LS_K, i.e., the shopping route being the closest to θ w.r.t. to ST, with $ST(LS_K) \leq ST(\theta)$[1],

[1]This corresponds to the last shopping route inserted into the skyline, or θ^{ST} in case no

conventionally dominates θ. In other words, it suffices to verify whether $ST(LS_K) \leq ST(\theta)$. If $ST(LS_K) > ST(\theta)$ then θ qualifies for insertion and it is necessary to verify if any route in the skyline becomes dominated due to the insertion of θ. First, it is necessary to verify whether $ST(LS_K) = ST(\theta)$: if that's the case, LS_K is removed from the skyline as it is conventionally dominated by θ. Subsequently, thanks to the order in which linearly non-dominated shopping routes are discovered, it is sufficient to verify whether θ's left neighbor, LS_K, is linearly dominated by LS_{K-1} (LS_K's left neighbor) and θ, i.e., verify if $\{LS_{K-1}, \theta\} \prec_L LS_K$ is true, and remove LS_K if this is the case. The procedure is iterated until θ's current left neighbor cannot be removed from the skyline or it represents the first route in the skyline, i.e., the shopping route with shopping time. Considering that linear skylines typically contain few elements, the overall cost of an insertion check can be assumed to be, on average, *constant*.

For the moment we assume said total order to be enforced by means of a min-priority queue Q, combined with some suitable generation scheme that progressively extends shopping routes according to the chosen cost criterion, and focuses on *which* criterion should be used to define the evaluation order. Consider an evaluation strategy that evaluates candidates according to their increasing shopping cost. As it will be shown later on, computing the cost of a shopping route w.r.t. a shopping list can be done in constant time with the use of appropriate data structures. On the other hand, computing a candidate's shopping time always requires to compute the fastest path visiting its stores, i.e., it is necessary to solve an instance of the trip planning query, which is NP-hard. Thus, using shopping time rather than shopping cost allows for an evaluation strategy with greater pruning potential and thus smaller computational costs. Now, note that evaluating in increasing order of shopping cost does not take into account the spatial information provided by a PSD query (i.e., shopper's and customer's delivery locations), hence this strategy may end up evaluating shopping routes where the associated fastest paths are long and thus likely being dominated by faster routes. Overall, evaluating in in-

route was previously inserted.

creasing order of shopping time is less penalizing when incurring in dominated candidates, as the cost of computing their shopping cost is negligible.

At this point we can start focusing on the baseline's presentation. First, we briefly describe three pre-computed lookup tables BSL-PSD uses to speed up key computations. Subsequently, we introduce the notion of skyline *upper bounds*, and specify how they are computed and used within the baseline. We then proceed to introduce the set of pruning criteria and the generation scheme the baseline uses to evaluate candidates. Finally, we conclude by formally introducing BSL-PSD.

We pre-compute three *lookup tables*. The first table keeps track of the stores where each product is sold, with the list of stores each product is associated with being ordered in increasing order of cost. This is similar to a typical text-based inverted list and allows us to determine in constant time a subset of stores yielding minimum shopping cost for any shopping list λ. The second table stores pairs $(product, store)$, with each pair being associated with the cost of the product at that store. Thus, for any shopping list λ it is possible to compute in constant time if a combination of stores can fulfill it, as well as its shopping cost. The third table stores the travel time of the fastest path connecting any pair of stores.

There are two important shopping routes BSL-PSD uses to delimit the candidates' search space, i.e., the one yielding *minimum shopping time*, θ^{ST}, and the other yielding *minimum shopping cost*, θ^{SC}. The former provides the *shopping cost upper bound*, SC^U, while the latter provides the *shopping time upper bound*, ST^U.

Computing θ^{SC}. Finding out a shopping route with minimum shopping cost requires to find out a subset of stores in T where each product in λ can be bought at minimum cost – this can be done in constant time by using the *first* pre-computed *lookup table*. Later on we show that during the candidates' evaluation it suffices to find the first shopping route with shopping cost equal to that of θ^{SC} to terminate the baseline's execution, as subsequent candidates are ensured to be longer (and thus dominated by such route).

Computing θ^{ST}. To find out the shopping route with minimum shopping time requires to compute a variant of the trip planning query, i.e., it requires to find out the combination of stores that (i) yields the fastest path connecting the shopper's location, the stores, and the customer's delivery location, and (ii) that satisfies λ. BSL-PSD generates (and thus evaluates) a combination of stores in increasing order of shopping time. Consequently, the first candidate that fulfills λ always corresponds to θ^{ST}. We elaborate more on this shortly.

BSL-PSD evaluates candidate shopping routes in increasing order of shopping time and constructs the linear skyline accordingly. To do so, it generates shopping routes according to the chosen order, coupled with a set of pruning criteria used to reduce the search space. We start by presenting the first pruning criterion, a lemma that allows to update the shopping cost upper bound SC^U as non-dominated routes are progressively inserted into the linear skyline.

Lemma 1. *Let Q be the min-priority queue used to order the evaluation of shopping routes in strict increasing order of shopping time. Let λ be a shopping list. Let then θ be the last candidate popped from Q: if θ fulfills λ and qualifies for insertion into the linear skyline, then it is possible to set $SC^U = SC(\theta)$.*

Proof. The mechanism used to generate shopping routes in strict increasing order of shopping time ensures that any route θ' popped from Q after θ yields $ST(\theta') \geq ST(\theta)$. Then, by virtue of the notion of linear skyline (Definition 7) θ' may qualify for insertion only if $SC(\theta') < SC(\theta) = SC^U$. □

The above lemma allows to progressively enforce stricter upper bounds on shopping cost, and thus limit the candidates' space as linearly non-dominated solutions are found out. Let us now introduce a lemma that allows to early terminate BSL-PSD.

Lemma 2. *Let θ be the last linearly non-dominated shopping route added to the linear skyline. If $SC(\theta) = SC(\theta^{SC})$ we are guaranteed to have found out all the linearly non-dominated shopping routes and the evaluation can terminate.*

Proof. We know that shopping routes are popped from Q in increasing order of shopping time. We also know that θ yields the minimum possible shopping cost. Consequently, any sequence θ' that fulfills λ popped from Q after θ necessarily has $ST(\theta') \geq ST(\theta)$ and $SC(\theta') \geq SC(\theta) = SC(\theta^{SC})$, i.e., Definition 7 ensures that θ' is dominated. □

In other words, the above lemma guarantees that once a shopping route with minimum shopping cost $SC(\theta^{SC})$ and that satisfies λ is found, the baseline can terminate as it is not possible to find further non-dominated routes.

Let us now introduce a slightly modified notation for shopping routes that serves to relate them to the notion of *ranked minimum detours*. Such notation will be used later on to illustrate the route generation scheme used by the baseline to evaluate candidates in increasing order of shopping time.

Definition 8 (Shopping routes and minimum detours). *We define a shopping route $\Theta = \langle \tau_{k_1}, \tau_{k_2}, \ldots, \tau_{k_{|\theta|}} \rangle$ to be the route where $\tau_{k_i} \in ST$ represents the i-th store visited by Θ and that yields the k_i-th minimum detour when added to the fastest path between the store that precedes it in Θ (or the shopper's location v_ω, if $i = 1$) and the customer's delivery location v_σ.*

Algorithm 1 presents BSL-PSD, along with the generation scheme used to evaluate shopping routes in strict increasing order of shopping time. First, BSL-PSD prunes from T those stores that do not offer any of the products required in λ (line 1, function PruneStores). This immediately reduces the candidates' search space. Next, BSL-PSD executes two single-source shortest path searches (lines 2 and 3): the first one originates from the shopper's location v_ω and targets the set of stores T as well as the delivery location v_σ. The second one originates from the delivery location v_σ and targets the set of stores T. These searches return the travel times between v_ω and v_σ, and between any pair (v_ω, τ) and (τ, v_σ), with $\tau \in T$. Such travel times are then appropriately combined with those in the lookup table holding the travel times between any pair of stores to support efficient implementations of the functions MinDetour and NextMinDetour (discussed shortly). More specifically, for every store $\tau \in T$ we sum the travel time with any other store $\tau' \in T$ with that

```
Algorithm 1: BSL-PSD
  Input : Road network $G$, set of stores $T \subseteq V$, shopper's current
          location $v_\omega$, customer's delivery location $v_\sigma$, shopping list $\lambda$,
          shopping route yielding minimum shopping cost $\theta^{SC}$.
  Output: Linear skyline $LS$.
1  $T \leftarrow$ PruneStores$(T, \lambda)$
2  $T \leftarrow$ DijkstraMultipleTarget$(v_\omega, T \cup \{v_\sigma\})$
3  $T \leftarrow$ DijkstraMultipleTarget$(v_\sigma, T)$
4  $LS \leftarrow \emptyset$
5  $SC^U \leftarrow +\infty$
6  $Q \leftarrow \{\langle \tau_{k_1=1} =$ MinDetour$(v_\omega, v_\sigma, \emptyset, T)\rangle\}$
7  while $Q \neq \emptyset$ do
8      $\theta = \langle \tau_{k_1}, \cdots, \tau_{k_{|\theta|}} \rangle, Q \leftarrow$ Pop$(Q)$
9      if SatisfyList$(\lambda, \theta)$ then
10         if $SC(\theta) < SC^U$ then
11             $LS \leftarrow$UpdateLS$(LS, \theta)$            // (Observation 1)
12             $SC^U \leftarrow SC(\theta)$
13         if $SC^U = SC(\theta^{SC})$ then return LS
14     $\theta^s = \langle \tau_{k_1}, \cdots, \tau_{k_{|\theta|}} \rangle \oplus$ MinDetour$(\tau_{k_{|\theta|}}, v_\sigma, \theta, T)$
15     $\theta^p = \langle \tau_{k_1}, \cdots, \tau_{k_{|\theta|-1}} \rangle \oplus$ NextMinDetour$(\tau_{k_{|\theta|-1}}, v_\sigma, \theta, T)$
16     $Q \leftarrow$ Push$(\{\theta^s, \theta^p\}, Q)$
17 return $LS$
```

between τ' and v_σ. This yields a list of travel times which, once *sorted*, allows finding quickly the store yielding the k-th minimum detour when added to the fastest path connecting τ and v_σ.

BSL-PSD then goes on to set the initial state of several entities, namely, that of the linear skyline LS, the shopping time upper bound ST^U, and the priority queue Q, which initially holds the partial shopping route containing the store minimizing the detour w.r.t. the fastest path connecting v_ω and v_σ. Note that such store is found via the function MinDetour, which uses the information computed previously. Subsequently (while cycle, line 7), BSL-PSD starts generating and evaluating shopping routes in increasing order of shopping time. For each candidate θ popped from Q (line 8), BSL-PSD first verifies whether θ fulfills λ (line 9). If such condition holds, the baseline goes on to verify whether $SC(\theta) < SC^U$ (line 10): if this condition does not hold, then θ is conventionally dominated (lemma 1) and can be discarded. Otherwise, θ can be inserted into the skyline according to the procedure outlined in observation 3.1.1 (function UpdateLS, line 11), and SC^U can be tightened (line 12, by virtue of lemma 1). If $SC^U = SC(\theta^{SC})$ the algorithm immediately terminates (line 13), as it is not possible to find further linearly non-dominated shopping routes (by virtue of lemma 2).

Lines 14–16 represent the generation scheme used by BSL-PSD to evaluate shopping routes in increasing order of shopping time, with $\oplus$ symbolizing the *append* operation. For each shopping route θ popped from Q the baseline generates two new shopping routes, namely, θ^s and θ^p. θ^s is the shopping route generated by adding a store at the end of θ that (1) does not already appear in θ and that (2) *minimizes* the detour distance when added to the fastest path between $\tau_{k_{|\theta|}}$ and the delivery location v_σ – such store is found out via the function MinDetour. θ^p is the shopping route generated by replacing the last store visited by θ, i.e., $\tau_{k_{|\theta|}}$, with the store yielding the $(k_{|\theta|} + 1)$-th minimum detour when added to the fastest path connecting $\tau_{k_{|\theta|-1}}$ and v_σ – such store is found out via the function NextMinDetour. Similarly to θ^s, note that we require the store found by NextMinDetour to not already appear in θ. Observe that both θ^s and θ^p have shopping time greater or equal than θ.

Finally, we can state two important features regarding BSL-PSD:

Theorem 1. *BSL-PSD evaluates all possible shopping routes.*

Proof. We prove it by induction on the size of shopping routes. First we show that the baseline examines every shopping route with $|\theta| = 1$. During the initialization phase BSL-PSD initializes the min-priority queue with the shopping route yielding the minimum detour distance between v_ω and v_σ, i.e., $\theta = \langle \tau_{k_1=1} \rangle$ (line 6). Then, every time some shopping route $\theta = \langle \tau_{k_1} \rangle$ is popped from Q, BSL-PSD generates a shopping route θ^p (line 15) where τ_{k_1} is replaced with τ_{k_1+1}, i.e., the shop that yields the k_1-th minimum detour between v_ω and v_σ is replaced with the one that yields the $(k_1 + 1)$-th minimum detour between v_ω and v_σ. Let us now assume that BSL-PSD generates all the shopping routes of length n. Then, for any route $\theta = \langle \tau_{k_1}, \ldots, \tau_{k_n} \rangle$ we know that the generation scheme generates a route $\theta^s = \langle \tau_{k_1}, \ldots, \tau_{k_n}, \tau_{k_{n+1}=1} \rangle$ (line 14), where the store $\tau_{k_{n+1}=1}$ yields the minimum detour distance when added to the fastest path connecting τ_{k_n} and v_σ. Then, during subsequent iterations line 15 guarantees that θ^s allows to generate a whole set of shopping routes of length $n+1$ where the shop yielding the k_{n+1}-th minimum detour between k_n and v_σ is replaced with that yielding the $(k_{n+1}+1)$-th minimum detour.

□

Theorem 2. *BSL-PSD evaluates shopping routes in strict increasing order of shopping time.*

Proof. We prove it by contradiction. First, observe that the min-priority queue Q ranks the shopping routes it holds in increasing order of shopping time. Hence, the only event that may violate the theorem's thesis is if a route popped from Q has shopping time smaller than those popped (and thus evaluated) previously. Note, however, that such event is impossible, as the generation scheme used between lines 14–16 ensures that the shopping routes θ^s and θ^p have shopping time always greater or equal than that of the popped route θ from which they are generated.

□

4.1.1 BSL-PSD's complexity

BSL-PSD first filters out from T stores that do not offer any product in λ. Thanks to the use of pre-computed lookup tables, the cost of such operation is $O(|\lambda| \cdot |T|)$. Let us now denote by $T' \subseteq T$ the subset of stores that offer at least one product in λ. Let us also denote by $N = O\big(|T'|!\big)$ the overall number of candidate shopping routes to possibly evaluate, by S the number of candidate routes θ^s generated at line 14, and by P the number of candidate routes θ^p generated at line 15, with $1 + S + P = N$.

The first major operations conducted by BSL-PSD are the two single-source shortest path searches at lines 2 and 3. Both searches have a common component cost of $O\big((|E|+|V|)\cdot log|V|\big)$, i.e., the cost inherent to conducting a Dijkstra search. Recall then that the second search is followed by the creation of a set of sorted lists, one per every store $\tau \in |T'|$, each holding the travel times that result from adding any other store in $|T'|$ in the fastest path connecting τ and v_ω (an information already available in the appropriate lookup table). Thus, creating and sorting such lists has cost $O(|T'| \cdot OC)$, where OC is the sorting algorithm cost.

Such operations are then followed by the baseline's time-dominant component, represented by the while loop (lines 7–16). The cost of executing such component can be expressed as $O\big(Nlog_2N + N2|T'| + S|\lambda| + |\lambda| \sum_{j=1}^{P} |\theta_j^p| + N\big)$. The first term represents the cost associated with the use of a min-priority queue to order the evaluation of candidates. The second term represents the cost of generating two candidate routes from each candidate popped from the queue (i.e., the cost of executing MinDetour and NextMinDetour). By using the aforementioned sorted lists, finding the k-th minimum detour from any store to the delivery location has cost $O(|T'|)$, which in turn yields $O(N2|T'|)$. The third term represents the cost of computing the shopping cost of candidates θ^s generated at line 14 from some candidate θ. Recall that each product in λ shall be bought at the store selling it for the lowest price among those in θ. Hence, computing the shopping cost of a candidate θ^s can be done in $O(|\lambda|)$ by (1) using the lookup table providing the selling price of any product in any store and (2) by keeping track of the minimum price each product in λ is bought among θ's stores. The fourth term represents the cost of computing the shopping cost of candidates θ^p generated at line 15. Recall that such operation requires to replace the last store in some θ with the one yielding the next minimum detour. Now, observe that some of the products in λ may be bought at the store being replaced, hence in the worst case its replacement requires to find out for each product in λ which store among those in θ^p sells it at minimum price. Finally, the fifth term represents the cost needed to check if candidate routes qualify for insertion into the linear skyline.

4.2 An approximated approach: APX-PSD

The baseline's major drawback lies in its necessity to possibly evaluate a number of shopping routes that is *factorial* in the number of stores, thus limiting its scalability and applicability to real-world scenarios. In order to overcome that, we propose APX-PSD, an approach that trades the optimality of linear skylines for a greatly reduced number of candidates to evaluate. The key idea behind APX-PSD is to consider stores at a *coarser granularity*, i.e., *partitions* of stores rather than individual stores to generate – and thus evaluate – shopping routes from those partitions, rather than the whole set of stores, that look the most "promising."

To partition the stores of a road network APX-PSD superimposes a point-region (PR) quad-tree [18] over the minimum bounding rectangle (MBR) enclosing the stores. This corresponds to the *root* quadrant of the quad-tree. The PR quad-tree is then constructed by recursively splitting each quadrant having a number of elements (stores) larger than a given capacity threshold in four sub-quadrants. Quadrants that are split during the construction process are the quad-tree *intermediate* nodes, while unsplit ones make up the *leaves*. Given that it relies only on the spatial location of the stores, APX-PSD can pre-compute a quad-tree over the stores of a road-network, along with information concerning travel time *between partitions* and *statistics* on the products each quadrant (at any level of the tree) holds. Such statistics are subsequently used to drive the generation and evaluation of candidate routes, i.e., to decide which quadrants are the most *promising* w.r.t. shopping time and shopping cost.

We first define the notions of *travel time* between *quadrants* and *travel time* between a *vertex* and some *quadrant*, as they are key to the evaluation strategy employed by APX-PSD.

Definition 9 (Travel time between quadrants). *Let Q be a PR quad-tree superimposed over the stores $T \subseteq V$. Let P_i and P_j be two quadrants of Q. Then, the travel time between P_i and P_j, denoted by $mTT(P_i, P_j)$, is defined by $mTT(\tau_k^i, \tau_l^j)$, with $\tau_k^i \in P_i$, $\tau_l^j \in P_j$, and $\nexists(\tau_m^i, \tau_n^j)$, with $\tau_m^i \in P_i$, $\tau_n^j \in P_j$,* *such that* $mTT(\tau^i_m, \tau^j_n) < mTT(\tau^i_k, \tau^j_l)$.

Definition 10 (Travel time between a vertex and a quadrant)**.** *Let* Q *be a PR quad-tree superimposed over the stores* $T \subseteq V$. *Let* $v \in V$ *be some vertex and* $P \in Q$ *be some quadrant. Then, the travel time between* v *and* P_j, *denoted by* $mTT(v, P)$, *is defined by* $mTT(v, \tau^j)$, *with* $\tau^j \in P$, *and* $\nexists(v, \tau^j_n)$, *with* $\tau^j_n \in P_j$, *such that* $mTT(v, \tau^j_n) < mTT(v, \tau^j_l)$.

Observe that Definition 9 requires to find out the pair of stores, each belonging to one of the considered quadrants, yielding minimum travel time, while Definition 10 requires to find out the store within a quadrant that *minimizes* travel time w.r.t. some given vertex. Finally note that travel times between partitions can be pre-computed. Both definitions allow APX-PSD to enforce upper bounds on shopping time when generating shopping routes from different partitions – to be discussed further shortly.

With a PR quad-tree in place, APX-PSD can then proceed to process PSD queries. Its evaluation strategy relies on a *depth-first* search (DFS) of the quad-tree driven by a *scoring* function we introduce below. Given the set of shopping routes under construction (initially empty), such function estimates how "good" each quad-tree quadrant, be it an intermediate node or a leaf, is in terms of shopping time and shopping cost w.r.t. the characteristics of a PSD query and the shopping routes generated so far, thus driving the generation and evaluation of shopping routes towards quadrants that are deemed the "most promising".

Definition 11 (Quad-tree quadrant scoring function)**.** *Let us consider a shopping list* λ, *a PR quad-tree* Q *superimposed over* $T \subseteq V$, *and a quadrant* $P_i \in Q$ *(either intermediate node or leaf) that needs to be scored w.r.t.* λ. *Let us assume that* P_i *must be reached from either the shopper's location* v_ω *or some other quadrant* $P \in Q$. *Let us further consider that* P_i *contains* m *of the products specified in* λ; *let us denote by* λ^{P_i} *the subset of such products, and by* $\lambda^{P_i}_k$ *the average cost of the* k*-th product within* λ^{P_i} *among* P_i*'s stores.*

Let us finally denote the minimum travel time needed to depart from v_ω, *visit one or more stores in* P_i, *and finally reach the customer's delivery location*

v_σ by $ST_{P_i}^{v_\omega} = mTT(v_\omega, P_i) + mTT(v_\sigma, P_i)$. Analogously, departing from any $P \in Q$ yields $ST_{P_i}^{P} = mTT(P, P_i) + mTT(v_\sigma, P_i)$. For the sake of simplicity, in this context we denote either v_ω or P by x. Let us finally denote by $maxPrice$ the price of the most expensive product in any store from λ. Then, we define the score of P_i w.r.t. x and λ as follows:

$$F(x, \lambda, P_i) = \begin{cases} \frac{ST_{P_i}^{x}}{ST^U} + \frac{\sum_{k=1}^{|\lambda^{P_i}|} \left(\lambda_k^{P_i} / maxPrice\right)}{m} & m > 0 \\ +\infty & m = 0 \end{cases} \tag{4.1}$$

We normalize each of the two terms on the right hand side of the equation to give the same importance to shopping time and shopping cost. Specifically, $ST_{P_i}^{x}$ is normalized w.r.t. the shopping time upper bound ST^U, while the average cost of each product is normalized w.r.t. the cost of the most expensive product available. Consequently, assuming that $m > 0$ we have $\forall(x, \lambda, P_i), F(v, \lambda, P_i) \in [0, 2]$.

Algorithm 2: APX-PSD

```
Input  : road network $G = (V, E)$, set of stores $T \subseteq V$, quad-tree $Q$
         superimposed over the MBR enclosing the elements in $T$,
         shopper's location $v_\omega$, delivery location $v_\sigma$, shopping list $\lambda$,
         shopping time upper bound $ST^U$.
Output: approximated linear skyline $LS$.
1 $Q, T \leftarrow$ PruneStores($T, \lambda$)
2 $T \leftarrow$ DijkstraMultipleTarget($v_\omega, T \cup \{v_\sigma\}$)
3 $T \leftarrow$ DijkstraMultipleTarget($v_\sigma, T$)
4 $P \leftarrow$ GetRoot(Q)
5 $LS \leftarrow$ Explore($P, Q, \lambda, v_\sigma, \emptyset, \emptyset, ST^U$)
6 return $LS$
```

Intuitively, the lower the score, the more “promising” a quadrant is. Observe also that the scoring function attempts to balance the importance given to shopping time and shopping cost. We thus expect APX-PSD to generate shopping routes with shopping time and cost having the tendency to distribute more toward the centers of the intervals in the corresponding optimal skyline. In fact, the experimental results shown in Section 5 confirm that a scoring function with such a feature makes APX-PSD *robust* to variations in store and products' price *distributions*.

Algorithm 3: Explore

```
Input  : current partition $P$, quad-tree superimposed over $G$, $Q$,
         shopping list $\lambda$, customer's delivery location $v_\sigma$, linear
         skyline $LS$, set of partial shopping routes $PR$, shopping
         time upper bound $ST^U$.
Output: set of partial shopping routes $PR$ (updated), linear skyline
         $LS$ (updated), shopping list $\lambda$ (propagated).
1  $\lambda' \leftarrow$ GetMissingProducts($\lambda, PR$)
2  if $\lambda' = \emptyset$ then return $PR, LS, \lambda$
3  if IsLeaf($P, Q$) then
4      $PR, LS \leftarrow$ ComputePartitionRoutes($\lambda', P, PR, LS$)
5  else
6      $src \leftarrow$ GetStart($PR$)
7      $Z = \{P_1, P_2, P_3, P_4\} \leftarrow$ Score ($Q, P, \lambda, src, v_\sigma, PR, ST^U$)
8      while $Z \neq \emptyset$ do
9          $z \leftarrow$ GetTopScore($Z$)
10         $PR, LS, \lambda \leftarrow$ Explore($z, Q, \lambda, LS, PR, ST^U$)
11         $Z \leftarrow Z \setminus \{z\}$, $src \leftarrow$ GetStart($PR$)
12         $Z \leftarrow$ ReScore ($Q, Z, \lambda, src, v_\sigma, PR, ST^U$)
13 return $PR, LS, \lambda$
```

At this point we are ready to introduce APX-PSD. Algorithms 2 and 3 present the pseudo-code behind our approach. Algorithm 2 starts by performing several preliminary operations. First, it removes from T stores that do not offer any of the products in λ, and updates T and Q accordingly (line 1). Next, it performs two single-source shortest path searches at lines 2–3 that, analogously to those in BSL-PSD, compute the fastest paths between the shopper's and customer's delivery locations and the stores in T. The algorithm then initiates to recursively perform a depth-first visit of the quad-tree, with the goal of constructing shopping routes from the tree's leaves that look the "most promising" and update the linear skyline accordingly. Such operations are implemented by the function Explore, invoked at line 5. Algorithm 3 provide the details.

Explore first determines the set of products that cannot be bought from the routes currently stored in PR and stores such set in λ' (line 1, function GetMissingProducts). Note that if $\lambda' = \emptyset$ then Explore can *terminate*, as this implies that the depth first search conducted within Q already found out shopping routes that can satisfy λ and inserted them into LS. EXPLORE then verifies if the currently considered quadrant P (initially the quad-tree root) is a *leaf* or not (line 3). If P is a leaf, then the function COMPUTEPARTITIONROUTES is executed (line 4). Such function first counts the number of products in λ' that can be bought from P's stores – let us suppose m products. Subsequently, the function generates the set of (partial) shopping routes from P's stores, where *each* such route allows to buy *exactly* those m products (stores that do not offer any of the products in λ' are ignored). COMPUTEPARTITIONROUTES then goes on to compute the *Cartesian product* between the set of routes currently stored in PR and that computed from P. The result then becomes the new PR's content. Each route in PR is subsequently checked to verify if its shopping time is above ST^U – in such case the route is removed from PR. Finally, COMPUTEPARTITIONROUTES verifies if the surviving routes buy all the products in λ (i.e., when $\lambda' = \emptyset$) and, if so, attempts to insert them into LS.

If P is an intermediate node of Q then EXPLORE first proceeds to determine in which partition the routes currently stored in PR terminate and set *src* accordingly (line 6, function GETSTART). Then, EXPLORE scores P's four sub-quadrants (line 5, function SCORE) and finally recursively invokes itself by considering such sub-quadrants in ascending order of score (line 10). Note that the function RESCORE (line 12) takes advantage of any update in PR to update the scores of partitions still within Z, and thus better direct the evaluation of candidate routes.

4.2.1 APX-PSD's complexity

APX-PSD first requires to pre-compute a quadtree, an operation that has cost $O(|T|)$. The two single-source shortest searches (lines 2 and 3) operate on $G = (V, E)$ and have cost $O\big((|E| + |V|) \cdot log|V|\big)$ each. Let us now focus on the recursive execution of EXPLORE. The cost of such execution is $O\big((|V^Q| + |E^Q|) + N' + N + N|T'||\lambda| + NlogN)$.

The first term is due to the DFS being conducted over Q: if V^Q and E^Q

denote, respectively, the nodes and edges in Q, then performing a DFS has cost $O(|V^Q| + |E^Q|)$. The second term, N', represents the cost of generating shopping routes from Q's leaves by COMPUTEPARTITIONROUTES: if L denotes the set of Q's leaves, then $N' = \sum_{i=1}^{|L|} |L_i|!$. The third term represents the cost of generating (partial) shopping routes by means of the Cartesian products conducted within COMPUTEPARTITIONROUTES, with $N = \prod_{i=1}^{|L|} |L_i|!$. The fourth term represents the cost of finding the store, within each of the N shopping routes, from which a product in λ can be bought at minimum price. Finally, the fifth term represents the cost of generating the final linear skyline, i.e., $O\big(NlogN\big)$.

Overall, APX-PSD has to evaluate much less candidate routes (if L denotes the set of Q's leaves, then $O(\prod_{i=1}^{|L|} |L_i|!)$) than BSL-PSD ($O(|T|!)$), thanks to the spatial partitioning imposed over the stores and how such partitioning is used to generate and evaluate candidate shopping routes. On the other hand, this limits the evaluation to a restricted subset of candidate routes, which explains APX-PSD's expected sub-optimality.

5

Experimental Evaluation

In order to evaluate the APX-PSD approach, we used datasets containing the road networks and POI locations for Amsterdam, Oslo and Berlin [2]. Since those POIs do not include actual stores we used the locations of gas stations and pharmacies (which are typically scattered throughout a city) as proxies for the locations of stores.

Figure 5.1 illustrates the store locations in Berlin, Oslo and Amsterdam as used in our experiments. Recall that pharmacies and gas stations were used as proxy for stores, due to (1) the absence of information about real stores in these cities, and (2) it is realistic to assume that pharmacies and gas stations are scattered over a city just like stores would be. The total number of vertices, edges and stores for all of the three maps are shown in Table 5.1.

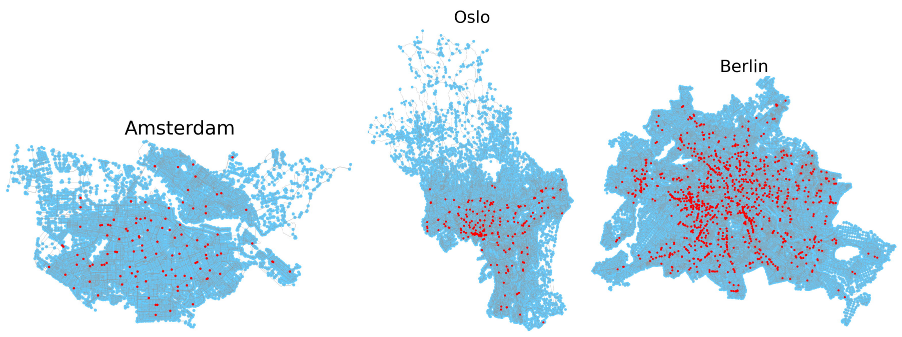

Figure 5.1: Distribution of stores in each city's road network. (Cities not shown to scale).

	Amsterdam	**Oslo**	**Berlin**
Vertices	106600	305175	428769
Edges	130091	330633	504229
Stores	100	207	768

Table 5.1: Metadata for datasets.

The parameters considered for the experimental evaluation are: (1) store cardinality (i.e., number of stores in a network), (2) distribution of a products' cost in the stores, (3) spatial distribution of differently sized stores (4) size of shopping lists, and (5) capacity of the quad-tree leaves.

Given that the networks are fixed, varying the store cardinality models the density of stores in a city. We also looked into the case when prices increase or decrease w.r.t. the distance of the store to the city's center. We considered three different sizes of stores, i.e., small, medium and large, according to the percentage of the total number of products they hold, i.e., 25%, 50% and 75%, respectively, out of a total of 1,000 products. Note that we chose to not have any store selling all the products to avoid a possibly trivial solution involving a single store. Even though stores are by default distributed randomly throughout the network, regardless of their sizes, we also investigated the effect of increasing or decreasing the size of stores according to their proximity to the city's center. The values used for each parameter are shown in Table 5.2.

Parameter	**Default Value**
Store Cardinality	10, **25**, 50
Product Cost Distribution	Rising, **Normal**, Declining
Store Size Spatial Distribution	Increasing, **Random**, Decreasing
Shopping list size	5, **10**, 15
quad-tree leaf capacity	4, **8**, 16

Table 5.2: Default Parameter Values.

We attempted to emulate a realistic scenario in terms of the distribution of the cost of products as well as of placement of differently sized stores. We thus divide the maps into *three concentric rings* w.r.t. the city center. The inner, middle and outer rings will host small, medium and large stores, respectively, when the spatial distribution of stores is set to "Increasing". Conversely, when that parameter is set to "Decreasing" the inner, middle and outer rings segment will host large, medium and small stores, respectively. Similarly, when the product cost distribution parameter is set to "Rising" (Declining) products in stores farther (closer) from the city's center are more expensive. The store closest (farther) to the city centre will sell it at minimum (maximum) price and the one farthest will sell it at the maximum (minimum) price. All the stores in between will sell the product at a price proportional to the distance between those two stores. For "Normal" distribution we first find a mean price for each product following a U(5,15) distribution. Once we have the mean price for a product, we assign prices of that product to different stores following a normal distribution with its mean price and a standard deviation of 2. As discussed in Section 4.2, APX-PSD's evaluation strategy relies on partitions of stores. Thus we observe the effects of the leaf capacity on the APX-PSD by varying the leaf capacity to 4, 8 and 16.

Finally, we test each value each considered parameter can assume by conducting 100 individual experiments, and report the average optimality gap, coverage gap and processing time. In each experiment we randomly select the shopper's and customer's delivery locations, as well as randomly generate a shopping list of the required size. Furthermore, we randomly select the required number of stores (25 stores in default setting) from those available in the networks. All the experiments were conducted on a virtual machine with an Intel(R)-Xeon(R) CPUs running at 2.30GHz and with 264GB of RAM.

5.1 Evaluation metrics

We evaluate our APX-PSD by measuring the quality of the results (effectiveness) as well as the efficiency of it. In this section, we discuss the metrics we use to measure such effectiveness and efficiency.

5.1.1 Effectiveness

Comparing an optimal linear skyline (opt-LS) to an approximated one (apx-LS) requires comparing two aspects of skylines: optimality and coverage. For that we propose the two measures presented next.

Consider Figure 5.2 where opt-LS $= \{A, B, C, D\}$ and apx-LS $= \{A', B', C'\}$. The area given by the polygon $OYABCDXO$ (shaded in green plus orange) represents the area A_{opt} not dominated by opt-LS, and similarly the polygon $OY'A'B'C'X'O$ (shaded in blue *plus* the one shaded in green) denotes the area A_{apx} not dominated by apx-LS. The smaller the difference between A_{apx} and A_{opt} the better, but in order to make the right comparison we need to consider only the portion of A_{apx} that intersects with A_{opt}, which in the case of this example is given by the polygon $OYABCGX'O$, which we denote by A_{cover}. Finally, the ratio $(A_{apx} - A_{cover})/A_{apx}$ represents the normalized "room for improvement" of the approximated solution. We call this measure the *Optimality Gap*.

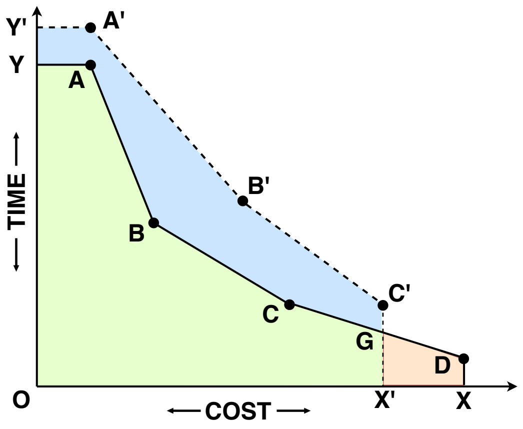

Figure 5.2: Area coverage by the approximate and optimal linear skyline.

The optimality gap does not consider part of A_{opt} that is not "covered" by A_{apx}, e..g, the orange shaded polygon $X'GDXX'$ in the example and which we denote as A_{miss}. This, which we name *Coverage Gap*, is a consequence of the skyline approximation, i.e., the more points it is missing, the larger such gap.

In what follows we compute this (normalized) coverage gap as (A_{miss}/A_{opt}) and, like the optimality gap, the smaller it is, the better.

Finally, note that in *all the experiments we conducted the optimality and coverage gaps were below 50% and 15% respectively*, quantifying that the approximated linear skylines were of good quality. As well, the processing time of APX-PSD was always smaller than 1 second and around two orders of magnitude smaller than BSL-PSD's.

5.1.2 Efficiency

We evaluate the efficiency of our PSD-PSD by comparing its processing time to the processing time of BSLPSD. The experiments are designed to test whether APX-PSD can provide a good solution in real-time. Even though we can not compare the BSL-PSD and APX-PSD in terms of effectiveness and efficiency for large store cardinality and shopping list size, we can confirm the scalability of APX-PSD reporting the processing time for these extreme cases.

From Sections 4.1 and 4.2 recall that we assume some information – namely the shortest path between stores and the stores partitioning – is pre-computed offline. We argue that such assumption is reasonable, since stores are seldom added to or removed from networks. Moreover such pre-computation represents a not overly expensive one-time cost.

City	**Shortest Path**	**Partitioning**	**Total**
Amsterdam	31.40	1.56	32.96
Oslo	202.78	2.32	205.10
Berlin	1072.31	4.76	1077.07

Table 5.3: Pre-computation run-time (in seconds)

Table 5.3 shows the time required to pre-compute the shortest path between every pair of stores and also perform the store partitioning. Note that while we have used a typical implementation of Dijkstra's algorithm for shortest paths computation, more efficient alternatives could be used as this is a step completely independent of the approaches being proposed in our work.

5.2 Effects of store cardinality

Figure 5.3 shows that the optimality gap increases when increasing the store cardinality, while the coverage gap decreases. We explain such behaviour by observing how quad-trees as well as the scoring function react to changes in store cardinality. When store cardinality increases, the number of stores in each quad-tree leaf increases accordingly. Since product costs are distributed uniformly in the default case, adding more stores smooths the average cost of each product in different partitions, which in turn reduces the impact of product costs in APX-PSD's scoring function. Therefore, the linear skyline will include routes with higher costs, which will decrease the coverage gap. Consequently, APX-PSD has to visit more leaves to complete the shopping list. Furthermore, with larger store cardinality BSL-PSD generates shopping routes with lower shopping time that APX-PSD fails to find, thus increasing the optimality gap.

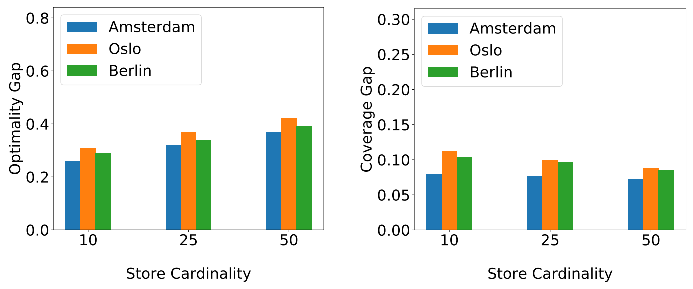

Figure 5.3: Effectiveness w.r.t. store cardinality

Figure 5.4 shows that BSL-PSD's processing time increases when store cardinality increases, mainly due to an increased number of stores (and thus potential candidates) to consider. On the other hand, APX-PSD's processing time exhibits small changes. Recall that the leaves' capacity in a quad-tree is fixed, thus increasing the store cardinality increases the tree's depth. Notice that, we varied the parameter upto 500 stores for Berlin but only up to 200 stores and 100 stores for Oslo and Amsterdam respectively. We could not test

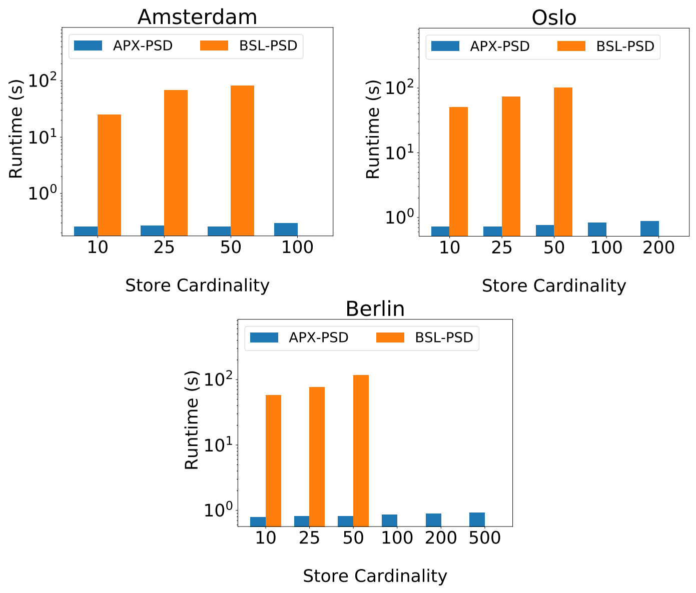

Figure 5.4: Efficiency w.r.t. store cardinality.

Amsterdam or Oslo with larger store cardinality due to the limitation of the datasets respectively, Recall that APX-PSD's time-dominant component deals with shopping routes generation and evaluation, rather than tree traversal, which explains the small impact on performance.

5.3 Effects of product cost distribution

In Figure 5.5 we can see that the "Declining" and "Rising" cases have comparable optimality gaps and higher coverage gaps to the "Normal" one. We argue that these results are due to the characteristics of the scoring function used by APX-PSD which, we recall, attempts to balance the importance given to shopping time and shopping cost. Since the product cost gradually decreases towards one direction for both "Declining" and "Rising", the scoring function manages to minimize cost better than "Normal" distribution. However, the optimality gap created due to creating partial routes from a leaf at a time remains comparable to the "Normal" distribution. Interestingly, both the optimality gap and coverage gap are the highest for Oslo in all of the cases, which we attribute to the skewed distribution of stores compared to the other cities (Figure 5.1).

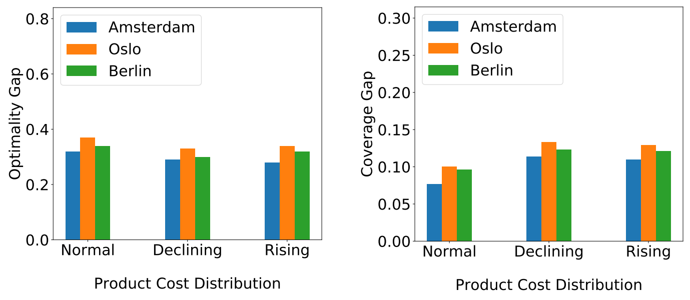

Figure 5.5: Effectiveness w.r.t. different cost distributions.

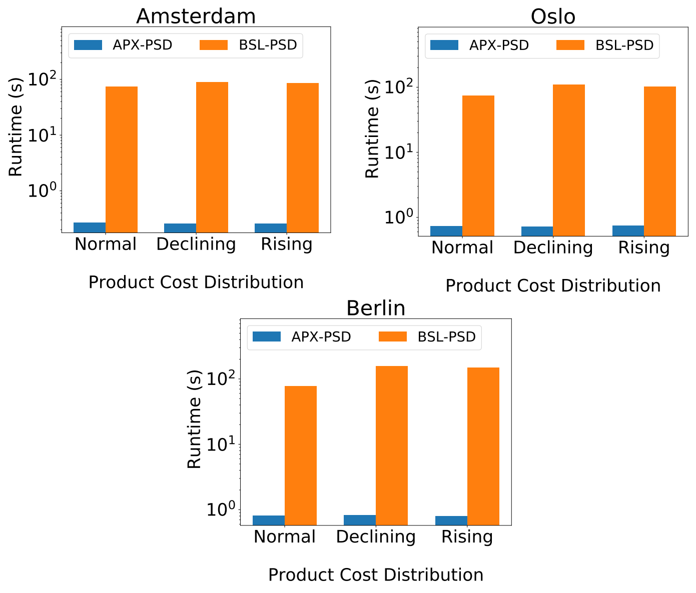

Figure 5.6: Efficiency w.r.t. different cost distributions.

Figure 5.6 shows that BSL-PSD's processing time using Berlin's network is larger for the "Declining" and "Rising" cases than for the "Uniform" one. In those two cases the low-cost products are distributed in certain areas of the map, and thus it is more likely to take longer to find routes with lower shopping costs if the shopper's and customer's delivery locations are far from those regions. Note that such differences are further amplified by Berlin's large network size. However, APX-PSD's processing time is not affected since it takes great advantage of pre-computed aggregated paths as it traverses the quad-tree hosting the store partitions.

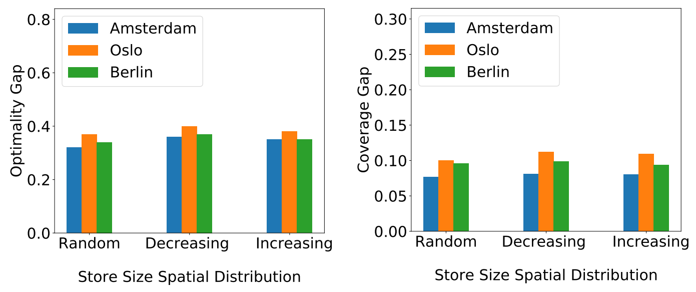

Figure 5.7: Effectiveness w.r.t. different distributions of store size.

5.4 Effects of store size spatial distribution

In Figure 5.7 we can see that the "Decreasing" and "Increasing" cases have comparable coverage and optimality gaps to the "Random" one. These results can be explained by taking into account the characteristics of APX-PSD's scoring function, which makes it insensitive to different distributions of store sizes (apart from the effects that can be observed on the shape of both optimal and approximated skylines).

BSL-PSD's processing time (Figure 5.8) is higher when dealing with the "Decreasing" and"Increasing" cases. Since the locations of larger stores are concentrated in certain areas, BSL-PSD takes longer to generate routes with minimum cost and terminate (depending on the shopper's and customer's delivery locations), an effect that is further compounded by the city's network size. As usual, Berlin exhibits the largest processing time for the same reason observed with varying cost distributions. Finally note that APX-PSD's processing time remains unaffected.

5.5 Effects of shopping list size

The optimality (coverage) gap increases (decreases) with the shopping list size as evidenced in Figure 5.9. Larger shopping lists require more traversals of the network. Recall that by construction, shopping routes are appended to existing partial routes. Such appending means that previous not-so-good choices remain and their effect are further compounded by new potentially not-so-good choices as the algorithm evolves, worsening the approximation. As a

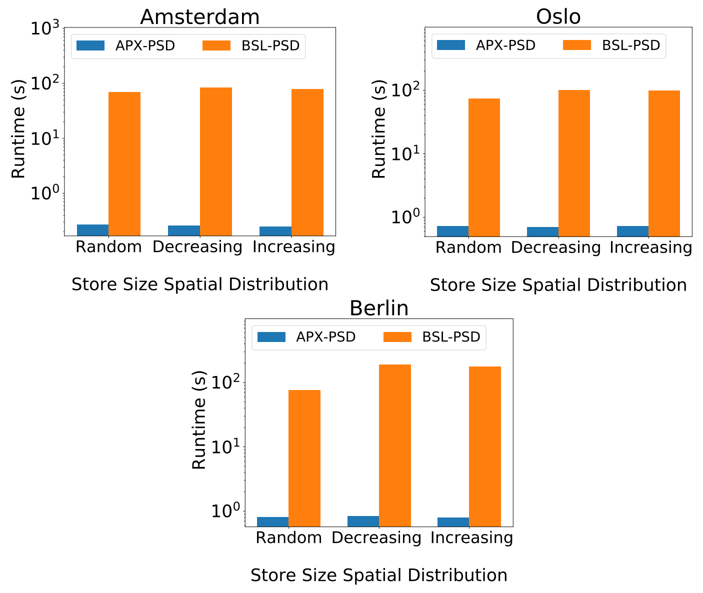

Figure 5.8: Efficiency w.r.t. different distributions of store size.

result both the shopping time and cost increase which increases the optimality gap and decreases the coverage gap.

BSL-PSD's processing time (Figure 5.10) increases when increasing the shopping list size. This can be explained by observing that, on average, large shopping lists require more stores per route to be satisfied, and thus likely require to evaluate more candidate routes. On the other hand APX-PSD is mildly affected by such increase, thanks to the noticeably smaller number of candidates it generates and evaluates by design.

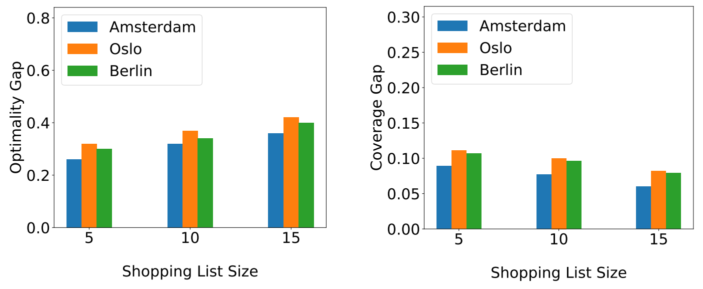

Figure 5.9: Effectiveness w.r.t. shopping list size.

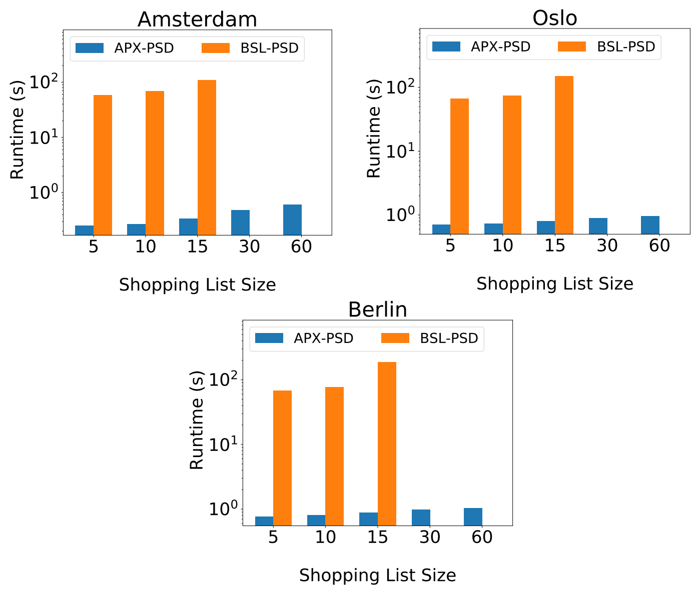

Figure 5.10: Efficiency w.r.t. shopping list size.

5.6 Effects of quad-tree leaf capacity

From the results shown in Figure 5.11 we see that the optimality and coverage gaps decrease (i.e., the overall result quality increases) as the leaf capacity increases. This can be explained by observing that leaves containing more stores allow APX-PSD to generate and evaluate more partial shopping routes, thus allowing to discover shopping routes with increasingly lower costs (and thus closer to the ones computed by the baseline). Finally, Figure 5.12 shows that APX-PSD's efficiency decreases as the leaf capacity increases, due to the increased number of partial shopping routes that APX-PSD generates and evaluates from the leaves it visits – indeed, from APX-PSD's complexity analysis (Sec. 4.2.1) recall that this represents the time-dominant component of this approach.

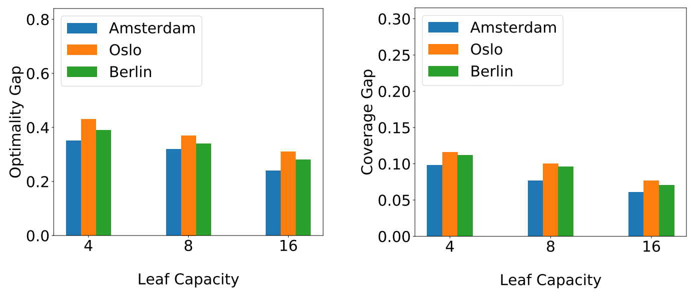

Figure 5.11: Effectiveness w.r.t. varying quad-tree leaf capacity

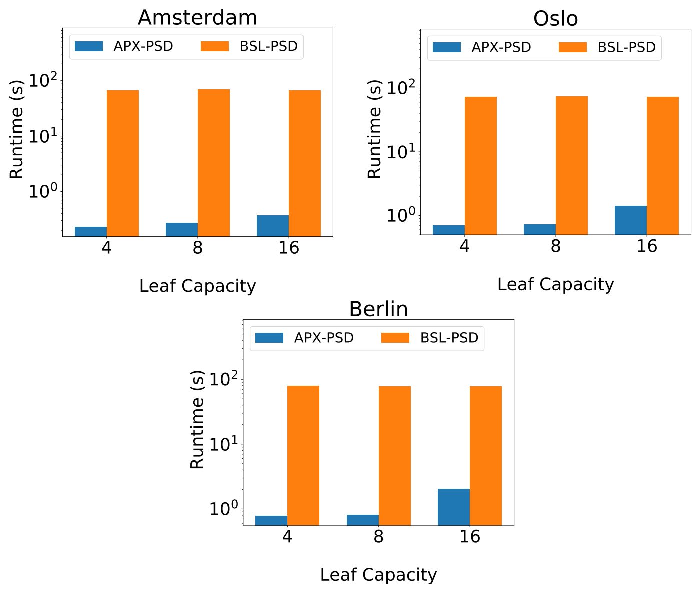

Figure 5.12: Efficiency w.r.t. with varying quad-tree leaf size

6

Conclusion

In this thesis we proposed a solution to what we called the "Personal Shopper's Dilemma" query, which is to decide on how to fulfil a customer's shopping list while minimizing travel time as well as shopping cost. The idea is to leave the shopper to decide, "on the fly" how to prioritize these two criteria. Given the query's NP-hardness we proposed a heuristic solution that leverages on the concept of linear skyline queries. In order to measure the effectiveness of the proposed heuristic we also proposed a metric to evaluate its loss w.r.t. an optimal solution. Using real city-scale datasets we show that our proposal is able to deliver good linear skylines yielding optimality and coverage gaps below 50% and 15% respectively two orders of magnitude faster than the optimal solution.

A direction for future work w.r.t. the PSD query itself would be allowing a shopper to find routes to serve multiple customers, possibly in different locations, and/or have multiple shoppers that could, for instance, bid on shopping lists of different customers after considering their perspective on those two criteria.

References

[1] E. Ahmadi, C. Costa, and M. Nascimento, "Best-compromise in-route nearest neighbor queries," in *ACM SIGSPATIAL*, 2017, 41:1–41:10.

[2] E. Ahmadi and M. Nascimento, *Datasets of roads, public transportation and points-of-interest in Amsterdam, Berlin and Oslo*, In: `https://sites.google.com/ualberta.ca/nascimentodatasets/`, 2017.

[3] S. Arora, "Polynomial time approximation schemes for euclidean tsp and other geometric problems," in *Proceedings of 37th Conference on Foundations of Computer Science*, IEEE, 1996, pp. 2–11.

[4] ——, "Approximation schemes for np-hard geometric optimization problems: A survey," *Mathematical Programming*, vol. 97, no. 1-2, pp. 43–69, 2003.

[5] S. Borzsony, D. Kossmann, and K. Stocker, "The skyline operator," in *IEEE ICDE*, 2001, pp. 421–430.

[6] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Closest pair queries in spatial databases," *ACM SIGMOD Record*, vol. 29, no. 2, pp. 189–200, 2000.

[7] C. F. Costa and M. A. Nascimento, "In-route task selection in crowdsourcing," in *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2018, pp. 524–527.

[8] C. F. Costa, M. A. Nascimento, J. A. Macêdo, Y. Theodoridis, N. Pelekis, and J. Machado, "Optimal time-dependent sequenced route queries in road networks," in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2015, pp. 1–4.

[9] C. F. Costa, M. A. Nascimento, and M. Schubert, "Diverse nearest neighbors queries using linear skylines," *GeoInformatica*, vol. 22, no. 4, pp. 815–844, 2018.

[10] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, 1984, pp. 47–57.

[11] G. R. Hjaltason and H. Samet, “Distance browsing in spatial databases,” *ACM Transactions on Database Systems (TODS)*, vol. 24, no. 2, pp. 265–318, 1999.

[12] H. Kriegel, M. Renz, and M. Schubert, “Route skyline queries: A multi-preference path planning approach,” in *IEEE ICDE*, 2010, pp. 261–272.

[13] F. Lettich, M. A. Nascimento, and S. Anwar, “Trade-off aware sequenced routing queries (or osr queries when pois are not free),” Submitted, 2020.

[14] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S. Teng, “On trip planning queries in spatial databases,” in *SSTD*, 2005, pp. 273–290.

[15] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, “Query processing in spatial network databases,” in *VLDB*, 2003, pp. 802–813.

[16] N. Roussopoulos, S. Kelley, and F. Vincent, “Nearest neighbor queries,” in *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, 1995, pp. 71–79.

[17] C. Salgado, M. Cheema, and D. Taniar, “An efficient approximation algorithm for multi-criteria indoor route planning queries,” in *ACM SIGSPATIAL*, 2018, pp. 448–451.

[18] H. Samet, “The quadtree and related hierarchical data structures,” *ACM Computing Surveys (CSUR)*, vol. 16, no. 2, pp. 187–260, 1984.

[19] M. Sharifzadeh, M. Kolahdouzan, and C. Shahabi, “The optimal sequenced route query,” *The VLDB Journal*, vol. 17, no. 4, pp. 765–787, 2008.

[20] M. Sharifzadeh and C. Shahabi, “Processing optimal sequenced route queries using voronoi diagrams,” *GeoInformatica*, vol. 12, no. 4, pp. 411–433, 2008.

[21] M. Shekelyan, G. Jossé, M. Schubert, and H. Kriegel, “Linear path skyline computation in bicriteria networks,” in *DASFAA*, 2014, pp. 173–187.