Trade-Off Aware Queries in Road Networks

by

Camila Ferreira Costa

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

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

This thesis discusses bi-criteria optimization queries in road networks, in particular queries that allow users to consider trade-offs between alternative solutions. In order to measure the proximity of two nodes in a road network, a cost optimal path between them is typically computed, for instance, the fastest path. However, optimal paths can depend on more than one cost criterion. For example, when planning a bicycle route, minimizing only the distance is usually not sufficient. The user may also be interested in taking into consideration the incline of the path. Since different criteria might conflict with each other, typically there is not a single solution (a path in the example given above) that optimizes all criteria at the same time. Therefore, in this thesis we rely on the notion of skyline queries to generate a set of interesting solutions such that the user can select the preferred criteria trade-off out of the returned options. Skyline queries find a set of non-dominated solutions that are optimal for any arbitrary combination of the considered criteria. An element dominates another if it is as good or better in all dimensions and better in at least one dimension. Within this context, we first consider k-Diverse Nearest Neighbors Queries. In the original definition of k-nearest neighbor (k-NN) queries there is no concern regarding diversity of the answer set, even though in some scenarios it may be interesting. For instance, if one is looking for restaurants close by, it may be more interesting to return restaurants of different categories or ethnicities which are nonetheless relatively close. Thus, differently from a traditional k-NN query, there are two competing criteria to be optimized: closeness and diversity. Using skyline queries we can find diverse k-NNs close to a given query point and that are optimal solutions for any combination of the weights a user could give to the two competing criteria. Next, we explore trade-off aware queries in the context of spatial crowdsourcing. Spatial crowdsourcing is a relatively new platform where requesters submit location-specific tasks to be performed by workers that need to travel to those locations. Examples of these tasks include taking pictures or performing small repairs. Tasks are assigned to proper workers based on a particular objective, such as maximizing the number of assigned tasks. Previous works focus on optimizing a single criterion. We, on the other hand, consider the scenario where a worker is traveling along a given path, for instance from work to home, and is willing to perform tasks, that are not far away from the original path, in exchange for a reward. We assume that the worker wants to minimize the detour from his/her original path while maximizing the reward received for performing tasks. We investigate two variants of this problem. The first one considers an offline setting where all tasks are known beforehand, whereas in the second variant tasks appear dynamically. We show that all proposed problems are NP-Hard and present exact (whenever applicable) and heuristic approaches to solve them. Moreover, within the context of each problem, we present an experimental evaluation using real datasets and varying several parameters, and discuss the results obtained. Finally, we conclude this thesis by presenting a summary of our findings for each proposed problem and suggestions for future work.

Preface

This thesis follows a paper-based format. Its body is composed of published papers and a paper recently submitted for review, as described in more details below.

Chapter 2 consists of a journal paper [14], which is an extension of a full paper [13] published at a conference:

- Camila F. Costa, Mario A. Nascimento, Matthias Schubert (2018).Diverse nearest neighbors queries using linear skylines. GeoInformatica 22(4): 815-844.

- Camila F. Costa, Mario A. Nascimento (2017). Towards Spatially- and Category-Wise k-Diverse Nearest Neighbors Queries. In Proc. of the 15th Intl. Symp. on Spatial and Temporal Databases: 163-181. (*Winner of the Symposium's Best Paper Award*)

Chapter 3 has been published as a journal paper [6], which is also an extension of a short paper [12] published in a conference proceedings:

- Camila F. Costa, Mario A. Nascimento (2020). In-Route Task Selection in Spatial Crowdsourcing. ACM Transactions on Spatial Algorithms and Systems 6(2): 7:1-7:45.

- Camila F. Costa, Mario A. Nascimento (2018). In-route task selection in crowdsourcing. In Proc. of the 26th ACM SIGSPATIAL Intl. Conf.

on Advances in Geographic Information Systems: 524-527. (*Winner of the Conference's Best Fast Forward Presentation Runner-Up Award*).

Chapter 3 consists of a full paper accepted for publication in the proceedings of a forthcoming conference:

- Camila F. Costa, Mario A. Nascimento. Online In-Route Task Selection in Spatial Crowdsourcing. To appear at the 28th ACM SIGSPATIAL Intl. Conf. on Advances in Geographic Information Systems.

Finally, during my Ph.D. program I have also co-authored the following papers which are out of the scope of this thesis [2], [15], [16], and thus are not part of this document:

- Elham Ahmadi, Camila F. Costa, Mario A. Nascimento (2017). Best-Compromise In-Route Nearest Neighbor Queries. In Proc. of the 25th ACM SIGSPATIAL Intl. Conf. on Advances in Geographic Information Systems: 41:1-41:10.

- Camila F. Costa, Mario A. Nascimento (2016). IDA 2016 Industrial Challenge: Using Machine Learning for Predicting Failures. 15th Intl. Symp. on Advances in Intelligent Data Analysis: 381-386. (*Winner of the Symposium's Industrial Challenge*).

- Camila F. Costa, Theodoros Chondrogiannis, Mario A. Nascimento, Panagiotis Bouros (2020). RRAMEN: An Interactive Tool for Evaluating Choices and Changes in Transportation Networks. In Proc. of the 23nd Intl. Conf. on Extending Database Technology: 599-602.

To my husband Toni

For all the kindness and patience throughout this journey.

"Se avexe não

Toda caminhada começa no primeiro passo

A Natureza não tem pressa

Segue o seu compasso

Inexoravelmente chega lá"

– A Natureza Das Coisas, Flávio José

"Rush not

Every journey begins with the first step

Nature takes its time

On its own pace

Inevitably prevails"

– The nature of things, Flávio José

Acknowledgements

I would like to express my gratitude to my supervisor Prof. Mario A. Nascimento for being the best mentor I could have asked for. Even after a bumpy start (which I'm sure he will never forget), he did not give up on me and always motivated me to explore my potential. I am really grateful for all the knowledge he shared with me along my academic journey and for inspiring me to grow as a researcher. Even with a very busy schedule he always found the time to meet with me to discuss ideas and guide me in the right direction. His guidance has empowered me to achieve great accomplishments that will have a long lasting impact in both my career and in my personal life.

I would also like to thank the other members of my examining committee, Prof. Joerg Sander, Prof. Zachary Friggstad, Prof. Ioanis Nikolaidis, and Prof. Cyrus Shahabi for their availability and insightful comments. I also thank Professors Matthias Schubert and Panagiotis Bouros with whom I had the privilege to collaborate during my Ph.D. program.

On a personal level, I would first like to thank my husband Toni for all the love, patience and encouragement during these years. I also thank my Mom, my brothers, my aunt Neiva and my grandma Maria for always rooting for my success. I was very fortunate to meet many people who made this journey easier and more pleasant. I would like to express my sincere gratitude to Gabriela for all the support and, especially, for helping us start a new life in

Edmonton. Speaking of beginnings, I would like to thank Marjorie, Priscila, Jadson, Leandro, Mariana, and Victor (bandido) for becoming our first social support network in Canada. I thank Ariane and Rodolfo for sticking with us through the ups and downs of this journey. Martin, Cacau, Camila and Levi, I am very grateful for your friendship and for all the sushi we've made together. Regina and Beda, thank you for being such good and supportive friends, and even our workout partners. Gutinho, thank you for always cheering us up. Chris, you were an amazing addition to the gang. I am also very grateful to Francesco, Antonio, Georg and Rubens, who lived in Edmonton for a short time, but made this journey more fun. Last, but not least, I would like to thank our good friends Andrade, Coutinho and Vituxo, who are physically distant, but are always there for us.

My work was partially supported by NSERC, Canada and CNPq's Science Without Borders Program, Brazil.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

When computing a cost-optimal path between two nodes of a road network, usually only a single criterion is taken into account, for instance, the travel time *or* the distance traveled. However, depending on the application, using a single cost criterion can be too restrictive. For example, when planning an electric car route, the driver may not only be interested in minimizing the travel time, but also the energy consumption. Similarly, when riding a bike, the user may wish to maximize the perception of safety, in addition to minimizing the travel distance. In order to employ multiple criteria when computing optimal solutions, a simple way is to combine all criteria and optimize the combined value. Such strategy requires the user to determine the importance of each criterion over the others and that the criteria are on the same scale. However, the user's preference may not be obvious or clear from the outset and some criteria may not be combinable, e.g., distance and perception of safety. A more principled way to deal with such problem is to determine all results that are optimal under any arbitrary combination of the criteria, which can be accomplished by using the notion of skyline queries [5]. The result set of a skyline query contains all non-dominated elements. An element dominates another if it is as good or better in all dimensions and better in at least one dimension. Figure 1.1 illustrates a classical example [5] where one wants to find hotels that are both inexpensive and close to the beach. The linked dark dots represent the result set, i.e., the non-dominated hotels. Once the skyline set is computed, the user can consider all interesting alternatives *w.r.t., his/her own preferences* and choose the one that better suits his/her needs.

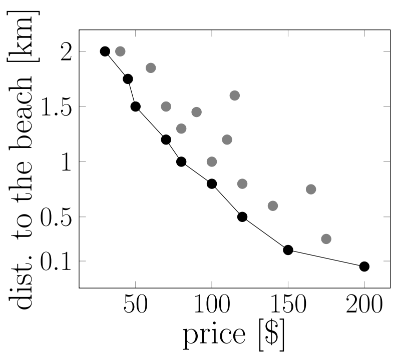


Figure 1.1: Illustration of bi-criteria skyline.

In this thesis, we first propose the use of the skyline operator to solve k-Diverse Nearest Neighbors queries. k-nearest neighbor (k-NN) [40] queries are well-known and widely used in a plethora of applications. Given a query point q, a set of points P and a distance metric, a k-NN query finds the set of k points $P' \in P$ such that no other point in $P \setminus P'$ is closer to q than those in P' according to that metric. One potential drawback of k-NN queries is the fact that the k-NNs are determined based *only* on their distance to the query and no assumption is made on how they relate to each other. Providing homogeneous result sets, i.e., ones where the elements are very similar to each other, may not add much information to the query result as whole. One such scenario is document ranking [7], where returning k documents that are close to the query, but diverse among themselves is likely to increase the amount of overall information one gathers.

This concern has motivated some research to incorporate diversity into similarity searches, i.e., to find elements that are as close as possible to the query, while, at the same time, being as diverse as possible. The relation between diversity and closeness has been mainly investigated within the domain of information retrieval, e.g., [7], [11], recommendation systems, e.g., [58], web retrieval, e.g., [39] and spatial query processing [33], [59].

We aim to incorporate the notion of diversity into spatial k-NN queries and investigate (k, r)-DNNs queries. A (k, r)-DNN query returns the k *diverse* nearest neighbors no farther than a radius r from a given query point. We consider three different notions of diversity: spatial, angular and categorical. For instance, a tourist visiting a city may want to explore points-of-interest (POIs) that are not too far from his/her location, but that at the same time cover different parts of the city. Such scenario illustrates spatial diversity. In the case of angular diversity, a tourist could be interested in visiting attractions that surround his/her location but are in different directions. Likewise, consider a user looking for restaurants close by, in which case it may be interesting to return different types of restaurant (maximizing the categorical diversity) so that the user could make a decision based on diversified options.

Previous works that dealt with similar problems, i.e., balancing closeness and diversity, rely on a user-provided linear combination of the relative importance of closeness over diversity. We, on the other hand, consider the case where the user does not need to specify weights for each criterion and aim at finding *all* results that are *optimal* under *any* linear combination of two competing criteria (i.e., closeness and diversity) by relying on the notion of linear skyline queries [43].

As our main contribution in the first part of this thesis we propose two approaches that leverage the notion of linear skyline queries in order to find the k diverse nearest neighbors within a radius r from a given query point, or

(k, r)-DNNs for short. Our proposed approaches return a relatively small set containing *all* optimal solutions for *any* linear combination of the weights a user could give to the two competing criteria, and we consider three different notions of diversity: spatial, categorical and angular. Our experiments, varying a number of parameters and exploring synthetic and real datasets, in both Euclidean space and road networks, respectively, show that our approaches are several orders of magnitude faster than a straightforward approach.

Next, in the second part of this thesis, we explore trade-off aware queries in the context of spatial crowdsourcing. Crowdsourcing is a relatively new paradigm which relies on the contributions of a large number of workers to accomplish tasks submitted by requesters, such as image tagging and language translation. The increasing popularity of mobile computing led to a shift from traditional web-based crowdsourcing to spatial crowdsourcing [49]. Spatial crowdsourcing consists of location-specific tasks that require people to physically be at specific locations to complete them, differently from web-based crowdsourcing. Some examples of spatial crowdsourcing applications include ride-sharing services, e.g., Uber [51] and Lyft [38], delivery services, e.g., DoorDash [22] and Instacart [30], handyman related jobs, e.g., TaskRabbit [47] and Handy [26], pet sitting, e.g., Rover [41] and Wag [54], moving services, e.g., Dolly [21] and Bellhops [4], lawn care services, e.g., LawnLove [37], IT services, e.g., HelloTech [27], and car repair, e.g., YourMechanic [57].

Tasks are assigned to workers based on a particular objective, such as maximizing the number of assigned tasks [32], [48], [56], minimizing the matching distance between workers and tasks [36], [56] or maximizing a given matching score [42], [45], [50]. Traditionally, published works which assume that a worker is assigned to multiple tasks, do not typically take into account the travel cost between tasks, e.g. [32], [42], [45], [48], [50], [56]. However, that cost directly affects the number of tasks the worker will be able to perform. Thus, even if those tasks are spatially close from the worker, they may not be completed depending, for example, on the worker's time/distance budget. Therefore, we consider the more generic problem of finding a *task schedule* for a worker, i.e., a feasible sequence of tasks, a problem that has also been considered in [18]–[20], [35].

In [18], [19], the authors focus on maximizing the number of tasks performed by a single worker assuming that the worker is willing to perform up to a predefined maximum number of tasks and that he/she must arrive at the task location before its deadline. [20] presents an extension of [18] which aims at maximizing the overall number of tasks performed by multiple workers considering that each task is assigned to up to one worker. The work in [35] aims at maximizing the number of tasks completed by a single worker in an online setting, i.e. where tasks appear dynamically.

Differently from [18]–[20] and [35], which aim at optimizing a single criterion, namemly maximizing the number of completed tasks, our goal is to find a compromise between two competing criteria in the context of task scheduling. More specifically, we consider the scenario where a user has a preferred path between two locations, for instance from work to home, and is willing to perform some tasks on the way to his/her destination. Additionally, we assume that each task is associated with a reward and that the user wants to maximize the total reward received for performing tasks while minimizing the detour from the preferred path. Since these are competing criteria, a single route can not typically optimize them at the same time. Therefore, we propose the In-Route Task Selection problem, which aims at finding different trade-offs between reward and detour using skyline queries.

In Chapter 3 we consider the offline version of the In-Route Task Selection problem, named IRTS, in which all tasks are known beforehand. As our main contribution in that chapter we propose exact and heuristic approaches to solve IRTS. Our experiments, using real city-scale datasets, show that while the exact approach serves as a benchmark, it does not scale due to the NP-hardness of the problem. The overall best heuristic approach, on the other hand, can solve relatively large instances of the IRTS problem within practical query processing time, e.g., at par with less effective greedy heuristics, while still producing very good approximate skyline sets, e.g., often yielding less than 10% relative error w.r.t. the exact solution.

Finally, in Chapter 4 we study the Online In-Route Task Selection (Online-IRTS) problem, which considers an online setting in which tasks appear dynamically. We also investigate the Online-IRTS problem using the paradigm of skyline queries in order to systematically explore different trade-offs between earned rewards and path deviation. Because of the online nature of the problem, i.e., irrevocable decisions about which task to perform have to be made without knowledge of future tasks, it is not possible to guarantee optimal solutions for the Online-IRTS query. Therefore, we propose two heuristic approaches, where one is based on local optimizations, and the other one is based on incremental solutions, along with a method to evaluate the quality of their solutions w.r.t. the optimal offline solution. Our experiments using city-scale realistic datasets show that the first approach is more effective whereas the second is more efficient, allowing one to choose which approach to use according to his/her priorities.

The remainder of this thesis is structured as follows. In Chapter 2 we discuss k-Diverse Nearest Neighbors queries. Chapter 3 presents the offline version of the In-Route Task Selection problem, while Chapter 4 considers such problem in an online setting. We note that the notation used in Chapters 2-4, as well as a discussion of related works, are contained within each corresponding chapter. Finally, Chapter 5 concludes this thesis with a summary of our findings and suggestions for future work.

Chapter 2

Diverse Nearest Neighbors Queries Using Linear Skylines

2.1 Introduction

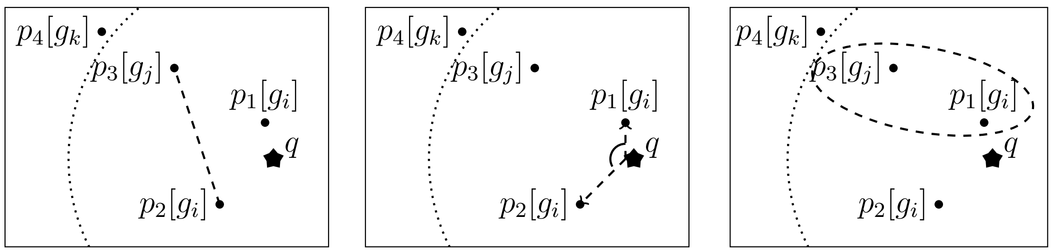
k-Nearest neighbor (k-NN) queries [40] have been extensively studied by researchers in several areas, such as spatial databases, data mining, information retrieval, pattern recognition and statistics. Namely, given a query point q and a set of points P, a k-NN query finds the set of k points $P' \in P$ such that no other point in $P \setminus P'$ is closer to q than those in P' according to a given metric.

One potential drawback of k-NN queries is the fact that the k-NNs are determined based *only* on their distance to the query, i.e., no assumption is made on how they relate to each other. Providing homogeneous result sets, i.e., ones where the elements are very similar to each other, may not add much information to the query result as whole. One such scenario is document ranking [7], where returning k documents that are close to the query, but diverse among themselves is likely to be more interesting as it increases the amount of overall information one gathers from the response.

This concern has motivated some researchers to incorporate diversity into similarity searches, i.e., to find elements that are as close as possible to the query, while, at the same time, being as diverse as possible. The relation between diversity and closeness has been mainly investigated within the domain of information retrieval, e.g., [7], [11], recommendation systems, e.g., [58], web retrieval, e.g., [39] and spatial query processing [33], [59].

In this chapter, we incorporate the notion of diversity into the realm of spatial k-NN queries and investigate (k, r)-DNNs queries. A (k, r)-DNN query returns the k *diverse* nearest neighbors no farther than a radius r from q. Next, we present the three different notions of diversity we investigate in this chapter: spatial, angular and categorical.

In *spatial diversity*, the diversity between two data points is given by the distance between them. For instance, a tourist visiting a city may want to explore points-of-interest (POIs) that are not too far from his/her location, but wants to have a variety of different parts of the city to choose from. In the example shown in Figure 2.1a, the distance between points p_2 and p_3 is greater than the pairwise distance of any two points. Therefore, the set $\{p_2, p_3\}$ represents the most diverse set for $k = 2$ when spatial diversity is considered.

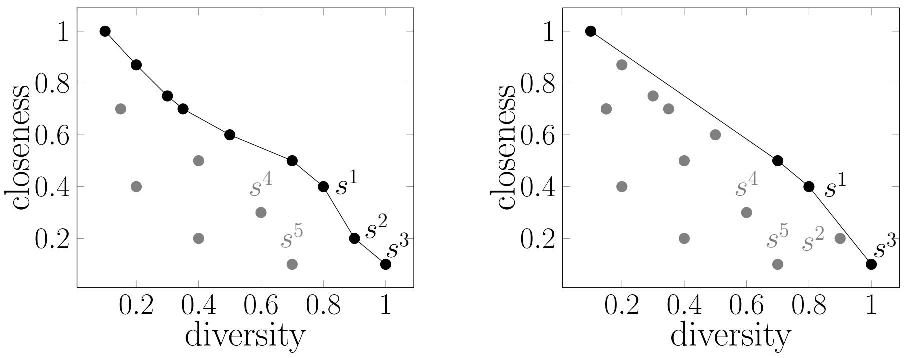


(a) Spatial (b) Angular (c) Categorical

Figure 2.1: Answer sets for spatial, angular and categorical diversities for $k = 2$. ($p_i[g_j]$ denote that POI p_i belongs to category g_j). The dotted arc represents query-defined radius r, hence p_4 cannot be part of the solution set.

In *angular diversity*, the diversity is modeled by the difference between the directions of data points with respect to the query point. For instance, a tourist could be interested in visiting attractions that surround his/her location but are in different directions. As shown in Figure 2.1b, the points returned when angular diversity is considered, i.e., $\{p_1, p_2\}$, are not necessarily far away from each other as the most spatially diverse points, $\{p_2, p_3\}$, but by virtue of being in different directions also bring in the notion of discovering different regions in a more principled manner. Regarding *categorical diversity*, the diversity is modeled by the difference between categories (or labels) of data points. As an example, if a user is looking for restaurants close by, it may be interesting to return different types of restaurant so that the user could make a decision based on diversified options. As illustrated in Figure 2.1c, although p_1 and p_2 are the two closest points from q, they belong to the same category g_i. Thus, $\{p_1, p_3\}$ would be a more diverse choice.

As we shall discuss shortly, previous works that dealt with similar problems, i.e., balancing closeness and diversity, suffer from two main shortcomings: (1) they rely on a particular user-provided linear combination of the relative importance of closeness over diversity, and (2) they do not provide exact solutions, given the problem is NP-hard [8]. The approaches we propose in this chapter overcome both such shortcomings. In order to find *all* results that are *optimal* under *any* given arbitrary combination of two competing criteria (e.g., closeness and diversity), we rely on the notion of skyline queries [5], more specifically linear skylines [43]. The result set of a skyline query contains elements which are not dominated by any other element. In the context of (k, r)-DNN queries, a solution is not dominated if there is no other solution with higher closeness *and* diversity than its own. In order to illustrate the concept of skyline queries within this context, consider Fig. 2.2a. Each point represents a solution with its corresponding closeness and diversity. All the points shown in gray have lower closeness and/or diversity than the points shown in black. This means that the gray points are dominated. On the other hand, the points shown in black represent the skyline, i.e. the set of all non-dominated solutions. However, skyline queries may return a large amount of result sets, potentially with similar costs, making the task of choosing the "right" one a difficult one.



(a) Conventional skyline (b) Linear skyline

Figure 2.2: Illustration of conventional vs linear skylines. The linked dark dots denote the frontier of non-dominated solutions returned by each approach.

Linear skyline queries have been proposed as a way to reduce the result set to a more intuitive one. Two important properties of linear skylines are that (1) they typically return a much smaller subset of the conventional skyline and (2) the solutions are optimal under any *linear* combination of the competing criteria. The linear skyline obtained from the conventional skyline shown in Figure 2.2a is depicted in Figure 2.2b.

The main contributions of this chapter are two algorithms to find optimal answers for (k, r)-DNN queries using linear skylines. In our proposed algorithms the candidate subsets are generated in decreasing order of closeness to the query point, i.e., increasing order of distance. This facilitates checking whether a candidate is part of the skyline or not, since all posterior subsets must have higher diversity than the previous ones in order to be non-dominated. Once the distance from q to a data point exceeds the given radius r, the algorithms stop, that is, only data points within distance r from q are considered.

Our extensive experiments, that investigate several different parameters and explore both Euclidean space as well as real data in road networks, show that our proposed methods find all optimal solutions for any linear combination of the optimization criteria several orders of magnitude faster than a straightforward approach.

The remainder of this chapter is structured as follows. In Section 2.2 we present a discussion of related work. Next, we formally define (k, r)-DNN queries in Section 2.3, and follow with our proposed solutions in Section 2.4. The experimental evaluation and results are shown and discussed in Section 2.5. Finally, Section 2.6 presents a summary of our findings.

2.2 Related Work

The concept of incorporating diversity into similarity search has its origins in information retrieval. The Maximal Marginal Relevance (*MMR*) model [7] is one of the earliest proposals to consider diversity to re-rank documents in the answer set, where at each step, the element with higher marginal relevance is selected. A document has high marginal relevance if it is both relevant to the query and has minimal similarity to previously selected documents.

In [53], Vieira *et al* survey several other approaches and show that, compared to those, their proposed methods Greedy Marginal Contribution (*GMC*) and Greedy Randomized with Neighborhood Expansion (*GNE*) are superior. Thus, we will review only those two methods. Similarly to *MMR*, *GMC* constructs the result set incrementally and the elements are ranked based on their maximum marginal contribution (a concept similar to maximal marginal relevance in [7]). In the *GNE* approach, a predefined number of result sets is found and the one that maximizes the optimization function is returned. Differently from *GMC*, in each iteration the element chosen is a random element among the top ranked ones. For each result set R constructed, the algorithm then performs swaps between elements in R and elements that offer a great chance to be more diverse. If any of these swaps leads to a better result, according to the optimization function, then it is set as the best solution. We choose to compare our proposed approaches to *GNE* because, although it is an approximate solution, it is the one that presented the highest precision w.r.t. the optimal solution (obtained through an exhaustive brute-force approach). Moreover, it is generic and can be applied to all of the three types of diversity considered in this chapter.

In the context of spatial diversity, several other approaches have been proposed. In Jain *et al* [31] the goal is to find the k-NNs to a given query point q such that the distance between any two points is greater than a predefined minimum diversity. The work proposed by Abbar *et al* [1] strives to find the most diverse set within a predefined radius in Hamming space. Note that the works above do not consider, as we do, the notion of a trade-off between closeness and diversity. Zhang *et al* [59] study the problem of diversified spatial keyword search on road networks, aiming at maximizing a linear function that combines both criteria. The keywords are used for filtering data points, i.e., only points that contain all query keywords are considered. The diversity between data points is given by their network distance and the diversity between keywords is not taken into account in such function.

Regarding angular diversity, Lee *et al* [34] presented the Nearest Surrounder

Query, which aims at finding the nearest objects from a query point from different angles. However, as in [1], [31], no compromise between closeness and diversity is considered. Kucuktunc *et al* [33] investigated the diversified k-NN problem based on angular similarity. The goal was to find a set of k points that optimizes a linear function which minimizes pairwise angular similarity and maximizes relevancy of the returned points.

Differently from our proposal, all of the works above focused on a single type of diversity and none of them supports the categorical diversity considered in this chapter. Moreover, all the approaches presented above that consider a trade-off between closeness and diversity, propose approximate solutions to the problem of optimizing a given linear combination of both criteria. We, on the other hand, find the set of *all* optimal solutions for *any* linear combination of these criteria, i.e., the linear skyline, discussed next.

The skyline operator was first introduced in [5]. Given a d-dimensional data set, a skyline query returns the points that are not dominated by any other point. In the context of (k, r)-DNN queries, a solution S is not dominated if there is no solution S' with higher closeness and higher diversity than S. One interesting aspect of skyline queries is that the user does not need to determine beforehand weights for closeness and diversity. The skyline query provides the user with a set of multiple and equally interesting solutions in the sense they are all non-dominated, for arbitrary weights. The users then make their decision based on the returned options. A drawback of skyline queries is that it may return a large answer set, which may make it difficult for users to interpret the results and choose the solution that better suits their preferences. To overcome this problem, [29], [46], [52] have proposed to reduce the amount of results by focusing on finding the k skyline points that best diversify the skyline result. Note that in this work we consider diversity as one of the criteria to be maximized, while those papers first find the skyline for any set of criteria and then select the skyline points that are more diverse among themselves. Shekelyan et al. [43] propose the more pragmatic concept of *linear* skyline queries. A linear skyline consists of a relatively small subset of the conventional skyline which represents optimal solutions under *all* linear combination functions of the criteria to be optimized. Thus, in this chapter we rely on the notion of linear skylines and detail how we use it in the next section.

2.3 Preliminaries and Problem Definition

The (k, r)-Diverse Nearest Neighbor ((k, r)-DNN) query incorporates diversity into traditional spatial k-NN queries. Given a set of points $P = \{p_1, p_2, \ldots p_n\}$, the goal of a (k, r)-DNN query is to find a set of solutions with k points from P, all within a radius r w.r.t. the query point q, that are at the same time close to q and as diverse as possible.

Given the set P, we define the closeness of a point $p_i \in P$ w.r.t the query q as the complement of their distance $d(p_i, q)$ relative to radius r. Thus maximizing the closeness of p_i w.r.t q is the same as minimizing $d(q, p_i)$. When a road network setting is considered, $d(q, p_i)$ is defined as the network distance, i.e., the length of the shortest path connecting those points. On the other hand, in the Euclidean space, $d(q, p_i)$ is defined as the Euclidean distance between q and p_i. The closeness of an object p_i w.r.t q is then defined as $clo(q, p_i) = 1 - \frac{d(q,p_i)}{r}$. Given the above, the closeness of a subset $S \subseteq P$ w.r.t. q can be defined as follows.

Definition 2.3.1 (Closeness and distance w.r.t a set)**.** For a given set $S \subseteq P$, the closeness of S w.r.t. q is defined as $\overline{clo}(q, S) = \min_{s_i \in S}\{clo(q, s_i)\}$, i.e., the minimum closeness between q and a point in S. Similarly, $\overline{d}(q, S)$ is defined as the distance from q to its farthest point in S.

Next, we define the three different notions of diversity considered in this chapter: spatial, angular and categorical.

Definition 2.3.2 (Spatial Diversity)**.** The spatial diversity between two points $p_i, p_j \in P$ is given by the distance between them, i.e., $div_{spa}(p_i, p_j) = d(p_i, p_j)$.

Spatial diversity is also adopted in [24], [59]. The idea is to find points that are well spatially distributed and consequently cover different areas. In Figure 2.1a, the pair (p_2, p_3) is the most spatially diverse since $div_{spa}(p_2, p_3)$ is greater than the diversity of any other pair of points.

Definition 2.3.3 (Angular Diversity)**.** The angular diversity between two points $p_i, p_j \in P$ w.r.t. q is given by the angle formed by segments $\overline{q, p_i}$ and $\overline{q, p_j}$, i.e., $div_{ang}(p_i, p_j) = \measuredangle(p_i, q, p_j)$.

For the sake of an example, consider Figure 2.1b. The most diverse pair with respect to angular diverse is (p_1, p_2) since the angle formed by them is greater than the angle formed by any other pair of points.

Regarding categorical diversity, we assume that there is a set of categories $G = \{g_1, g_2, ..., g_{|G|}\}$ and that each point $p_i \in P$ is labeled with a category denoted by $g(p_i) \in G$. Naturally, categories are non-exclusive, i.e., more than one point may belong to the same category. We model the diversity between categories as a matrix $M_{|G|\times|G|} = (m_{j,l})$, and for simplicity, we assume that $0 \leq m_{j,l} \leq 1$ and that $m_{j,l} = 0 \Leftrightarrow j = l$.

Definition 2.3.4 (Categorical Diversity)**.** The categorical diversity between a pair of points $p_i, p_j \in P$ is given by $div_{cat}(p_i, p_j) = m_{g(p_i),g(p_j)}$, i.e., the diversity between their categories.

The definitions presented above describe how different types of diversity are calculated for pairs of data points. Let us now present the more general definition of diversity within a non-singleton set $S \subseteq P$.

Definition 2.3.5 (Diversity of a set)**.** The diversity of a set $S \subseteq P$ is defined as the minimum pairwise diversity between the points of S, i.e., $\overline{div}(S) = \min_{p_i, p_j \in S}\{div(p_i, p_j)\}$, where div is one of the three diversity cases discussed above: $div_{spa}, div_{ang}, div_{cat}$.

Note that the angular diversity of a set also depends on q, since all computed angles depend on q, unlike the other types of diversity we discuss in this chapter. Nonetheless, for the sake of clarity we omit q, which is a constant, from the definition.

Next, we define the notions of conventional and linear skyline domination in terms of closeness and diversity. In what follows we use $s^i = (\overline{div}(S^i), \overline{clo}(q, S^i))$ to denote a vector representing the diversity and closeness of a set S^i.

Definition 2.3.6 (Conventional skyline)**.** Let S^i and S^j be k-sized subsets of P. Then, in the context of (k, r)-DNN queries, S^i dominates S^j, denoted as $S^i \prec S^j$, if

$$\begin{aligned} &\overline{clo}(q, S^i) > \overline{clo}(q, S^j) \text{ and } \overline{div}(S^i) \geq \overline{div}(S^j) \quad \text{or} \\ &\overline{clo}(q, S^i) \geq \overline{clo}(q, S^j) \text{ and } \overline{div}(S^i) > \overline{div}(S^j) \end{aligned}$$

That is, S^i is better in one criteria and at least as good as S^j in the other one. (Note that the notion of goodness is w.r.t. the criteria we aim to maximize.) Thus, the conventional skyline, i.e., the set of all non-dominated subsets, is given by

$$\{S^i \subseteq P \text{ s.t. } \nexists S^j \subseteq P : S^j \prec S^i, |S^i| = |S^j| = k\}.$$

In order to illustrate the concept of conventional dominance, consider the points $\{s^1, \ldots, s^5\}$ shown in Figure 2.2 representing the solutions $\{S^1, \ldots, S^5\}$. The solution S^4 is dominated by solution S^1, since S^1 is both closer to q and more diverse than S^4. Similarly, S^5 is dominated by S^1, S^2 and S^3.

A linear skyline consists of the subset of the conventional skyline which is optimal under all linear combination functions, i.e., for any weights (importance) that a user could give to either criteria. Next we present the definition of w-dominance which determines whether a set linearly dominates another one for a particular weight vector $w = (w_{div}, w_{clo})$, where w_{div} and w_{clo} denote the importance given to the diversity and closeness aspects of the answer sets.

Definition 2.3.7 (Linear w-Dominance)**.** A set S^i linearly w-dominates another set S^j, where $w \in \mathbb{R}^2_{\geq 0}$ and $w \neq (0,0)$, if and only if $w^T s^i > w^T s^j$ [43]. For simplicity in the remainder of this chapter, we refer to linear w-dominance simply as w-dominance.

For instance, in Figure 2.2, the set S^1 w-dominates S^3 for $w = (0.5, 0.5)$, since $0.5 \times 0.8 + 0.5 \times 0.4 > 0.5 \times 1 + 0.5 \times 0.1$. However, this is not a sufficient condition for determining whether S^3 is a linearly dominated set in the general sense. Note that, for example, S^3 w-dominates S^1 for $w = (1, 0)$. As formalized in the following definition, a set is only considered to be linearly dominated if it is either conventionally dominated or w-dominated for every possible w.

Definition 2.3.8 (Linear Skyline)**.** Let $\mathcal{S}$ be a set of solutions in a two-dimensional space. A subset $\mathcal{S}' \subseteq \mathcal{S}$ linearly dominates a solution $S^j \in \mathcal{S}$, denoted as $\mathcal{S}' \prec_L S^j$, if and only if

$$(\exists S^i \in \mathcal{S}' \; s.t. \; S^i \prec S^j) \vee (\forall w \in \mathbb{R}^2_{\geq 0} \; \exists S^i \in \mathcal{S}' \; s.t. \; w^T s^i > w^T s^j)$$

The maximal set of *linearly non-dominated* solutions is referred to as *linear skyline* [43].

In other words, a solution S^i is linearly non-dominated w.r.t. a set $\mathcal{S}'$ if it is not conventionally dominated by any solution in $\mathcal{S}'$ *and* there is a vector w and a solution $S^j \in \mathcal{S}'$ such that S^i w-dominates S^j. For the set of solutions $\mathcal{S}' = \{S^1, S^2, S^3, S^4, S^5\}$ shown in Figure 2.2, S^1 and S^3 are considered linearly non-dominated since they are conventionally non-dominated and, as shown above, S^1 w-dominates S^3 for $w = (0.5, 0.5)$ and S^3 w-dominates S^1 for $w = (1, 0)$. Thus, both solutions satisfy the conditions to be considered linearly non-dominated. On the other hand, although S^2 is conventionally non-dominated,

Notation	Meaning
P	Set of data points
q	Query point
k	Answer set size
r	Query radius
$d(p_i, p_j)$	Distance between the data points p_i and p_j
$\overline{d}(q, S)$	Distance of the set S w.r.t. q
$clo(q, p_i)$	Closeness of an object p_i w.r.t q
$\overline{clo}(q, S)$	Closeness of a set S w.r.t. q
$div_{spa}(p_i, p_j)$	Spatial diversity between two points $p_i, p_j \in P$
$div_{ang}(p_i, p_j)$	Angular diversity between two points $p_i, p_j \in P$
$div_{cat}(p_i, p_j)$	Categorical diversity between two points $p_i, p_j \in P$
$\overline{div}(S)$	Diversity (spatial, angular or categorical) of a set S
S^i	k-sized subset of P
s^i	Vector representing the diversity and closeness of a set S^i
$S^i \prec S^j$	S^j is conventionally dominated by S^i
$\mathcal{S}' \prec_L S^j$	S^j is linearly dominated by the set of solutions $\mathcal{S}'$

Table 2.1: Notation used in Chapter 2.

there is no weight vector w for which S^2 w-dominates either S^1 or S^3. S^4 and S^5 are conventionally dominated solutions and thus they are also linearly dominated.

Finally, the problem addressed in this chapter can be defined as follows.

Problem Definition. *Given a query q, a positive integer k, a radius r and a set of data points P, the (k, r)-DNN query aims at finding the set of all k-sized linearly non-dominated subsets $S \subseteq P$ within distance r from q.*

Throughout this chapter we use the notation presented in Table 2.1.

2.4 Proposed Solutions

As our main contribution in this chapter we present two algorithms that provide provably optimal solutions to the (k, r)-DNN query using linear skylines. Both algorithms analyze the candidate subsets in decreasing order of closeness, which means that the subsequent subsets necessarily need to have a higher diversity than the previous ones in order to be non-dominated. This allows us to establish a lower bound to the diversity of the next non-dominated result sets. Throughout this chapter, this lower bound is denoted by *minDiv*. Both proposed solutions, named *Recursive Range Filtering* (*RRF*) and *Pair Graph* (*PG*), use such lower bound to filter out dominated subsets. *RRF* works recursively by combining a partial solution S' with points in a candidate set C, one at time, that have a higher diversity to S' than *minDiv*. *PG* strives to reduce the number of generated combinations by pruning subsets that contain pairs of elements $(p_i, p_j) \subseteq P$ such that $div(p_i, p_j) \leq minDiv$ and therefore could not be in a non-dominated solution.

Our proposed approaches are general in the sense that they work for all the three types of diversity considered in this chapter. However, note that for categorical diversity only the closest points from each category are needed in order to find the whole skyline set. Farther points would lead to solutions with lower closeness and that would not have higher diversity, since the diversity between two points is determined by their categories. Therefore, such solutions would be dominated. This means that the algorithms can stop once the closest point from each category is found.

RRF and *PG* are both based on Algorithm 1. They differ in how they find the set with maximum diversity for a fixed closeness, which is done by the method *findSetMaxDiv* used within Algorithm 1. We will start by describing the general framework of Algorithm 1 and we present the different approaches for computing *findSetMaxDiv* when describing *RRF* and *PG* in more detail.

Algorithm 1 takes as input a query point q, the set of data points P, the size of the answer sets k and the radius r. The first subset S generated is the closest one, i.e., the one with minimum $\overline{d}(q, S)$ (line 1). The elements $s_i \in S$ are removed from P and added to the set of previously examined points P' (lines 2-3). S is then added to the list LS, which stores linearly non-dominated solutions in decreasing order of closeness.

Note that S is linearly non-dominated: (1) it is conventionally non-dominated since no other solution can dominate it in terms of closeness and (2) it w-dominates any possible solution for $w = (0, 1)$. Therefore, S satisfies the two conditions to be considered linearly non-dominated, as per Definitions 2.3.7 and 2.3.8 above. Next, $minDiv$ is set to $\overline{div}(S)$. The variable $minDiv$ represents the minimum diversity that the subsequent solutions must have in order to be non-dominated. All subsets S' with $\overline{div}(S') \leq minDiv$ are discarded since they are dominated solutions and therefore do not belong to LS.

The remaining elements $p_d \in P$ are examined in increasing order of $d(q, p_d)$, until $d(q, p_d)$ exceeds the query radius r (lines 6-10). For each such p_d, the

Algorithm 1: MinDistMaxDiv

```
Data: Query $q$, set of data points $P$, result set size $k$, radius $r$
Result: All linearly non-dominated solutions $S \subseteq P$, $|S|$ = k and
        $\overline{d}(q, S) \leq r$
$S \leftarrow k\text{-}NN(q, P)$;
$P \leftarrow P \setminus S$;
$P' \leftarrow S$;
$LS.add(S)$;
$minDiv \leftarrow \overline{div}(S)$;
while $|P| > 0$ do
    $p_d \leftarrow \underset{p_i \in P}{\text{argmin}}(d(q, p_i))$;
    if $d(q, p_d) > r$ then
        return $LS$;
    end
    $P \leftarrow P \setminus p_d$;
    $C \leftarrow$ all $p_i \in P'$ such that $div(p_d, p_i) > minDiv$;
    $S \leftarrow findSetMaxDiv(args)$;
    if $S$ is not null then
        $j \leftarrow |LS|$;
        $LS.add(S)$;
        $minDiv \leftarrow \overline{div}(S)$;
        while $j > 2$ & $\{S^{j-1}, S\} \prec_L S^j$ do
            Delete $S^j$ from $LS$
            j=j-1
        end
    end
    $P' \leftarrow P' \cup p_d$;
end
return $LS$;
```

method $findSetMaxDiv$ looks for the set S with maximum $\overline{div}(S) > minDiv$ such that $p_d \in S$ and $S \setminus p_d \subseteq C$. Where C is a list containing all $p_i \in P'$ such that $div(p_d, p_i) > minDiv$ (line 12). This step eliminates the elements that cannot be combined with p_d in order to obtain a non-dominated solution. Note that $\overline{clo}(q, S) = clo(q, p_d)$ since p_d is the farthest point from q in S. Also, for a particular p_d, all subsets examined by *findSetMaxDiv* have the same closeness, since they necessarily contain p_d. This implies that for each p_d there is up to one non-dominated solution, which is the one with the highest diversity.

If a non-dominated solution S is found by *findSetMaxDiv*, S is added to the end of LS (line 16) and $minDiv$ is updated to $\overline{div}(S)$. As proved in [43], if S is a conventionally non-dominated solution, then it is also linearly non-dominated in LS and thus it can be added to that list. After adding S to LS, some solutions in LS may become linearly dominated and thus must be discarded. Let S^j be the jth element in LS. Shekelyan et al. [43] showed that any solution S^j is only linearly dominated in LS if it is dominated by both of its neighbors S^{j-1} and S^{j+1}. Thus, in order to determine if S^j is linearly dominated, it is sufficient to check whether $\{S^{j-1}, S^{j+1}\} \prec_L S^j$. Let j be the size of LS before inserting the new non-dominated solution S. We perform a left traversal in LS (lines 18-21) and first check whether $\{S^{j-1}, S\} \prec_L S^j$ holds. If so, S^j is removed from LS. After that, S^{j-1} becomes the left neighbor of S. Subsequently, we need to examine whether $\{S^{j-2}, S\} \prec_L S^{j-1}$ holds. This process will be terminated when the first element of LS has been examined or the current left neighbor of S is a linearly non-dominated set.

In order to check whether $\{S^{j-1}, S^{j+1}\} \prec_L S^j$ holds, we follow the procedure proposed in [43]. Intuitively, S^j is linearly dominated by S^{j-1} and S^{j+1} if the vector s^j lies below the line between s^{j-1} and s^{j+1}. For instance, in Figure 2.2b, S^2 is linearly dominated by S^1 and S^3 because s^2 lies below the line between s^1 and s^3. This observation can be formalized as follows. Let n be the normal vector of the line between s^{j-1} and s^{j+1}, such that $n^T s^{j-1} = n^T s^{j+1}$, then $\{s^{j-1}, s^{j+1}\} \prec_L s^j$ iff $n^T s^j < n^T s^{j-1} = n^T s^{j+1}$.

Next, we prove that the set of solutions generated by Algorithm 1 is correct and complete. For this we assume that the *findSetMaxDiv* method is correct and finds the corresponding non-dominated solution for a given $p_d \in P$, if such solution exists. We prove such assumption, i.e., the correctness of the *findSetMaxDiv* method used in each of our proposed solutions in the following subsections, when RRF and PG are presented. Moreover, we also assume that the left traversals [43] performed in LS in order to discard linearly dominated solutions work correctly.

Theorem 1. *LS does not contain linearly dominated solutions.*

Proof. The first solution S found by Algorithm 1 is the closest one and thus is not linearly dominated. All subsequent solutions cannot dominate previous solutions in terms of closeness, since they are found in increasing order of distance. Therefore, those solutions must have a higher diversity than $minDiv$, the highest diversity found so far, in order to be non-dominated. For each point $p_d \in P$, a new solution S is returned by the $findSetMaxDiv$ method only if $\overline{div}(S) > minDiv$. Moreover, $findSetMaxDiv$ finds the corresponding non-dominated solution for p_d, i.e., the one with highest diversity, if such solution exists. Therefore, since the solutions that are not conventionally dominated but are linearly dominated are correctly discarded, no linearly dominated solution can be part of LS. $\square$

Theorem 2. *All linearly non-dominated solutions are in LS.*

Proof. Let us suppose by contradiction that there is a linearly non-dominated solution S that is not part of LS. Assuming $\overline{d}(q, S) = d(q, p_d)$, $p_d \in P$ implies that S was not found by the $findSetMaxDiv$ method when p_d was the current element with minimum distance w.r.t. q in P. This is a contradiction since $findSetMaxDiv$ is guaranteed to find the corresponding non-dominated solution for any $p_d \in P$ if such solution exists. (This last statement is proven, in the context of each proposed method, next) $\square$

2.4.1 Recursive Range Filtering (RRF)

The *findSetMaxDiv* method used in the Recursive Range Filtering (RRF) solution, which in this context we denote as *RRF-findSetMaxDiv* to remove any ambiguity, assumes that p_d, the point in P with minimum distance from q, is part of the non-dominated result set S, if such set exists. Also, as explained above, p_d is the farthest element from q in S. A list of candidates C, containing points that can be combined with the partial set $S' = \{p_d\}$ in order to obtain a non-dominated solution, is given as input to *RRF-findSetMaxDiv*. One element c_i from C is chosen at a time to also be part of S'. Elements that cannot be combined with c_i are removed from C and the algorithm is recursively called for the new partial set S' and list of candidates C.

Algorithm 2 shows how the *RRF-findSetMaxDiv* method works. It takes as input a partial set S', which initially only contains the point p_d, a list C of candidate points that can be combined with S', the number n of elements that need to be added to S' to form a set of size k and the current k-sized set with maximum diversity $setMaxDiv$, which is initially empty. If $n > 1$ (line 8), the candidates c_i in C are examined one at a time. More specifically, we check whether the list of candidates C_i obtained after adding c_i to S' contains at least $n-1$ points (line 12). If so, the function is recursively called for the new partial set S'_i and list of candidates C_i (line 14). Otherwise, it is not possible to obtain a non-dominated solution S such that $S'_i \subseteq S$ and therefore S'_i can be discarded. Note that we avoid creating the same subset more than once by not adding previously examined candidate points $c_j \in C$ such that $1 \leq j < i$ to C_i (line 10). That is because all the possible sets containing $S' \cup c_j$ have already been created, including the ones that contain c_i, and adding c_j to C_i would just create duplicates and make the algorithm less efficient.

In the base case (line 1), only one point still needs to be added to S'.

We then simply create a solution S for each $c_i \in C$ and set the one with highest diversity to $setMaxDiv$. When no other set of size k can be built, the algorithm returns $setMaxDiv$ (line 18).

Algorithm 2: *RRF-findSetMaxDiv*

```
Data: Partial set $S'$, list of candidates $C$, $n = k - |S'|$, $setMaxDiv$ and $minDiv$
1  if $n$=1 then
2      for $i$ from 1 to $|C|$ do
3          $S \leftarrow S' \cup c_i$;
4          if $\overline{div}(S) > \overline{div}(setMaxDiv)$ then
5              $setMaxDiv \leftarrow S$;
6          end
7      end
8  else
9      for $i$ from 1 to $|C| - n + 1$ do
10         $C_i \leftarrow C[i+1, \ldots, |C|]$;
11         $C_i \leftarrow$ all $c_j \in C_i$ such that $div(c_i, c_j) > minDiv$;
12         if $|C_i| \geq n - 1$ then
13             $S'_i \leftarrow S' \cup c_i$;
14             $setMaxDiv \leftarrow findSetMaxDiv(S'_i, C_i, n-1, setMaxDiv, minDiv)$;
15         end
16     end
17 end
18 return $setMaxDiv$;
```

Figure 2.3 illustrates one iteration of Algorithm 2 for $k = 4$, $p_d = p_6$ and considering spatial diversity. We assume that the points $P' = \{p_1, p_2, p_3, p_4, p_5\}$ have already been examined and the following linearly non-dominated solutions have been generated $LS = \{(\{p_1, p_2, p_3, p_4\}, clo = 0.55, div = 0.5), (\{p_2, p_3, p_4, p_5\}, clo = 0.44, div = 1)\}$. Therefore, at this point $minDiv = 1$.

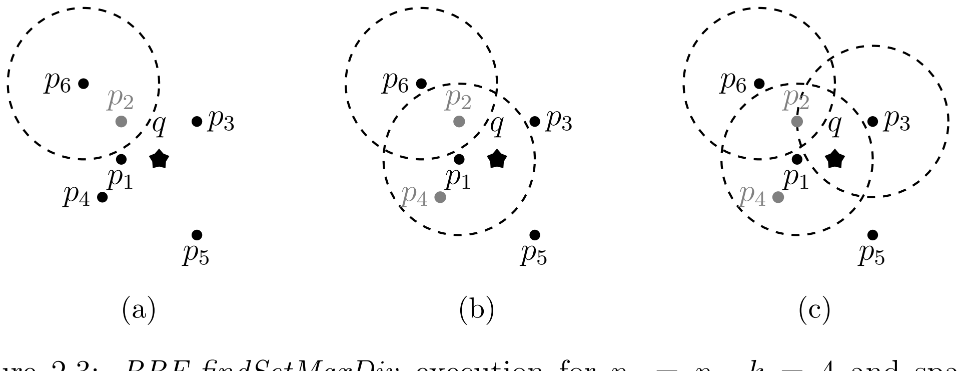


Figure 2.3: *RRF-findSetMaxDiv* execution for $p_d = p_6$, $k = 4$ and spatial diversity.

The method takes as input $S' = \{p_6\}$, $C = [p_1, p_3, p_4, p_5]$, $n = 3$ ($i.e., k-1$) and $setMaxDiv = \emptyset$. Note that, as shown in Figure 2.3a, p_2 cannot be combined with p_6 and thus is not part of C. The first candidate point p_1 is added to S', which discards p_4 from C, as shown in Figure 2.3b. The algorithm is then recursively called for $S' = \{p_6, p_1\}$, $C = [p_3, p_5]$. The candidate point p_3 is added to S' and the only remaining candidate is $C = [p_5]$, as illustrated in Figure 2.3c. The base case, where $n = 1$, is reached and a solution is created for each point in C. The solution $S^1 = \{p_6, p_1, p_3, p_5\}$ with $\overline{div}(S^1) = 1.12$ is built and becomes the current $setMaxDiv$. When the second candidate point of $C = [p_3, p_5]$, i.e., p_5, is considered to be added to $S' = \{p_6, p_1\}$ (omitted in the example), the corresponding candidate set will be empty. As explained above, this avoids creating the same set S^1 twice. Next, the second candidate from $C = [p_1, p_3, p_4, p_5]$, i.e., p_3, is added to $S' = \{p_6\}$, creating the new candidate set $C = [p_4, p_5]$. Note that p_1 is not added to C because all the possible sets containing p_1 have already been generated. The solution $S^2 = \{p_3, p_4, p_5, p_6\}$ with $\overline{div}(S^2) = 1.35$ is found and becomes the current $setMaxDiv$. When p_4 and p_5 are considered to be added to $S' = \{p_6\}$ (omitted in the example), they are pruned because there are not enough points to be combined with them in order to create a non-dominated solution that has not been generated yet.

Theorem 3. For any p_d, the element in P with minimum distance to q, the method *RRF-findSetMaxDiv* finds the corresponding non-dominated solution S, where $\overline{d}(q, S) = d(q, p_d)$, if such a solution exists.

Proof. Let us assume that S exists and the method does not find it. This means that at least one $s_i \in S$, $s_i \neq p_d$ has been discarded by *RRF-findSetMaxDiv*. There are two possible cases:

1. $div(s_i, s'_j) \leq minDiv$ holds for at least one $s'_j \in S'$, $S' \subseteq S \setminus s_i$.

This is a contradiction to the assumption that S is a non-dominated solution and thus $div(s_i, s_j) > minDiv$ for all $s_i, s_j \in S$.

2. There is no subset $S' \subseteq S \setminus s_i$ such that $S' \cup s_i$ can be combined with at least $n = k - |S'| - 1$ points.

   This is also a contradiction to the fact that S is a solution with k points and thus $S' \cup s_i$ can be combined with $S \setminus \{S' \cup s_i\}$.

Thus, such set S must be found by the method *RRF-findSetMaxDiv*. $\square$

2.4.2 Pair Graph Approach (PG)

The idea behind the PG approach is to combine pairs of elements (p_i, p_j) such that $div(p_i, p_j) > minDiv$ in order to construct sets of size k that can potentially be a non-dominated solution. Two pairs can be combined if they have an element in common. For instance, set $S^1 = (p_i, p_j)$ can be combined with set $S^2 = (p_j, p_l)$ if $div(p_i, p_j) > minDiv$ and $div(p_j, p_l) > minDiv$, obtaining the set $S^3 = \{p_i, p_j, p_l\}$ of size three. Similarly, S^3 can be combined with a pair (p_l, p_m) to obtain the set $S^4 = \{p_i, p_j, p_l, p_m\}$ of size four if $div(p_l, p_m) > minDiv$. However, note that there is no guarantee that $\overline{div}(S^4) > minDiv$, unless $div(p_i, p_l) > minDiv$, $div(p_i, p_m) > minDiv$ and $div(p_l, p_m) > minDiv$.

A non-dominated solution $S \subseteq P$ for which the candidate point p_d is the farthest one from q, can only contain previously examined elements $p_i \in P'$ such that $div(p_d, p_i) > minDiv$, i.e., the points in the list of candidates C. Therefore, the set of pairs $\mathcal{P}$ needed for following the aforementioned strategy are all (p_d, p_i) for $p_i \in C$ and all (p_i, p_j) for $p_i, p_j \in C$ such that $div(p_i, p_j) > minDiv$.

In order to facilitate the construction of sets of size greater than two by combining the pairs in $\mathcal{P}$, we build a graph $G(V, E)$, where the set of vertices is $V = C \cup p_d$. Also, for each pair $(p_i, p_j) \in \mathcal{P}$ with $div(p_i, p_j) > minDiv$, we create an edge (p_i, p_j) and add it to the set of edges E. The cost of an edge (p_i, p_j) is given by $div(p_i, p_j)$. Note that finding sets of size k can be accomplished by finding paths with k vertices in G.

Figure 2.4 shows the graph G built for the example shown in Figure 2.3, i.e., for $p_d = p_6$, $k = 4$ and spatial diversity. Since all the sets of size $k = 4$ must contain p_6, in order to find paths in G that may represent a non-dominated solution, we assume that the starting point is p_6. Consider the following possible paths of size 4 starting from p_6: $PT^1 = (p_6, p_1, p_3, p_5)$, $PT^2 = (p_6, p_3, p_4, p_5)$ and $PT^3 = (p_6, p_1, p_3, p_4)$. The diversity of PT^1 is 1.12 because this is the minimum cost of an edge between any two vertices of PT^1. Since $1.12 > 1 = minDiv$, PT^1 is a candidate to be added to skyline. Similarly, the diversity of PT^2 is 1.35, which makes PT^2 a better candidate than PT^1. However, PT^3 does not have a diversity higher than $minDiv$ since the elements in PT^3 do not form a clique. More specifically, there is no edge (p_1, p_4) in E, and thus $div(p_1, p_4) \leq minDiv$. Therefore, PT^2 represents the set with highest diversity.

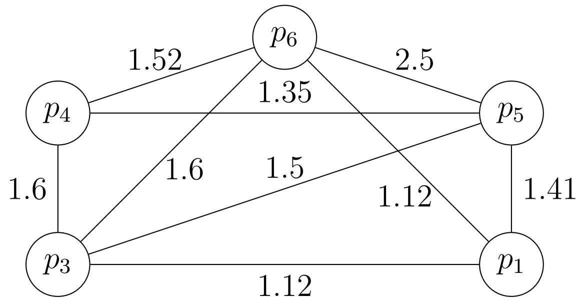


Figure 2.4: Graph G built for the example shown in Figure 2.3.

Algorithm 3 shows all steps of the *PG-findSetMaxDiv* method. In order to find paths of a given size k, we use a solution based on Dijkstra's classical algorithm. First, all edges $(u, v) \in E$, representing paths of size 2, are added to a priority *queue* (lines 5 to 7). The paths PT in *queue* are ordered by decreasing order of an upper bound $ub(PT)$ that represents an optimistic value to the diversity of a set of size k that contains the vertices of PT. Let us assume that the size of PT is $p \leq k$ and that $l = k - p$. As exemplified above, in order to obtain a non-dominated solution of size k that contains the vertices of PT, PT can only be extended by adding l vertices that are neighbors of all vertices of PT. Let MN be this set of mutual neighbors. If $|MN| < l$, then it is not possible to find a non-dominated solution that contains the vertices of PT, therefore PT can be discarded. Otherwise, for each $pt_i \in PT$ let $div^l(pt_i, mn_j)$ be its l-th largest diversity to an element $mn_j \in MN$. We refer to the minimum of these values as $div^m(PT, MN)$. Since $div^m(PT, MN)$ is an upper bound to the diversity of PT to l other elements, if $div^m(PT, MN) \leq minDiv$, PT can also be discarded. The same occurs if $\overline{div}(PT) \leq minDiv$. Therefore, the upper bound $ub(PT)$ to the diversity of a set S of size k containing the vertices of PT is given by $\min\{\overline{div}(PT), div^m(PT, MN)\}$.

At each step, the path PT with highest $ub(PT)$ is removed from *queue* (line 9) and expanded if it contains less than k vertices (line 12). Let v be the last vertex in the path PT. For each neighbor u of the vertex v, PT is extended by adding u. If $ub(PT \cup u) > minDiv$ then the path $(PT \cup u)$ is added to the queue (lines 15 and 16). If PT has size k (line 10), PT is returned. Note that no other posterior path PT' of size k could have higher diversity than PT. Since PT' is not the first set of size k removed from *queue*, then $ub(PT') \leq ub(PT)$, which means that in an optimistic scenario the diversity of PT' is at most equal to $div(PT)$. If no path of size k is found, the algorithm returns a *null* answer.

Theorem 4. For any p_d, *PG-findSetMaxDiv* finds the non-dominated solution S, where $\overline{d}(q, S) = d(q, p_d)$ and $|S| = k$, if such a solution exists.

Algorithm 3: *PG-findSetMaxDiv*

```
Data: List of candidates $C$, result set size $k$, point $p_d$ and $minDiv$
1  $\mathcal{P} \leftarrow$ all pairs $(p_d, p_i)$, $p_i \in C$;
2  $\mathcal{P} \leftarrow \mathcal{P} \cup$ all pairs $(p_i, p_j) \subset C$, $div(p_i, p_j) > minDiv$;
3  $G(V, E) \leftarrow buildGraph(\mathcal{P})$;
4  $queue \leftarrow \emptyset$ ;            // $queue$ stores the paths to be expanded
5  for $(v, u) \in E$ do
6      $queue$.add($(v, u)$);
7  end
8  while $|queue| > 0$ do
9      $PT \leftarrow queue.dequeue()$;
10     if $|PT| = k$ then
11         return $PT$;
12     else if $|PT| < k$ then
13         $v \leftarrow PT[|PT|]$ ;            // $v$ is the last vertex of $PT$
14         for $(v, u) \in E$ do
15             if $ub(PT \cup u) > minDiv$ then
16                 $queue$.add($PT \cup u$);
17             end
18         end
19     end
20 end
21 return $null$;
```

Proof. We divide this proof into two parts. First we prove that the algorithm does not discard any pair of points necessary for finding S, if S exists. Then we show that given all the necessary pairs, *PG-findSetMaxDiv* finds S.

Let us suppose that S exists and the algorithm discards a pair $(s_i, s_j) \subset S$. There are two possible cases:

1. $s_i = p_d$ or $s_j = p_d$

   This means that *PG-findSetMaxDiv* discarded a pair containing p_d. Since S is a non-dominated solution, $div(p_d, s_p) > minDiv$ for any $s_p \in S, s_p \neq p_d$. However, all pairs (p_d, s_p) such that $div(p_d, s_p) > minDiv$ are maintained by our algorithm, a contradiction to the assumption that (s_i, s_j) was discarded.

2. $s_i \neq p_d$ and $s_j \neq p_d$

   The pair (s_i, s_j) is only discarded by the algorithm if $div(p_d, s_i) \leq minDiv$ and/or $div(p_d, s_j) \leq minDiv$. Since $\{p_d, s_i, s_j\} \subseteq S$ and S is a non-dominated solution, $div(p_d, s_i) > minDiv$ and $div(p_d, s_j) > minDiv$, a contradiction.

Therefore no necessary pair is discarded.

Next, we need to prove that *PG-findSetMaxDiv* is able to find S. We do so, again by contradiction, assuming that $S = (s_1, s_2, ..., s_i, s_{i+1}, ..., s_k)$ exists and is not found. This means that S is not removed from the queue in the method. This happens if, during the expansion of a s_i, $2 \leq i < k$, the partial solution $S' = (s_1, s_2, ..., s_i, s_{i+1})$ is not inserted into the queue, which in turn occurs in the following situations:

1. There is no edge $(s_i, s_{i+1}) \in E$.

   This is a contradiction since as proven above, no pair $(s_i, s_j) \subseteq S$ is discarded by the method and consequently there is an edge $(s_i, s_{i+1}) \in E$.

2. $ub(S') \leq minDiv$.

   Therefore $\overline{div}(S') \leq minDiv$ and/or $div^m(S', MN) \leq minDiv$. Since $S' \subseteq S$, $\overline{div}(S') \geq \overline{div}(S)$. As S is a non-dominated solution, $\overline{div}(S) > minDiv$ and consequently $\overline{div}(S') > minDiv$, a contradiction. If $div^m(S', MN) \leq minDiv$ then there is no non-dominated solution of size k that contains the points in S'. A contradiction to the fact that S is non-dominated and $S' \subseteq S$.

$\square$

2.4.3 Baseline Approach

As a baseline for comparison we use a straightforward, brute-force algorithm (*BF*). *BF* generates all possible subsets $S \subseteq P$ of size k, such that $\overline{d}(q, S) \leq r$, in an arbitrary order. If S is a conventionally non-dominated solution, we check whether S is also linearly non-dominated in LS. If so, S is added to LS. Note that, differently from the previous approaches, S is not necessarily the last element in LS, since LS is sorted in decreasing order of closeness and S is found in an arbitrary order. Therefore, after inserting S into LS, we need to perform a left and a right traversal in LS in order to remove potentially linearly dominated solutions. Algorithm 4 shows the pseudo-code of the *BF* approach.

Algorithm 4: *BF* - Brute Force

```
Data: Query $q$, set of data points $P$, result set size $k$, radius $r$
Result: All linearly non-dominated solutions $S \subseteq P$, $|S| = k$ and $\overline{d}(q,S) \leq r$
C ← all $p_i \in P$ such that $d(q, p_i) \leq r$;
LS ← $\emptyset$;
for each set $S \subseteq C$ of $k$ size do
    if $S$ is not conventionally dominated then
        $j$ ← position of $S$ in $LS$;
        if $\{S^{j-1}, S^{j+1}\} \not\prec_L S$ then
            $LS.add(S)$;
            $i = j$;
            while $i > 2$ & $\{S^{i-1}, S\} \prec_L S^i$ do
                Delete $S^i$ from $LS$;
                $i = i - 1$;
            end
            $i = j + 1$
            while $i < |LS| - 1$ & $\{S, S^{i+1}\} \prec_L S^i$ do
                Delete $S^i$ from $LS$;
                $i = i + 1$;
            end
        end
    end
end
return $LS$;
```

2.5 Experiments

We performed experiments using real road networks and synthetically generated data sets in order to evaluate our methods. We first investigated how *PG* and *RRF* perform in comparison to the brute force solution (*BF*) presented in Section 2.4. Then, we compare how they compare to each other in terms of efficiency (recall that both are proved exact thus effectiveness is not a concern) w.r.t. a number of parameters. Finally, we show how *GNE* (discussed in Section 2.2) compares to our proposed approaches in terms of processing time as well as quality of the results (recall that *GNE*'s results are not guaranteed to be exact).

The algorithms were developed in Java and the experiments were conducted on a Linux-based virtual machine with 4 vCPU and 8 GB main memory. All the graphs shown next present the average results (for 50 runs) in logarithmic scale. If an algorithm was not able to find the solution in less than 10 minutes, the computation was aborted and (optimistically and simplistically) assumed to take 10 minutes.

The real datasets used in our experiments reflect the road networks of Amsterdam (AMS), Oslo (OSLO) and Berlin (BER), as of March/2017 [3]. We assume that the eateries (restaurants and coffee shops) of those networks serve as POIs. Table 2.2 summarizes the details of the datasets used in our experiments.

	Amsterdam	Oslo	**Berlin**
#vertices	106,599	305,174	**428,768**
#edges	130,090	330,632	**504,228**
#POIs	824	958	**3,083**
diameter (km)	38.8	57.4	**72.8**

Table 2.2: Summary of the real datasets used in our experiments (**bold** defines default values).

We also evaluated how our proposed approaches scale with problem size and how they behave under different data distributions within an Euclidean Space, namely Uniform (default), Gaussian and Clustered. The clustered data set was created using the generator described in [25] based on multivariate normal distribution. The dataset generated contains three clusters, each with roughly the same number of points. We also considered datasets with different sizes, where each data point represents a POI. Table 2.3 shows the values examined in our experiments.

$\|P\|$ **(# POIs)**	1000, **5000**, 25000
Distribution	**Uniform**, Gaussian, Clustered

Table 2.3: Parameters for generating the synthetic datasets and their values (**bold** defines default values).

For both real and synthetic datasets, we also varied the size of the answer sets k and the radius r as a percentage of the network's diameter, as shown in Table 2.4. As usual, when varying one parameter the others were kept at their default values. In both datasets, the query points were assumed to follow the actual POIs' distribution, i.e., POIs were selected randomly as query points.

Answer set size (k)	3, **5**, 7, 10
Query radius (r)	1%, **5%**, 10%, 20%

Table 2.4: Experimental parameters and their values (**bold** defines default values).

With respect to the spatial diversity, since the computation of network distances on the real road networks can be very costly, by default we assume that the distance between any pair of POIs is pre-computed. Nevertheless, for the sake of completeness, we also evaluated the effect of this pre-processing step in the query time.

Regarding categorical diversity, a category was assigned to each POI and we varied the number of categories from 5 to 100, where the default value is

20. The categories were evenly distributed among POIs, meaning that each category has roughly the same number of POIs. The diversity between pairs of categories was computed as follows. For each category g, we generated a point in a 2-dimensional space representing g. The diversity between categories was then given by the Euclidean distance between the points representing them. Therefore, the closer two points are, the less diverse their categories are. By doing that, the diversities also exhibit some notion of "transitivity", that is, if category A is similar to B and B is similar to C than A cannot be too dissimilar from C, which is an intuitive property to observe.

2.5.1 Comparison with *BF*

We evaluated how the baseline approach *BF* compares to our proposed approaches in terms of performance. Table 2.5 shows the results obtained for the spatial diversity[1] with all parameters set to their default values. Both of our approaches are two orders of magnitude faster than *BF*. We also note that the performance of *BF* degrades fast as the parameters increase since it generates all the possible subsets of size k within distance r from q in order to find the ones that belong to the linear skyline. As an example, for $r > 5\%$, it fails to find the skyline set in less than 10 min in most cases.

Approach	**Real Data [s]**	**Synthetic Data [s]**
BF	493.3	16.6
RRF	4.4	0.1
PG	2.8	0.2

Table 2.5: Processing time, in seconds, for all approaches using all default values.

The result above reflect the default setting where the pair-wise POIs dis-

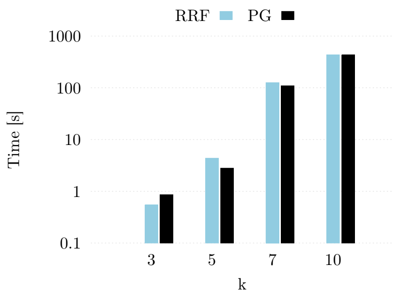
[1]Since it finds all possible sets of size k regardless of the diversity considered, *BF*'s processing time does not vary with the type of diversity; hence we omit the results for those.

tances for the real dataset were pre-computed. Our preliminary experiments indicated that there is little difference when those distances are computed at query time. This suggests that *BF*'s processing time is mostly due to the actual computation of the possible non-dominated sets of size k.

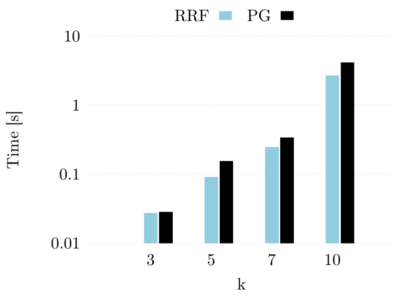
Given that *BF* is clearly a non-competitive approach, in what follows we only report the experimental results obtained for our proposed approaches *RRF* and *PG*.

2.5.2 Effect of k

As expected and shown in Figures 2.5, 2.6 and 2.7, the running time of all solutions increases with k for both real and synthetic datasets and for all three types of diversity. This is simply because more points need to be evaluated when forming a set.



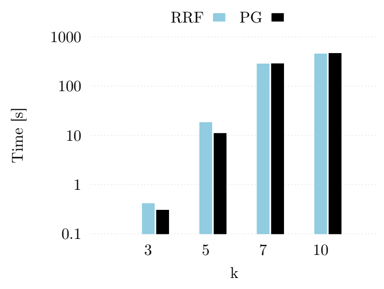
(a) Real Dataset



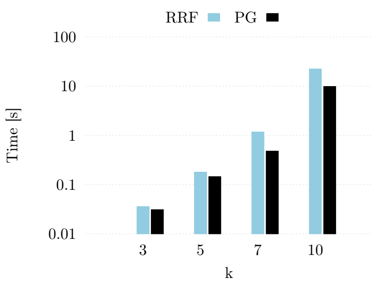
(b) Synthetic Dataset

Figure 2.5: Varying the answer set size k for spatial diversity.

We note that all solutions perform better for a synthetic dataset where the points are uniformly distributed than for the real network. Even though there are more POIs in the synthetic dataset (5000), due to the data distribution a range around a point in that dataset tends to include less points than in the real dataset. Therefore, there are less possible combinations of points to be examined. Moreover, in the real network, there are several concentrations of POIs. Although a range around a point $p \in P$ will tend to exclude the points

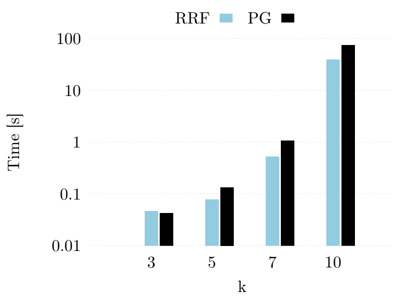


(a) Real Dataset

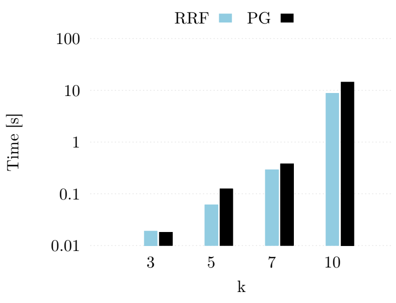


(b) Synthetic Dataset

Figure 2.6: Varying the answer set size k for angular diversity.



(a) Real Dataset



(b) Synthetic Dataset

Figure 2.7: Varying the answer set size k for categorical diversity.

closer to p, based on $minDiv$, points from other clusters will most likely not be excluded. Thus, there are potentially more candidates to be combined with p in order to form sets of size k than in a uniform distribution.

As shown in Figure 2.5b, for the spatial diversity *RRF* outperforms *PG* for all observed values for the synthetic dataset. However, this is not always the case. For the real network, *PG* tends to outperform *RRF*, as shown in Figure 2.5a. When the number of candidate points to be combined with a given point p is sufficiently large, the pruning strategy employed by *PG* is more powerful than the one used in *RRF*. *RRF* finds all the possible sets of size k with diversity higher than $minDiv$. The greater the number of candidate points, the greater the number of potential sets to be generated. On the other hand, in the *PG* approach, partial subsets that offer a greater chance to be part of the solution with maximum diversity are expanded first. Once the first solution of size k is found, the algorithm stops.

Regarding angular diversity, *PG* tends to outperform *RRF* even for the synthetic dataset, as shown in Figure 2.6b. This is because when compared to the spatial diversity, the overall diversity of a set tends to be lower since it is limited by the maximum angle that can be formed between the points in the set. With an overall lower diversity, more pairs of points will have “high” diversity and thus cannot be pruned. As also evidenced in the results obtained for spatial diversity, *PG* deals better with the cases where there are more candidate points to be combined with partial sets in order to obtain a k-sized non-dominated set.

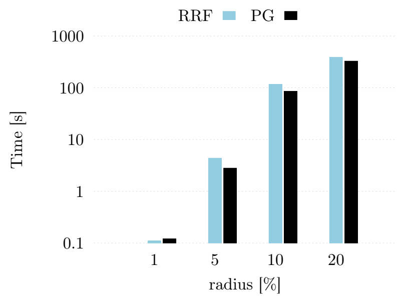
Unlike the results obtained for spatial diversity, when considering the angular diversity *RRF* was slightly faster than PG for the real dataset when $k \geq 7$. When new points are added to a set, the diversity of such set tends to decrease. For angular diversity and considering the real dataset, particularly, this diversity decreases more sharply. Therefore, the strategy of first exploring sets with high diversity, employed by *PG*, may not be very effective for high values of k since adding new points to such sets may decrease their diversity significantly and, thus, the estimate provided by *PG* is not very precise.

Figure 2.7 shows that the execution time for the categorical diversity is much shorter than when a spatial or angular diversity is considered. This is because only the closest points from each category are needed in order to find the skyline set. This experiment also shows that *RRF* outperforms *PG* for most examined values. This can be explained by the fact that *RRF* is more efficient when there is a smaller number of candidate points that are diverse enough to be combined with a partial set in order to obtain a non-dominated k-sized set, which is the case for categorical diversity.

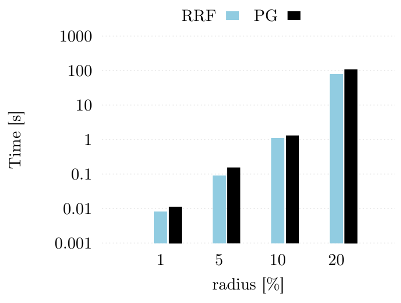
2.5.3 Effect of the radius r

Figures 2.8, 2.9 and 2.10 show that the running time of all solutions also increases with the radius. A larger radius includes more POIs and thus there is a larger number of subsets that could be part of the skyline, requiring the algorithms to examine more combinations.

We note that both of our solutions exhibit the same behavior as in the previous experiment for each type of diversity. For the spatial diversity, *PG* tends to outperform *RRF* for the real dataset, while *RRF* is faster for the synthetic dataset. Regarding the angular diversity, *PG* tends to be more efficient for both datasets, while *RRF* is more efficient when the categorical diversity is considered. We also note that when r is sufficiently small, the gains from *PG*'s pruning are not outweighed by the extra cost of computing the upper bound to the diversity of partial subsets and the costs involving the queue operations. This explains why *RRF* is faster than *PG* for $r = 1\%$ for all types of diversity.



(a) Real Dataset

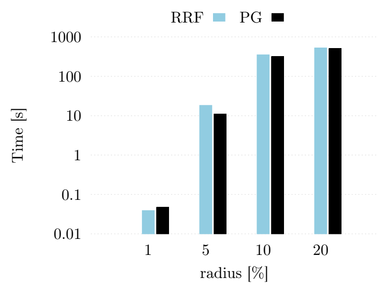


(b) Synthetic Dataset

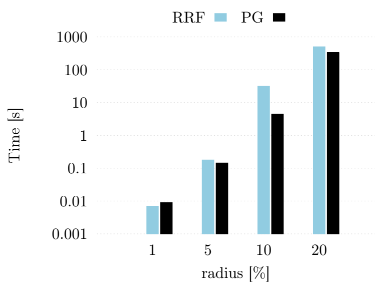
Figure 2.8: Varying the radius r for spatial diversity.

2.5.4 Effect of the datasets

Figures 2.11a, 2.12a and 2.13a show the results obtained for different road networks and for the three types of diversity. The larger the network in terms of

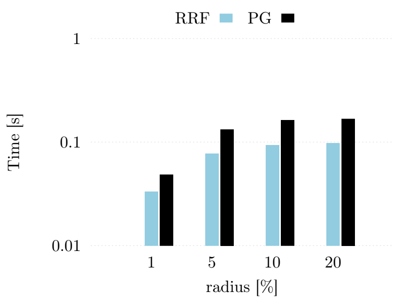


(a) Real Dataset

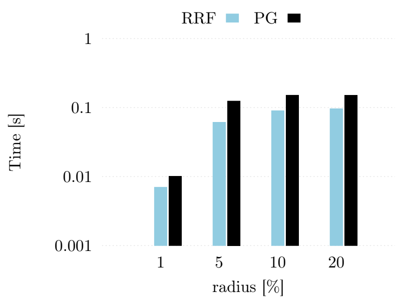


(b) Synthetic Dataset

Figure 2.9: Varying the radius r for angular diversity.



(a) Real Dataset

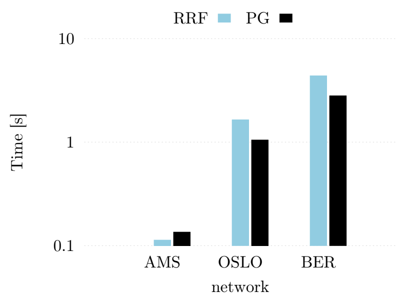


(b) Synthetic Dataset

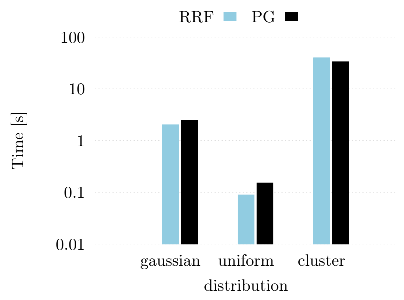
Figure 2.10: Varying the radius r for categorical diversity.

number of POIs, the greater the number of POIs that will tend to be included within a radius from the query point. Consequently, more combinations of points will need to be examined in order to determine the ones that belong to the linear skyline. This explains why all approaches perform better for the AMS network and worse for the BER network. The same reasoning can also be used to explain the results obtained when the number of points in the synthetic dataset was varied, as shown in Figures 2.11c, 2.12c and 2.13c.

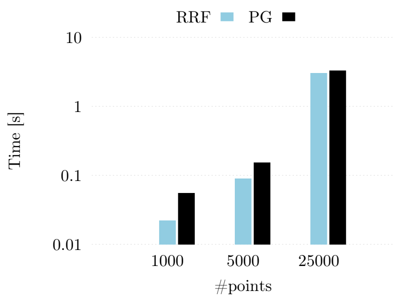
Regarding the distribution of points in the synthetic datasets, the spatial and angular diversities exhibited the same behavior, as shown in Figures 2.11b and 2.12b. More specifically, all solutions perform better for the uniform distribution. Since the data points are more scattered in the space, the neighborhood of a point tends to include less points than in the other two distributions. The lower the number of points, the faster it is to find the skyline set. In the



(a) road networks (real datasets)



(b) distribution (synthetic datasets)

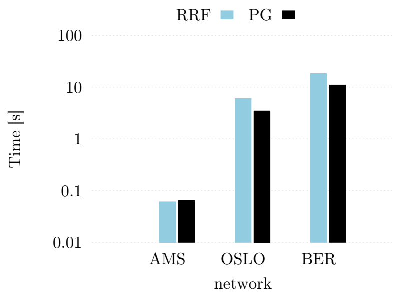


(c) size (synthetic datasets)

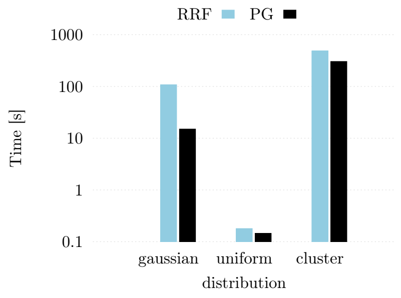
Figure 2.11: Varying the datasets for spatial diversity.

clustered distribution, there is more than one concentration of points, which means that, although a range around a point will probably eliminate points in the same cluster, it will most likely not include points from other clusters. Therefore, since the diversity between points in different clusters will tend to be high, the number of subsets that can potentially be a non-dominated solution will also increase, requiring more combinations of points to be generated and checked. This is also the reason why *PG* outperforms *RRF* for the clustered distribution. Since a Gaussian distribution has only one concentration of points, it does not suffer from this drawback. This explains why all approaches perform better for a Gaussian distribution when compared to a clustered distribution.

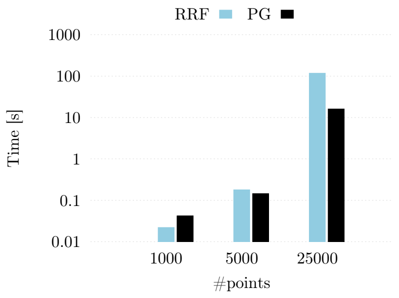
Varying the distribution of the points does not have the same effect on the categorical diversity as on the spatial and angular diversities. The categori-



(a) road networks (real datasets)

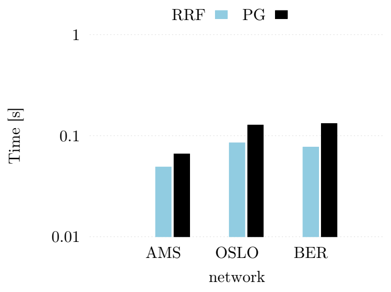


(b) distribution (synthetic datasets)

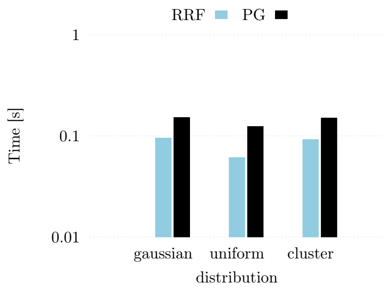


(c) size (synthetic datasets)

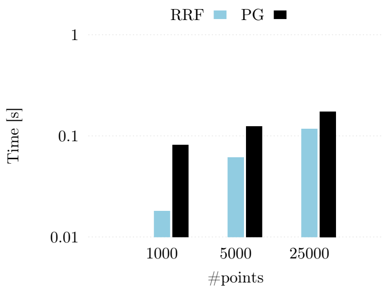
Figure 2.12: Varying the datasets for angular diversity.



(a) road networks (real datasets)



(b) distribution (synthetic datasets)



(c) size (synthetic datasets)

Figure 2.13: Varying the datasets for categorical diversity.

cal diversity is not as sensitive to this parameter as the other diversities, as shown in Figure 2.13b. This can be explained by the fact that, in the synthetic dataset, the position of points has no relation with the diversity between them. Consequently, the distribution does not influence the number of candidates (points with high diversity) to be combined with partial sets. The only parameter that varies with the distribution is the number of points selected within the given radius. This explains why our approaches are slightly faster for the uniform distribution.

2.5.5 Effect of the pre-processing (for spatial diversity)

In this experiment we evaluated how pre-computing the network distances between pairs of POIs influences the processing time. The results are shown in Figure 2.14. Recall that such pre-processing of pair-wise POI distance is only used when computing spatial diversities. For the other types of diversity such distance is irrelevant. The pre-processing time for the AMS, OSLO and BER networks were, respectively, 40 sec, 2.2 min and 14.5 min. For the *RRF* and *PG* approaches the computation of the pair-wise POI distances corresponds to 63% and 71% of the total time, respectively, confirming that the pre-processing of distance is worthwhile.

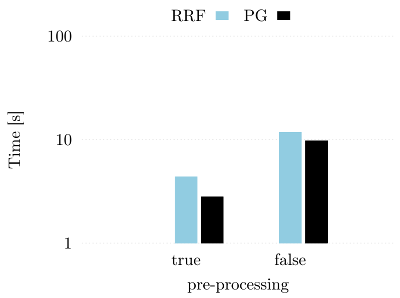
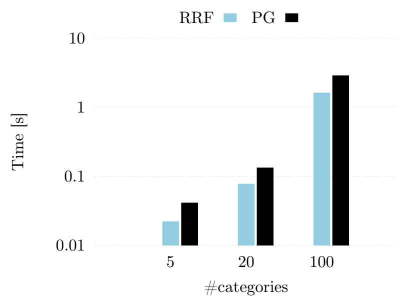


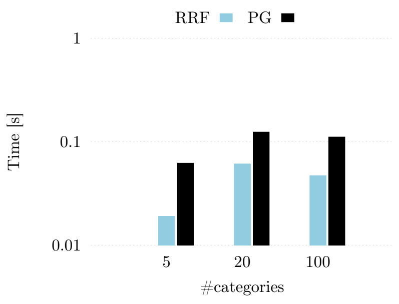
Figure 2.14: Effect of pre-processing the network distances.

2.5.6 Effect of the number of categories (for categorical diversity)

In this experiment we evaluated how our proposed approaches behave w.r.t. categorical diversity when varying the number of categories from 5 to 100. As shown in Figure 2.15, the running time of both approaches tends to increase with the number of categories. Since for categorical diversity only the closest points from each category are needed, the higher the number of categories, the higher the number of points that need to be examined in order to find the skyline set. As shown in Figure 2.15b, although the processing time tends to increase with the number of categories, our approaches are faster for 100 categories than for 20. When more categories are considered, more diverse solutions are found faster, which increases the pruning power of both approaches, since sets are pruned based on the highest diversity found so far. For 100 categories, this greater pruning power outweighs the fact that more points need to be examined.



(a) Real dataset



(b) Synthetic dataset

Figure 2.15: Varying the number of categories.

2.5.7 GNE

As mentioned in Section 2.2, *GNE* provides an approximate result for a given trade-off between similarity and diversity whereas we obtain the optimal one for *all* linear trade-offs. Nonetheless, for the sake of completeness we evaluated how sub-optimal *GNE*'s results are as well as how much faster they are obtained. We fixed λ, the trade-off between closeness and diversity given as input to the *GNE* algorithm, at 0.5. Since *GNE* requires both criteria to be on the same scale, we also normalized the diversity criterion by dividing the corresponding values by the maximum possible value for the given diversity function. For instance, for div_{spa}, the maximum value is $2 \times r$, since it is the maximum possible distance between any two points. Similarly, the maximum value for div_{ang} is 180°. Since we assume that $0 \leq div_{cat} \leq 1$, it does not need to be normalized.

Table 2.6 shows *GNE*'s computation time as well as the degree of sub-optimality (denoted as "gap") obtained for the three types of diversity considered in this chapter when all parameters are set to their default values. Regarding spatial diversity in the real dataset, we also evaluated how *GNE* performs when the network distances are computed at query time and when they are pre-computed. Note that the latter case is not applicable to the synthetic dataset since no pre-computation is done.

	Real Dataset		**Synthetic Dataset**	
Diversity	**Time [s]**	**Gap**	**Time [s]**	**Gap**
Spatial (with pre-processing)	0.98	18.43%	N/A	N/A
Spatial (w/o pre-processing)	12	17.75%	0.64	22.8%
Angular	0.45	6.09%	0.09	2.84%
Categorical	0.26	8.33%	0.06	7.85%

Table 2.6: GNE's results for the real and synthetic datasets.

With respect to the real dataset, *GNE*'s performance degrades significantly when the network distances are computed at query time. For each p_i within the radius r, *GNE* finds the $(k-1)$ more diverse points w.r.t. p_i, which requires multiple network distance computations. We note that in this case our proposed approaches are able to find the whole linear skyline faster than

GNE finds a single approximate solution to a given trade-off. On the other hand, when the distances are pre-computed, *GNE* performs better than our approaches. More specifically, the time that *PG* takes to find the skyline set is equivalent to running *GNE* around 3 times. For angular diversity, *GNE* is significantly faster than *PG*, while both of our approaches are faster than *GNE* for categorical diversity. Regarding the synthetic dataset, when the parameters are set to their default values, *GNE* is up to twice as fast as our proposed approaches.

It is important to stress that the results above are for *GNE* to provide an answer for *one* trade-off pair. It is not unreasonable to think that a user may want to try several combinations. Our approaches do not require the user to choose any trade-off as it finds optimal solutions to all linear ones. Moreover, even if the user were to know which trade-off he/she would like, *GNE* is not guaranteed to find optimal solutions.

In general, the quality of the results produced by *GNE* are worse for spatial diversity. When compared to the other two types of diversity, in spatial diversity there are more points that have the potential to lead to more diverse sets. This is evidenced in Section 2.5.7 where we show that the number of non-dominated sets found for spatial diversity is much larger than for the other diversities. Therefore, the number of points that offer a great chance to be diverse w.r.t. the previously selected ones is larger, which increases the chances of *GNE* making a poor decision. We also note that for the real dataset the results should ideally be the same regardless of whether the distances are pre-computed or not, but as one can see in Table 2.6, that is not case for *GNE*. This is due to the randomization that takes place within that algorithm as discussed in Section 2.2.

In angular diversity, the diversity of a set is restricted by the maximum angle that can be formed between k points. This means that more pairs of points will tend to have a diversity more similar to that of the pairs in the optimal solution, increasing the chance of GNE producing solutions closer to the optimal ones. Similarly, in categorical diversity the number of points that could be diverse w.r.t. previously selected points is limited by the the number of categories. Therefore, *GNE* is less prone to errors since there are less points to choose from when this type of diversity is considered.

Conventional Skyline vs Linear Skyline

In order to support the claim that a linear skyline set is significantly smaller than its corresponding conventional skyline set, we performed an experiment to compare the size of both sets for the three diversities considered in this chapter. Table 2.7 shows the results of this experiment. When spatial diversity is considered, more conventionally non-dominated solutions are found because there are more points that potentially lead to more diverse sets. In this case, the linear skyline set is around 90% smaller than the conventional skyline for the real dataset and 82% smaller for the synthetic dataset.

	Real Dataset		**Synthetic Dataset**	
Diversity	**Skyline**	**Linear Skyline**	**Skyline**	**Linear Skyline**
Spatial	47	4	28	5
Angular	15	5	11	5
Categorical	6	4	6	3

Table 2.7: Conventional Skyline vs Linear Skyline for the real and synthetic datasets.

In angular diversity the diversity of sets slowly increases as more points are examined, however, not as many points lead to more diverse sets as when the spatial diversity is considered. In categorical diversity the diversity of sets tends to increase faster than when the other two types of diversity are considered. Consequently, less points have diversity high enough w.r.t. the previously examined points in order to obtain a new non-dominated set. This explains why fewer conventionally non-dominated sets are found for categorical diversity. Nevertheless, for the real dataset, the linear skyline showed to be around 67% and 33% smaller for angular and categorical diversities, respectively. While for the synthetic dataset, the linear skyline is around 55% smaller for the angular diversity and 50% for the categorical diversity. These figures clearly support our reasoning that by using linear skylines it would be much simpler for the user to choose a solution.

We also note that the numbers shown in Table 2.7 for the conventional skyline tend to be smaller for the synthetic dataset when compared to the default case of Berlin's network because when a uniform distribution is considered in the Euclidean space, less points tend to be included within the same radius, leading to less non-dominated sets.

2.6 Conclusion

In this chapter we have addressed the $(k, r)-$DNN query which is a novel variation of k-NN queries. This query returns sets containing k elements that are as close as possible to the query point, while, at the same time, being as diverse as possible. We assume that the user specifies the maximum distance he/she is willing to travel in order to visit a POI, i.e., the query radius r. We have considered three different notions of diversity, namely spatial, angular and categorical. Previously proposed solutions are approximate and require the user to determine a specific weight for each type of diversity. Our proposed approaches based on linear skylines find *all* results that are *optimal* under *any* arbitrary linear combination of the two competing criteria, namely closeness and diversity.

We proposed two solutions to (k, r)-DNN queries using the notion of skylines. *Recursive Range Filtering (RRF)* strives to reduce the number of combinations generated by recursively combining a partial set with candidate points that are diverse enough to be part of a non-dominated solution. *Pair Graph (PG)* works by pruning subsets that contain pairs of points that have low diversity and could not be together in a non-dominated solution. Experiments varying a number of parameters showed that (1) both *RRF* and *PG* are orders of magnitude faster than a straightforward solution, and (2) *PG* outperforms *RRF* when there is a greater number of candidate points that can be combined with previously selected points, and *RRF* is more efficient otherwise. Also, we confirmed that our approaches are more effective than *GNE* by virtue of guaranteeing optimal results, as well as more generic since they do not require the user to set any particular trade-off.

Chapter 3

In-Route Task Selection in Spatial Crowdsourcing

3.1 Introduction

Crowdsourcing is a relatively new computing paradigm which relies on the contributions of a large number of workers to accomplish tasks, such as image tagging and language translation. The increasing popularity of mobile computing led to a shift from traditional web-based crowdsourcing to spatial crowdsourcing [49].

Spatial crowdsourcing consists of location-specific tasks, which require people to physically be at specific locations to complete them. Examples of these tasks include taking pictures, answering questions about a certain location in real time or perform physical chores. Tasks are assigned to suitable workers based on one or more objectives, such as maximizing the number of assigned tasks, maximizing a given matching score, minimizing the total amount of reward paid out by task requesters or maximizing the net reward earned by workers after deducting traveling costs. In the cases where a worker is assigned to multiple tasks, the travel cost between tasks has typically not been taken into account, e.g., [32], [48], [50]. However, that cost may directly affect the number of tasks the worker could be able to perform, depending, for example, on the worker's time/distance budget. In this chapter we consider the particular problem of finding a task schedule for a worker, i.e., a feasible task sequence, a problem that has also been considered in [18], [20].

Differently from [18], [20], we consider that a worker is traveling a given path between two given locations and is willing to possibly perform tasks on the way to the destination, likely deviating from the original path, as long as the total time spent by him/her does not exceed a given budget. We refer to this problem as *In-Route Task Selection* (IRTS). On the one hand, it makes sense to consider performing tasks that minimize the deviation from the original path to the location of the tasks plus the time taken for completing them. On the other hand, considering that each task is associated with a reward, it also makes sense to maximize the total reward received for performing tasks. Clearly, intermediary solutions exist as well, i.e., the worker may have his/her own reasons for prioritizing (or alternatively penalizing) the deviation vs reward trade-off in different ways. In fact, the novelty and non-trivial complexity of the IRTS problem, in the context of spatial crowdsourcing, comes from the fact that there are two *competing criteria* to be optimized at the same time: deviation from an intended path and total reward collected.

In this chapter we address two different variants of the IRTS problem. In the first one, named IRTS-SP, we assume that the worker only specifies the starting point s and the destination d and that he/she is willing to consider other paths that deviate as little as possible, in terms of its cost, from the cost of the shortest path connecting s and d. In the second variant, named IRTS-PP, we assume that the worker has a *preferred path* from s to d and is willing to consider other paths as long as he/she travels *along that one preferred path* for as long as possible. A typical example for the former is when workers rely on GPS-based routers, whereas the latter is practically relevant in cases where the worker prefers a path for non-objective reasons, e.g., availability of public transit, bicycle-friendliness or perceived safety.

The IRTS-SP and IRTS-PP problems can be informally formulated as follows. (Note that we assume, for both problems, that the worker has a total temporal budget b, including travel time and completion time for tasks, which cannot be exceeded.) In the IRTS-SP problem, the worker specifies a starting point s and destination d, and the goal is to maximize the total reward collected by the worker by performing tasks while also minimizing the difference between the time cost of the new path connecting s to d, including the time needed to complete tasks on the way, and the travel time of the shortest path connecting them. In the context of IRTS-PP, we consider the worker's preferred path PP, and also aim at maximizing the total reward earned by the worker but now minimizing the total deviation from PP. The fundamental difference between those two variants, which demands different approaches, is how "detour" is defined. For IRTS-SP, there is no concern about the path itself, only the path's cost matters, whereas for IRTS-PP the worker wants to stick to his/her preferred path as much as possible.

In order to illustrate the IRTS-SP and the IRTS-PP problems, consider the simple scenario shown in Figure 3.1, where there are three available tasks $T = \{t_1, t_2, t_3\}$ associated with their corresponding rewards, \$3, \$4 and \$5, and time for completion, 2, 4, and 3, respectively. Also, assume that the worker's time budget is $b = 32$.

Let us first consider the IRTS-SP problem and assume that the worker's starting location is s and d is his/her destination. The shortest path connecting

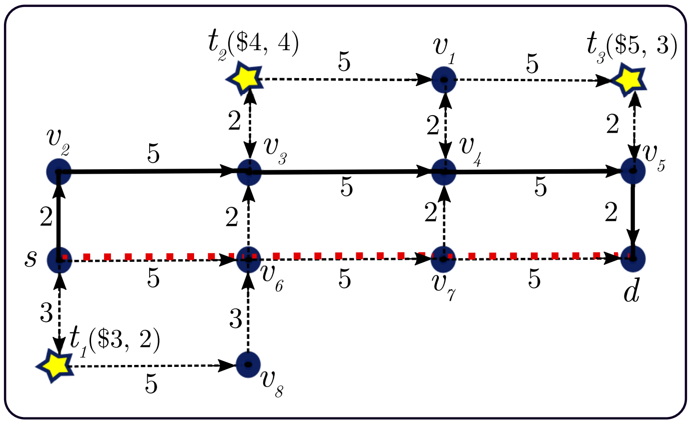


Figure 3.1: Preferred path PP from s to d (in bold), the corresponding shortest path (dotted red line), and tasks t_1, t_2 and t_3 (denoted by star vertices) with their corresponding rewards and time for completion. Edge costs denote the time needed to traverse them.

s to d is $SP = \langle s, v_6, v_7, d \rangle$, represented by a red dotted line, with cost equal to 15. The path that minimizes the detour incurred from traveling to at least one task location[1] and completing such task(s) is $SP_1 = \langle s, t_1, v_8, v_6, v_7, d \rangle$ with reward \$3 and a total cost of 23, hence a detour of 8 units. Recall that in the context of IRTS-SP, the "detour" is a measure of how much longer the path is, including the time needed to perform the selected tasks (in this case t_1 costing 2 units) w.r.t. the actual shortest path. On the other hand, if the user wants to maximize his/her reward, the path $SP_2 = \langle s, v_6, v_3, t_2, v_1, t_3, v_5, d \rangle$ (with reward \$9 and detour 15) would be the best option. Let us now consider other alternative paths. $SP_3 = \langle s, v_6, v_7, v_4, v_1, t_3, v_5, d \rangle$ yields a total detour of 11 and reward of \$5, while $SP_4 = \langle s, v_6, v_3, t_2, v_1, v_4, v_5, d \rangle$ yields a total detour of 12 and reward of \$4. Finally, $SP_5 = \langle s, t_1, v_8, v_6, v_3, t_2, v_1, t_3, v_5, d \rangle$ yields a total detour of 21 and reward of \$12. However, since its cost is 36, which is greater than b, SP_5 is not a feasible option.

Let us now consider the IRTS-PP problem. The path in bold represents the worker's preferred path $PP = \langle s, v_2, v_3, v_4, v_5, d \rangle$. On the one hand, if

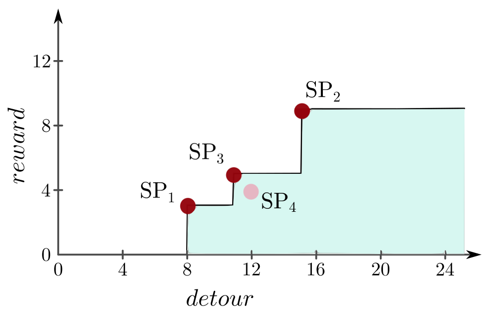
[1]A solution that does not perform any task is of no practical interest.

the worker wants to minimize the detour from PP incurred from traveling to at least one task location and completing such task(s), the best path is $PP_1 = \langle s, v_2, v_3, v_4, v_5, t_3, v_5, d\rangle$ with reward \$5 and detour 7, because the edge $\langle v_5, t_3\rangle$ that does not belong to the preferred path needed to be traversed twice and t_3 takes 3 units of time to be performed. On the other hand, if the user wants to maximize his/her reward, the path $PP_2 = \langle s, v_2, v_3, t_2, v_1, t_3, v_5, d\rangle$ (with reward \$9 and detour 21) would be the best option. Note that both paths are feasible since they satisfy the maximum budget b; the total time of PP_1 is 26, while PP_2 takes 30 units of time. Clearly there are other alternative paths. $PP_3 = \langle s, v_2, v_3, t_2, v_3, v_4, v_5, d\rangle$ yields a total detour of 8 and reward of \$4, $PP_4 = \langle s, t_1, s, v_2, v_3, v_4, v_5, d\rangle$ yields a total detour of 8 and reward of \$3. $PP_5 = \langle s, v_2, v_3, t_2, v_3, v_4, v_5, t_3, v_5, d\rangle$ yields a total detour of 11 and reward of \$9, however, since PP_5's total cost is 34, which is greater than b, PP_5 is not a feasible option.

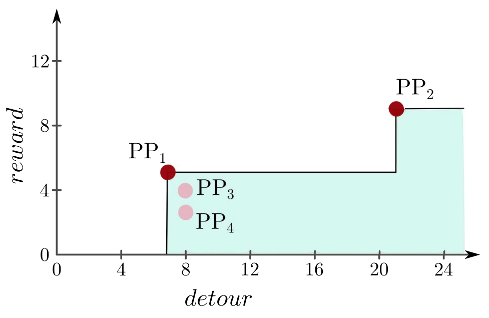
As shown above, a single route does not typically optimize both criteria, i.e., minimize detour and maximize reward at the same time; in the IRTS-SP (IRTS-PP) case, SP_1 (PP_1) minimizes detour and SP_2 (PP_2) maximizes reward. A simplistic way to address this problem is to assign weights to both criteria and optimize the resulting combination. However, such an approach would depend primarily on the worker's preferences, which, besides possibly not being obvious or clear from the outset, adds an extra parameter to the problem. Moreover, we believe that not requiring workers to specify beforehand their preferences is of practical relevance as it empowers them to consider all interesting alternatives by themselves. In this context, a more principled way to deal with the IRTS problem is to determine *all* results that are optimal under *any* arbitrary combination of the two criteria. Fortunately, that can be achieved by using the notion of *skyline queries* [5]. In generic terms, the result set of a skyline query contains objects which are not dominated by any other one. An object o_i is dominated by another object o_j if, for each criterion, o_i is at most as good as o_j, and, for at least one criterion, o_j is strictly better than o_i.

Figure 3.2 illustrates graphically the concept of skyline queries. It shows the corresponding detours and rewards of the candidate paths considered above (obtained from Figure 3.1) for both variants IRTS-SP (Figure 3.2a) and IRTS-PP (Figure 3.2b). The shaded area contains the dominated paths. Let us first consider the IRTS-SP variant. Paths SP_1 and SP_2 are not dominated since they are the ones that minimize the incurred detour and maximize the reward received by the worker, respectively. Similarly, path SP_3 is also non-dominated since there is no other path with shorter detour *and* higher reward than its own. On the other hand, path SP_4 is dominated by path SP_3. Therefore, the skyline set for this case contains paths SP_1, SP_2 and SP_3. Let us now consider the IRTS-PP variant. Path PP_2 is non-dominated since it maximizes the reward the worker could receive considering the specified budget $b = 32$. On the other hand, paths PP_3 and PP_4 are dominated since they yield a greater detour than that of PP_1, but offer a smaller reward. Note that PP_1 is non-dominated since there is no other path with smaller detour than its own. Therefore, paths PP_1 and PP_2 are not dominated and thus are equally interesting and should be offered as alternatives to the worker, who can decide by him/herself how to prioritize the trade-off between deviation and reward.

Our main objective in this chapter is to devise a skyline-based approach in order to solve both variants of the IRTS problem. A straightforward application of the skyline query requires the generation of all possible paths in order to compute the skyline set. Instead, we exploit properties from each of the IRTS problems in order to efficiently and incrementally generate, as well as



(a) IRTS-SP variant.



(b) IRTS-PP variant.

Figure 3.2: Skylines for the example illustrated in Figure 3.1 for variants IRTS-SP (left) and IRTS-PP (right).

prune paths, thus reducing the search space substantially. In this context, the main contributions we offer are two-fold. First, we define the IRTS problem and two variants thereof (IRTS-SP and IRTS-PP). Those are new extensions of the task scheduling problem in spatial crowdsourcing which empower the worker w.r.t. trade-offs between competing criteria. Second, after showing that both IRTS-SP and IRTS-PP problems are NP-hard, we present, for each one of them, an exact as well as a few heuristic approaches that can be used to solve city-scale instance problems efficiently.

The remainder of this chapter is structured as follows. In Section 3.2 we present relevant related works and contrast them to ours. We present the formal definition of the IRTS problem, showing that both IRTS-SP and IRTS-PP are NP-hard, in Section 3.3. Our proposed exact and heuristic algorithms, which form the core of our contribution, are presented in Section 3.4, followed by their experimental evaluation using real city-scale datasets in Section 3.5. Finally, Section 3.6 presents a summary of our findings.

3.2 Related Work

The literature in spatial crowdsourcing presents many different ways to assign tasks to workers, for instance, maximizing the number of assigned tasks [32], [48], maximizing a given matching score [10], [45], [50], [60] or minimizing the total amount of reward paid out by requesters while maximizing the number of assignments [17].

Kazemi and Sahabi [32] study the maximum task assignment (MTA) problem in spatial crowdsourcing which aims at maximizing the overall number of assigned tasks. It considers that a worker only accepts tasks within his/her spatial region and is willing to perform up to a predetermined number of tasks. To et al. [48] introduce a framework for crowdsourcing hyper-local information. A task can only be answered by workers who are already within a radius r from the task location at a time when the task is valid. The goal is to maximize task assignment given a budget, which is the maximum number of workers that can be selected.

Tong et al. [50] study the Global Online Micro-task Allocation in spatial crowdsourcing (GOMA) problem, which aims at maximizing a total matching utility. The utility for a task-worker pair (t, w) is given by the payoff of task t times the success ratio of w completing tasks. Song et al. [45] propose the trichromatic online matching (TOM) in real-time spatial crowdsourcing problem, which aims at matching three types of objects, namely tasks, workers and workplaces. The goal is to maximize a total utility score representing the satisfaction of the matching involving the corresponding objects. Zheng and Chen [60] study the Task Assignment with Mutual Benefit Awareness (TAMBA) problem. TAMBA aims at maximizing the mutual benefit of the workers and tasks, which is measured in terms of the expected answer quality for a worker w and task t, given by the probability of w's acceptance of t

multiplied by the expected rating of w's completing t.

Within the context of reward-based task assignment, Dang and Cao [17] propose the Maximum Task Minimum Cost Assignment (MTMCA) problem. MTMCA considers that workers have multiple skills and that each task has a type. The goal is to maximize the number of assignments and subsequently minimize the total amount of money spent by the requesters, assuming that the price of tasks is attributed by workers. The Multi-Skill Spatial Crowdsourcing (MS-SC) problem is presented in [9]. It aims at assigning multi-skilled workers to complex spatial tasks such that skills between workers and tasks match each other, and workers benefits are maximized. Each task is associated with a budget, which represents the maximum amount the requester is willing to pay for that task. The workers' rewards are given by the sum of the remaining budget of the completed tasks after deducting traveling costs.

Differently from the works above, [18], [20] deal with assigning a task sequence to workers and, thus, take the travel cost between tasks into account, similarly to what we do. In [18], the authors focus on maximizing the number of tasks performed by a single worker assuming that the worker is willing to perform up to a predefined maximum number of tasks and that he/she must arrive at the task location before its deadline. An extension of that problem is proposed in [20], and it aims at maximizing the overall number of tasks performed by multiple workers considering that each task is assigned to at most one worker. As in [18], we focus on the scenario where workers self-select the tasks they want to perform from a list of published tasks, termed Worker Selected Tasks (WST) [32] mode. However, differently from [18], [20], we assume that a worker is willing to perform tasks while traveling from a given starting point to a destination but likely without deviating too much from the original path (or travel time thereof) he/she was going to take. Additionally, we consider that each task is associated with a reward and that the worker also wants to maximize the total reward received for performing tasks, a notion not considered in either [18] or [20].

All of the works above focus on either optimizing a single criterion or on also optimizing a secondary criterion, which serves as a tie-breaker. Cheng et al. [10] study the reliable diversity-based spatial crowdsourcing (RDB-SC) problem, which assigns workers to tasks such that tasks can be accomplished with high reliability and spatial/temporal diversity. Thus, there are two criteria to be optimized simultaneously: reliability and diversity. However, while still relying on the notion of dominance, they do not find a skyline set, instead they select the *one* solution that dominates the most solutions as the best one. We, on the other hand, aim at providing the worker with *all* optimal choices by returning the skyline set of solutions.

The IRTS problem can also be seen as an interesting combination of two seemingly unrelated problems: In-Route Nearest Neighbor queries and the Orienteering Problem. The problem of searching for nearest neighbors with respect to a given (preferred) path has been previously defined as the In-Route Nearest-Neighbor (IRNN) query [44]. Within the context of multiple competing criteria in IRNN queries, [2] focuses on the trade-off yielded by minimizing the detour incurred for visiting a *single* point of interest (POI) and also minimizing the total cost of the path at the same time. On the other hand, [28] aims at minimizing the cost for reaching a POI, as opposed to the total cost of the path, and the detour incurred. In the Orienteering Problem (OP) [23], it is given a graph $G(V, E)$, where each vertex $v \in V$ is associated with a positive score, and a budget b. OP aims at finding the route from a given starting point s that maximizes the total score while the total travel cost does not exceed b. The main differences between IRTS and IRNN and OP are the following. IRNN considers deviating towards one single POI, while IRTS considers multiple tasks (which can be seen as POIs). In fact, IRTS can be seen as a generalization of the IRNN problem. The OP problem, on the other hand, does not consider the notion of trade-off between travel cost and rewards at all, and in this respect, IRTS can be considered a non-trivial extension to OP.

3.3 Preliminaries

We assume that the worker's movement is constrained by an underlying road network, which is modeled as a directed graph $G(V, E, C)$, where V is a set of vertices that represent the road intersections and end-points, E is the set of edges modelling all road segments and C indicates the costs of edges in E. In our case, the cost of traversing an edge connecting vertices v_i and v_j is given by the time it takes to traverse the corresponding road network segment that connects those vertices and is denoted by $c(v_i, v_j)$.

We define a path $P_i = \langle v_i^1, v_i^2, ..., v_i^n \rangle$ in G as a sequence of vertices such that any two consecutive vertices v_i^j and v_i^{j+1}, for $1 \leq j < n$, are directly connected by an edge $(v_i^j, v_i^{j+1}) \in E$. The shortest path from the worker's starting point s to his/her destination d is denoted by SP. Likewise, the *preferred path* of a worker is denoted by PP.

A worker w is an individual[2] who is willing to perform tasks in exchange for rewards while traveling from his/her origin to the destination. We assume that the worker has a budget b which represents the maximum time he/she wishes to spend, including the time required to perform the selected tasks and the total travel time.

[2]This is just a practical assumption, nothing in our study prevents considering the worker to be a device, e.g., a robot or an autonomous vehicle.

We also assume that all tasks are located on an edge of the network. If a given task t is not placed on an existing vertex $v \in V$, we replace that edge, say (v_j, v_l), in G with two new edges (v_j, t) and (t, v_l), adjusting the costs of the affected edges accordingly. Note that this implies that some of the vertices in the graph are now tasks rather than actual road intersections or the like. Thus, we further assume that every vertex v has a non-negative reward $\rho(v)$ and a time for completion $\delta(v)$ associated to it, where $\rho(v) = 0$ and $\delta(v) = 0$ if v does not represent a task.

It is important to note that some vertices traversed in a path may represent tasks which are *not* completed, and therefore their cost and reward should not be considered as part of the path's overall cost and reward. To address that, we add a binary variable, $\pi(v_i^j)$, to indicate whether a task at a vertex $v_i^j \in P_i$ was completed or not, that is:

$$\pi(v_i^j) = \begin{cases} 1 & \text{iff } v_i^j \text{ is a vertex representing a task } \textit{and} \text{ is completed} \\ 0 & \text{otherwise} \end{cases}$$

The reward of a path and travel and detour costs can now be defined as follows:

Definition 3.3.1 (Reward of a path)**.** Given a path P_i in G, its total reward is given by the sum of the rewards of the completed vertices (tasks) in it (recall that vertices which are not tasks or are tasks that have not been completed do not contribute to the path's reward), i.e.,

$$R(P_i) = \sum_{v_i^j \in P_i} \rho(v_i^j) \times \pi(v_i^j)$$

Definition 3.3.2 (Travel Cost)**.** Given a path P_i in G, its travel cost is given by the sum of the costs of the edges in it, i.e.,

$$TC(P_i) = \sum_{j=1}^{n-1} c(v_i^j, v_i^{j+1}).$$

While the definitions of reward of a path and travel cost apply to both variants of the IRTS problem, each one has its own detour function. More specifically, IRTS-SP considers the extra traveling cost w.r.t. SP's cost, whereas for the IRTS-PP variant, the detour is calculated w.r.t. the actual edge deviation from a given preferred path PP; these notions are formalized next.

Definition 3.3.3 (Displacement Detour and Total Detour w.r.t. the Shortest Path)**.** Given a path P_i connecting s to d, the displacement detour of P_i w.r.t. the shortest path SP is defined as the difference between the travel costs of P_i and SP:

$$DDSP(P_i, SP) = TC(P_i) - TC(SP).$$

Similarly, the total detour of P_i w.r.t. SP is given by the sum of its displacement detour and the total time required for performing the completed tasks in P_i, i.e.:

$$TDSP(P_i, SP) = DDSP(P_i, SP) + \sum_{v_i^j \in P_i} \delta(v_i^j) \times \pi(v_i^j).$$

Definition 3.3.4 (Displacement Detour and Total Detour w.r.t. a Preferred Path)**.** Given a path P_i and a preferred path PP connecting s to d, the displacement detour of P_i w.r.t. PP is defined as the sum of the costs of the edges in P_i that do not belong to PP. That is:

$$DDPP(P_i, PP) = \sum_{j=1}^{n-1} d(v_i^j, v_i^{j+1}, PP),$$

where $d(v_i^j, v_i^{j+1}, PP) = c(v_i^j, v_i^{j+1})$ if $(v_i^j, v_i^{j+1}) \not\subset PP$ or null otherwise. The total detour of P_i w.r.t. PP is equal to the sum of its displacement detour and the total time required for performing the completed tasks in P_i, i.e.:

$$TDPP(P_i, PP) = DDPP(P_i, PP) + \sum_{v_i^j \in P_i} \delta(v_i^j) \times \pi(v_i^j).$$

Finally, we present the definition of feasible path, i.e., a path that can be completed by the worker within the given budget b.

Definition 3.3.5 (Feasible Path)**.** A path P_i is said to be feasible, i.e., of potential interest to the worker, if its travel time plus the time taken for performing the tasks in it is not greater than the given budget b. More formally, a path is feasible if the following condition is satisfied:

$$TC(P_i) + \sum_{v_i^j \in P_i} \delta(v_i^j) \times \pi(v_i^j) \leq b.$$

As mentioned earlier, IRTS aims at providing the user with a set of optimal alternative feasible paths for different trade-offs between detour and reward. In order to do so, we rely on the notion of *skyline queries*, which was first introduced in [5]. Given a d-dimensional data set, a skyline query returns the points that are not dominated by any other point. In the context of the IRTS problem, a path P_i is not dominated if there is no other path P_j with smaller detour *and* higher reward than P_i. A powerful aspect of skyline queries is that the user does not need to determine beforehand weights for detour and reward. The skyline is a set of equally interesting solutions in the sense that they are all non-dominated, for arbitrary weights. The skyline set found for the IRTS-SP problem can be formally defined as follows.

Definition 3.3.6 (Skyline)**.** Let $\mathcal{P}$ be a set of paths in a two-dimensional cost space. A path $P_m \in \mathcal{P}$ dominates another path $P_n \in \mathcal{P}$, denoted as $P_m \prec P_n$, if

$$TDSP(P_m, SP) < TDSP(P_n, SP) \quad \wedge \quad R(P_m) \geq R(P_n) \quad \vee$$

$$TDSP(P_m, SP) \leq TDSP(P_n, SP) \quad \wedge \quad R(P_m) > R(P_n)$$

That is, P_m is better in one criterion and at least as good as P_n in the other one. The set of non-dominated paths, i.e. $\{P_m \in \mathcal{P} \mid \nexists P_n \in \mathcal{P} : P_n \prec P_m\}$, is called a skyline.

We note that the same definition also applies to the IRTS-PP variant by replacing the detour cost function $TDSP(P_i, SP)$ with IRTS-PP's own detour cost function, $TDPP(P_i, PP)$, defined above.

The two variants of the IRTS problem considered in this chapter can now be formally defined as follows. (For ease of reference, Table 3.1 summarizes the notation used throughout this chapter.)

Problem Definition (IRTS-SP)**.** *Given a worker w with his/her starting location s, destination d, budget b, and a set of available tasks T (embedded in some vertices of the network G), the IRTS-SP problem aims at finding the set of all non-dominated paths, w.r.t. $TDSP(\cdot, SP)$ and $R(\cdot)$, from s to d, that contains at least one task $t_i \in T$ and whose total time does not exceed b.*

Problem Definition (IRTS-PP)**.** *Given a worker w with his/her corresponding preferred path PP between vertices s and d, a budget b, and a set of available tasks T (embedded in some vertices of the network G), the IRTS-PP problem aims at finding the set of all non-dominated paths, w.r.t. $TDPP(\cdot, PP)$ and $R(\cdot)$, from s to d, that contains at least one task $t_i \in T$ and whose total time does not exceed b.*

Notation	*Meaning*
SP and PP	The shortest path and the worker's preferred path, respectively, between s and d, i.e., the worker's origin and destination
$P_i = \langle v_i^1, v_i^2, ..., v_i^n \rangle$	A path P_i
v_i^j	The j-th vertex in P_i
$c(v_i, v_j)$	Cost of the edge connecting v_i to v_j
$d_e(v_i, v_j)$	Euclidean distance between vertices v_i and v_j
$\pi(v_i^j)$	A binary flag indicating whether a task at vertex v_i^j is completed or not
$TC(P_i)$	Travel cost of path P_i
$DDSP(P_i, SP)$	Displacement detour of path P_i w.r.t. the shortest path between v_i^1 and v_i^n
$TDSP(P_i, SP)$	Total detour of path P_i w.r.t. the shortest path between v_i^1 and v_i^n
$DDPP(P_i, PP)$	Displacement detour of path P_i w.r.t. PP
$TDPP(P_i, PP)$	Total detour of path P_i w.r.t. PP
$\rho(v_j)$	Reward of a vertex (task) v_i
$R(P_i)$	Reward of path P_i
$\delta(v_j)$	Time for completion of a vertex (task) v_i
$P_i \prec P_j$	P_j is dominated by P_i
$P_{sp}^{v_i,v_j}$	Shortest path between vertices v_i and v_j (Sec. 3.4)
$P_{pp}^{v_i,v_j}$	Path between v_i and v_j which minimizes the detour w.r.t. PP (Sec. 3.4)
$P_{sp}^{[t_i^1,...,t_i^n]}$	Path from s to d that visits the task sequence $[t_i^1, \ldots, t_i^n]$ and minimizes the travel cost (Sec. 3.4)
$P_{pp}^{[t_i^1,...,t_i^n]}$	Path from s to d that visits the task sequence $[t_i^1, \ldots, t_i^n]$ and minimizes the detour w.r.t. PP (Sec. 3.4)
$P_i[v_r : v_s]$	Sub-path in P_i between v_r and v_s (Sec. 3.4)

Table 3.1: Notation used in Chapter 3.

In the following, we prove that both IRTS-SP and IRTS-PP are NP-Hard, therefore justifying the need for efficient heuristics in order to solve non-trivially sized problems.

Theorem 5. IRTS-SP is NP-Hard.

Proof. Solving IRTS-SP involves finding the path from s to d with the highest reward within the given budget. This is precisely what is returned by the Orienteering Problem (OP) [23] if we consider that all tasks take 0 units of time to be performed. Therefore, IRTS-SP is at least as hard as OP. Since OP is NP-Hard, IRTS-SP is also NP-Hard. $\square$

It is worth emphasizing that OP finds *only one* of the paths belonging to the skyline set, while IRTS-SP finds *all* non-dominated paths in terms of detour and reward.

Next, in order to establish IRTS-PP's complexity, recall that finding its skyline set includes finding the path with the highest reward such that its total time cost is under a given budget b and the detour is minimum. We denote this IRTS-PP's sub-problem as the Maximum Reward Minimum Detour (MRMD) problem and show that MRMD's decision version is NP-Complete and from that we can show that the IRTS-PP problem is NP-hard.

Lemma 1. The decision problem of MRMD, i.e., to decide whether there exists a valid path with reward at least L_r and detour at most $U_d \leq b$, is NP-Complete.

Proof. We prove this theorem by a reduction from the decision version of the Traveling Salesman Problem (TSP). Given a complete graph $G(V, E)$, where the cost of an edge $(v, u) \in E$ is given by the travel cost between vertices $v, u \in V$, the TSP aims at finding the shortest possible route from a given vertex v that visits every vertex in G exactly once and returns to v. The decision version of the TSP seeks to answer the following question: Given a graph $G(V, E)$, is there a tour from a given vertex $v \in V$ with cost at most T_{max}?

In order to transform a given TSP problem into an instance of MRMD we make the following assumptions:

- The starting location v represents the worker's starting location and all the remaining vertices represent tasks (and are referred as such in the following).

- The travel cost of an edge linking task i to task j is defined as in TSP. Each task taskes 0 units of time to be performed and is associated with a reward of 1. Additionally, the budget b in MRMD is equal to T_{max}.

- The worker's destination d is equal to v and the preferred path is given by $PP = \langle s, d \rangle$.

- L_r is given by $|V| - 1$, i.e., the sum of the rewards of all tasks.

The decision problem of the constructed MRMD instance is: Given a worker and the set of tasks T, can we find a path that includes all the tasks and whose detour is at most $b = T_{max}$?

We now show that TSP has a Yes-instance if and only if MRMD has a Yes-instance. A solution to TSP visits every vertex with cost at most T_{max}. This means that all tasks can be completed, maximizing the reward, with cost at most T_{max}. We note that since the only edge in the preferred path is from v to itself, the detour cost of any path is equal to its travel cost and, thus, it is up to T_{max}. On the other hand, if the MRMD problem has a Yes-instance, then it completes all the tasks and the detour cost is no greater than $b = T_{max}$. Thus, the corresponding path is a TSP route with cost no more than T_{max}.

Therefore, since TSP's decision problem is NP-complete it follows that MRMD's decision problem is also NP-Complete. □

Theorem 6. IRTS-PP is NP-Hard

Proof. MRMD's decision problem is NP-complete which implies that MRMD's optimization version is NP-Hard. Since solving MRMD is required to solve IRTS-PP then it follows that IRTS-PP is also NP-Hard. □

3.4 Proposed Approaches

In this section we present our solutions to both IRTS-SP and IRTS-PP. A straightforward approach to solve both of these variants is to find all possible paths from s to d passing through at least one task, and add the non-dominated ones to the skyline set. However, such an approach is clearly not practical even for very small problems. Therefore, for each IRTS variant, we first propose an exact approach that finds the set of all non-dominated paths by pruning sub-paths that are provably not part of a complete non-dominated path from s to d. Next, since the exact approach does not scale to larger instances due to the NP-hardness of the IRTS problem, we propose some heuristics that approximate the exact skyline.

3.4.1 IRTS-SP

Exact Solution

IRTS-SP is a simpler problem than IRTS-PP in the sense that, for a given set of tasks, minimizing the total detour incurred for traveling to the locations of those tasks is equivalent to minimizing the total travel cost of a path connecting them. This means that, for a given pair of tasks, we only need to examine one path connecting them, namely the fastest one. On the other hand, as we will show in Section 3.4.2, in order to solve IRTS-PP we may need to examine multiple paths between pairs of tasks.

Based on the above observation, in order to find the exact skyline for IRTS-SP, we build a directed graph of tasks $GT_{sp} = (V_{sp}, E_{sp})$. The set of vertices V_{sp} contains s, d, and a set of feasible tasks that can be completed within the given budget. Each edge connecting two vertices v and u in E_{sp} represents the fastest path from v to u in the original graph G and is associated with the travel cost of such path. The graph of tasks GT_{sp} obtained for the example shown in Figure 3.1 is illustrated in Figure 3.3.

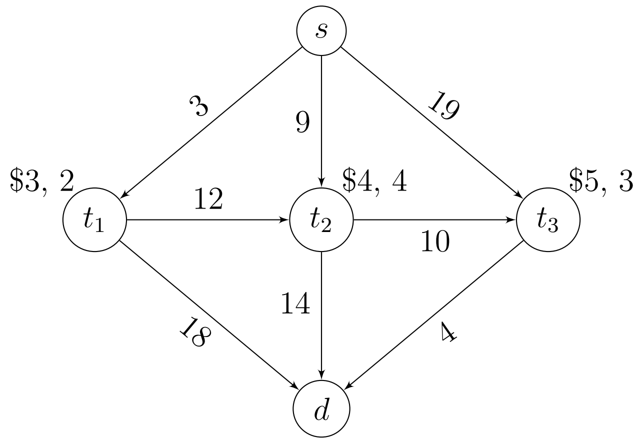


Figure 3.3: Graph GT_{sp} built for the example shown in Figure 3.1. The value on each edge represents the corresponding travel cost.

Algorithm 5 shows how GT_{sp} is constructed in details. We first compute the fastest path from s to all tasks $t_i \in T$ that could be completed within the given budget (line 3). This step is performed by calling algorithm *shortestPathToSet* (explained next) and serves to filter the feasible tasks. Next, for each feasible task t_i, we find the fastest path $P_{sp}^{t_i,t_j}$ connecting it to each task t_j that can be completed after t_i within the remaining budget (line 11), i.e., after deducting the time for traveling to t_i and completing it. An edge is created for each such $P_{sp}^{t_i,t_j}$ and added to the the set of edges (lines 13-16).

```
Algorithm 5: $GT_{sp}$ construction
  Input: Graph $G = (V, E)$, starting point $s$, destination $d$, set of tasks
         $T = \{t_1, ..., t_m\}$, and budget $b$.
  Output: Graph of tasks $GT_{sp} = (V_{sp}, E_{sp})$.
1  $V' \leftarrow s, d$
2  $E' \leftarrow \emptyset$
3  $\mathcal{P}_{sp} \leftarrow shortestPathToSet(s, d, G, T, b)$
4  $T' \leftarrow$ all tasks found in $\mathcal{P}_{sp}$
5  for each task $t_i$ in $T'$ do
6      $V_{sp} \leftarrow V_{sp} \cup t_i$
7      $P_{sp}^{s,t_i} \leftarrow$ shortest path from $s$ to $t_i$ in $\mathcal{P}_{sp}$
8      $e \leftarrow$ new edge from $s$ to $t_i$ with cost $TC(P_{sp}^{s,t_i})$
9      $E_{sp} \leftarrow E_{sp} \cup e$
10     $remainingBudget \leftarrow b - TC(P_{sp}^{s,t_i}) - \delta(t_i)$
11     $\mathcal{P}_{sp}^{t_i} \leftarrow shortestPathToSet(t_i, d, G, T' \cup d, remainingBudget)$
12     $T'_i \leftarrow$ all tasks found in $\mathcal{P}_{sp}^{t_i}$
13     for each $t_j$ in $T'_i$ do
14         $P_{sp}^{t_i,t_j} \leftarrow$ shortest path from $t_i$ to $t_j$ in $\mathcal{P}_{sp}^{t_i}$
15         $e \leftarrow$ new edge from $t_i$ to $t_j$ with cost $TC(P_{sp}^{t_i,t_j})$
16         $E_{sp} \leftarrow E_{sp} \cup e$
17     end
18 end
19 return $(V_{sp}, E_{sp})$
```

Complexity Analysis (GT_{sp} construction). In order to build GT_{sp} we execute *shortestPathToSet* up to $|T| + 1$ times in G, i.e., once for s and at most once for each task in T. Let C_{SPS} be the complexity of *shortestPathToSet* (discussed next), the complexity of Algorithm 5 in the worst case is $O(|T| \times C_{SPS})$.

Within Algorithm 5, the call to *shortestPathToSet* (described in Algorithm 6) is used to find the shortest paths from s to every feasible task t_i considering the given budget b (line 3), and then from each such t_i to the destination and to the tasks that can be visited afterward considering the remaining budget (line 11). *shortestPathToSet* (Algorithm 6) performs a network expansion based on the classical Dijkstra's algorithm. We maintain a queue Q that stores paths which are dequeued in increasing order of travel cost. For each dequeued path P, we first check if its last vertex v has been

Algorithm 6: *shortestPathToSet*

```
Input: Origin $o$, final destination $d$, graph $G = (V, E)$, set of target
       vertices $S_v$, maximum cost $maxCost$.
Output: Set $\mathcal{P}$ of shortest paths from $o$ to the vertices in $S_v$.
1  $CS, \mathcal{P}, Q \leftarrow \emptyset$
2  $Q.add(\langle o \rangle)$
3  while $Q \neq \emptyset$ do
4      $P \leftarrow Q.dequeue()$
5      $v \leftarrow$ last vertex of $P$
6      if $CS.contains(v)$ then
7          continue
8      end
9      $CS.add(v)$
10     if $S_v.contains(v)$ then
11         if $TC(P) + \delta(v) + \frac{d_e(v,d)}{maxSpeed} \leq maxCost$ then
12             $\mathcal{P}.add(P)$
13         end
14         $S_v.remove(v)$
15         if $S_v = \emptyset$ then
16             return $\mathcal{P}$
17         end
18     end
19     for each $u$ neighbor of $v$ in $E$ do
20         if $u \notin CS$ then
21             $P_u \leftarrow$ extend $P$ with $u$
22             $lb \leftarrow TC(P_u) + \frac{d_e(u,d)}{maxSpeed}$
23             if $lb > maxCost$ then
24                 continue
25             end
26             $Q.add(P_u)$
27         end
28     end
29 end
30 return $\mathcal{P}$
```

found before (line 6). If so, there is another path P_j from s to v that is shorter than P, thus it is not worth expanding P. If not, we check whether v is one of the target vertices, i.e., if it belongs to S_v (line 10). If such condition is satisfied, we estimate the total time of the shortest path from o to d passing through v as $TC(P) + \delta(v) + \frac{d_e(v,d)}{maxSpeed}$ (line 11), where $d_e(u, d)$ is the Euclidean distance from u to d and $maxSpeed$ is the maximum speed allowed in the given network. If such estimate does not exceed $maxCost$, v can potentially be completed within the budget and P is added to the result set $\mathcal{P}$ (line 12). Next, v is removed from S_v. If S_v becomes empty, the search stops and $\mathcal{P}$ is returned since all shortest paths to the target vertices have already been found. If P is not pruned, it is expanded with each neighbor u of v in E, creating a new path P_u (lines 19-21). Then, we estimate the total travel cost of the shortest path from the origin o to the destination d passing through u (line 22) as $lb = TC(P_u) + \frac{d_e(u,d)}{maxSpeed}$. If $lb > maxCost$, it is not worth expanding P^u, since d can not be reached within the given budget. Otherwise, P_u is added to Q (line 26). Besides the termination condition discussed above, the algorithm also stops when the queue becomes empty.

Complexity Analysis (*shortestPathToSet*). Since *shortestPathToSet* performs a network expansion similar to one in Dijkstra's algorithm, its complexity in the worst case is $O(|E| + |V| \times log|V|)$.

In the following we prove that *shortestPathToSet* correctly finds all feasible tasks T' that should be connected to s in the graph GT_{sp}, i.e., each task $t_i \in T$ such that $TC(P_{sp}^{s,t_i}) + \delta(t_i) + TC(P_{sp}^{t_i,d}) \leq b$, as well as their corresponding shortest paths. Similarly, we also prove that for every $t_i \in T'$, it finds all tasks that can be completed after t_i considering the remaining budget, i.e., each $t_j \in T'$ such that $TC(P_{sp}^{s,t_i}) + \delta(t_i) + TC(P_{sp}^{t_i,t_j}) + \delta(t_j) + TC(P_{sp}^{t_j,d}) \leq b$. We note that although the actual value of $TC(P_{sp}^{t_i,d})$ is not known before *shortestPathToSet* is invoked, in this algorithm we use a lower bound, namely $\frac{d_e(t_i,d)}{maxSpeed}$, to estimate such cost, hence no feasible tasks are discarded.

Lemma 2. *shortestPathToSet* correctly finds the set of feasible tasks to be connected to s and to each task in GT_{sp}.

Proof. Let us first consider the case where *shortestPathToSet* is used to find the set $T' \subseteq T$ of feasible tasks from s within Algorithm 5 (line 3). By contradiction, let us assume that there is a feasible task $t_i \in T$, i.e., $TC(P_{sp}^{s,t_i}) + \delta(t_i) + TC(P_{sp}^{t_i,d}) \leq b$, that does not belong to T'. This means that the shortest path P_{sp}^{s,t_i} from s to t_i was not added to the result set $\mathcal{P}$ in *shortestPathToSet*. Let us analyze the two possibilities:

1. P_{sp}^{s,t_i} was added to Q

   (a) P_{sp}^{s,t_i} was never dequeued.

   In this case Q is not empty, but the algorithm terminated before P_{sp}^{s,t_i} could be dequeued. Therefore, line 14 of Algorithm 6 was reached for a certain path P, which was examined before P_{sp}^{s,t_i}. This means that P was the first path found for the last target vertex remaining in S_v, and paths to every other vertices in S_v have been previously found. Since paths are examined in increasing order of cost, this contradicts the assumption that P_{sp}^{s,t_i} is the shortest path to $t_i \in S_v$.

   (b) P_{sp}^{s,t_i} was dequeued.

   Then, it must have been the case that P_{sp}^{s,t_i} was pruned on line 7 or 11 of Algorithm 6. If line 7 was reached, there is a shorter path P' from s to t_i, since paths are dequeued from Q in increasing order of cost, which contradicts the fact that P_{sp}^{s,t_i} is the shortest path connecting s to t_i. In the second case (line 11), P_{sp}^{s,t_i} was pruned because $TC(P_{sp}^{s,t_i}) + \delta(t_i) + \frac{d_e(t_i,d)}{maxSpeed} > b$, which is a contradiction to the assumption that t_i is a feasible task and thus $TC(P_{sp}^{s,t_i}) + \delta(t_i) + TC(P_{sp}^{t_i,d}) \leq b$, and, consequently $TC(P_{sp}^{s,t_i}) + \delta(t_i) + \frac{d_e(t_i,d)}{maxSpeed} \leq b$, since $\frac{d_e(t_i,d)}{maxSpeed}$ is a lower bound to $TC(P_{sp}^{t_i,d})$.

2. P_{sp}^{s,t_i} was not added to Q

   This means that for some vertex u in P_{sp}^{s,t_i}, the shortest path $P_{sp}^{s,u}$ was not added to Q. Let v be the vertex visited immediately before u in P_{sp}^{s,t_i} and assume that $P_{sp}^{s,v}$ is dequeued. $P_{sp}^{s,u}$ is discarded if one the following conditions is satisfied:

(a) v has been found before

Therefore, v has been found with smaller cost, which contradicts the assumption that $P_{sp}^{s,v}$ is the shortest path from s to v and, consequently, that P_{sp}^{s,t_i} is the shortest path to t_i.

(b) The path $P_{sp}^{s,v}$ is expanded, but $lb = TC(P_u) + \frac{d_e(u,d)}{maxSpeed} > b$, where P_u is the path obtained after extending $P_{sp}^{s,v}$ with u

Since P_u is a subpath of P_{sp}^{s,t_i} and by assumption $TC(P_{sp}^{s,t_i}) + TC(P_{sp}^{t_i,d}) \leq b$, then $TC(P_u)+TC(P_{sp}^{u,d}) \leq b$ also holds. As $\frac{d_e(u,d)}{maxSpeed}$ is a lower bound to $TC(P_{sp}^{u,d})$, the inequality $TC(P_u)+\frac{d_e(u,d)}{maxSpeed} > b$ is a contradiction.

Now, let us consider the case where *shortestPathToSet* is used to find the set of feasible tasks from each task $t_i \in T'$ within Algorithm 5 (line 11). We need to prove that *shortestPathToSet* finds the shortest paths from t_i to every task $t_j \in T'$ such that $TC(P_{sp}^{s,t_i}) + \delta(t_i) + TC(P_{sp}^{t_i,t_j}) + \delta(t_j) + TC(P_{sp}^{t_j,d}) \leq b$. Therefore, t_j is feasible if $TC(P_{sp}^{t_i,t_j})+TC(P_{sp}^{t_j,d})+\delta(t_j) \leq b-TC(P_{sp}^{s,t_i})-\delta(t_i)$. The proof for this case is similar to the one presented above when we consider that the starting node o given as input to *shortestPathToSet* is t_i, the set of target vertices is $S_v = T'$ and $maxCost = b - TC(P_{sp}^{s,t_i}) - \delta(t_i)$. □

Once GT_{sp} is built as discussed above, we are able to find the exact skyline containing non-dominated paths, in terms of reward and detour, from s to d. Algorithm 7 shows the pseudo-code of our solution. It invokes the algorithm *EXCT_REC-SP* (Algorithm 8), which recursively expands a given path P until the budget is exceeded. Initially, P contains only the starting point s (line 1 of Algorithm 7). In each call of *EXCT_REC-SP*, we analyze the feasible paths that can be obtained by expanding P. We first check whether the destination can be reached within the remaining budget from the last vertex v of P (line 2). If not, P is not further expanded and the recursion stops. Otherwise, we consider the edges that connect v to any other vertex $u \in V_{sp}$ (line 5). If the current neighbor u of v is the destination d (line 7), P is extended with u, generating a new path P_u (line 7). If P_u is not dominated, P_u is added to the skyline set $\mathcal{S}$ and any path dominated by it is removed from $\mathcal{S}$ (lines 9-12). We note that P_u is not further expanded since it is a path to the destination (line 13). On the other hand, if u is a task (line 15), we first check if u has already been expanded when constructing P. If not, P is expanded creating the new path P_u, and $EXCT\_REC\text{-}SP$ is recursively called for P_u and considering the remaining budget $remainingBudget = b - TC(P_{sp}^{v,u}) - \delta(u)$, i.e., the budget after deducting the cost to reach u and to complete it.

Algorithm 7: *EXCT-SP*

```
Input: Graph of tasks $GT_{sp} = (V_{sp}, E_{sp})$, starting point $s$, destination $d$
       and budget $b$.
Output: Set $\mathcal{S}$ of non-dominated paths w.r.t. $TDSP(\cdot, SP)$ and $R(\cdot)$ with
        cost up to $b$.
1 $P \leftarrow \langle s \rangle$
2 $\mathcal{S} \leftarrow \emptyset$
3 $EXCT\_REC\text{-}SP(GT_{sp}, d, b, P, \mathcal{S})$
4 return $\mathcal{S}$
```

In order to check the dominance of a path and update the skyline set $\mathcal{S}$ efficiently (lines 9 and 11 of Algorithm 8), we follow the strategy proposed in [43]. More specifically, we maintain an ordering of $\mathcal{S}$ as a tuple $(P_1, \ldots, P_K)$ where $\forall i < j \in 1, .., K$, the following conditions hold: $TDSP(P_i, SP) < TDSP(P_j, SP)$ & $R(P_i) < R(P_j)$. In order to determine whether a given path P is dominated in $\mathcal{S}$ we find its left neighbor P_k, which is the closest path to P w.r.t. $TDSP(\cdot, SP)$ where $TDSP(P_k, SP) \leq TDSP(P_i, SP)$. If P is not dominated by P_k then it is not dominated in $\mathcal{S}$ and thus it can be inserted into such set. Determining the left neighbor of P in $\mathcal{S}$ can be performed

Algorithm 8: *EXCT_REC-SP*

```
Input: Graph of tasks $GT_{sp} = (V_{sp}, E_{sp})$, final destination $d$, budget $b$,
       path to be expanded $P$ and current skyline set $\mathcal{S}$.
$v \leftarrow$ last vertex of $P$
if $TC(P_{sp}^{v,d}) > b$ then
    return
end
for each edge $e$ from $v$ in $E_{sp}$ do
    $u \leftarrow e.toNode$
    if $u = d$ then
        $P_u \leftarrow$ extend $P$ with $u$
        if $P_u$ is not dominated in $\mathcal{S}$ then
            $\mathcal{S}.add(P_u)$
            remove any paths dominated by $P_u$ from $\mathcal{S}$
        end
        continue
    end
    else
        if $u$ has already been expanded when constructing $P$ then
            continue
        end
        $P_u \leftarrow$ extend $P$ with $u$
        $remainingBudget \leftarrow b - TC(P_{sp}^{v,u}) - \delta(u)$
        $EXCT\_REC\text{-}SP(GT_{sp}, d, remainingBudget, P_u, \mathcal{S})$
    end
end
```

using a binary search w.r.t. $TDSP(\cdot, SP)$. Thus, the complexity of checking whether P has to be inserted into $\mathcal{S}$ is $O(\log |\mathcal{S}|)$. In case P is inserted into $\mathcal{S}$, we need to check whether it dominates other paths in $\mathcal{S}$, starting with P_k. If P_k is dominated by P, it is removed from $\mathcal{S}$. Thereafter, we perform a right traversal in $\mathcal{S}$ starting from P_{k+1}. If P dominates P_{k+1}, then P_{k+1} is removed from $\mathcal{S}$ and P_{k+2} becomes P's right neighbor. This traversal stops when P does not dominate its right neighbor or when there is no path left to be considered. Such operation has a time complexity of $O(|\mathcal{S}|)$ in the worst case. However, our experiments have shown that the results of the dominance checks (line 9) are negative for the majority of cases, i.e., often yielding cost $O(\log |\mathcal{S}|)$ rather than $O(\log |\mathcal{S}| + |\mathcal{S}|)$.

Figure 3.4 shows the paths explored by *EXCT-SP* for the graph GT_{sp} shown in Figure 3.3. The feasible path $P_1 = \langle s, t_1, d \rangle$ is found with cost 23 and reward \$3. Since the cost of the shortest path SP is 15, P_1's detour is 8. As the skyline set $\mathcal{S}$ is empty, P_1 is added to $\mathcal{S}$. Next, $P_2 = \langle s, t_2, t_3, d\rangle$ is found with cost 30 and reward \$9, and is added to $\mathcal{S}$. Then, path $P_3 = \langle s, t_2, d\rangle$ is found with cost 27 and reward \$4 and it is also added to the skyline set. Finally, $P_4 = \langle s, t_3, d\rangle$ is found and added to $\mathcal{S}$, while P_3 is discarded since it is dominated by P_4. Thus, the final skyline set returned is $\mathcal{S} = \{P_1, P_2, P_4\}$.

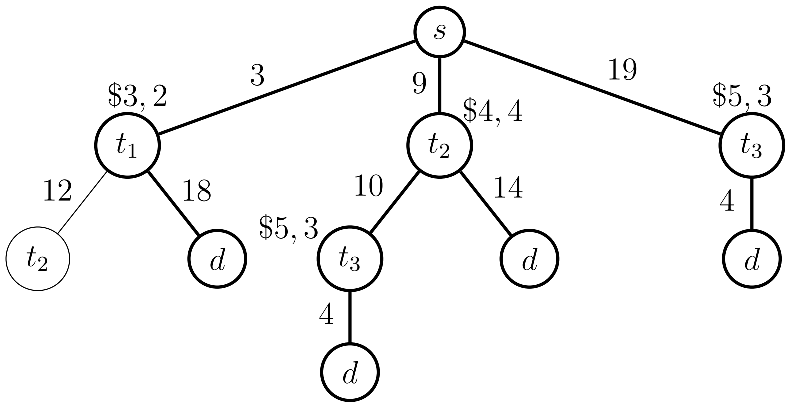


Figure 3.4: Paths explored by *EXCT-SP* for the GT_{sp} shown in Figure 3.3. The feasible paths from s to d are shown in bold.

Complexity Analysis **(*EXCT-SP*).** In the worst case, *EXCT-SP* generates all permutations of tasks. Considering the time for checking the dominance of each generated path and for updating the skyline set, *EXCT-SP*'s complexity in the worst case is $O(|T|! \times (\log|\mathcal{S}| + |\mathcal{S}|)) = O(|T|! \times |\mathcal{S}|)$. However, we note that the number of tasks in a path is limited by the budget given by the worker and, in practice, it is very unlikely that such budget is large enough to include all tasks in the graph. Moreover, as discussed above, the skyline set will not be updated for the majority of generated paths.

Theorem 7. The skyline set $\mathcal{S}$ found by *EXCT-SP* is the exact one.

Proof. We need to prove that $\mathcal{S}$ is complete, i.e., it includes all non-dominated paths, and also correct, i.e., that no dominated path is part of $\mathcal{S}$.

First, by contradiction, let us assume that there is a feasible non-dominated path P_i from s to d that is not part of $\mathcal{S}$. Then there are two cases to be analyzed:

1. P_i was found by *EXCT-SP*, but was discarded

   This means that the condition on line 9 of Algorithm 8 was not satisfied. Therefore, P_i is dominated in $\mathcal{S}$, which is a contradiction to the assumption that P_i is a non-dominated path.

2. P_i was not found by *EXCT-SP*

   In this case, for some task t_j in P_i, the subpath $P_i[s : t_j]$ was not found. Let t_k be the task visited immediately before t_j in P_i and let us assume that the subpath $P_i[s : t_k]$ was found. When *EXCT_REC-SP* was called for $P_i[s : t_k]$, $P_i[s : t_j]$ was not found if one of the following conditions was satisfied:

   (a) $TC(P_{sp}^{t_k,d}) > remainingBudget$ (line 2)

   Therefore, $TC(P_{sp}^{t_k,d}) > b - TC(P_i[s : t_k]) - \sum_{v_l \in P_i[s:t_k]} \delta(v_l) \times \pi(v_l)$ and, consequently, $TC(P_i[s : t_k]) + \sum_{v_l \in P_i[s:t_k]} \delta(v_l) \times \pi(v_l) + TC(P_{sp}^{t_k,d}) > b$. However, since P_i is feasible, we have that $TC(P_i) + \sum_{v_l \in P_i} \delta(v_l) \times \pi(v_l) \leq b$. Furthermore, once $P_i[s : t_k]$ is a sub-path of P_i, we also have that $TC(P_i[s : t_k]) + TC(P_{sp}^{t_k,d}) \leq TC(P_i)$ and $\sum_{v_l \in P_i[s:t_k]} \delta(v_l) \times \pi(v_l) \leq \sum_{v_l \in P_i} \delta(v_l) \times \pi(v_l)$. Thus, $TC(P_i[s : t_k]) + TC(P_{sp}^{t_k,d}) + \sum_{v_l \in P_i[s:t_k]} \delta(v_l) \times \pi(v_l) \leq b$, which is a contradiction.

   (b) There is no edge $e \in E_{sp}$ from t_k to t_j representing the path $P_i[t_k : t_j]$.

   This can happen in one of the following situations: (I) there is no edge at all between t_k and t_j in E_{sp} or (II) the edge connecting t_k to

t_j in E_{sp} represents a path P^{t_k,t_j} different from $P_i[t_k : t_j]$. Condition (I) is a contradiction since P_i is a feasible path, which means that is possible to visit t_j after t_k within the budget and thus there is an edge connecting these vertices in E_{sp}. Regarding (II), since P^{t_k,t_j} is the shortest path connecting t_k and t_j, $TC(P^{t_k,t_j}) < TC(P_i[t_k : t_j])$. This means that $P_i[t_k : t_j]$ could be substituted with P^{t_k,t_j} in P_i, generating a new shorter path $P^{s,d}$, which dominates P_i. This contradicts the assumption that P_i is not dominated.

(c) t_j has already been expanded when constructing $P_i[s : t_k]$

Let t_l be the task visited after the second time t_j is visited in P_i (if no tasks are visited after t_j, $t_l = d$). Since by assumption P_i is not dominated, there is an edge $e \in E_{sp}$ connecting t_k to t_l whose travel cost is equal to that of path $P_i[t_k : t_l]$. Therefore, unless there is another non-dominated path P_m with the same reward and cost as P_i (in this case P_m could substitute P_i in the skyline set), P_i can be found by following edge e, which contradicts the assumption that P_i is not found.

Next, we prove that no dominated path is part of $\mathcal{S}$. By contradiction, suppose that there is a dominated path P_d in $\mathcal{S}$. Since as proved above all non-dominated paths are part of $\mathcal{S}$, P_d can not belong to $\mathcal{S}$ since it is dominated by at least one path in $\mathcal{S}$ and all dominated paths are removed from $\mathcal{S}$ (line 11 of Algorithm 8). This is a contradiction to the assumption that P_d is part of $\mathcal{S}$. $\square$

Heuristic Solutions

Due to the hardness of the IRTS-SP problem, the exact approach does not scale to large sized instances. Therefore, we developed a few heuristics that approximate the exact skyline, which are presented next.

k-NN Graph Heuristic. In order to avoid generating all possible permutations of tasks, the k-NN Graph Heuristic (*kGH-SP*) limits the number of neighbors of each task t_i in GT_{sp} to a given $k \ll |T|$, leading to GT_{sp}^k, which is a smaller version of GT_{sp}. More specifically, when building GT_{sp}^k we only connect a task t_i to its k closest tasks, which, intuitively, have a greater chance of leading to shorter paths. (We defer the discussion on what would be suitable values for k to use to Section 3.5.)

In order to find the non-dominated paths in this reduced graph we follow the same procedure as *EXCT-SP*, with the only difference being the graph given as input to Algorithm 7, which in this case is GT_{sp}^k (as opposed to GT_{sp}).

Complexity Analysis (kGH-SP). In the worst case, the complexity for building GT_{sp}^k is the same as the one for building GT_{sp}. With regard to *kGH-SP*, when a path P is expanded, up to k new paths of size $|P| + 1$ are created. Since initially one path is created for each feasible task in T, the total number of paths generated in the worst case is $O(|T| \times k^{|T|})$. Let $\mathcal{S}_{app}$ be the skyline set computed by *kGH-SP*. Considering the cost for updating $\mathcal{S}_{app}$, *kGH-SP*'s time complexity in the worst case is $O(|T| \times k^{|T|} \times |\mathcal{S}_{app}|)$.

Minimum Detour Heuristic. The Minimum Detour Heuristic (*MDH-SP*) greedily expands a path in G with its closest feasible task. The pseudo-code of *MDH-SP* is shown in Algorithm 9. We first compute the fastest path from s to all tasks $t_i \in T$ that could be completed within the given budget by calling algorithm *shortestPathToSet* (line 2). Then, for each such t_i, we compute the fastest path $P_{sp}^{t_i,d}$ to the destination by invoking *shortestPath* (explained next). Next, *MDH_REC-SP* (Algorithm 10) is invoked for P_{sp}^{s,t_i}, which is expanded until the budget is exceeded. Let v be last vertex of P. *MDH_REC-SP* first extends P with the fastest path from v to d, creating a new path P_u (line 5). If P_u is not dominated, it is added to the set $\mathcal{S}_{app}$ and any path dominated by P_u is removed from $\mathcal{S}_{app}$ (lines 6 to 9). Next, *MDH-SP* selects the closest task u from v that is not in P and can be completed within the remaining budget by invoking *getClosestTask* (Algorithm 11, explained next). If such vertex exists, a new path P_u is created by expanding P with $P_{sp}^{v,u}$ (line 12) and *MDH_REC-SP* is recursively called for P_u (lines 12) and considering the remaining budget, i.e., the budget after deducting the travel cost to u and the time for completing u. If there is no task that can be completed after v, the recursion stops. A path P is also not further expanded when $TC(P_{sp}^{v,d}) > b$ (line 2), meaning that the destination can not be reached within the given budget b in G.

Algorithm 9: Minimum Detour Heuristic (*MDH-SP*)

```
   Input: Graph $G = (V, E)$, starting point $s$, destination $d$, set of tasks $T$
          and budget $b$.
   Output: Set $\mathcal{S}_{app}$ of non-dominated paths, in terms of detour cost and
           reward, from $s$ to $d$ with cost up to $b$.
1  $\mathcal{S}_{app} \leftarrow \emptyset$
2  $\mathcal{P}_{sp} \leftarrow shortestPathToSet(s, d, G, T, b)$
3  $T' \leftarrow$ all tasks found in $\mathcal{P}_{sp}$
4  for each task $t_i$ in $T'$ do
5      $remainingBudget \leftarrow b - TC(P_{sp}^{s,t_i}) - \delta(t_i)$
6      $P_{sp}^{t_i,d} \leftarrow shortestPath(t_i, d, G, remaningBudget)$
7      $P \leftarrow P_{sp}^{s,t_i}$
8      $MDH\_REC{-}SP(G, remainingBudget, P, \mathcal{S}_{app})$
9  end
10 return $\mathcal{S}_{app}$
```

Complexity Analysis (MDH-SP). In the worst case, *MDH-SP* invokes *shortestPath* up to $|T|$ times. Therefore, the complexity of this phase is $O(|T| \times C_{SP})$, where C_{SP} is the complexity of *shortestPath* (analyzed next). Next, *MDH_REC-SP* is called for each task $t_i \in T'$. For each such task,

Algorithm 10: *MDH_REC-SP*

```
Input: Graph $G = (V, E)$, budget $b$, path to be expanded $P$ and current
       skyline set $\mathcal{S}_{app}$.
1  $v \leftarrow$ last vertex of $P$
2  if $TC(P_{sp}^{v,d}) > b$ then
3      return
4  end
5  $P_u \leftarrow$ extend $P$ with $P_{sp}^{v,d}$
6  if $P_u$ is not dominated in $\mathcal{S}_{app}$ then
7      $\mathcal{S}_{app}.add(P_u)$
8      remove any paths dominated by $P_u$ from $\mathcal{S}_{app}$
9  end
10 $P_{sp}^{v,u} \leftarrow getClosestTask(v, G, T \setminus P, b)$
11 if $u$ is not null then
12     $P_u \leftarrow$ extend $P$ with $P_{sp}^{v,u}$
13     $remaningBudget \leftarrow b - TC(P_{sp}^{v,u}) - \delta(u)$
14     $EXCT\_REC\text{-}SP(G, remaningBudget, P_u, \mathcal{S}_{app})$
15 end
```

MDH_REC-SP is recursively called up to $|T'|$ times, therefore it is invoked up to $|T|^2$ times in total. In each call of *MDH_REC-SP*, *getClosestTask* is invoked once and the skyline set can also be updated. Thus, in the worst case, the total cost of all calls to *MDH_REC-SP* is $O(|T|^2 \times (C_{GCT} + |\mathcal{S}_{app}|))$, where C_{GCT} is the time complexity of *getClosestTask* (analyzed next). Therefore, the total time complexity of *MDH-SP* is $O(|T| \times C_{SP} + |T|^2 \times (C_{GCT} + |\mathcal{S}_{app}|))$.

shortestPath finds the fastest path between two given vertices v and u in G by performing an A* search. The network expansion is guided by the heuristic $h(P) = \frac{d_e(l,u)}{maxSpeed}$, where P is a path whose last vertex is l. In other words, the path with the smallest value of $lb = TC(P_{sp}^{v,l}) + \frac{d_e(l,u)}{maxSpeed}$ is expanded first. Moreover, we also use the remaining budget as an upper bound to the cost of the shortest path between v and u and, therefore, if the lower bound lb of a path P exceeds such budget, P is pruned. For instance, when *shortestPath* is invoked in Algorithm 9 (line 6), the remaining budget is the budget after deducting the cost to reach t_i from s and the time for completing t_i.

Complexity Analysis (shortestPath). In the worst case, *shortestPath* has the same complexity as *shortestPathToSet* (Algorithm 6). However, in practice, *shortestPath* tends to be much faster than *shortestPathToSet* since the network expansion is guided towards a particular destination and it is able to prune more paths.

In order to find the closest feasible task from a given vertex v we invoke *getClosestTask* (Algorithm 11). It first finds a set of candidate tasks that can possibly be completed after v within the budget (lines 2-6). Next, the candidates are examined in increasing order of $\frac{d_e(v,t)}{maxSpeed}$ and the path P_{clo} to the closest task is updated as closer tasks are found. If the current task t has

Algorithm 11: *getClosestTask*

```
Input: Vertex $v$, graph $G = (V, E)$, set of candidate tasks $T'$, and budget $b$.
Output: The closest feasible task $t_i \in T'$ from $v$ that can be completed
        within the budget $b$.
1  $candidates \leftarrow \emptyset$
2  for $t_i \in T'$ do
3      if $\frac{d_e(v,t_i)}{maxSpeed} + \delta(t_i) + TC(P_{sp}^{t_i,d}) \leq b$ then
4          $candidates \leftarrow candidates \cup t_i$
5      end
6  end
7  if $candidates = \emptyset$ then
8      return null
9  end
10 sort $candidates$ by $\frac{d_e(v,t_i)}{maxSpeed}$
11 $P_{clo} \leftarrow null$
12 for $i$ from $0$ to $|candidates| - 1$ do
13     $t \leftarrow candidates[i]$
14     if $\frac{d_e(v,t)}{maxSpeed} < TC(P_{clo})$ then
15         $remaningBudget \leftarrow b - \delta(t) - TC(P_{sp}^{t,d})$
16         $P_{sp}^{v,t} \leftarrow shortestPath(v, t, G, remaningBudget)$
17         if $TC(P_{sp}^{v,t}) < TC(P_{clo})$ then
18             $P_{clo} \leftarrow P_{sp}^{v,t}$
19         end
20     end
21     else
22         return $P_{clo}$
23     end
24 end
25 return $P_{clo}$
```

a lower bound smaller than the cost to reach the closest task so far (line 14), its actual travel cost is computed (line 16) by invoking *shortestPath*. If t is closer to v than the current closest task, t becomes the closest task and P_{clo} is updated (lines 17-19). On the other hand, if the current task t has a lower bound greater than $TC(P_{clo})$ (line 21), then all the other unexamined tasks also do, and thus P_{clo} is returned as the path to the closest task (line 22).

Complexity Analysis (getClosestTask). In the worst case, *getClosestTask* invokes *shortestPath* $|T'|$ times. Therefore, in this case, its complexity is equal to $O(|T'| \times (|E| + |V| \times log|V|))$.

Maximum Reward Heuristic. The Maximum Reward Heuristic (*MRH-SP*) is similar to *MDH-SP* except that, instead of expanding a path P with the closest task from it, it selects the feasible task with the highest reward among the tasks that are not part of P.

Algorithm 12: *getHighestRewardTask*

```
Input: Vertex $v$, graph $G = (V, E)$, set of candidate tasks $T'$ and budget $b$.
Output: Task $t \in T'$ with the highest reward that can be completed after
        $v$ within the budget $b$.
1  $candidates \leftarrow \emptyset$
2  for $t_i \in T'$ do
3      if $\frac{d_e(v,t_i)}{maxSpeed} + \delta(t_i) + TC(P_{sp}^{t_i,d}) \leq b$ then
4          $candidates \leftarrow candidates \cup t_i$
5      end
6  end
7  if $candidates = \emptyset$ then
8      return null
9  end
10 sort $candidates$ in decreasing order of $\rho(t_i)$
11 for $i$ from 0 to $|candidates| - 1$ do
12     $t \leftarrow candidates[i]$
13     $remaningBudget \leftarrow b - \delta(t) - TC(P_{sp}^{t,d})$
14     $P_{sp}^{v,t} \leftarrow shortestPath(v, t, G, remaningBudget)$
15     if $P_{sp}^{v,t}$ is not null then
16         return $P_{sp}^{v,t}$
17     end
18 end
19 return null
```

MRH-SP follows the same procedure as *MDH-SP*, with the only difference being the line 8 of Algorithm 10, where the feasible task with the highest reward should be selected (rather than the closest task). In order to find such task, we invoke *getHighestRewardTask* (Algorithm 12). The set of candidate tasks is selected as in *getClosestTask* (lines 1-6). Next, the candidates are examined in decreasing order of reward (lines 7-8). The first task that can be completed within the given budget is returned (lines 10-11). If there is no feasible task, the algorithm returns null (line 19).

Complexity Analysis (MRH-SP). In the worst case, *getHighestRewardTask* has the same complexity as *getClosestTask* and thus, in this case, *MRH-SP* has the same time complexity as *MDH-SP*.

3.4.2 IRTS-PP

Exact Solution

IRTS-PP is fundamentally different from IRTS-SP due to the fact that traveling from a task t_i to a task t_j through the path that yields the minimum displacement detour w.r.t. PP does not guarantee optimality. For instance, consider tasks t_2 and t_3 in Figure 3.1. The path between these tasks that yields the minimum displacement detour w.r.t. PP is $P_{pp}^{t_2,t_3} = \langle t_2, v_3, v_4, v_5, t_3 \rangle$, with a total detour of 4 and cost of 14. $P_{pp}^{t_2,t_3}$ leads to the complete path $PP_5 = \langle s, v_2, v_3, t_2, v_3, v_4, v_5, t_3, v_5, d \rangle$ (discussed in Section 3.1), with cost equal to 34, which is greater than the budget $b = 32$. Therefore, if we considered $P_{pp}^{t_2,t_3}$ to be the only possible path between t_2 and t_3, no complete path containing t_2 and t_3 would be part of the resulting skyline set. However, as shown in Section 3.1, the shorter path $PP_2 = \langle s, v_2, v_3, t_2, v_1, t_3, v_5, d \rangle$, with detour 21 and cost 30, does connect t_2 to t_3 and belongs to the exact skyline.

As illustrated above, in order to guarantee optimality and find all feasible and non-dominated paths, it is not sufficient to consider the path $P_{pp}^{t_i,t_j}$ between two tasks t_i and t_j that yields the minimum detour w.r.t. PP. We may also need to explore shorter paths, even though they may yield a greater detour. In fact, as we will prove shortly, the set of paths that may need to be explored is limited to the set of non-dominated paths, in terms of travel cost and detour, between two feasible tasks. To exemplify this, let us consider tasks t_2 and t_3 again. The shortest path $P_{sp}^{t_2,t_3} = \langle t_2, v_1, t_3 \rangle$ connecting these tasks is non-dominated and thus it may need to be considered. Another possibility is path $P_{i}^{t_2,t_3} = \langle t_2, v_1, v_4, v_5, t_3 \rangle$, with a displacement detour of 7 and cost 14. However, $P_{i}^{t_2,t_3}$ has the same cost as $P_{pp}^{t_2,t_3} = \langle t_2, v_3, v_4, v_5, t_3 \rangle$, but yields a greater detour. Therefore, $P_{i}^{t_2,t_3}$ is an uninteresting path since it is dominated by $P_{pp}^{t_2,t_3}$, and thus it can be discarded. On the other hand, both paths $P_{pp}^{t_2,t_3}$ and $P_{sp}^{t_2,t_3}$ are non-dominated, in terms of travel and displacement detour costs, and may need to be considered further as possibly being part of a complete path from s to d that belong to the exact skyline. As we will show next, we may not need to compute all non-dominated paths between a given pair of tasks. Therefore, in order to avoid computing unnecessary non-dominated paths, we compute them on the fly, i.e., as needed.

Algorithm 13 shows the pseudo-code of the exact approach $EXCT\text{-}PP$. We first compute the shortest paths from s to all tasks $t_i \in T$ that could be completed within the given budget (line 1), and the paths that yield the minimum detour w.r.t. PP from s to each such task (line 2). The first step is performed by invoking algorithm *shortestPathToSet* (Algorithm 6), while the second one is performed by invoking *minDetourPathToSet*. *minDetourPathToSet* is similar to *shortestPathToSet* except that paths are explored in increasing order of displacement detour w.r.t. PP (rather than travel cost).

Next, the same process is repeated for each feasible task $t_i \in T'$ (lines 4-8). In order to exemplify the paths computed from lines 1-8, let us consider the example shown in Figure 3.1. Table 3.2 shows the shortest paths from s and each feasible task t_i to every other feasible task t_j and to the destination. Similarly, Table 3.3 shows the corresponding paths yielding the minimum detour. For instance, $P_{sp}^{t_2,t_3}$ has a cost of 10 and yields a displacement detour of 10, while $P_{pp}^{t_2,t_3}$ has a cost of 14 and yields a detour of 4.

Algorithm 13: *EXCT-PP*

```
Input: Original graph $G = (V, E)$, preferred path $PP$, set of tasks $T$, and
       budget $b$.
Output: Set $\mathcal{S}$ of non-dominated paths w.r.t. $TDPP(\cdot, PP)$ and $R(\cdot)$
        with cost up to $b$.
1  $\mathcal{P}_{sp}^{s} \leftarrow shortestPathToSet(s, d, G, T, b)$
2  $\mathcal{P}_{pp}^{s} \leftarrow minDetourPathToSet(s, d, G, T, b, PP)$
3  $T' \leftarrow$ all tasks found in $\mathcal{P}_{sp}$
4  for each task $t_i$ in $T'$ do
5      $remainingBudget \leftarrow b - TC(P_{sp}^{s,t_i}) - \delta(t_i)$
6      $\mathcal{P}_{sp}^{t_i} \leftarrow shortestPathToSet(t_i, d, G, T, remainingBudget)$
7      $\mathcal{P}_{pp}^{t_i} \leftarrow minDetourPathToSet(t_i, d, G, T, remainingBudget, PP)$
8  end
9  $\mathcal{S} \leftarrow \emptyset$
10 $tree \leftarrow \langle s \rangle$
11 $buildTree(tree, s, d, \langle s \rangle, \langle s \rangle, b, \mathcal{S})$
12 $expandTree(tree, G, d, b, \langle s \rangle, \mathcal{S})$
13 return $\mathcal{S}$
```

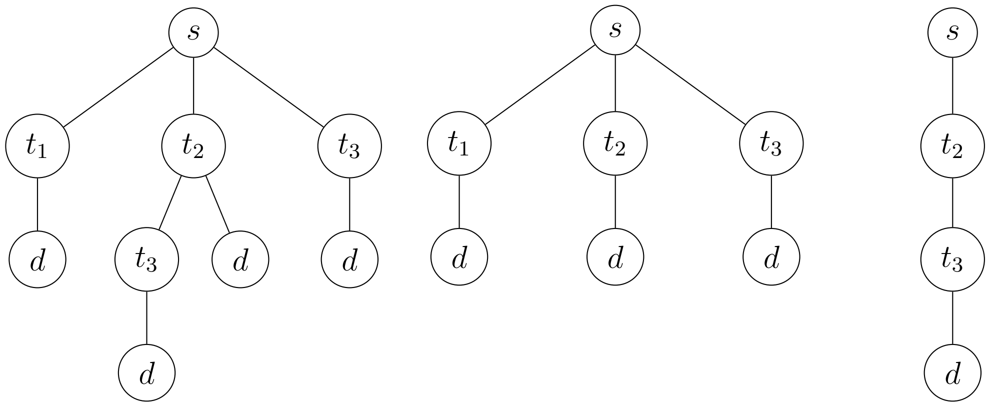
Table 3.2: Fastest paths.

from	**to**	**cost**	**detour**
s	t_1	3	3
s	t_2	9	2
s	t_3	19	2
t_1	t_2	12	5
t_1	d	18	18
t_2	t_3	10	10
t_2	d	14	2
t_3	d	4	2

Table 3.3: Paths yielding the minimum detour w.r.t. PP.

from	**to**	**cost**	**detour**
s	t_1	3	3
s	t_2	9	2
s	t_3	19	2
t_1	t_2	12	5
t_1	d	22	3
t_2	t_3	14	4
t_2	d	14	2
t_3	d	4	2

Subsequently, using the information described above, we build a tree containing task sequences for which we will compute the corresponding non-dominated paths, in terms of detour and travel costs. More specifically, these sequences are the ones that can not be completed if one follows the path yielding the minimum detour between each pair of tasks, but that can be completed by following the corresponding shortest paths. Figures 3.5a and 3.5b illustrate the task sequences that can be completed by following the paths shown in Table 3.2 and Table 3.3, respectively. Note that the only path to d contained in the tree on the left, but not contained in tree on the right is $P = \langle s, t_2, t_3, d\rangle$, as illustrated in Figure 3.5c. Therefore, P can be completed by following the shortest paths between each pair of tasks, but can not be completed if one follows the corresponding paths yielding the minimum detour. This means that P needs to be further examined since there may be another path that visits tasks t_2 and t_3, in this order, whose total cost does not exceed the budget, but that yields a smaller detour than that of the shortest path that visits those tasks.



(a) Paths that can be completed by following the shortest path between each pair of nodes.

(b) Paths that can be completed by following the path that yields the minimum detour between each pair of nodes.

(c) Path to be further examined.

Figure 3.5: Using the information from Tables 3.2 and 3.3 to build a tree containing paths that need to be further examined.

Figure 3.6 illustrates how the tree shown in Figure 3.5c is built. While building such tree, we also compute paths that can potentially be part of the skyline set $\mathcal{S}$, as explained next. Initially, the tree has only one node, i.e., the starting point s (Figure 3.6a). Next, we consider all tasks t_i that can be reached from s within the given budget (as shown in Table 3.2). Let us first consider task t_1. Since the task sequence $[t_1]$ can be completed by following the path $P_{pp}^{[t_1]}$ yielding the minimum detour between each pair of nodes, no other path that visits $[t_1]$ needs to be explored as it is dominated by $P_{pp}^{[t_1]}$. Due to this reason, $P = \langle s, t_1, d \rangle$ is not added to the tree. Moreover, since the task sequence $[t_1]$ will not be further explored, we need to check whether $P_{pp}^{[t_1]}$ is dominated by other paths in $\mathcal{S}$. Since $\mathcal{S}$ is empty, $P_{pp}^{[t_1]}$ is added to $\mathcal{S}$. Next, t_1 is added as a child of s for further expansion (Figure 3.6b). Then, the algorithm is recursively called for the sub-tree rooted at t_1. No tasks can be completed after t_1 within the given budget, thus t_1 is removed from the tree because no path visiting t_1 first needs to be further explored. Next, we consider task t_2. The task sequence $[t_2]$ can also be completed by following the path $P_{pp}^{[t_2]}$ yielding the minimum detour between each pair of nodes. Since $P_{pp}^{[t_2]}$ is not dominated, it is added to $\mathcal{S}$, while $P_{pp}^{[t_1]}$ is removed. Next, t_2 is added as a child of s (Figure 3.6c) and the algorithm is recursively called for the sub-tree rooted at t_2. The task sequence $[t_2, t_3]$ can be completed by following the shortest path $P_{sp}^{[t_2,t_3]}$, but can not be completed if one follows the path $P_{pp}^{[t_2,t_3]}$ yielding the minimum detour between each pair of nodes. Therefore, $P = \langle s, t_2, t_3, d \rangle$ needs to be added to the tree for further examination (Figure 3.6d). The same process is repeated for task t_3, which is added to tree (Figure 3.6e), while $P_{pp}^{[t_3]}$ is added to $\mathcal{S}$, discarding $P_{pp}^{[t_2]}$. Finally, the sequence $[t_3]$ is removed from the tree since no path that visits t_3 first needs to be further examined. The resulting tree is shown in Figure 3.6f. Algorithm 14 shows the pseudo-code of *buildTree* and describes each step illustrated above in details.

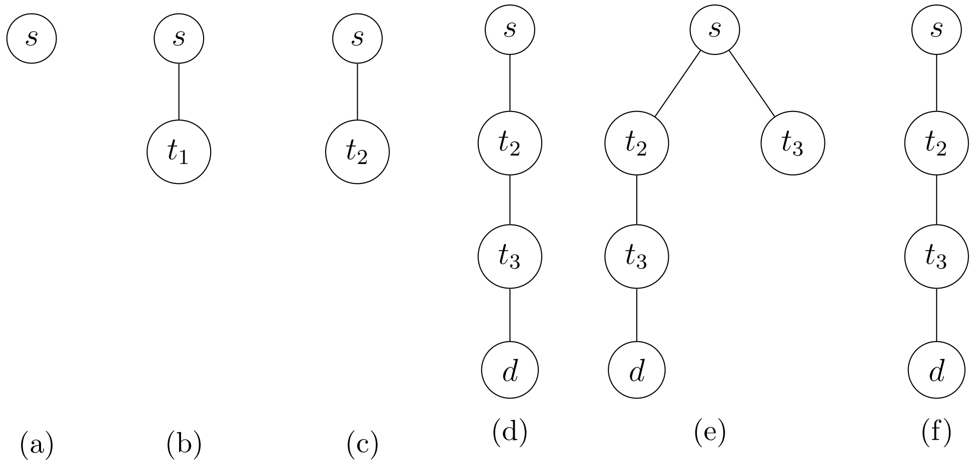


Figure 3.6: Illustration of the construction of a tree containing paths that need to be further examined.

Complexity Analysis **(***buildTree***).** In the worst case, *buildTree* examines all permutations of the tasks in T. Considering the cost of updating the skyline set, its total time complexity in the worst case is $O(|T|! \times |\mathcal{S}|)$.

Lemma 3. *buildTree* finds all feasible non-dominated paths P, in terms of reward and total detour, that can be completed by following the path that yields the minimum detour between each consecutive pair of tasks in P. Moreover, it also creates a path in the tree representing each task sequence that can not be completed in such case, but can be completed by following the corresponding shortest paths.

Proof. Let us first consider that P is a feasible non-dominated path in which each sub-path between two consecutive tasks t_i and t_j is $P_{pp}^{t_i,t_j}$. By contradiction, let us assume that P is not added to the skyline set $\mathcal{S}$ in *buildTree*. This can happen in two cases:

1. P is found (line 5) but not added to $\mathcal{S}$

   In this case, the condition on line 6 is not satisfied. Therefore, P is domi-

Algorithm 14: *buildTree*

```
Input: Tree tree, origin o, destination d, shortest path $P_{sp}$ from s to o by
       following the task sequence to node tree, corresponding path $P_{pp}$
       that yields the minimum detour, budget b and current skyline set
       $\mathcal{S}$.
1  for $P_{sp}^{o,t_i} \in \mathcal{P}_{sp}^{o}$ do
2      if $TC(P_{sp}) + TC(P_{sp}^{o,t_i}) + \delta(t_i) + TC(P_{sp}^{t_i,d}) \leq b)$ then
3          if $P_{pp} \neq null$ & $\mathcal{P}_{pp}^{o}$ contains $P_{pp}^{o,t_i}$ then
4              if $TC(P_{pp}) + TC(P_{pp}^{o,t_i}) + \delta(t_i) + TC(P_{pp}^{t_i,d}) \leq b)$ then
5                  $P_d \leftarrow P_{pp} + P_{pp}^{o,t_i} + P_{pp}^{t_i,d}$
6                  if $P_d$ is not dominated in $\mathcal{S}$ then
7                      add $P_d$ to $\mathcal{S}$
8                      remove any paths dominated by $P_d$ from $\mathcal{S}$
9                  end
10                 $child \leftarrow tree.addChild(t_i)$
11                 $buildTree(child, t_i, P_{sp} + P_{sp}^{o,t_i}, P_{pp} + P_{pp}^{o,t_i}, \mathcal{S})$
12             end
13             else
14                 $child \leftarrow tree.addChild(t_i)$
15                 $child.addChild(d)$
16                 $buildTree(child, t_i, P_{sp} + P_{sp}^{o,t_i}, null, \mathcal{S})$
17             end
18         end
19         else
20             $child \leftarrow tree.addChild(t_i)$
21             $child.addChild(d)$
22             $buildTree(child, t_i, P_{sp} + P_{sp}^{o,t_i}, null, \mathcal{S})$
23         end
24     end
25 end
26 if tree is leaf then
27     $tree.removeBranchRecursively()$
28 end
```

nated in $\mathcal{S}$, which contradicts the assumption that it is a non-dominated path.

2. P is not found on line 5 by *buildTree*

   This means that, for some task t_j, the sub-path $P[s : t_j]$ was not built by *buildTree*. Let t_i be the task visited immediately before t_j in P and let us assume that the sub-path $P[s : t_i]$ was built by *buildTree*. When *buildTree* was called for $P[s : t_i]$, $P[s : t_j]$ was not built if one of the following conditions was satisfied:

(a) $TC(P_{sp}[s:t_i]) + TC(P_{sp}^{t_i,t_j}) + \delta(t_j) + TC(P_{sp}^{t_j,d}) > b$ (line 2).

Where $P_{sp}[s:t_i]$ visits the same tasks as P from s to t_i, but follows the shortest path between each pair of tasks. Since $TC(P_{sp}[s:t_i]) \leq TC(P[s:t_i])$, $TC(P_{sp}^{t_i,t_j}) \leq TC(P_{pp}^{t_i,t_j})$, and $TC(P_{sp}^{t_j,d}) \leq TC(P_{pp}^{t_j,d})$, then $TC(P[s:t_i]) + TC(P_{pp}^{t_i,t_j}) + \delta(t_j) + TC(P_{pp}^{t_j,d}) > b$ also holds. Therefore, $TC(P[s:t_j]) + \delta(t_j) + TC(P_{pp}^{t_j,d}) > b$. As $P[s:t_j]$ is a sub-path of P and $TC(P) \leq b$, this inequality is a contradiction to the assumption that P is feasible.

(b) $TC(P_{pp}[s:t_i]) + TC(P_{pp}^{t_i,t_j}) + \delta(t_i) + TC(P_{pp}^{t_j,d}) > b$ (line 4).

As explained above, this is a contradiction to the fact that P is a feasible path in which each sub-path between two consecutive tasks t_i and t_j is $P_{pp}^{t_i,t_j}$.

Now, let us consider that P is a path in which each sub-path between two consecutive tasks t_i and t_j is $P_{pp}^{t_i,t_j}$, but P is not feasible. Moreover, let us assume that the task sequence ts visited by P can be completed by following the corresponding shortest path P_{sp}^{ts} for ts. Therefore, *buildTree* must create a path in the tree representing ts. By contradiction, let us assume that ts is not created in *tree*. Let t_i and t_j be two consecutive tasks in P_{sp}^{ts} visited before the last visited task t_l. Let us further assume that *buildTree* is invoked for t_i (when t_i is the origin o), but it is not invoked for t_j. This means that none of the lines 11, 16 or 22 was reached for t_j, which in turn can happen in one of the following cases:

1. $TC(P_{sp}^{ts}[s:t_i]) + TC(P_{sp}^{t_i,t_j}) + \delta(t_j) + TC(P_{sp}^{t_j,d}) > b$ (line 2).
   In this case, none of the lines above can be reached. However, this inequality is a contradiction to the fact that P_{sp}^{ts} is a feasible path.

2. The condition on line 3 is satisfied and thus line 22 is not reached

Therefore, $P^{ts}_{pp}[s : t_i] \neq null$ and $\mathcal{P}^{t_i}_{pp}$ contains P^{t_i,t_j}_{pp}. Whether the condition on line 4 is satisfied or not, t_j is added as a child of t_i (lines 10 and 14) and *buildTree* is recursively called for t_j (lines 11 and 16), which is a contradiction to the assumption that *buildTree* is not invoked for t_j.

3. The condition on line 3 is not satisfied and thus lines 10 and 14 are not reached

   However, in this case, line 22 is reached. t_j is added as a child of t_i and *buildTree* is recursively called tor t_j.

Now, let us assume that t_j is the task visited immediately before the last visited task t_l in P^{ts}_{sp}. When *buildTree* is called for t_j, t_l must be added as a child of t_j and d must be added as a child t_l, otherwise there will be no complete path from s to d in tree that visits the task sequence in P^{ts}_{sp}. The only case in which d will not be added as a child of t_l is when the condition on line 4 is satisfied. However, this is a contradiction to the assumption that P is not feasible.

Finally, let us consider the case where the task sequence in P^{ts}_{sp}. i.e., ts, is added to *tree*, but is then discarded (lines 26-28). This means that for some task t_i in ts, its sub-tree is null. However, as shown above, if t_i is not the last task in ts, there will be an edge in the tree connecting t_i to next task t_j in ts. Similarly, if t_i is the last task in ts, there will be an edge connecting t_i to d. Therefore, in none of the cases above t_i is a leaf. □

Once the tree is built, the remaining non-dominated paths, in terms of detour and reward, are computed by invoking *expandTree*. *expandTree* follows a similar procedure as *EXCT_REC-SP* (Algorithm 8), except that it expands a tree, as opposed to a graph. Algorithm 15 shows the pseudo-code of *expandTree*, which recursively expands a given path P until the budget is exceeded. Initially P contains only the starting point s (line 9 of Algorithm 13). In each call of *expandTree*, we analyze the feasible paths that can be obtained by expanding P following the task sequences in *tree*. We first check whether the destination can be reached within the remaining budget from the last vertex v of P (line 2). If not, P is not further expanded and the recursion stops. Otherwise, we consider the edges that connect v to any other node u in the tree (line 5). For each such u, we examine all non-dominated paths, in terms of travel and detour costs, from v to u (line 6). For each non-dominated path P^i_{nd}, we check whether its travel cost exceed the remaining budget (line 8). If so, P^i_{nd} is not further examined. Otherwise, we create a new path P_u by extending P with the path P^i_{nd}. If the current child u of v is the destination d (line 12) and P_u is not dominated (line 13), P_u is added to $\mathcal{S}$ and any path dominated by it is removed from $\mathcal{S}$ (lines 14-15). We note that P_u is not further expanded since it is a path to the destination. On the other hand, if u is a task (line 18), *expandTree* is recursively called for P_u and considering the remaining budget $remainingBudget = b - TC(P^i_{nd}) - \delta(u)$, i.e., the budget after deducting the cost to reach u through P^i_{nd} and to complete u.

For the tree illustrated in Figure 3.6f, *expandTree* will examine the paths shown in Figure 3.7. The only feasible path, shown in bold, is $P = \langle s, t_2, t'_3, d\rangle$ with reward \$9 and detour 21. Since it is not dominated in the current skyline set $\mathcal{S}$, it is added to $\mathcal{S}$, which now contains all non-dominated paths.

Complexity Analysis. In the worst case, *expandTree* generates all possible permutations of the tasks in T. However, *expandTree* may explore more than one path for each such permutation, since it considers a set of non-dominated paths connecting two nodes. Let $maxND$ be the maximum number of non-dominated paths between any two vertices. The maximum number of

Algorithm 15: *expandTree*

```
Input: Tree to be expanded $tree$, graph $G = (V, E)$, final destination $d$,
       budget $b$, path to be expanded $P$ and current skyline set $\mathcal{S}$.
1  $v \leftarrow$ last vertex of $P$
2  if $TC(P_{sp}^{v,d}) > b$ then
3      return
4  end
5  for each child $u$ of tree do
6      $\mathcal{P}_{nd} \leftarrow nonDominatedPaths(v, u, G, PP)$
7      for $P_{nd}^i$ in $\mathcal{P}_{nd}$ do
8          if $TC(P_{nd}^i) > b$ then
9              continue
10         end
11         $P_u \leftarrow$ extend $P$ with $P_{nd}^i$
12         if $u = d$ then
13             if $P_u$ is not dominated in $\mathcal{S}$ then
14                 $\mathcal{S}.add(P_u)$
15                 remove any paths dominated by $P_u$ from $\mathcal{S}$
16             end
17         end
18         else
19             $remainingBudget \leftarrow b - TC(P_{nd}^i) - \delta(u)$
20             $expandTree(tree, G, d, remainingBudget, P_u, \mathcal{S})$
21         end
22     end
23 end
```

paths generated by *expandTree* is given by $\sum_{k=0}^{|T|-2} \frac{(|T|-2)!}{k!} \times maxND^{(|T|-k-1)} = O(|T|! \times maxND^{|T|})$. For each generated path, the skyline set can be updated once, which has a time complexity of $O(|\mathcal{S}|)$. Additionaly, in the worst case, we have to compute the non-dominated paths between each pair of tasks. Since there are at most $|T|^2$ such pairs, the total complexity of such computation is $O(|T|^2 \times C_{NDP})$, where C_{NDP} is the time complexity of *nonDominatedPaths* (discussed next).

Lemma 4. *expandTree* finds all remaining feasible non-dominated paths, in terms of reward and total detour, that are not found by *buildTree*.

Proof. Let us consider that P is a feasible non-dominated not found by *buildTree*. By contradiction, let us assume that P is also not found by *expandTree*. There are two cases to be analyzed:

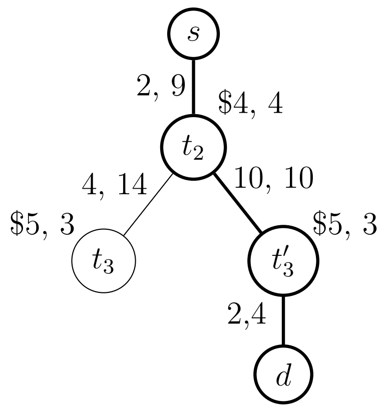


Figure 3.7: Paths explored by *expandTree* for the tree shown in Figure 3.6f. The feasible paths from s to d are shown in bold.

1. P was built by *expandTree*, but was discarded

   This means that the condition on line 13 of Algorithm 15 was not satisfied. Therefore, P is dominated in $\mathcal{S}$, which is a contradiction to the assumption that P is non-dominated.

2. P was not built by *expandTree*

   In this case, for some task t_j in P, the sub-path $P[s : t_j]$ was not built by *expandTree*. Let t_i be the task visited immediately before t_j in P and let us assume that the sub-path $P[s : t_i]$ was built by *expandTree*. When *expandTree* was called for $P[s : t_i]$, $P[s : t_j]$ was not built if one of the following conditions was satisfied:

   (a) There is no edge connecting t_i to t_j in *tree*.

   Since P is feasible and was not added to the skyline set $\mathcal{S}$ in *expandTree*, then there is a path in *tree* representing the task sequence visited by P, as proved in Lemma 3. As t_i and t_j are visited in sequence in P, there must be an edge connecting t_i and t_j in *tree*, which is a contradiction to the assumption that such edge does not exist.

   (b) There is no path in $\mathcal{P}_{nd}^{t_i,t_j}$ representing the path connecting t_i to t_j

in P.

Since all paths in $\mathcal{P}_{nd}^{t_i,t_j}$ are non-dominated, in terms of detour and travel costs, the path $P[t_i : t_j]$ connecting t_i to t_j in P is dominated. This means that there is another path P^{t_i,t_j} from t_i to t_j that dominates $P[t_i : t_j]$. Thus, by substituting $P[t_i : t_j]$ with P^{t_i,t_j} in P we can obtain a feasible path P_{nd} which has smaller travel and detour costs than P and has the same reward, since both paths visit the same set of tasks. Therefore, P is dominated by P_{nd}, which is a contradiction to the assumption that P is a non-dominated path.

□

In order to compute the required non-dominated paths, in terms of travel and detour costs, between a pair of nodes within Algorithm 15, Algorithm 16 (*nonDominatedPaths*) is invoked. If the non-dominated paths between the given origin o and target t have already be computed, they do not need to be computed again and the previously computed result is returned (lines 1-2). Otherwise, if $P_{sp}^{o,t} = P_{pp}^{o,t}$, no further paths need to be computed since any path between o and t is dominated by $P_{sp}^{o,t}$ and $P_{pp}^{o,t}$, and thus it sufficient to return $P_{sp}^{o,t}$ (or $P_{pp}^{o,t}$). If none of the conditions above is satisfied, the corresponding non-dominated paths are computed. The detour cost of the shortest path $P_{sp}^{o,t}$ between the origin o and the target t is used as an upper bound *maxDetour* (line 10) to the detour cost of any non-dominated path from o to t. Similarly, the travel cost of $P_{pp}^{o,t}$ is used as an upper bound *maxCost* (line 11) to the travel cost of any non-dominated path from o to t. Note that any path from o to t that yields a greater detour than *maxDetour* or a greater travel cost than *maxCost* is dominated by $P_{sp}^{o,t}$ or $P_{pp}^{o,t}$, respectively, thus it can be discarded. The algorithm maintains a queue Q which stores paths that are dequeued in increasing order of displacement detour cost w.r.t. PP. At each step, a path

P is dequeued from Q. If the last vertex v of P has already been found with smaller cost (line 20), meaning that there is a path P_j from o to v with smaller cost *and* detour than P, then P can be discarded. Note that P is dominated by P_j and it can be safely pruned since, as proved in [43], a sub-path of a non-dominated path is also a non-dominated path. Therefore, expanding P does not lead to a non-dominated path from o to t. On the other hand, if P is a path to t and is non-dominated (lines 26-27), P is added to the set of feasible non-dominated paths $\mathcal{P}_{nd}^{o,t}$ and all paths dominated by P are removed from $\mathcal{P}_{nd}^{o,t}$ (lines 28-29). Moreover, in this case, *maxCost* can also be updated (line 30) since all subsequent paths must have a smaller travel cost than that of P in order to be non-dominated. If none of the conditions above applies to P, it is expanded with each neighbor u of v in E, creating a new path P_u. If the estimated cost to the destination $lb = TC(P_u) + \frac{d_e(u,d)}{maxSpeed}$ is not greater than the maximum cost *maxCost*, P_u is added to Q (line 40). The algorithm stops when Q is empty (line 14) or when the detour yielded by a dequeued path P is greater than *maxDetour* (lines 16-17).

Complexity Analysis **(*nonDominatedPaths*).** *nonDominatedPaths* performs a network expansion similar to the one in *shortestPathToSet*, except that each vertex may be inserted into Q and expanded more than once in *nonDominatedPaths*. In the worst case, *nonDominatedPaths* finds all simple paths connecting o and t in G. In a complete graph, a vertex v can be expanded up to $\sum_{k=0}^{|V|-3} \frac{(|V|-3)!}{k!} = e(|V|-3)!$ times, i.e., considering all permutations of any number of vertices. Therefore, taking into account the queue operations, the complexity in the worst case is $O(|V|! \times (|E| + |V| \times \log |V|))$. However, we note that road networks are sparse graphs and therefore the algorithm is expected to perform much better in practice. Moreover, the detour and travel costs of a

Algorithm 16: *nonDominatedPaths*

Input: Origin o, target t, graph $G = (V, E)$ and preferred path PP.

Output: Set $\mathcal{P}_{nd}^{o,t}$ of feasible non-dominated paths, in terms of travel cost and detour, from o to t.

```
if $\mathcal{P}_{nd}^{o,t}$ has been previously computed then
    return $\mathcal{P}_{nd}^{o,t}$
end
$\mathcal{P}_{nd}^{o,t} \leftarrow \emptyset$
if $P_{sp}^{o,t} = P_{pp}^{o,t}$ then
    $\mathcal{P}_{nd}^{o,t} \leftarrow \mathcal{P}_{nd}^{o,t} \cup P_{sp}^{o,t}$
    return $\mathcal{P}_{nd}^{o,t}$
end
$maxDetour \leftarrow DDPP(P_{sp}^{o,t}, PP)$
$maxCost \leftarrow TC(P_{pp}^{o,t})$
$Q, foundCost \leftarrow \emptyset$
$Q.add(\langle o \rangle)$
$foundCost.put(o, 0)$
while $Q \neq \emptyset$ do
    $P \leftarrow Q.dequeue()$
    if $DDPP(P, PP) > maxDetour$ then
        return $\mathcal{S}$
    end
    $v \leftarrow$ last vertex of $P$
    if $v$ has already been found with smaller cost than $TC(P)$ then
        continue
    end
    else
        $foundCost.put(v, TC(P))$
    end
    if $v = t$ then
        if $P$ is non-dominated then
            $\mathcal{P}_{nd}^{o,t}.add(P)$
            remove any paths dominated by $P$ from $\mathcal{S}$
            $maxCost \leftarrow TC(P)$
        end
        continue
    end
    for each $u$ neighbor of $v$ in $E$ do
        $P_u \leftarrow$ extend $P$ with $u$
        $lb \leftarrow TC(P_u) + \frac{d_e(u,d)}{maxSpeed}$
        if $lb > maxCost$ then
            continue
        end
        $Q.add(P_u)$
    end
end
return $\mathcal{P}_{nd}^{o,t}$
```

path from o to t are limited by $maxDetour$ and $maxCost$, respectively, which means that it is very unlikely that all simple paths from o to t will be explored by *nonDominatedPaths*.

Lemma 5. The algorithm *nonDominatedPaths* finds all feasible non-dominated paths, in terms of travel and detour costs, from a given origin o to a target t.

Proof. By contradiction, let us assume that there is a feasible non-dominated path P_i from o to t that is not found by *nonDominatedPaths*.

Let us first consider the simple case where $P_{sp}^{o,t} = P_{pp}^{o,t}$ (line 5). Since P_i is not found, then $P_i \neq P_{sp}^{o,t}$. P_i is not dominated by $P_{sp}^{o,t}$ if $DDPP(P_i, PP) < DDPP(P_{sp}^{o,t}, PP)$. However, since by assumption $P_{sp}^{o,t} = P_{pp}^{o,t}$, then $P_{sp}^{o,t}$ yields the same detour as $P_{pp}^{o,t}$, which is the minimum possible. Therefore, $DDPP(P_i, PP) < DDPP(P_{sp}^{o,t}, PP)$ does not hold. Similarly, P_i is not dominated by $P_{pp}^{o,t}$ if $TC(P_i) < TC(P_{pp}^{o,t})$, which is not possible since $P_{pp}^{o,t} = P_{sp}^{o,t}$. Therefore, both cases contradict the assumption that P_i is non-dominated.

Now, let us assume that $P_{sp}^{o,t} = P_{pp}^{o,t}$ does not hold. Let us analyze the possible cases:

1. P_i was added to Q

    (a) P_i was not dequeued

    In this case Q is not empty, but the algorithm terminated before P_i could be dequeued. This means that for some dequeued path P, the algorithm reached line 16, meaning that $DDPP(P, PP) > maxDetour$. Since the paths are examined in increasing order of detour, $DDPP(P_i, PP) > DDPP(P, PP) > maxDetour$. Therefore, P_i is dominated by the shortest path $P_{sp}^{o,t}$, which contradicts the assumption that P_i is a non-dominated path.

(b) P_i was dequeued

Therefore, when P_i was dequeued from Q one of the following conditions was satisfied: (a) $DDPP(P_i, PP) > maxDetour$ or (b) t was already found with smaller cost. Condition (a) is a contradiction to the assumption that P_i is non-dominated and thus $DDPP(P_i, PP) \leq maxDetour$. If condition (b) is satisfied, then there is a path P_j from o to t which is shorter than P_i *and* yields a smaller detour, since it was dequeued before. Therefore, P_i is dominated by P_j, which contradicts the assumption that it is a non-dominated path.

2. P_i was not added to Q

This means that for some vertex u in P_i, the sub-path $P_i[o : u]$ was not added to Q. Let v be the vertex visited immediately before u in P_i and let us assume that the sub-path $P_i[o : v]$ is dequeued from Q. $P_i[o : u]$ is discarded in one of the following cases:

(a) v was previously found with a smaller cost than that of $P_i[o : v]$ and thus $P_i[o : v]$ was not expanded.

This means that there is another path from o to v with smaller cost and detour than $P_i[o : v]$. Therefore, $P_i[o : v]$ is dominated and, consequently, P_i is also dominated, which contradicts the assumption that P_i is a non-dominated path.

(b) $P_i[o : v]$ was expanded, but $lb = TC(P_i[o : u]) + \frac{d_e(u,d)}{maxSpeed} > maxCost$.

Then there is a path P_j, such that $TC(P_j) = maxCost$, which was found before $P_i[o : u]$. Since paths are examined in increasing order of detour, then $DDPP(P_i[o : u], PP) > DDPP(P_j, PP)$.

As $\frac{d_e(u,d)}{maxSpeed}$ is a lower bound to $TC(P_{sp}^{u,d})$, then $TC(P_i[o : u]) + TC(P_{sp}^{u,d}) > maxCost$ also holds. This means that $TC(P_i) > maxCost$ and, thus, P_i is dominated by P_j, which is a contradiction to the assumption that P_i is a non-dominated path.

□

Now that all algorithms used within $EXCT\text{-}PP$ have been presented, we analyze its total time complexity and prove that the skyline set returned by it is the exact one.

Complexity Analysis **(*$EXCT\text{-}PP$*).** In the first phase of the $EXCT\text{-}PP$ algorithm, *$shortestPathToSet$* and *$minDetourPathToSet$* are invoked up to $|T| + 1$ times each. Each of such calls has a time complexity of $O(|E|+|V|\times\log|V|)$ in the worst case. Therefore, the total complexity of this phase is $O(|T|\times(|E|+|V|\times\log|V|))$. As discussed above, the time complexity of *$buildTree$* and *$expandTree$* are $O(|T|!\times|\mathcal{S}|)$ and $O(|T|!\times maxND^{|T|}\times|\mathcal{S}|+|T|^2\times C_{NDP})$, respectively. Since $O(|T|\times(|E|+|V|\times\log|V|)) = O(|T|^2\times C_{NDP})$ and $O(|T|!\times|\mathcal{S}|) = O(|T|!\times maxND^{|T|}\times|\mathcal{S}|)$, the total time complexity of $EXCT\text{-}PP$ is $O(|T|!\times maxND^{|T|}\times|\mathcal{S}|+|T|^2\times C_{NDP})$ in the worst case.

Theorem 8. The skyline set $\mathcal{S}$ found by $EXCT\text{-}PP$ is the exact one.

Proof. We need to prove that the set $\mathcal{S}$ returned by $EXCT\text{-}PP$ includes all non-dominated paths, i.e., it is complete, and also that no dominated path is part of $\mathcal{S}$, i.e., it is correct.

By contradiction, let us first assume that there is a feasible non-dominated path P from s to d not in $\mathcal{S}$. This means that P is not found by *$buildTree$* nor by *$expandTree$*. However, as proved in Lemma 3, if P is a path in which each sub-path between two consecutive tasks t_i and t_j is $P_{pp}^{t_i,t_j}$, then P is found by *expandTree*. If this is not the case, P is found by *expandTree*, as proved in Lemma 4, since it is feasible and non-dominated. This contradicts the assumption that P is not found.

Next, we prove that no dominated path is part of $\mathcal{S}$. By contradiction, suppose that there is a dominated path P_d in $\mathcal{S}$. Since as proved above all non-dominated paths are part of $\mathcal{S}$, P_d can not belong to $\mathcal{S}$ since it is dominated by at least one path in $\mathcal{S}$ and all dominated paths are removed from $\mathcal{S}$. This is a contradiction to the assumption that P_d is part of $\mathcal{S}$. $\square$

Heuristic Solutions

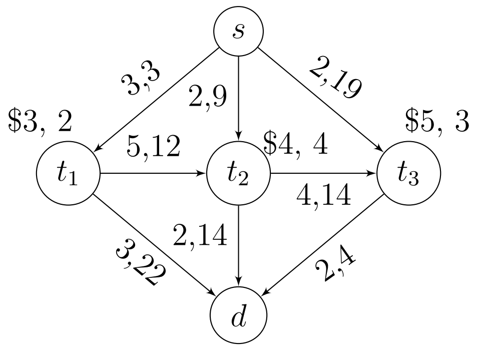
Due to the hardness of the IRTS-PP problem, the exact approach does not scale to large sized instances. Therefore, we developed a few heuristics that approximate the exact skyline by prioritizing the path that yields the minimum detour between tasks. Based on that, we first propose a Detour Oriented heuristic (*DOH*). Next, we show how variants of the *kGH-SP*, *MDH-SP* and *MRH-SP* heuristics, presented in Section 23, can be used for finding approximate skylines for the IRTS-PP problem.

Detour Oriented Heuristic (DOH). The Detour Oriented Heuristic (*DOH*) is based on a graph of tasks $GT_{pp} = (V_{pp}, E_{pp})$ similar to GT_{sp}, but where each edge (in E_{pp}) connecting two vertices v and u represents the path between v and u that yields the minimum detour w.r.t. PP. Besides containing s and d, V_{pp} includes the set of feasible tasks from T that can be completed considering that one travels between two locations along the path that yields the minimum detour w.r.t. PP.

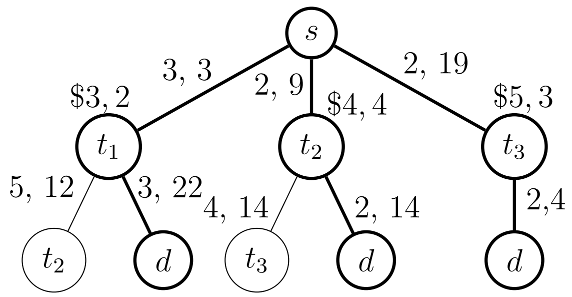
In order to build GT_{pp} we follow a procedure similar to one for building GT_{sp} (Algorithm 5), except that instead of invoking *shortestPathToSet* on lines 3 and 11, we call *minDetourPathToSet* (also used in *EXCT-PP*). As discussed above, *minDetourPathToSet* finds the paths yielding the minimum detour from a given vertex $v \in V$ to a set of target vertices. In the context of GT_{pp}'s construction, it is first used for finding the tasks $t_i \in T$ that can be completed with cost up to b by following the path P_{pp}^{s,t_i} from s to t_i. Next, it is used for finding all feasible tasks t_j that can be reached from a feasible task t_i with cost up to the remaining budget $b - TC(P_{pp}^{s,t_i}) - \delta(t_i)$ by following the path $P_{pp}^{t_i,t_j}$. The graph obtained for the example shown in Figure 3.1 is illustrated in Figure 3.8a. We note that, in the worst case, the time complexity for building GT_{pp} is the same as the one for building GT_{sp}.

DOH finds all feasible paths in GT_{pp} and stores the non-dominated ones in the approximate skyline set $\mathcal{S}_{app}$. Paths are expanded from s and pruned if their travel cost exceeds the given budget. In fact, in order to find $\mathcal{S}_{app}$, *DOH* follows the same procedure described in the *EXCT-SP* algorithm, except that the graph given as input to Algorithm 7 is GT_{pp} (rather than GT_{sp}), and the recursion stops when $TC(P_{pp}^{v,d}) > b$ (line 2 of Algorithm 8), i.e., when the destination d cannot be reached within the given budget by traveling along the path from v (last vertex of P) to d that yields the minimum detour w.r.t. PP.

Figure 3.8b shows the paths explored by *DOH* for the graph GT_{pp} shown in Figure 3.8a. *DOH* first finds the feasible path $P_1 = \langle s, t_1, d \rangle$ with detour 8 and reward \$3. Since P_1 is not dominated, it is added to the approximate skyline set $\mathcal{S}_{app}$. Next, path $P_2 = \langle s, t_2, d \rangle$ is found with detour 8 and reward \$4. Since its reward is greater than that of P_1 and both paths yield the same detour, P_2 is added to $\mathcal{S}_{app}$ and P_1 is removed (as it is now dominated by P_2). Then, path $P_3 = \langle s, t_3, d \rangle$ is found with detour 7 and reward \$5. Since it dominates P_2, it is added to $\mathcal{S}_{app}$ and P_2 is discarded. We note that *DOH* does not produce exact results. In this example, it does not find path $P = \langle s, t_2, t_3, d \rangle$, which is part of the exact skyline. The travel cost of P in GT_{pp} is equal to 34, which exceeds the budget $b = 32$, meanwhile it would be possible to find a path including t_2 and t_3 in the original graph G that does not exceed the budget.



(a) Graph GT_{pp} built for the example shown in Figure 3.1. The values on each edge represent the corresponding detour and travel costs, respectively.



(b) Paths explored by DOH for the graph GT_{pp} shown in Figure 3.8a. The feasible paths from s to d are shown in bold.

Figure 3.8: Graph GT_{pp} built for the example shown in Figure 3.1 and paths explored by DOH for such graph.

Complexity Analysis **(*DOH*).** In the worst case, DOH generates all possible permutations of tasks of size 1 to $|T|$ if each task is connected to every other task in GT_{pp}. Therefore, its complexity in this case is similar to *EXCT-SP*'s.

kGH-PP, MDH-PP and MRH-PP. We also use variants of the *kGH-SP*, *MDH-SP* and *MRH-SP* approaches, presented in Section 23, as heuristics for finding approximate skylines for the IRTS-PP problem. These variants, denoted *kGH-PP*, *MDH-PP* and *MRH-PP* are similar to *kGH-SP*, *MDH-SP* and *MRH-SP*, respectively, except that they consider that the path taken between two tasks t_i and t_j is $P_{pp}^{t_i,t_j}$, rather than $P_{sp}^{t_i,t_j}$.

More specifically, in *kGH-PP* the corresponding graph GT_{pp}^{k} is built by connecting each task t_i to its k closest tasks in terms of detour, which, intuitively, have a greater chance of leading to paths with shorter deviation from the preferred path PP. In order to find the non-dominated paths in this reduced graph, we follow the same procedure as DOH, with the only difference being the graph given as input to Algorithm 7, which in this case is GT_{pp}^{k} (as opposed to GT_{pp}). We note that $kGH\text{-}PP$ has the same time complexity as $kGH\text{-}SP$ in the worst case.

In $MDH\text{-}PP$, each path P is expanded with the feasible task that yields the minimum displacement detour. $MDH\text{-}PP$ is similar to $MDH\text{-}SP$ (Algortihm 9) except that the path taken between two tasks t_i and t_j is $P_{pp}^{t_i,t_j}$, as opposed to $P_{sp}^{t_i,t_j}$. Such paths are computed by invoking $minDetourPathToSet$ whenever $shortestPathToSet$ and $shortestPath$ are called within Algortihm 9. Moreover, the vertex u selected on line 8 of Algorithm 10 is the one such that $DDPP(P_{pp}^{v,u}, PP)$ is minimized. We note that in order to find u, it is not possible to follow the same procedure as $getClosestTask$ (Algorithm 11). More specifically, $\frac{d_e(v,t_i)}{maxSpeed}$ is not a lower bound to $DDPP(P_{pp}^{v,t_i}, PP)$. Therefore, $\frac{d_e(v,t_i)}{maxSpeed}$ can not be used for finding promising candidates, as we do in $getClosestTask$. Thus, we follow a procedure similar to $minDetourPathToSet$, i.e., we examine paths in G in increasing order of displacement detour w.r.t. PP. The network expansion stops when the first task t_i that can be completed within the remaining budget is found. t_i is then returned and used for expanding the current path P within Algorithm 10. In the worst case, $minDetourPathToSet$ is invoked by $MDH\text{-}PP$ up to $|T| + |T|^2$ times, therefore, in this case, the time complexity of $MDH\text{-}PP$ is $O(|T|^2 \times (|E| + |V| \times \log |V|))$.

Finally, as in $MRH\text{-}SP$, $MRH\text{-}PP$ also selects the feasible task t_i with the highest reward when expanding a path P. However, the travel cost from the last vertex v of P to t_i is given by the cost of path P_{pp}^{v,t_i}. *MRH-PP* follows the same procedure as *MRH-SP* and both algorithms have the same time complexity in the worst case.

3.5 Experiments

We evaluated the performance of our approaches, as well as the accuracy of the approximation algorithms, varying several parameters and using real road networks. The real datasets used in our experiments reflect the road networks of Amsterdam (AMS), Oslo (OSLO) and Berlin (BER), as of March/2017 [3]. In order to have a somewhat realistic set of task locations we use the location of eateries (restaurants and coffee shops) on those networks as locations of (pseudo) tasks. Table 3.4 summarizes the details of the datasets used in our experiments and Figure 3.9 illustrates the used road networks and the location of tasks onto them. The travel time of each edge $(v, u) \in E$ is computed as the Euclidean distance $d_e(v, u)$ divided by an average speed of 40 km/h. It is worth emphasizing that, although this is a simplistic assumption, the approaches proposed in this chapter work for any graph where the travel time of each edge is a function of its length.

	Amsterdam	**Oslo**	Berlin
#vertices	106,599	**305,174**	428,768
#edges	130,090	**330,632**	504,228
#tasks	824	**958**	3,083

Table 3.4: Summary of the real datasets used in our experiments (**bold** defines default values).

Table 3.5 shows the parameters varied in our experiments, besides the real datasets. The preferred path's cost ranges from 15 min to 60 min. We consider that the shortest path between two locations is selected as the traveler's pre-

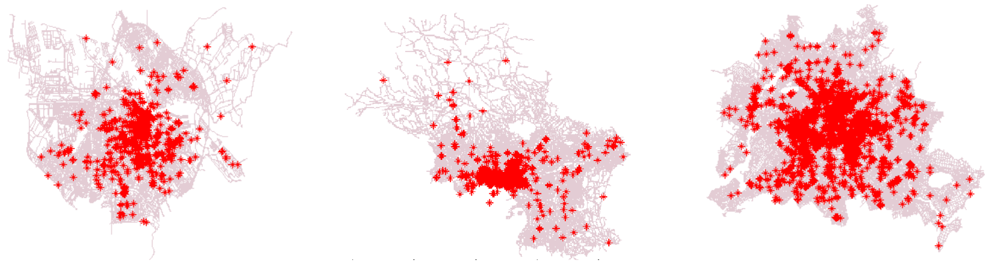

(a) Amsterdam (b) Oslo (c) Berlin

Figure 3.9: Locations of tasks in Amsterdam, Oslo and Berlin (overlaid on those cities' road networks).

ferred path. The worker's starting point s is randomly selected among all the vertices of the network. Then, we perform a Dijkstra search to select the destination d. More specifically, the network is expanded from s and, as soon as the distance from s to a vertex v removed from the queue exceeds the given travel time, the expansion stops and v becomes the destination. It is important to stress that selecting the preferred path as the shortest path connecting s and d is not a requirement, in fact *any* path could be used, we resorted to using shortest paths only for simplicity. While by doing that we do have $SP = PP$, it is also important to note that what differentiates IRTS-SP from IRTS-PP is not the initial path itself but the notion of detour cost ($TDSP(\cdot, SP)$ and $TDPP(\cdot, PP)$, respectively).

Parameter	Range
Cost of the preferred path/shortest path [min]	15, **30**, 60
Time budget (in addition to $TC(PP)$ or $TC(SP)$) [min]	30, **60**, 90
Number of available tasks ($\|T\|$)	10, **20**, 40, 80
Average task completion time ($\delta(t_i)$)	5, **15**, 30

Table 3.5: Experimental parameters and their values (**bold** denotes default values).

We also assume that the worker has a time budget between 30 min to 90 min besides $TC(SP)$ or $TC(PP)$, depending on the IRTS variant being considered. Inspired by the experiments in [18], we also varied the number of tasks $|T|$, available to the worker, between 10 and 80 tasks selected (randomly and off-line) among all tasks that can actually be completed within the given budget. (We note that for all cases evaluated in this chapter there are at least $|T|$ feasible tasks.) We also varied the average time it takes for completing a task from 5 to 30 min by following a normal distribution where the average time for completion is the distribution's mean. Finally, we assume the rewards paid out to workers to be proportional to the time it takes for completing the corresponding task. Nonetheless, there is nothing in the approaches presented that would prevent one from using an arbitrary reward scheme.

For each set of experiments, we vary the value of one parameter, and fix the other parameters to their default values. Moreover, we ran 50 cases and report the average of the results. The experiments were performed in a virtual machine with Intel(R) Xeon(R) CPU E5-2650 (8 cores @ 2.30GHz) and 16GB RAM, running Ubuntu.

For each variant of the IRTS problem, we first report the processing time of all proposed approaches and the relative error [55] of the approximate skylines, found by the heuristic approaches, w.r.t. the corresponding exact skylines. In order to illustrate how this relative error is measured, consider Figure 3.10, which represents the exact and approximate skylines found by the $EXCT$-PP and DOH approaches, respectively, for the example in Figure 3.1 and considering the IRTS-PP variant. The area delimited by the line in bold represents the area dominated by the exact skyline. Since DOH does not find path PP_2, the skyline set found by it only covers the area delimited by the red dashed line. Therefore, the colored area represents the area that is dominated by the exact skyline, but is not dominated by the approximate one. The relative error measures how large such area is w.r.t. the area dominated by the exact skyline. More formally, let $A_{\mathcal{S}_{EXCT}}$ be the area dominated by the exact skyline, and, similarly, let $A_{\mathcal{S}_{APP}}$ be the area dominated by the approximate skyline. The normalized relative error is given by:

$$error = \frac{A_{\mathcal{S}_{EXCT}} - A_{\mathcal{S}_{APP}}}{A_{\mathcal{S}_{EXCT}}}$$

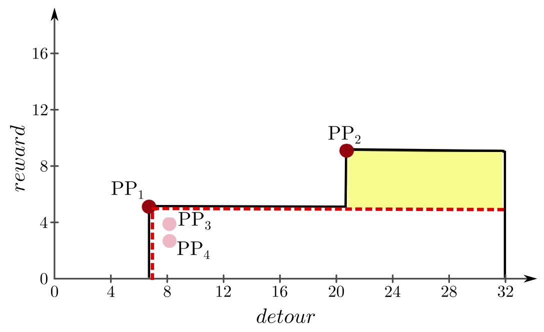


Figure 3.10: Relative error of an approximate skyline w.r.t. the corresponding exact skyline.

As we shall see in the following, the relative error of our approaches was never above 25%, in fact the majority of the time it was below 10%, thus all figures reporting that measure are scaled between 0 and 25% (rather than the usual 0-100%), in order to offer a more detailed view of its behavior.

In order to further inform the comparison of the quality of the heuristic approaches w.r.t the exact one, we also evaluated their *recall*, i.e., the percentage of non-dominated solutions (the skyline set) found by them, as well as their *precision*, i.e., percentage of non-dominated solutions in their result sets. We note that a low relative error does not necessarily mean a high precision and recall, since many points in the approximate skyline can be very close but yet distinct from the ones in the exact skyline, and thus the precision and recall obtained could be low.

Before presenting all results, recall that the kGH approach has k as an input parameter. In order to determine which value to use we ran preliminary experiments using different values of $k \in \{1, 2, 5, 10\}$. We observed that $k = 5$

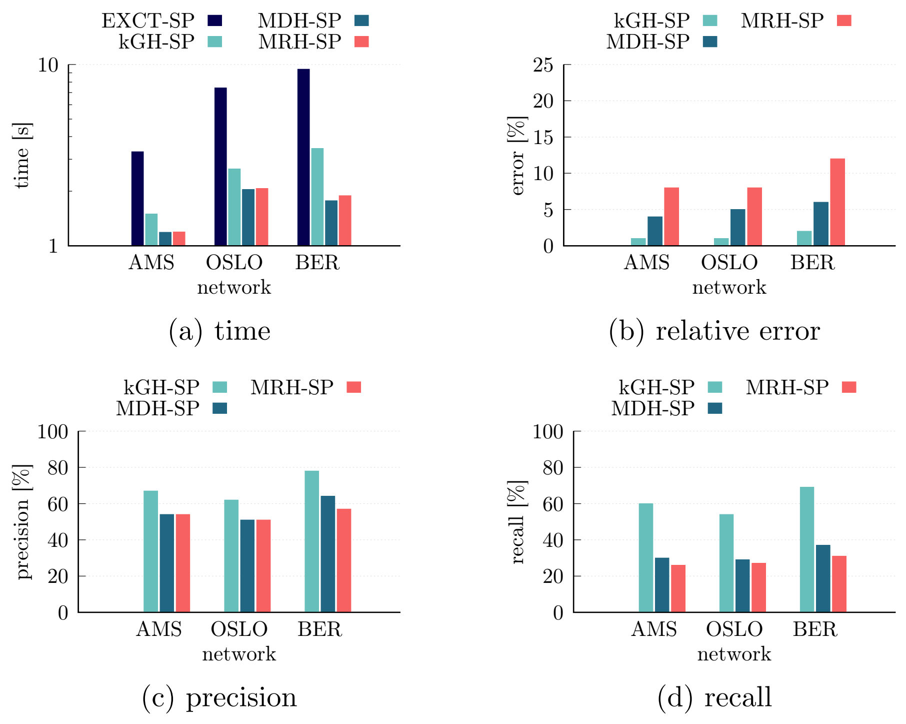
offers a good trade-off between processing time and quality of the results. Given those results, we decided to use $k = 5$ in all experiments that follow.

3.5.1 IRTS-SP

Effect of the road network

As shown in Figure 3.11a, *EXCT-SP* and *kGH-SP* are slower for the BER network. Such network is larger than the other ones, in terms of number of vertices and edges, which implies that it takes longer to compute the many shortest paths required by those approaches. On the other hand, *MDH-SP* and *MRH-SP* are slightly slower for the OSLO network. This is due to the fact that there is a higher concentration of tasks around the same area in such network (see Figure 3.9b). *MDH-SP* finds the closest task from a path by examining tasks in increasing order of Euclidean distance (divided by the maximum speed). Since tasks are more concentrated, the Euclidean distance to them will tend to be very similar, requiring *MDH-SP* to compute more shortest paths in order to find the actual closest task in terms of travel time. On the other hand, *MRH-SP* selects the task with the highest reward when expanding a path. Since tasks are closer to each other in the OSLO network, there is a higher chance that the selected task will also be close to the path and thus the remaining budget will tend to be larger. This, in turn, implies that more tasks can be visited in sequence, requiring more shortest paths to be computed.

Figure 3.11b shows that all heuristics present very low relative error, i.e., within 12%, when the network is varied. *MRH-SP* produces worse results than the other approaches. This is because it finds a smaller set of solutions (paths) than the other approaches. This implies that the area dominated by the skyline produced by *MRH-SP* will tend to be smaller if compared



(a) time (b) relative error

(c) precision (d) recall

Figure 3.11: Processing time, relative error, precision and recall for IRTS-SP w.r.t. the network.

to the result produced by the other approaches. We note that this behavior will be observed repeatedly in other experiments. Moreover, the relative error of *MRH-SP* is slightly higher for the BER network. This approach selects tasks based only on their reward. If the selected task does not lead to a path belonging to the exact skyline, the created path P_d will be dominated by a path P_{nd}. Since the tasks are more scattered in the BER network, the difference between the travel time of P_{nd} and of P_d will tend to be large, thus increasing the difference between the areas dominated by the exact skyline and the one produced by *MRH-SP*.

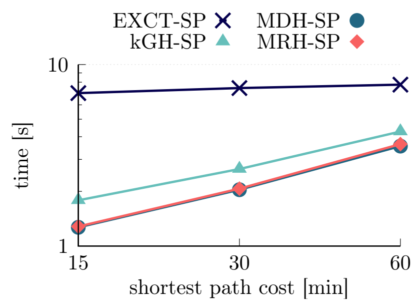
Figures 3.11c and 3.11d show that all heuristic approaches produce better results, in terms of precision and recall, for the BER network. Since the tasks are more scattered in that network, less tasks will tend to be reachable from a particular task within the remaining budget. Therefore, since the heuristic approaches for IRTS-SP consider a subset of these feasible tasks, the difference between the ones considered by *EXCT-SP* and by the non-exact approaches becomes smaller, increasing the chances of choosing the ones that lead to non-dominated paths.

It is also worth noting that *kGH-SP* is much faster than *EXCT-SP* while producing better results than *MDH-SP* and *MRH-SP*, especially in terms of recall. This is because *kGH-SP* examines more task sequences than the greedy heuristics, increasing the chance of finding the ones belonging to the exact skyline.

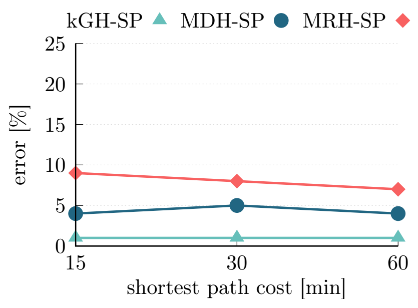
Effect of the cost of the shortest path

Figure 3.12a shows that, as expected, the processing time of our approaches increases with the travel cost from s to d since it takes longer to compute the required shortest paths. However, *EXCT-SP* is less affected by this parameter than the other approaches. This is due to the fact that the other approaches compute the shortest path from each feasible task t_i to d by performing an A* search which guides the expansion towards d. The farther d is, the longer it takes to compute such shortest paths. On the other hand, in *EXCT-SP*, the shortest paths from t_i to every other feasible task t_j and to d are computed in a single network expansion which only stops when the budget is exceeded. The time taken by such expansion increases slightly when the travel cost to d increases.

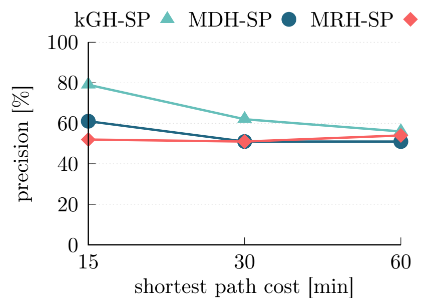
The precision and recall of the *kGH-SP* and *MDH-SP* approaches decrease with the shortest path cost, as shown in Figures 3.12c and 3.12d, respectively. A longer path also means a greater budget, which implies that paths can be expanded with tasks that are farther away and still be feasible. However, *kGH-SP* and *MDH-SP* prioritize closer tasks and thus may not



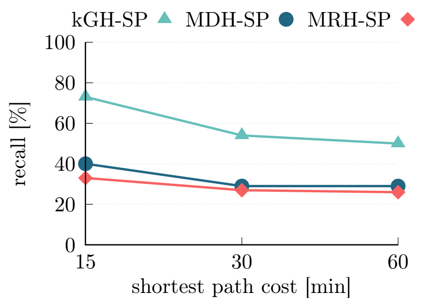
(a) time



(b) relative error



(c) precision



(d) recall

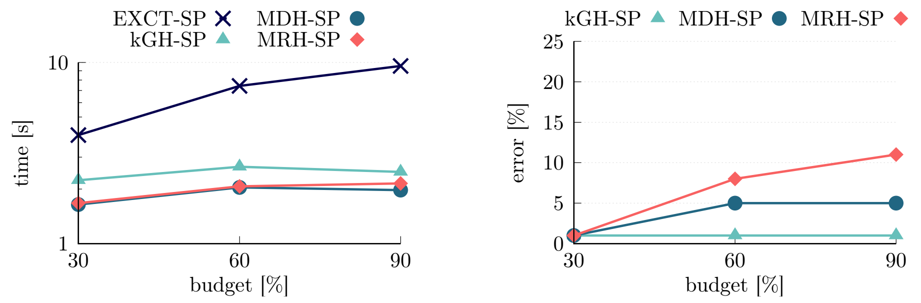
Figure 3.12: Processing time, relative error, precision and recall for IRTS-SP w.r.t. the shortest path cost.

examine some paths containing tasks that are farther away from each other, but that can be completed within the given budget.

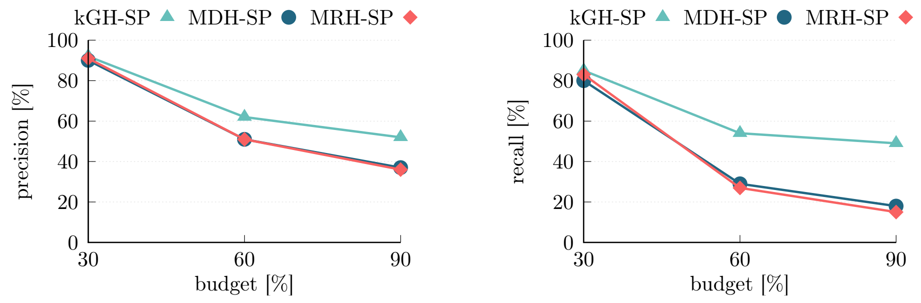
MRH-SP presents a behavior different than *kGH-SP* and *MDH-SP* in terms of relative error and precision, as shown in Figures 3.12b and 3.12c, respectively. Since it is driven by reward, it may select tasks that are farther from the current path. However, as discussed above, the exact skyline may contain paths in which tasks are farther away from each other (as long as the reward is high enough). Therefore, in this case, *MRH-SP* has a higher chance of producing solutions that are closer (or equal) to the ones in the exact skyline. This explains why the relative error of such approach tends to decrease slightly with the cost of the shortest path, while its precision increases slightly. On the other hand, the number of paths in the exact skyline tends to increase with the cost of the shortest path. Thus, there is a higher chance that more of those paths are not examined by $MRH\text{-}SP$, which affects its recall, as shown in Figure 3.12d .

Effect of the budget

As expected and shown in Figure 3.13a, the processing time of all approaches increases with the budget, simply because there are more feasible task sequences to be considered within a larger budget. We also note that $EXCT\text{-}SP$ is the most affected solution due to the higher number of permutations of tasks it examines, which rapidly increases with the number of tasks, as we will show in the next experiment.



(a) time (b) relative error



(c) precision (d) recall

Figure 3.13: Processing time, relative error, precision and recall for IRTS-SP w.r.t. the budget.

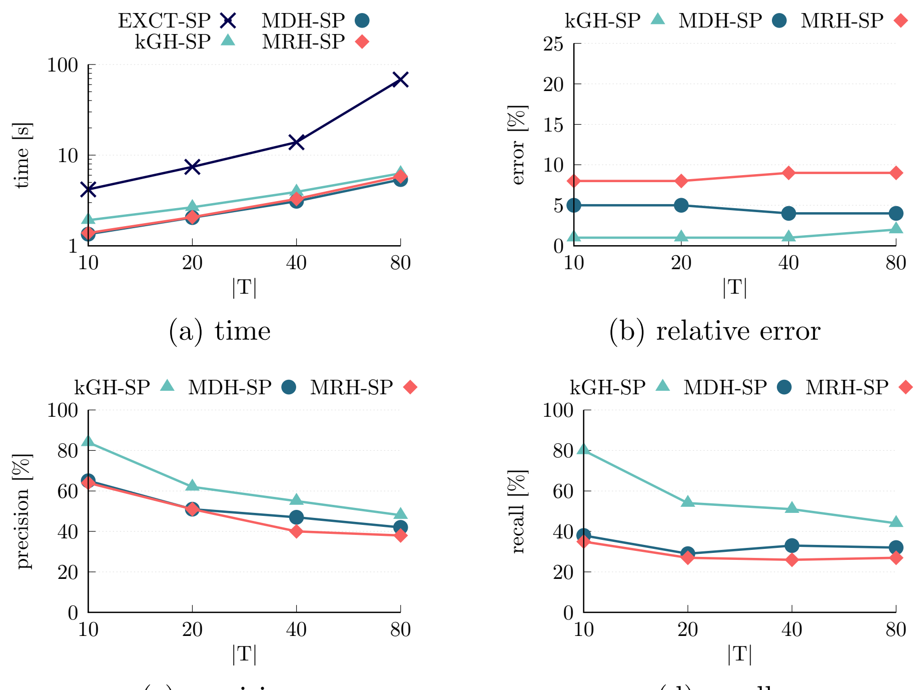
The error of the greedy heuristics tends to increase with the budget, as shown in Figure 3.13b, even though it is still below 10% for most cases. A larger budget means that more tasks can be performed in sequence, however, the greedy approaches may not be able to produce some longer paths, in terms of number of tasks, due to the poor choices they can make. Although MDH-SP selects the closest task when expanding a path, in a further expansion the remaining options may not be good, thus consuming a lot of the budget and not leaving enough time for other tasks to be performed. Similarly, MDH-SP selects tasks based only on their reward, potentially leading to poor choices in terms of travel time. Therefore, the difference between the area dominated by the skylines found by these approaches and the area dominated by the exact skyline will tend to increase. On the other hand, since kGH-SP explores more options when expanding a path, it has a higher chance of producing paths that are more similar to the non-dominated ones, even when the budget increases.

As shown in Figures 3.13c and 3.13d, the precision and recall of all heuristic approaches decrease with the budget. Intuitively, the higher the number of feasible task sequences, the greater the chance that some of them will not be considered by the heuristics.

Effect of $|T|$

Figure 3.14a shows that the processing time of all approaches increases with the number of feasible tasks, with $EXCT$-SP being more affected by that parameter. This is due to the high number of permutations of tasks that such approach may check when looking for non-dominated paths, which rapidly increases with the number of tasks.

Figure 3.14b suggests that the relative error of the results produced by our heuristics is not significantly affected by the number of tasks, being always below 10%. As shown in Figures 3.14c and 3.14d, the precision and recall of all heuristics tend to decrease with the number of tasks. Moreover, kGH-SP is the most affected approach. This can be explained by the fact that in such



(a) time (b) relative error

(c) precision (d) recall

Figure 3.14: Processing time, relative error, precision and recall for IRTS-SP w.r.t. $|T|$.

approach each task is only connected to its k closest tasks in the graph GT_{sp}^{k}. However, the number of feasible neighbors of a given task increases with the number of tasks. This means that more neighbors are not even considered, and, consequently, more feasible combinations of tasks will not be examined by *kGH-SP*.

Effect of the time for completing a task

Figure 3.15a shows that the processing time of all approaches decreases with the time it takes for completing a task because there are less feasible task sequences to be examined by each approach. *EXCT-SP* is the approach most affected by this parameter due to the high number of task sequences it has to evaluate, which increases when tasks take a shorter time to be performed.

As shown in Figures 3.15b, 3.15c and 3.15d the quality of results produced

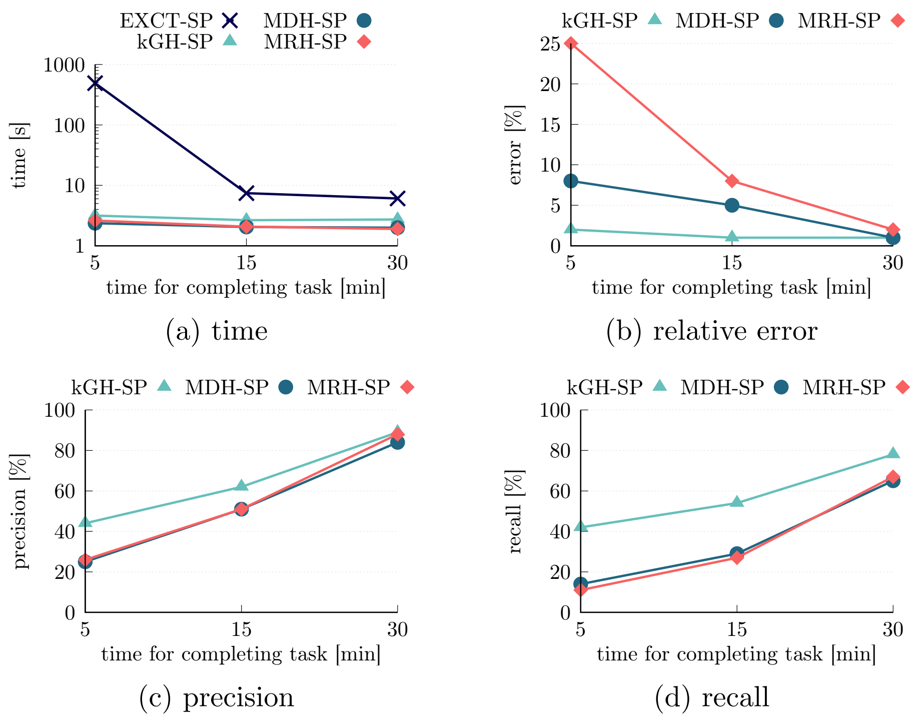


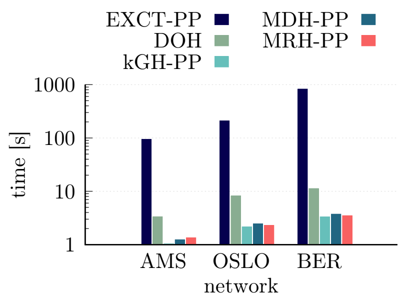
Figure 3.15: Processing time, relative error, precision and recall for IRTS-SP w.r.t. the for completing a task.

by our heuristics tends to increase with the time it takes for performing a task. This is because, the longer the tasks take to be performed, the lower will be the remaining budget after completing a task. This means that there will be less tasks that can be completed within such a budget. Therefore, since there are fewer options to choose from, there is a greater chance of picking a task that leads to non-dominated paths.

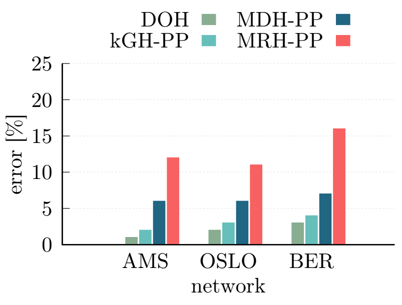
3.5.2 IRTS-PP

Effect of the road network

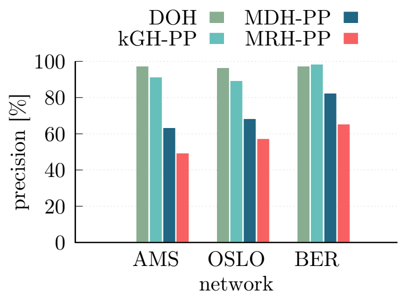
As shown in Figure 3.16a, although all approaches are affected by the network in terms of processing time, $EXCT\text{-}PP$, unsurprisingly, is the most affected. This is due to the high number of task sequences and the multiple paths between each pair of tasks it examines. Moreover, all approaches are slower for the BER network because it takes longer to compute the paths required by each approach in a larger network.



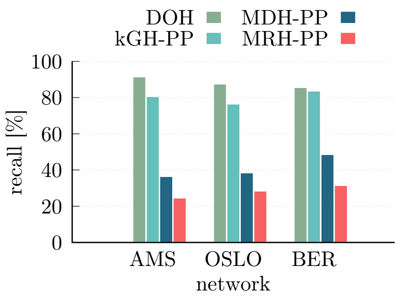
(a) time



(b) relative error



(c) precision



(d) recall

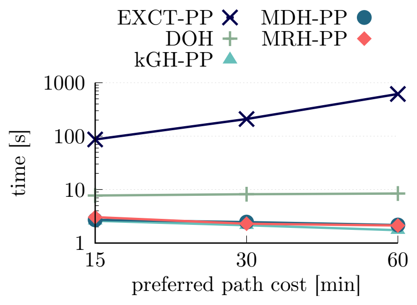
Figure 3.16: Processing time, relative error, precision and recall for IRTS-PP w.r.t. the network.

As the result obtained for *MRH-SP*, *MRH-PP*'s relative error is worse for the BER network, as shown in Figure 3.16b. This is also due to the distribution of tasks in such network. We also note that, although in general *DOH* produces better results than the other heuristics, in some cases *kGH-PP* may be slightly more precise than *DOH*. This is the case for the BER network, as shown in Figure 3.16c. *DOH* generates all non-dominated paths also generated by *kGH-PP*, however, its skyline set tends to be larger than the one produced by *kGH-PP* since it examines more task sequences. Due to a more scattered distribution of tasks in the BER network, the distance between some pairs of tasks may be greater. Since *DOH* considers only one path between each pair of tasks, i.e., the one yielding the minimum detour, the travel cost of some of those paths may exceed the budget, even though the same task sequences could be completed by following a shorter path. Therefore, *DOH* discards such task sequences, which may be part of he exact skyline, and thus paths dominated by the corresponding non-dominated paths will remain in its skyline set, which explains its lower precision. This is also the reason why *DOH*'s recall is lower for the BER network. On the other hand, similarly to the results obtained for IRTS-SP (and for the same reason), the other heuristic approaches produce results with higher precision and recall for the BER network, as shown in Figures 3.16c and 3.16d, respectively.

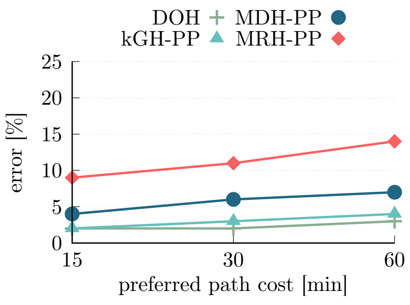
Effect of the cost of the preferred path

Figure 3.17a shows that while the processing time of *EXCT-PP* and *DOH* increases with the cost of the preferred path, *kGH-PP*, *MDH-PP* and *MRH-PP* are faster for longer paths. A longer preferred path also means a larger budget, which implies that more task sequences can be completed. Moreover, the number of non-dominated paths, in terms of travel and detour costs, between two tasks tends to increase with the budget. Since *EXCT-PP* may compute all such paths and it examines all feasible task sequences, its performance degrades fast with the cost of the preferred path. As *DOH* examines all feasible task sequences that can be completed by following the path that yields the minimum detour, and the number of such paths increases with the budget, *DOH*'s processing time also tends to increase with the cost of the preferred path.

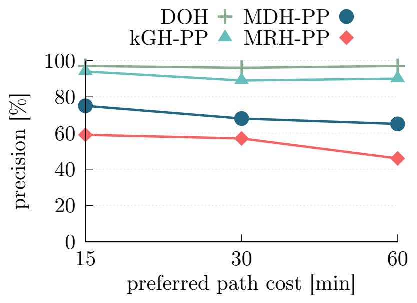
A larger budget also means that the worker can travel more in exchange for a shorter detour from the preferred path. As *kGH-PP* and *MDH-PP* prioritize paths that yield shorter detours, they may expand a path with a task that yields a short detour but not a small travel cost, thus consuming



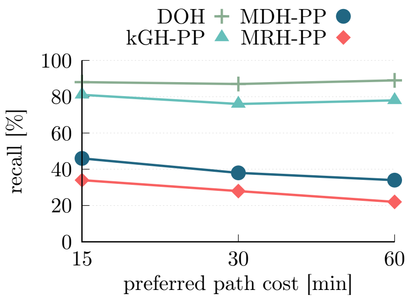
(a) time



(b) relative error



(c) precision



(d) recall

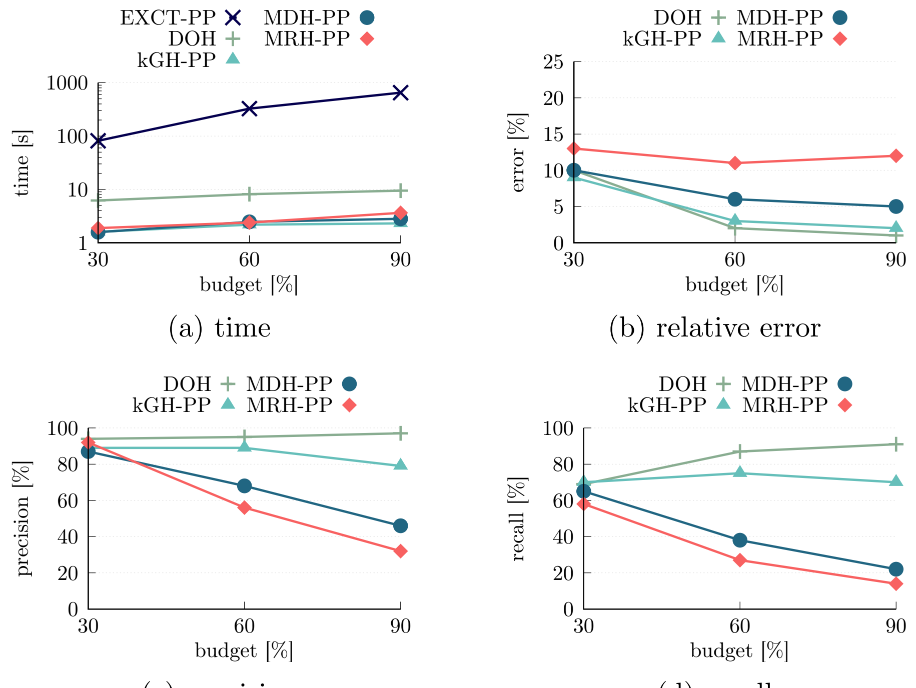
Figure 3.17: Processing time, relative error, precision and recall for IRTS-PP w.r.t. preferred path cost.

more from the budget. The remaining budget may not be large enough to complete more tasks and thus the path expansion is terminated. On the other hand, although *MRH-PP* prioritizes tasks with high reward, more tasks are feasible when the budget is larger. Therefore, *MRH-PP* may select farther tasks and the remaining budget may not be large enough for the path to be further expanded.

The quality of the results produced by the heuristics tends to decrease with the cost of the preferred path, as shown in Figures 3.17b, 3.17c and 3.17d. The number of paths in the exact skyline increases with the travel cost from s to d. Therefore, there is a higher chance that more of such paths will not be considered by the heuristics approaches.

Effect of the budget

As expected and observed in Figure 3.18a, the processing time of all approaches increases with the budget, simply because there are more feasible task sequences to be considered within a larger budget. As in the previous experiments, $EXCT\text{-}PP$ is the most affected approach due to the higher number of task sequences and sub-paths it examines.



(a) time (b) relative error

(c) precision (d) recall

Figure 3.18: Processing time, relative error, precision and recall for IRTS-PP w.r.t. the budget.

The error of the heuristic approaches tends to decrease with the budget, as shown in Figure 3.18b, being most of the time below 10%. A larger budget means that the worker can travel more in exchange for a shorter detour from the preferred path. This benefits the heuristics since they prioritize paths that yield shorter detours even if they are longer, thus generating paths closer to the ones in the exact skyline. Figures 3.18c and 3.18d show that, while the precision and recall increase for DOH as the budget increases, these values decrease for *kGH-PP*, *MDH-PP* and *MRH-PP*. This is due to the fact that the number of tasks that can be completed potentially increases with the budget. Since *kGH-PP* only extends a path with k tasks, the number of feasible tasks not considered by it in such expansion increases with the budget, thus decreasing the chance of producing paths that belong to the exact skyline. Similarly, *MDH-PP* and *MRH-PP* only extend a path with a single task and the likelihood of making a poor choice increases with the number of tasks. On the other hand, *DOH* considers all tasks that can be completed by following the path yielding the minimum detour w.r.t. *PP*, which explains why *DOH* performs better.

Effect of $|T|$

Figure 3.19a shows that, as expected, the processing time of all approaches increases with the number of tasks, but *EXCT-PP* is the most affected one. This is due to the high number of permutations of tasks that this approach may check when looking for non-dominated paths, which increases rapidly with the number of tasks.

Figure 3.19b shows that while the relative error of *MRH-PP* slightly increases with the number of tasks, the error of the other approaches slightly decreases. This can be explained by the fact that *MRH-PP* selects tasks based only on their reward, and the likelihood of choosing a task that leads to paths farther from the ones in the exact skyline increases with the number of tasks. On the other hand, since the other heuristics prioritize detour, the likelihood of producing paths closer to the ones in the exact skyline increases with the number of task options.

Figure 3.19c shows that the precision of all approaches tends to decrease with the number of tasks. Due to a higher number of task options, the heuris-

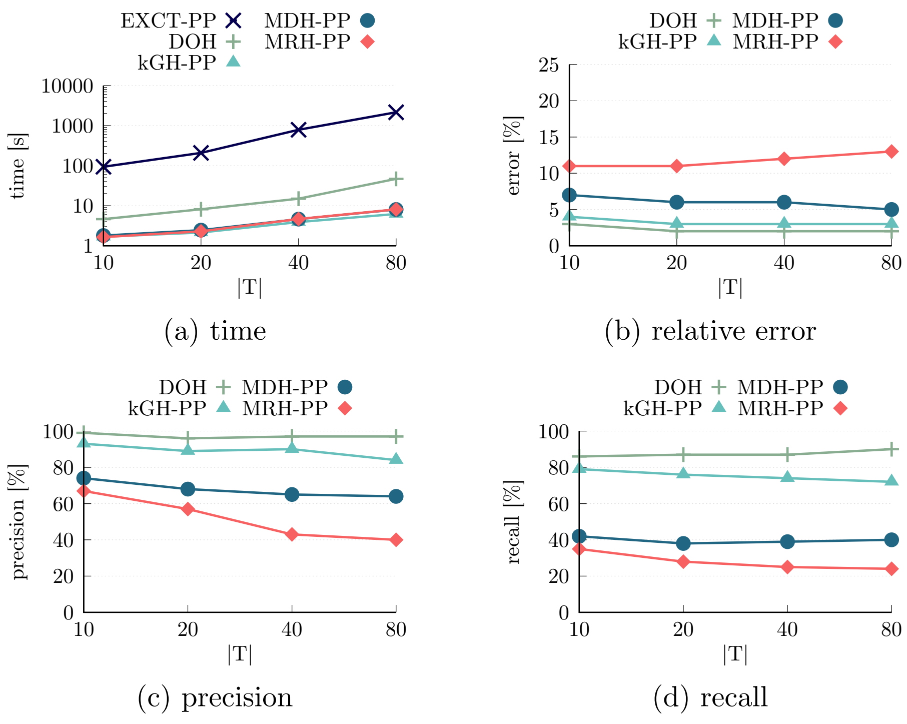
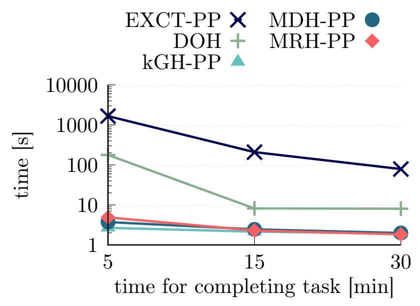


Figure 3.19: Processing time, relative error, precision and recall for IRTS-PP w.r.t. $|T|$.

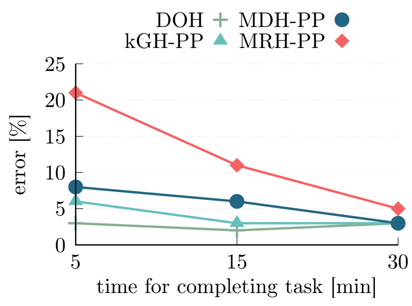
tics are more likely to produce task sequences that do not belong to the exact skyline. As shown in Figure 3.19d, *DOH*'s recall slightly increases with the number of tasks, while the recall of all other approaches decreases. More task sequences are examined by *DOH* when there are more tasks available. Therefore, the probability of generating paths that belong to the exact skyline is higher. On the other hand, the number of paths examined by the other heuristics is limited, and the likelihood of examining feasible task sequences that belong to the exact skyline decreases with the number of available tasks.

Effect of the time for completing a task

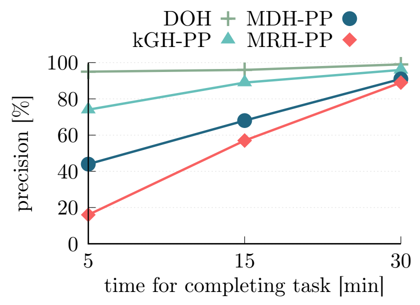
Figure 3.20 shows that the processing time of all approaches decreases with the time it takes for completing a task because there are less feasible task sequences to be examined by each approach. *EXCT-PP* and *DOH* are more affected by this parameter due to the higher number of task sequences they evaluate, which increases when tasks take a shorter time to be performed. Additionally, *EXCT-PP* may also examine multiple paths between pairs of tasks.



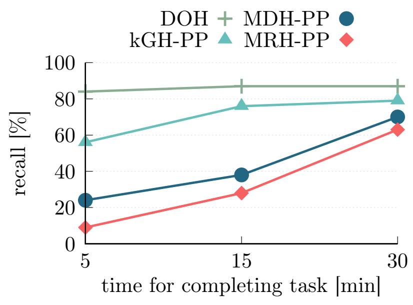
(a) time



(b) relative error



(c) precision



(d) recall

Figure 3.20: Processing time, relative error, precision and recall for IRTS-PP w.r.t. the time for completing a task.

As shown in Figures 3.20b, 3.20c and 3.20d the quality of results produced by our heuristics tends to increase with the time it takes for performing a task. This is because, the longer the tasks take to be performed, the lower will be the remaining budget after completing a task. This means that there will be less tasks that can be completed within such a budget. Therefore, since there are fewer options to choose from, there is a greater chance of picking a task that leads to non-dominated paths. However, *DOH* is only slightly affected by this parameter, while *MRH-PP* is drastically affected. Again, this is because the chance of *MRH-PP* making a poor choice increases with the number of feasible tasks, while DOH examines all feasible task sequences in GT_{pp}.

3.6 Conclusion

In this chapter we presented two variants of the *In-Route Task Selection* (IRTS) problem, which is a new and practically relevant problem in the context of spatial crowdsourcing. The IRTS-SP problem considers that the worker's initial route is the shortest path connecting his/her starting point and destination, whereas the IRTS-PP problem considers that the worker has an arbitrary preferred path instead, likely due to non-objective criteria. In both cases the worker is willing to consider the trade-off between a limited detour and rewards collected by completing tasks during such detour. Given the competing nature of those two criteria, we investigated this problem using the skyline paradigm, and after proving the NP-hardness of the problems, we proposed a few heuristic approaches and also analysed their complexity.

For the case of IRTS-SP, our experimental results, using real datasets at the city scale, showed that $kGH\text{-}SP$ is the best alternative among the heuristics. It produces consistently better results than the greedy heuristics, while being only slightly slower. In the case of IRTS-PP, the results showed that the heuristic guided by the detour's cost (DOH) can obtain solutions with low relative error rates. However, even though it is at least one order of magnitude faster than the exact approach $EXCT\text{-}PP$, its processing time can increase rapidly with the number of tasks and when tasks take a shorter time to be completed. In such cases, $kGH\text{-}PP$ is a better alternative since it can produce results closer to the ones produced by DOH, while still being slightly faster than the greedy heuristics.

Chapter 4

Online In-Route Task Selection in Spatial Crowdsourcing

4.1 Introduction

Crowdsourcing is a computing paradigm which relies on the contributions of a large number of workers to accomplish tasks, such as image tagging and language translation. The increasing popularity of mobile computing led to a shift from traditional web-based crowdsourcing to spatial crowdsourcing [49].

Spatial crowdsourcing consists of location-specific tasks, which require people to physically be at specific locations to complete them. Examples of these tasks include taking pictures, answering questions about a certain location in real time or perform physical chores. Tasks are assigned to suitable workers based on one or more objectives, such as maximizing the number of assigned tasks, e.g. [32], [48], maximizing a given matching score, e.g. [45], [50], minimizing the total amount of reward paid out by task requesters, e.g. [17] or maximizing the net reward earned by workers after deducting traveling costs, e.g. [9].

In our previous work [6] we defined a new type of spatial crowdsourcing query, namely In-Route Task Selection (IRST) query. Consider a city's road network, a worker's given path in such a network, and a static number of known tasks in the network, each yielding a reward and, likely, a detour from the worker's original path. Assuming that it would be in the best interest of the worker to maximize the rewards collected while minimizing the detour from the worker's original path, the IRTS query seeks to find *all* solutions that are optimal under *any* given trade-off between those two criteria. In this chapter we deal with a similar scenario, but considering a fundamental difference: *tasks appear dynamically.* That is, any devised solution has now to suggest paths to workers without any knowledge of tasks that may appear in the future. We refer to this new problem as the *Online In-Route Task Selection* (Online-IRTS) query.

In order to illustrate the Online-IRTS query consider the simple scenario shown in Figure 4.1, which contains four tasks with their corresponding release and expiration time, time for completion and reward. For instance, task t_1 is released at time 0, expires at time 7, requires 1 time unit to be completed and yields a reward of \$1. Let us assume that the worker is at s at time 0 and will travel towards d, originally using the shortest path, but is willing to deviate from such path to complete tasks as long as d is reached by time 16. Moreover, assume that the travel cost for one grid edge is one time unit and the distance between two points is given by their Manhattan distance. Finally, we denote the reward associated with a path by $R(\cdot)$ and the time detour of that path by $DT(\cdot)$, i.e., the difference between the total time taken by the given path, including its travel time and the time required for completing the tasks in it, and the cost of the original (shortest) path.

At time 0, there are two tasks available to be performed: t_1 and t_2. This

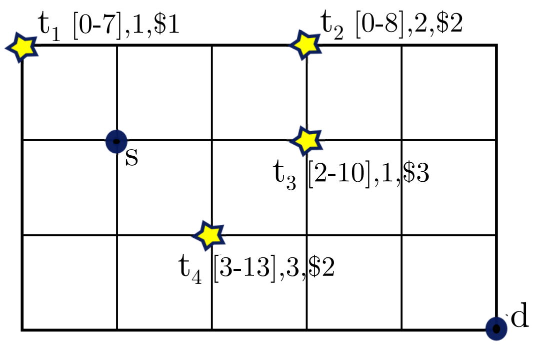


Figure 4.1: Sample scenario where the worker's starting point is s, destination is d, and tasks t_1, t_2, t_3 and t_4 (denoted by star vertices) with their corresponding [release time-expiration] range, time for completion and reward. The travel cost for one edge on the grid is one time unit.

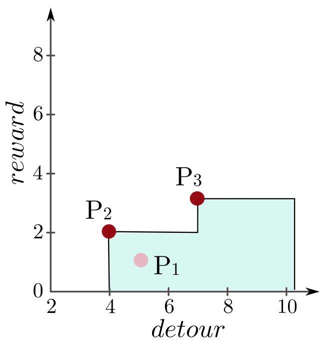
leads to three possible paths[1] shown in the upper half of Table 4.1. Note that some paths are simply not feasible, thus not considered, e.g., path $\langle t_2, t_1 \rangle$ is not a feasible option because the worker would arrive at t_1 at time 8, whereas t_1 expires at time 7. In the context of skylines, path P_1 is dominated by path P_2 since P_2 offers a higher reward for a shorter detour. Therefore, the skyline set at time 0 contains paths P_2 and P_3. This is graphically illustrated in Figure 4.2a, in which the shaded area contains the dominated paths. It should be noted, however, that the approach proposed in [6] would completely ignore tasks that could become available at a later time, i.e., t_3 and t_4 in the example above, and would return $\{P_2, P_3\}$ as the query's answer.

In this chapter, we consider tasks that may appear at a subsequent iteration. More specifically, we (1) return the skyline set at time 0 to the worker, who in turn (2) chooses a path and performs the next task in it, and then we (3) update the skyline set considering the worker's selection as well as the new available tasks. For instance, considering the scenario above, let us assume that among the paths returned to the worker, i.e. $\{P_2, P_3\}$ (recall that P_1 is dominated), he/she prefers path P_3 because he/she is interested in the reward

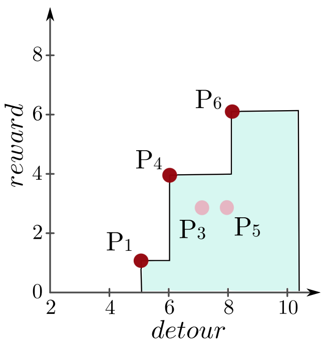
[1]We make the reasonable assumption that the worker wants to complete at least one task, otherwise there would be no point in considering the Online-IRTS query.

Time	*Potential Path*	$R(\cdot)$	$DT(\cdot)$
$t = 0$	$P_1 = \langle t_1 \rangle$	1	5
	$P_2 = \langle t_2 \rangle$	2	4
	$P_3 = \langle t_1, t_2 \rangle$	3	7
$t = 3$ (only paths containing t_1 need to be considered)	$P_1 = \langle t_1 \rangle$	1	5
	$P_3 = \langle t_1, t_2 \rangle$	3	7
	$P_4 = \langle t_1, t_3 \rangle$	4	6
	$P_5 = \langle t_1, t_4 \rangle$	3	8
	$P_6 = \langle t_1, t_2, t_3 \rangle$	6	8

Table 4.1: Possible paths considering tasks available at times 0 and 3. $R(.)$ denotes the reward associated with a path and $DT(.)$ denotes its total time detour.



(a) Skyline at time 0.



(b) Skyline at time 3.

Figure 4.2: Skyline for the example illustrated in Figure 4.1 at time 0 and at time 3 (assuming that the worker has travelled to task t_1).

offered by such path. After traveling to t_1 and completing it at time 3, two new tasks became available: t_3 and t_4. Therefore, there are new paths for the worker to contemplate, as illustrated in the lower half of Table 4.1. At this point, the worker may choose to go directly to the destination or travel along the originally selected path P_3 or P_4, P_5 or P_6. Clearly, in the context of skyline queries, and as illustrated in Figure 4.2b, P_3 is not an interesting option anymore since it deviates more from the original path and yields a lower reward than P_4, i.e., it is dominated by P_4. Similarly, P_5 is also dominated by P_3 and P_6. Now, let us assume that, among the remaining paths $\{P_1, P_4, P_6\}$, the worker prefers P_4. After performing task t_3, no new tasks became available and no existing task can be completed within the remaining budget, therefore the worker travels to the destination d on path $P_4 = \langle t_1, t_3 \rangle$, thus receiving a total reward of \$4 for a time detour of 6 time units w.r.t. the original (shortest) path. It is important to note that the answer to the Online-IRTS query is a *single* route, which is built incrementally; whereas in [6] the solution for the *offline* IRTS is a *skyline*, i.e., a set of routes, oblivious to tasks that may become available at a later time.

As shown in the example above, even though Online-IRTS uses the notion of dominated paths as the guideline to offer solutions to the worker, at each point he/she makes decisions, i.e., choose tasks to complete, that will likely affect potential future choices. Moreover, since during the process of constructing a path Online-IRTS has no knowledge about tasks that may appear in the future, we can only compute skylines based on the tasks that are known at the time a decision has to be made. Therefore, due to those reasons, *there is no guarantee that an "optimal" path is produced*, i.e., a path belonging to the exact skyline set $\mathcal{S}^*$ found by an optimal offline solution that knows all tasks beforehand. For instance, in the example shown in Figure 4.1 an optimal "omniscient" offline solution would find a skyline set containing all the paths shown in Table 4.2. The solution returned by the Online-IRTS query, $P_4 = \langle t_1, t_3 \rangle$, *considering the worker's choices*, is not part of the exact offline skyline set. Clearly, P_4 is dominated by P_2^*, which yields a reward of \$5 and a time detour of 5. However, the choice made at time 3, i.e. completing task t_3, cannot be retracted thus leading to a sub-optimal solution w.r.t. the offline solution.

Given the discussion above, it is important to have some metric to quantify the quality of a solution for the Online-IRTS query. Let us assume that $\mathcal{S}^*$ contains all optimal paths w.r.t. any combination of the two cost criteria assuming all tasks and their properties were known beforehand, and that P_i

Path	$R(\cdot)$	$DT(\cdot)$
$P_1^* = \langle t_3 \rangle$	3	1
$P_2^* = \langle t_2, t_3 \rangle$	5	5
$P_3^* = \langle t_1, t_2, t_3 \rangle$	6	8
$P_4^* = \langle t_2, t_3, t_4 \rangle$	7	10

Table 4.2: Exact skyline set found by an optimal offline solution.

is the obtained answer to the Online-IRTS query. We propose to measure the quality of P_i by evaluating how much it could be improved w.r.t. the paths in $\mathcal{S}^*$, as illustrated in Figure 4.3. As discussed above, path P_4 is dominated by path P_2^*. The area dominated by P_2^* is represented by the rectangle delimited by the black dashed line. Clearly, such area is larger than the one dominated by P_4, delimited in red. The metric we propose measures how large the difference between those areas is, i.e., how much room there is for improvement. Such metric is motivated by the fact that even though the worker is the one who chooses which task to perform next as he/she progresses in his/her path, obtaining a final path that dominates a relatively large area also means that good choices were offered to the worker in each iteration.

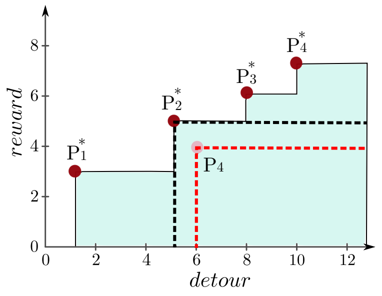


Figure 4.3: Skyline found by an offline optimal approach vs the path selected by the worker for the example shown in Figure 4.1.

The main contributions of this chapter, besides the formulation of the Online-IRTS query itself, are two heuristic approaches that build feasible paths *incrementally*, i.e., task after task, with the goal of ultimately approximating a path in the exact skyline set found by an (omniscient) offline optimal approach.

The first approach, named Local Optimum Heuristic (LOH), computes locally optimal sklyines after each task is executed, whereas the second, Incremental Heuristic (IH), explores a tree of non-dominated paths. Our extensive experiments using realistic city-scale datasets show that while LOH tends to produce better solutions than IH, it can also be several times slower, thus allowing the user to make an informed decision on which approach to use depending on how he/she prioritizes effectiveness or efficiency, and, therefore, being better served by LOH or IH, respectively.

The remainder of this chapter is structured as follows. In Section 4.2 we present relevant related works and contrast them to ours. We present some preliminaries needed to formalize the Online-IRTS problem in Section 4.3. Our proposed heuristic algorithms, which form the core of our contribution, are presented in Section 4.4, followed by their experimental evaluation in Section 4.5. Finally, Section 4.6 concludes this chapter with a summary of our findings.

4.2 Related Work

The literature in spatial crowdsourcing presents many different ways to assign tasks to workers, for instance, maximizing the number of assigned tasks [32], [48], maximizing a given matching score [10], [45], [50], [60], or minimizing the total amount of reward paid out by requesters, while maximizing the number of assignments [17]. In what follows we focus on works more related to the Online-IRTS query. A more detailed discussion of other works related to the topic of spatial crowdsourcing in general can be found in Section 3.2.

The works presented in [18], [20] deal with assigning a task sequence to workers and, thus, take the travel cost between tasks into account, similarly to what we do. In [18], the authors focus on maximizing the number of tasks performed by a single worker, while in [20] they aim at maximizing the number of tasks performed by multiple workers. The focus on both works is on a single optimization criteria, whereas we consider two (worker's reward and detour). Those works also do not consider an online environment, but rather a static one where all tasks are known beforehand.

Within the context of multi-criteria optimization, Cheng et al. [10] study the reliable diversity-based spatial crowdsourcing problem. The main differences between their work and ours are: (a) they focus on maximizing the reliability and diversity of tasks, whereas we focus on the worker's perspective and aim at maximizing his/her reward while minimizing detour, (b) they aim at assigning several workers at different times to accomplish each task, whereas we assign a single worker to a task, (c) they assign a single task to each worker and thus do not compute a task schedule for workers, as we do, and, finally, (d) they consider the general direction a worker moves to filter out tasks that are not in the worker's travel direction, while in our case the worker specifies a particular destination and can move freely in the network subjected only to his/her temporal budget.

The Online-IRTS problem is related to the In-Route Nearest Neighbor (IRNN) query [44] and the Orienteering Problem (OP) [23]. IRNN queries search for nearest neighbors with respect to a preferred path. Within the context of bi-criteria optimization in IRNN queries, [2], [28] aim at finding optimal trade-offs between the costs incurred for visiting a *single* point of interest (POI). The OP problem aims at finding the route from a given starting point that maximizes a total score while the total travel cost does not exceed a given budget. The main differences between Online-IRTS and IRNN and OP are the following. Unlike Online-IRTS, IRNN and OP focus on an offline scenario. IRNN considers deviating towards one single POI, while Online-IRTS considers multiple tasks. Finally, the OP problem does not consider any notion of trade-off between travel cost and rewards.

Finally, the most similar works to ours are [6], [35]. [6], as mentioned in the previous section, proposes the (offline) IRTS query in which the skyline set of paths is computed considering only a set of static and known tasks, unlike in the Online-IRTS query where tasks are dynamic and skylines are used to compute a path "on-the-fly." Although [35] assumes that tasks appear dynamically, as we do, it aims at optimizing a single criterion, i.e., the number of completed tasks, while we consider the trade-off between detour and reward. Another essential difference between Online-IRTS and the problem studied in [35] is that in [35] the algorithm decides which task the worker should travel to next, whereas in Online-IRTS at each iteration the worker is provided with a set of interesting paths for his/her own decision.

4.3 Preliminaries

We assume that the worker's movement is constrained by an underlying road network, which is modeled as a directed graph $G(V, E, C)$, where V is a set of vertices that represent the road intersections and end-points, E is the set of edges modelling all road segments and C indicates the costs of edges in E. In our case, the cost of traversing an edge connecting vertices v_i and v_j is given by the time it takes to traverse the edge that connects those vertices and is denoted by $c(v_i, v_j)$.

We define a path $P_i = \langle v_i^1, v_i^2, ..., v_i^n \rangle$ in G as a sequence of vertices such that any two consecutive vertices v_i^j and v_i^{j+1}, for $1 \leq j < n$, are directly connected by an edge $(v_i^j, v_i^{j+1}) \in E$. The shortest path from the worker's starting point s to his/her destination d is denoted by SP.

A worker w is an individual who is willing to perform tasks in exchange for rewards while traveling from his/her origin s to the destination d. We assume that the worker departs from s at time τ_s and has a budget b which represents the maximum time he/she wishes to spend, including the time required to perform the selected tasks and the total travel time.

We also assume that all tasks are located on an edge of the network. If a given task t is not placed on an existing vertex $v \in V$, we replace that edge, say (v_j, v_l), in G with two new edges (v_j, t) and (t, v_l), adjusting the costs of the affected edges accordingly. Note that this implies that some of the vertices in the graph are now tasks rather than actual road intersections or the like. Thus, every vertex v has a non-negative reward $\rho(v)$, a finite time for completion $\delta(v)$, a release time $\tau_r(v) \geq 0$ and a finite deadline $\tau_d(v)$ associated to it, where $\rho(v) = 0$, $\delta(v) = 0$, $\tau_r(v) = 0$ and $\tau_d(v) = \infty$ if v does not represent a task. It is important to note that some vertices traversed in a path may represent tasks which are *not* chosen to be completed, and therefore their cost and reward should not be considered as part of the path's overall cost and reward. To address that, we add a binary variable, $\pi(v_i^j)$, to indicate whether a task at a vertex $v_i^j \in P_i$ was completed or not, that is, $\pi(v_i^j)$ is set to 1 iff v_i^j is a task *and* is completed, or to 0 otherwise.

Given the above, the reward of a path, travel and detour costs, and path feasibility can now be defined as follows.

Definition 4.3.1 (Reward of a path)**.** Given a path P_i in G, its total reward is given by the sum of the rewards of the completed vertices (tasks) in it (recall that vertices which are not tasks or are tasks that have not been completed do not contribute to the path's reward), i.e.,

$$R(P_i) = \sum_{v_i^j \in P_i} \rho(v_i^j) \times \pi(v_i^j).$$

Definition 4.3.2 (Travel Cost)**.** Given a path P_i in G, its travel cost is given by the sum of the costs of the edges in it, i.e.,

$$TC(P_i) = \sum_{j=1}^{n-1} c(v_i^j, v_i^{j+1}).$$

Definition 4.3.3 (Detour Cost)**.** Given a path P_i connecting s to d, the detour of P_i w.r.t. the shortest path SP is defined as the difference between the travel costs of P_i and SP plus the time necessary for performing the completed tasks in P_i:

$$DT(P_i) = TC(P_i) - TC(SP) + \sum_{v_i^j \in P_i} \delta(v_i^j) \times \pi(v_i^j).$$

Definition 4.3.4 (Feasible Path)**.** A path P_i is feasible, i.e., of potential interest to the worker, if its total time is not greater than the budget b, and if it satisfies the tasks temporal constraints. That is,

1. $TC(P_i) + \sum_{v_i^j \in P_i} \delta(v_i^j) \times \pi(v_i^j) \leq b$
2. For each completed task t_j in P_i:

$$a(t_j, P_i) \geq \tau_r(t_j) \;\; \& \;\; a(t_j, P_i) + \delta(t_j) \leq \tau_d(t_j)$$

Where $a(t_j, P_i)$ is the arrival time at task t_j by following P_i.

As mentioned earlier, we aim at providing the user with a set of interesting alternative feasible paths for different trade-offs between detour and reward. In order to do so, we rely on the notion of *skyline queries*, which was first introduced in [5]. Given a d-dimensional data set, a skyline query returns the points that are not dominated by any other point. In the context of the Online-IRTS query, a path P_i is not dominated if there is no other path P_j with smaller detour *and* higher reward than P_i. A powerful aspect of skyline queries is that the user does not need to determine beforehand weights for detour and reward. The skyline is a set of equally interesting solutions in the sense that they are all non-dominated, for arbitrary weights. A skyline set found in the context of the Online-IRTS query can be formally defined as follows.

Definition 4.3.5 (Skyline)**.** Let $\mathcal{P}$ be a set of paths in the two-dimensional space "detour cost vs reward". A path $P_m \in \mathcal{P}$ dominates another path $P_n \in \mathcal{P}$, denoted as $P_m \prec P_n$, if

$$(DT(P_m) < DT(P_n) \quad \wedge \quad R(P_m) \geq R(P_n)) \quad \vee$$

$$(DT(P_m) \leq DT(P_n) \quad \wedge \quad R(P_m) > R(P_n))$$

That is, P_m is better in one criterion and at least as good as P_n in the other one. Then, $\mathcal{P}$'s skyline is defined as the set of non-dominated paths, i.e. $\{P_m \in \mathcal{P} \mid \nexists P_n \in \mathcal{P} : P_n \prec P_m\}$.

The Online-IRTS problem can now be formally defined as follows. (For ease of reference, Table 4.3 summarizes the notation used throughout this chapter.)

Problem Definition (Online-IRTS query)**.** *Let G be a road network and assume that tasks appear dynamically in G, where each task is associated with a unique vertex v (and vice-versa), a non-negative reward $\rho(v)$, a finite time for completion $\delta(v)$, a release time $\tau_r(v) \geq 0$ and a finite deadline $\tau_d(v)$. Given a worker w originally traveling on the shortest path from a starting location s*

to a destination d (both vertices in G), and a time budget b, the Online-IRTS query informs the worker w of a set of good and diverse feasible path options towards d, w.r.t. both $DT(\cdot)$ and $R(\cdot)$, at the initial point in time and after executing each task.

Notation	*Meaning*
$G(V, E, C)$	Directed graph with sets of vertices V, edges E and C being costs of edges in E.
SP	The shortest path between s and d, i.e., the worker's origin and destination
τ_s	Time the worker departs from s
$P_i = \langle v_i^1, ..., v_i^n \rangle$	A path P_i
v_i^j	The j-th vertex in P_i
$c(v_i, v_j)$	Cost of the edge connecting v_i to v_j
$\pi(v_i^j)$	A binary flag indicating whether a task at vertex v_i^j is completed or not
$TC(P_i)$	Travel cost of path P_i
$DT(P_i)$	Detour cost of path P_i w.r.t. SP
$\rho(v_j)$	Reward of a vertex (task) v_i
$R(P_i)$	Reward of path P_i
$\delta(v_j)$	Time for completion of a vertex (task) v_i
$\tau_r(v_j)$	Release time of a vertex (task) v_i
$\tau_d(v_j)$	Deadline of a vertex (task) v_i
$a(v_i^j, P_i)$	Arrival time at vertex v_i^j by following P_i
$P_i \prec P_j$	P_j is dominated by P_i
$tt(v_i, v_j)$	Travel time of the fastest path from v_i to v_j
$\mathcal{S}^*$	Optimal offline skyline

Table 4.3: Notation used in Chapter 4.

4.4 Proposed Approaches

In this section we propose two heuristics, named Local Optimum Heuristic (LOH) and Incremental Heuristic (IH), to solve the On-IRTS problem. Since the set of all tasks is not known in advance, both heuristics compute skyline sets based on the tasks available at the beginning of each iteration. LOH finds the exact local skyline set in each iteration, while IH finds an approximation of such set.

4.4.1 Local Optimum Heuristic (LOH)

In each iteration, the Local Optimum Heuristic (LOH) computes the local optimum skyline $\mathcal{S}_{LOC}$, i.e., the exact skyline considering the worker's current location and the tasks available at the beginning of the iteration. $\mathcal{S}_{LOC}$ is then returned to the worker who selects a path P and travels to the next task in P, which becomes the new source location when computing the local skyline for the next iteration.

Algorithm 17 shows the pseudocode of the LOH heuristic. First, it computes the exact local skyline considering the tasks available at τ_s and that the worker departs from s (line 2). Then, among the paths returned to the worker, he/she selects a path P (line 3) and travels to the first task t_i in P. After completing t_i, a new local skyline is computed considering the worker's current location and the new available tasks at the current time τ_{it} (line 15). This process is repeated until no more paths can be built considering the remaining budget (line 8) or if the worker chooses to travel to d next (line 10).

In order to find the exact skyline for each iteration (lines 2 and 15), we invoke EXCT-LOC which leverages on the exact algorithm EXCT-SP originally proposed in [6] to solve the (offline) IRTS-SP query. This is feasible because we are searching for a local solution w.r.t. only what is known at a point in time, just like the offline version of the IRTS query. The only modification needed on the original EXCT-SP algorithm is to consider the temporal constraints of tasks, which did not exist in [6].

Given the worker's current location s_w, the time τ_{it} at the beginning of the

Algorithm 17: LOH

```
Input: Graph $G = (V, E)$, starting point $s$, start time $\tau_s$, destination $d$ and
       budget $b$.
Output: Feasible path $P_{final}$ from $s$ to $d$
1  $P_{final} \leftarrow \langle s \rangle$
2  $\mathcal{S}_{LOC} \leftarrow$ EXCT-LOC($s, d, \tau_s, b, P_{final}$)
3  $P \leftarrow$ path selected by worker from $\mathcal{S}_{LOC}$
4  $i \leftarrow 1$
5  $\tau_{it} \leftarrow \tau_s$
6  $b_{it} \leftarrow b$
7  $s_p \leftarrow s$
8  while $P$ is not null do
9      $s_w \leftarrow i$-th task in $P$
10     if $s_w = d$ then
11         break
12     end
13     $P_{final}.append(s_w)$
14     $\tau_{it} \leftarrow \tau_{it} + tt(s_p, s_w) + \delta(s_w)$
15     $b_{it} \leftarrow b_{it} - tt(s_p, s_w) - \delta(s_w)$
16     $\mathcal{S}_{LOC} \leftarrow$ EXCT-LOC($s_w, d, \tau_{it}, b_{it}, P_{final}$)
17     $P \leftarrow$ path selected by worker from $\mathcal{S}_{LOC}$
18     $i \leftarrow i + 1$
19     $s_p \leftarrow s_w$
20 end
21 $P_{final}.append(d)$
22 return $P_{final}$
```

current iteration, and the remaining budget b_{it}, EXCT-LOC (Algorithm 18) first builds a directed graph of tasks $GT = (VT, ET)$. The set of vertices VT contains s, d, and a set of available tasks that can be completed within the remaining budget b_{it}. Each edge connecting two vertices v and u in ET represents the fastest path from v to u in the original graph G and is associated with the travel time $tt(v, u)$ of such path.

GT connects s_w to each task t_i available at τ_{it} such that $\tau_{it} + tt(s_w, t_i) + \delta(t_i) \leq \tau_d(t_i)$, i.e, t_i can be completed before its deadline after traveling from s_w to t_i. Moreover, if the current iteration is not the first one, we also connect s_w to d in order to give the worker the choice to travel to the destination after completing a task. Similarly, GT also connects every pair of available tasks t_i, t_j such that $\tau_{it} + tt(s_w, t_i) + \delta(t_i) + tt(t_i, t_j) + \delta(t_j) \leq \tau_d(t_j)$. Every task in GT is also connected to the destination d. The graph of tasks GT obtained for the first iteration of example shown in Figure 4.1 is illustrated in Figure 4.4. Note that there is no direct edge from t_2 to t_1 because after completing t_2 the worker would arrive at t_1's location at time 8, whereas t_1 expires at time 7.

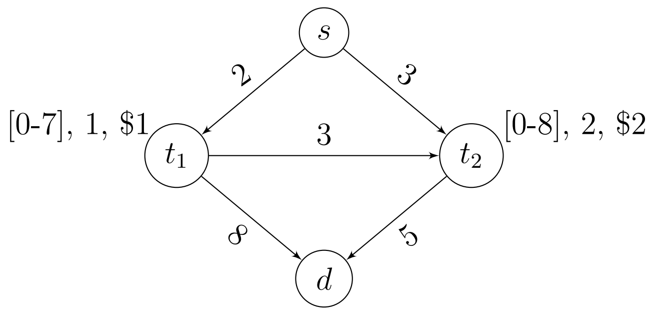


Figure 4.4: Graph GT built for the first iteration of the example shown in Figure 4.1. The value on each edge represents the corresponding travel cost.

Once the graph of tasks $GT = (VT, ET)$ is built (line 1 of Algorithm 18), EXCT-LOC finds the local exact skyline by calling EXCT_REC-LOC (Algorithm 19). EXCT_REC-LOC recursively expands a given path P, which initially is the partial path P_w already taken by the worker, until the budget is exceeded. In each call of EXCT_REC-LOC, we analyze the feasible paths that can be obtained by expanding P. We first check whether the destination can be reached within the remaining budget from the last vertex v of P (line 2). If not, P is not expanded further and the recursion stops. Otherwise, we consider the edges that connect v to any other vertex $u \in VT$ (line 4). If the current neighbor u of v is the destination d (line 6), P is extended with u, generating a new path P_u (line 7). If P_u is not dominated, P_u is added to the skyline set $\mathcal{S}_{LOC}$ and any path dominated by it is removed from $\mathcal{S}_{LOC}$ (lines 8-10). We note that P_u is not further expanded since it is a path to the destination (line 11). On the other hand, if u is a task (line 12), we first check if u is already part of P. If not, we check whether u is active at the time at the worker arrives there and whether it can be completed before its deadline (line 16). If both conditions are satisfied, P is expanded creating the new path P_u, and EXCT_REC-LOC is recursively called for P_u and considering the remaining budget $rb = b - tt(v, u) - \delta(u)$, i.e., the budget after deducting the cost to reach u and to complete it.

Algorithm 18: EXCT-LOC

```
Input: Graph $G = (V, E)$, starting point $s_w$,destination $d$, start time $\tau_{it}$,
       budget $b_{it}$, and current path $P_w$ taken by the worker.
Output: Set $\mathcal{S}_{LOC}$ of non-dominated paths considering the tasks available
        at $\tau_{it}$.
1 $GT \leftarrow$ build graph of tasks considering the tasks available at $\tau_{it}$
2 $\mathcal{S}_{LOC} \leftarrow \emptyset$
3 EXCT_REC-LOC($GT, d, b_{it}, P_w, \tau_{it}, \mathcal{S}_{LOC}$)
4 return $\mathcal{S}_{LOC}$
```

Algorithm 19: EXCT_REC-LOC

```
Input: Graph of tasks $GT = (VT, ET)$, destination $d$, budget $b$, path to
       be expanded $P$, start time $\tau$, and current skyline set $\mathcal{S}_{LOC}$.
1  $v \leftarrow$ last vertex of $P$
2  if $tt(v, d) > b$ then
3      return
4  end
5  for each edge e from v in ET do
6      $u \leftarrow e.toNode$
7      if $u = d$ then
8          $P_u \leftarrow$ extend $P$ with $u$
9          if $P_u$ is not dominated in $\mathcal{S}_{LOC}$ then
10             $\mathcal{S}_{LOC}.add(P_u)$
11             remove any paths dominated by $P_u$ from $\mathcal{S}_{LOC}$
12         end
13         continue
14     end
15     else
16         if $u$ is in $P$ then
17             continue
18         end
19         $at \leftarrow \tau + tt(v, u)$
20         if $at \geq \tau_r(u)$ & $at + \delta(u) \leq \tau_d(u)$ then
21             $P_u \leftarrow$ extend $P$ with $u$
22             $rb \leftarrow b - tt(v, u) - \delta(u)$
23             EXCT_REC-LOC($GT, d, rb, P_u, at + \delta(u), \mathcal{S}_{LOC}$)
24         end
25     end
26 end
```

Figure 4.5 shows the paths explored by EXCT-LOC for the graph shown in Figure 4.4. The feasible path $P_1 = \langle s, t_1, d \rangle$ is found with a detour cost of 5 and a reward of \$1. As the skyline set $\mathcal{S}_{LOC}$ is empty, P_1 is added to $\mathcal{S}_{LOC}$.

Next, $P_2 = \langle s, t_1, t_2, d \rangle$ is found with a detour of 7 and a reward of \$3, and is added to $\mathcal{S}_{LOC}$. Then, path $P_3 = \langle s, t_2, d \rangle$ is found with a time detour of 4 and a reward of \$2. P_3 is added to the skyline set while P_1 is discarded since it is dominated by P_3. Therefore, $\mathcal{S}_{LOC} = \{P_2, P_3\}$.

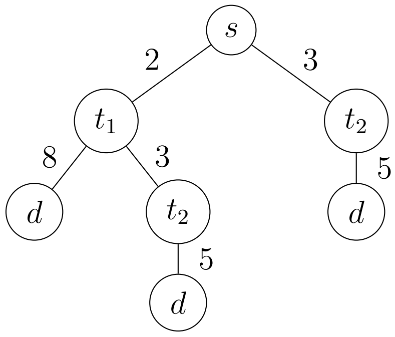


Figure 4.5: Paths explored by EXCT-LOC for the GT shown in Figure 4.4.

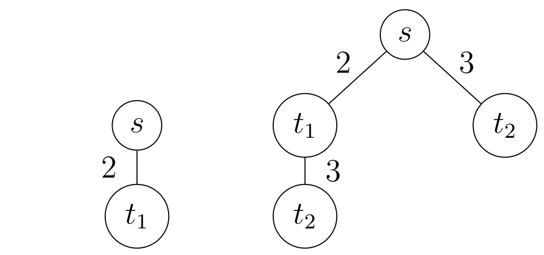
LOH's Complexity Analysis. Let T_i be the set of active tasks in iteration i. The complexity for building the graph of tasks used in LOH is $O(|T_i| \times (|E| + |V| \times \log|V|))$, since we invoke Dijkstra's algorithm from each task in T_i in order to compute the travel times to every other task in such set. In the worst case, LOH may generate all permutations of tasks of size 1 to $|T_i|$. Therefore, in such case $O(|T_i|!)$ paths can built in each iteration. Moreover, the complexity for checking whether each of those paths is dominated is $O(\log|S_{LOC}|)$, assuming that the paths are maintained in a list sorted in increasing order of detour. In case a path is non-dominated, and thus is inserted into the list, it can take $O(|S_{LOC}|)$ in the worst case for such list to be updated. Therefore, the total complexity of this phase of the algorithm is $O(|T_i|! \times |S_{LOC}|)$ in the worst case.

4.4.2 Incremental Heuristic (IH)

Depending on the number of available tasks at a given iteration, the LOH heuristic may take too long to run since it examines all feasible paths containing at least one task to determine the non-dominated ones. In light of this, we developed another heuristic approach, named Incremental Heuristic (IH), which expands a tree that contains the current non-dominated paths, instead of expanding the graph of tasks GT. While this renders IH to be faster than LOH, it may not produce the exact local skyline for a given iteration, which LOH does.

Similarly to LOH, IH also executes Algorithm 17 to produce the final path taken by the worker. However, instead of invoking EXCT-LOC in order to compute the local skyline for the given iteration (lines 2 and 15 in Algorithm 17) it invokes APP-LOC (Algorithm 20). Moreover, IH also maintains a tree of non-dominated paths that is updated in each call of APP-LOC.

Algorithm 20 shows the pseudocode of APP-LOC which takes as input a tree of tasks TT rooted at s_w, i.e., the worker's current location. In the beginning of the first iteration, TT contains only the worker's starting location s. Then, each new available task is considered at a time and is inserted at each feasible position in TT (line 4). In each iteration the set of new tasks NT contains only available tasks that were not considered in previous iterations.



(a) Tree after inserting t_1.
(b) Tree after inserting t_2.

Figure 4.6: Tree after inserting tasks t_1 and t_2 at time 0.

Algorithm 20: APP-LOC

```
Input: Tree of tasks $TT$ rooted at $s_w$, destination $d$, start time $\tau_{it}$, budget
       $b_{it}$, and current path $P_w$ taken by the worker.
Output: Set $\mathcal{S}_{LOC}$ of non-dominated paths considering the tasks available
        at $\tau_{it}$ and updated tree $TT$.
1 $NT \leftarrow$ new tasks available at $\tau_{it}$ that were not available in the previous
  iteration
2 $\mathcal{S}_{LOC} \leftarrow \emptyset$
3 for $t_i$ in $NT$ do
4     addTask($TT, t_i, d, \tau_{it}, b_{it}, P_w, \mathcal{S}_{LOC}$)
5     $TT \leftarrow$ remove all dominated paths from $TT$
6 end
7 $\mathcal{S}_{LOC} \leftarrow$ updateSkylineSet($\mathcal{S}_{LOC}$)
8 return $\mathcal{S}_{LOC}, TT$
```

Figure 4.6 illustrates the tree of non-dominated paths after the insertion of tasks t_1 and t_2 in the first iteration of the example illustrated in Figure 4.1. Algorithm 21 outlines the process of how a task is inserted into the tree. When inserting t_1, the path $\langle s, t_1, d\rangle$ is generated (line 4) and added to the skyline set $\mathcal{S}_{LOC}$ (line 5-7). t_1 is added as a child of s (Figure 4.6a) and since there is no other position in TT where t_1 can be inserted to generate a new path, the recursion stops (lines 10-12). Next, when inserting t_2, the path $\langle s, t_2, d\rangle$ is created and added to $\mathcal{S}_{LOC}$, causing the path $\langle s, t_1, d\rangle$ to be discarded. Since s already has a child, i.e., t_1, the algorithm first tries to insert t_2 before t_1 (lines 13-19). The path $\langle s, t_2, t_1\rangle$ is generated (line 14) and trimInfeasiblePaths (Algorithm 22) is invoked. trimInfeasiblePaths removes all paths rooted at the new node n that become invalid after inserting n. In our example, since t_1 can only be completed after its deadline by following the path $\langle s, t_2, t_1\rangle$, such path is pruned (lines 1-2 of Algorithm 22) and it is not inserted into the tree. Next, Algorithm 21 tries to insert t_2 after t_1 (lines 21-27) and generates the path $\langle s, t_1\rangle$ (line 22). Then, addTask is recursively called (line 25) for $\langle s, t_1\rangle$ with t_1 as the new root. The path $\langle s, t_1\rangle$ is extended with t_2 and d (line 4) and is added to $\mathcal{S}_{LOC}$. Since the root of the tree, t_1, is a leaf, t_2 is added as a child of t_1 (see Figure 4.6b) and the recursion stops (lines 10-12).

Algorithm 21: addTask

```
Input: Sub-tree root $r$, new node $n$ to be inserted, destination $d$, time $\tau$,
       budget $b$, path $P$, and current local skyline $\mathcal{S}_{LOC}$.
Output: Updated $r$ (or null if $n$ could not be inserted)
$at \leftarrow \tau + tt(r, n)$
if $at + \delta(n) > \tau_d(n)$ or $tt(r, n) + \delta(n) + tt(n, d) > b$ then
    return null
end
$P_n \leftarrow$ extend $P$ with $n$ and $d$
if $P_n$ is not dominated in $\mathcal{S}_{LOC}$ then
    $\mathcal{S}_{LOC}.add(P_n)$
    remove any paths dominated by $P_n$ from $\mathcal{S}_{LOC}$
end
$r_{aux} \leftarrow r$
$n_{aux} \leftarrow n$
if $r$ is leaf then
    $r.addChild(n_{aux})$
    return $r$
end
for child $c$ of $r$ do
    $P_c \leftarrow$ extend $P$ with $n$ and $c$
    $\tau_c \leftarrow \tau + tt(r, n) + \delta(n) + tt(n, c) + \delta(c)$
    $b_c \leftarrow b - tt(r, n) - \delta(n) - tt(n, c) - \delta(c)$
    $c_{aux} \leftarrow$ trimInfeasiblePaths$(c, d, \tau_c, b_c, P_c, \mathcal{S}_{LOC})$
    if $c_{aux} \neq null$ then
        $n_{aux}.addChild(c_{aux})$
    end
end
$r_{aux}.addChild(n_{aux})$
for child $c$ of $r$ do
    $P_c \leftarrow$ extend $P$ with $c$
    $\tau_c \leftarrow \tau + tt(r, c) + \delta(c)$
    $b_c \leftarrow b - tt(r, c) - \delta(c)$
    $c_{aux} \leftarrow$ addTask$(c, n, d, \tau_c, b_c, P_c, \mathcal{S}_{LOC})$
    if $c_{aux} \neq null$ then
        $r_{aux}.addChild(c_{aux})$
    end
end
$r.addChild(n_{aux})$
if $r_{aux}$ is not null then
    return $r_{aux}$
end
else
    return null
end
```

```
Algorithm 22: trimInfeasiblePaths
  Input: New sub-tree root $n$, destination $d$, time $\tau$ after completing $n$,
         budget $b$, path $P$, and current local skyline $\mathcal{S}_{LOC}$.
  Output: Updated $r$ (or null if $n$ could not be inserted
1 if $\tau > \tau_r(n)$ or $tt(n,d) > b$ then
2     return null
3 end
4 $P_d \leftarrow$ extend $P$ with $d$
5 if $P_d$ is not dominated in $\mathcal{S}_{LOC}$ then
6     $\mathcal{S}_{LOC}.add(P_d)$
7     remove any paths dominated by $P_d$ from $\mathcal{S}_{LOC}$
8 end
9 if $n$ is leaf then
10    return $n$
11 end
12 $n_{aux} \leftarrow n$
13 for child $c$ of $n$ do
14    $P_c \leftarrow$ extend $P$ with $c$
15    $\tau_c \leftarrow \tau + tt(n,c) + \delta(c)$
16    $b_c \leftarrow b - tt(n,c) - \delta(c)$
17    $c_{aux} \leftarrow$ trimInfeasiblePaths($c, d, \tau_c, b_c, P_c, \mathcal{S}_{LOC}$)
18    if $c_{aux} \neq null$ then
19        $n_{aux}.addChild(c_{aux})$
20    end
21 end
22 return $n_{aux}$
```

After each new task is inserted into the tree, APP-LOC maintains only the non-dominated paths from the root to a leaf in TT (line 5 of Algorithm 20). For instance, if $\langle s, t_1, t_2, d\rangle$ was dominated, the whole sub-tree rooted at t_1 (shown in Figure 4.6b) would be removed before the next iteration since the path $\langle s, t_1, d\rangle$ is also dominated. Finally, addTask only tests the dominance of paths that contain at least one task in the set of new tasks NT. However, there may be other non-dominated paths in the tree that do not contain any of those tasks. Therefore, we invoke updateSkylineSet (line 6 of Algorithm 20) to generate such paths and update $\mathcal{S}_{LOC}$ accordingly.

Once the local skyline set $\mathcal{S}_{LOC}$ produced by APP-LOC is returned to the worker, he/she chooses a path P from such set and travels to next task t_i in P. Then, the sub-tree rooted at t_i becomes the new tree that will be given as input to APP-LOC (Algorithm 20) in the next iteration. For instance, if the worker chooses path $\langle s, t_2, d\rangle$ in the first iteration, the sub-tree rooted at t_2

becomes the new tree in the next iteration, which is then extended with the new tasks that became available, i.e., t_3 and t_4.

IH's Complexity Analysis. In order to compute the travel times between pairs of tasks in IH we also invoke Dijkstra's algorithm from each active task in T_i, as we do in LOH. The complexity for maintaining the local skyline set S_{LOC} is also the same as in LOH. Even though IH may generate $O(|T_i|!)$ paths in the worst case, in practice it is consistently faster than LOH, since it only expands non-dominated paths that remain in the tree. Moreover, when a task t_j is considered for insertion into the tree, if it does not lead to any non-dominated paths it is discarded and is not considered in future iterations.

4.4.3 Offline Exact Approach

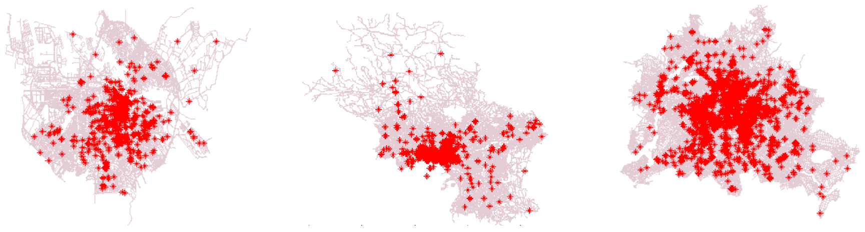
In order to obtain an offline optimal skyline set S^* we follow the same procedure as the EXCT-SP algorithm proposed in [6], except that we take into consideration the active time of tasks, which was not part of the problem setting in [6]. More specifically, when building the graph of tasks used for finding the non-dominated paths, we consider only the tasks that are active while the worker is available, i.e., tasks that are released before the worker leaves the system and that expire after the worker becomes available. Moreover, when expanding a path with a task t_i, we check whether t_i has already been released when the worker arrives and whether it can be completed before it expires. If such conditions are not satisfied, the path is pruned and not further expanded.

4.5 Experiments

We evaluated the performance of our approaches, as well as their accuracy w.r.t. the offline optimal solution, i.e., where all tasks, rewards and deadlines are known beforehand. In order to mimic a realistic set of task locations we used the location of eateries (restaurants and coffee shops) on the actual road networks of Amsterdam (AMS), Oslo (OSLO) and Berlin (BER)[2] as locations of (pseudo) tasks; metadata about those datasets can be seen in Table 4.4 and Figure 4.7 illustrates them visually. The travel time of each edge $(v, u) \in E$ is computed as the Euclidean distance $d_e(v, u)$ divided by an average speed of 40 km/h. Note that such an assumption is not a limitation of the proposed approaches; they work for any connected graph where the cost of each edge is a function of its length.

	Amsterdam	**Oslo**	Berlin
#vertices	106,599	**305,174**	428,768
#edges	130,090	**330,632**	504,228
#tasks	824	**958**	3,083

Table 4.4: Summary of the real datasets used in our experiments (**bold** defines default values).



(a) Amsterdam (b) Oslo (c) Berlin

Figure 4.7: Locations of tasks in Amsterdam, Oslo and Berlin (overlaid on those cities' road networks).

Table 4.5 shows the parameters varied in our experiments, in addition to the real datasets. The shortest path's cost, from the worker's starting location s to the destination d, ranges from 15 min to 60 min. s is randomly selected among all the vertices of the network, and then, we perform a Dijkstra-based search to select d. More specifically, the network is expanded from s and, as

[2]As of March/2017 [3].

soon as the travel time from s to a vertex v removed from the queue exceeds the given shortest path's cost, the expansion stops and v becomes the destination d. We also assume that the worker has a time budget between 60 and 240 min and that the time he/she becomes available follows a normal distribution whose mean is 12pm and standard deviation varies from 120 to 360 min.

Parameter	Range
Cost of shortest path	15, **30**, 60
Worker's time budget	60, **120**, 240
Worker's start time std dev	120, **240**, 360
Average task completion time	5, **15**, 30
Task density [%]	25, 50, **100**
Task active time	60, **120**, 240
Task release time std dev	120, **240**, 360
α policy	Incr., Decr., Constant, $\boldsymbol{\mathcal{N}(0.5, 0.2)}$

Table 4.5: Experimental parameters and their values (**bold** denotes default values). All numerical values, except task density, are in min.

In order to evaluate the effect of task density on a given network we experimented varying the number of tasks between 25% and 100% of all possible tasks. We also varied the time it takes for completing a task by following a normal distribution. The distribution's mean was set to 5, 15 and 30 min, while the standard deviation was fixed at 1/3 of the respective distribution's mean. We assume the rewards paid out to workers to be proportional to the time it takes for completing the corresponding task. Nonetheless, there is nothing in the approaches presented that would prevent one from using an arbitrary reward scheme. We consider that the active time of tasks also follows a normal distribution. We varied the distribution's mean from 60 to 240 min and set the standard deviation to 1/4 of the corresponding mean. Moreover, we assume that the time the tasks become available is determined by a distribution equal to one used to determine the worker's availability.

Finally, in order to simulate the worker's choice in each iteration, in our experiments we select the path that maximizes the following linear combination of the two criteria: $\mathcal{F}(P) = \alpha \times \left(1 - \frac{DT(P)}{maxDT}\right) + (1 - \alpha) \times \left(\frac{R(P)}{maxR}\right)$ for $0 \leq \alpha \leq 1$. Therefore, the higher (lower) the value of α the more importance is given to the detour (reward) of the paths. We adopted 4 policies for how α behaves as the algorithms progresses: it can increase, decrease, remain constant or vary following a normal distribution. In the cases where α increases or decreases, we assume that its value is initially set to a random value between 0 and 0.5 (0.5 and 1), and then increases (decreases) exponentially in order to simulate scenarios where the importance rapidly shifts from reward to detour (vice-versa), as the worker completes tasks. When α is drawn from a Normal distribution, we model the case where the worker's priority changes in a non-deterministic way. It is important to stress that in those cases, a non-constant α reflects our basic assumption that the worker can change his/her priorities with time, hence requiring the use of the skyline paradigm where both criteria are possibly weighted differently in each iteration. Finally, we also performed experiments where the value of α is drawn from a U(0,1) distribution before the first iteration and kept constant afterwards.

For each set of experiments, we vary the value of one parameter, fix the other parameters to their default values and report the average results of 100 runs. The experiments were performed in a virtual machine with Intel(R) Xeon(R) CPU E5-2650 (8 cores @ 2.30GHz) and 16GB RAM, running Ubuntu.

Regarding performance, in the following we report the average processing time per iteration, i.e., how long the worker needs to wait in average for the local skyline to be computed after he/she finishes a task (or before leaving the origin). Also, all charts have the same scale in order to facilitate comparisons.

Finally, in order to evaluate the quality of the paths produced by our heuristics we measure the relative error of a final path P obtained by them w.r.t. the paths in the optimal offline skyline $\mathcal{S}^*$. In order to illustrate how this relative error is measured, let us consider Figure 4.2 again, which shows $\mathcal{S}^*$ and the final path P_4 produced following the choices made by the worker. Path P_4 is dominated by path P_2^*, and thus the area dominated by P_2^*, represented by the area delimited in black, is larger than the one dominated by P_4, which is delimited in red. The idea behind the relative error metric is that the difference between those areas reflects how much room there is for improvement as any other point in that area would dominate the one obtained. Clearly the smaller this area the better, in fact it is null if the produced path belongs to $\mathcal{S}^*$. Note that a path P can be dominated by multiple paths in $\mathcal{S}^*$. Let us denote the set of such paths as $\mathcal{P}_d$ and as $A_{\mathcal{P}_d}$ the area representing the union of the areas dominated by each path in $\mathcal{P}_d$. Similarly, let A_P be the area dominated by path P, then, the normalized relative error is given by:

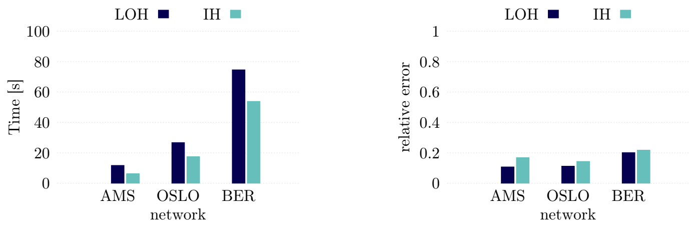
$$error = \frac{A_{\mathcal{P}_d} - A_P}{A_{\mathcal{P}_d}}$$

4.5.1 Effect of the road network

As expected and as as shown in Figure 4.8a, the efficiency of our approaches decreases with the size of the network and the number of tasks in it[3]. A greater number of tasks implies that there are more permutations of tasks to be considered. Moreover, the larger the network, in terms of number of vertices and edges, the more time is consumed to compute the required shortest paths. This explains why our approaches are slower for the BER network. The slightly larger relative error in BER's case (Figure 4.8b) is due to the network size and how tasks are distributed (see Figure 4.7c). Tasks will tend to be farther away

[3]Recall from Table 4.4 that the larger the network the more tasks it may have.

from each other in such network, and if the worker chooses to travel to a task that does not lead to a path in $\mathcal{S}^*$, it will be harder to compensate for a "bad choice" since there will potentially be less feasible tasks that can be combined with the worker's current path within the remaining budget.

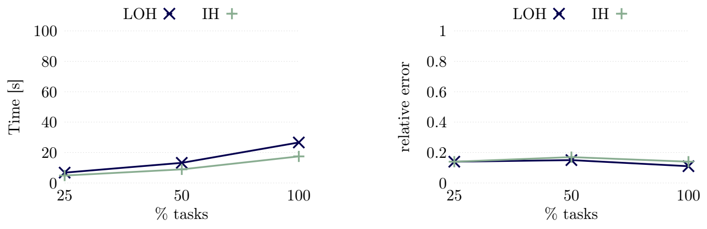


(a) average time (b) relative error

Figure 4.8: Processing time and error w.r.t. the network.

4.5.2 Effect of task density

Figure 4.9a shows that, as expected, the processing time of our heuristics increases with the task density since there are more permutations of tasks to be examined. On the other hand, Figure 4.9b shows that when the task density increases, the relative error of our approaches slightly increases and then decreases for the OSLO network. Although a greater number of task options increases the chance of the worker making a poor decision, a lower density also means that tasks will tend to be farther away from each other, which can potentially lead to a higher relative error, as discussed in Section 4.5.1.

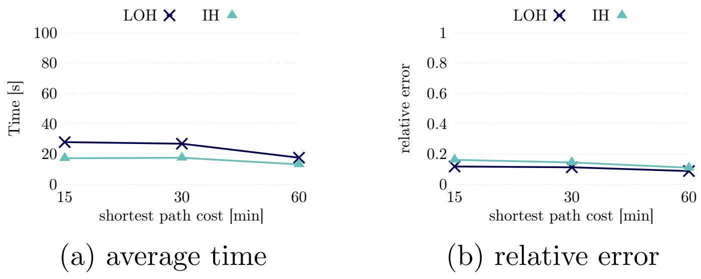


(a) average time (b) relative error

Figure 4.9: Processing time and error w.r.t. the task density.

4.5.3 Effect of the cost of the initial shortest path

Figures 4.10a and 4.10b show that the average processing time of both heuristics, as well as their relative error, tend to decrease with the cost of the shortest path SP between s and d. This is because the time the worker has to perform tasks decreases with the increase in cost of the shortest path, since his/her budget is fixed. Therefore, fewer tasks become feasible within a more limited budget, which positively affects the processing time and relative error of our heuristics.

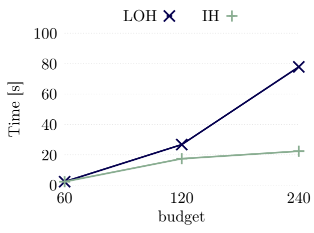


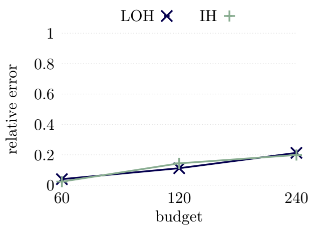
(a) average time (b) relative error

Figure 4.10: Processing time and error w.r.t. SP's cost.

4.5.4 Effect of the worker's budget

As shown in Figure 4.11a, and as expected, the processing time of both heuristics increase with the worker's budget simply because it allows for more feasible paths to be examined, which strongly impacts LOH's performance. Figure 4.11b shows that the error w.r.t. $\mathcal{S}^*$ also increases with the worker's budget for both approaches. A larger budget means that more tasks can be performed in sequence, and thus the worker has to make more decisions, and, consequently, there is a greater chance of the worker making poor choices, diminishing the chance of generating paths close to the ones in $\mathcal{S}^*$.



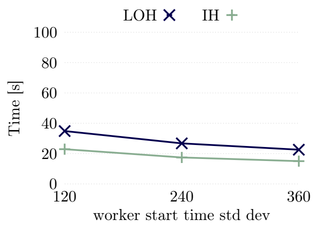


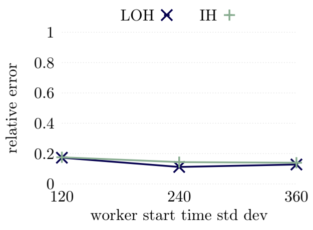
(a) average time (b) relative error

Figure 4.11: Processing time and error w.r.t. the worker's budget.

4.5.5 Effect of the worker's starting time

Figure 4.12 suggests that the processing time of our heuristics as well as their relative error tend to decrease with the standard deviation of the time the worker becomes available. A smaller standard deviation means that the worker will tend to become available when more tasks are active (due to how the release time of tasks is distributed). Therefore, there will potentially be more tasks to expand a path with, which makes our approaches slightly slower and slightly less effective, as shown in Figures 4.12a and 4.12b, respectively.



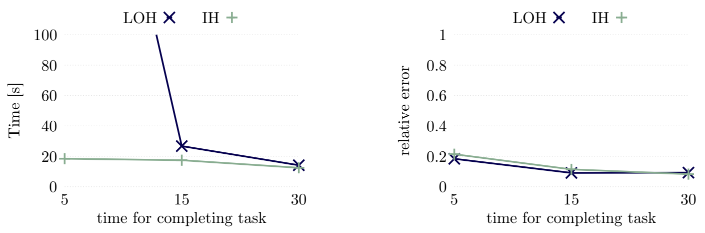


(a) average time (b) relative error

Figure 4.12: Processing time and error w.r.t. the standard deviation of the worker's start time.

4.5.6 Effect of the average time for completing a task

Figure 4.13a shows that the processing time of our approaches decrease with the average time it takes to perform a task, however, LOH's performance is more affected by such parameter. When tasks take a shorter time to be completed, more of them can be included in the worker's path within his/her budget, therefore there are more feasible paths to be examined, which directly affects LOH's processing time. As shown in Figure 4.13b, the error w.r.t. $\mathcal{S}^*$ tends to decrease with the time it takes to complete a task for both heuristics. With tasks taking longer to be completed, less tasks can be performed in sequence within the worker's budget, which positively affects our heuristics.



(a) average time (b) relative error

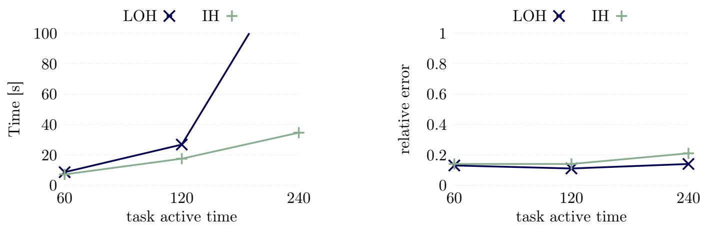
Figure 4.13: Processing time and error w.r.t. the time for completing a task. For completion time 5, LOH's average processing time is 364 sec.

4.5.7 Effect of the active time of the tasks

As shown in Figure 4.14a the processing time of both approaches increase with the active time of tasks, however, LOH is significantly more affected by this parameter. When tasks are active for a longer time, more tasks will tend to be active while the worker is available and, thus, there are more tasks to be considered by each heuristic, which strongly affects LOH's performance. As shown in Figure 4.14b, IH's relative error increases with this parameter also due to a greater number of task options. We note that even though LOH's error is only slightly affected for the values examined, it is expected that it will also tend to increase when tasks are active for a longer time.

4.5.8 Effect of the release time of the tasks

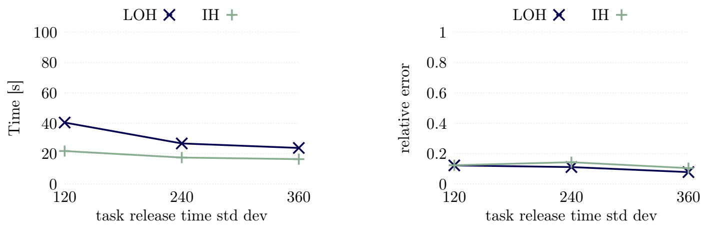
Figure 4.15a shows that the processing time of our heuristics decreases with the standard deviation of time the tasks become active. With a smaller stan-



(a) average time (b) relative error

Figure 4.14: Processing time and error w.r.t. the active time of tasks. When the task active time is 240 min, LOH's average processing time is 154 sec.

dard deviation more tasks will tend to be available when the worker becomes active, and thus there will be more tasks to be examined by each approach. This parameter also slightly affect the quality of the results produced by our heuristics, as shown in Figure 4.15b, since when fewer tasks are active better solutions tend to be produced.

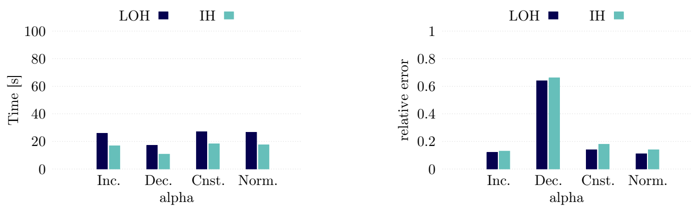


(a) average time (b) relative error

Figure 4.15: Processing time and error w.r.t. the standard deviation of the tasks' release time.

4.5.9 Effect of α (trade-off between detour and reward)

Figures 4.16a and 4.16b show how our proposed approaches behave as we vary the policies on how the worker prioritizes detour or reward. Neither approach seems to be particularly sensitive to how α varies, except when it decreases, i.e., when the workers interest shifts from minimizing detour to maximizing reward as he/she progresses in his/her path. In this case, the worker will travel to more tasks, as he/she will seek more rewards. Interestingly, even though more iterations will be performed, each will tend to require progressively less processing time as the worker's budget is consumed after each task, which, in turn, will tend to limit the number of feasible paths. This explains the shorter average processing time in this case. On the other hand, more iterations yield more opportunities for "bad choices", which leads to a higher relative error.



(a) average time (b) relative error

Figure 4.16: Processing time and error w.r.t. α.

4.5.10 Summary

The main observations from the experiments above are:

- In line with the complexity analysis of the algorithms (Sections 4.4.1 and 4.4.2), more tasks available imply more permutations of tasks to be considered. While this affects both approaches' efficiency negatively, the effect is more pronounced in LOH's case than IH's as IH expands only non-dominated paths. This effect is greatly compounded when the worker has more time available, e.g., when his/her time budget is larger or tasks can be accomplished fast.

- If the worker pursues rewards aggressively, he/she will naturally tend to perform more tasks, increasing the chances of sub-optimal choices, which degenerates the effectiveness of both approaches, but affecting IH slightly more.

- Finally, w.r.t. effectiveness neither approach seemed to be particularly affected by the investigated parameters, though IH does not fare as well as LOH, as IH typically examines less paths, thus potentially missing some good ones.

4.6 Conclusion

In this chapter we presented the Online-IRTS query, a variant of the *In-Route Task Selection* (IRTS) [6] query in which tasks appear dynamically. Online-IRTS provides the worker with a set of non-dominated paths in an online manner using the notion of skyline queries. As new tasks become available, and according to the selections made by the worker in previous iterations, the skyline set is updated so that the worker can contemplate new interesting paths yielding different trade-offs between detour and reward.

In order to solve the Online-IRTS query we proposed two heuristic approaches that build a feasible path incrementally, i.e., task after task, with the goal of ultimately approximating a path in the offline optimal skyline set. Our first proposed heuristic, named Local Optimum Heuristic (LOH), builds the local exact skyline in each iteration considering all available tasks, while the Incremental Heuristic (IH) expands non-dominated paths in order to build an approximate local skyline. Our experimental results using realistic datasets showed that LOH produces paths that yield a relative error up to 36% smaller than that of the solutions produced by IH, while IH is up to 20 times faster than LOH.

Chapter 5

Conclusion

In this thesis we proposed the use of skyline queries to solve bi-criteria optimization problems in road networks. Skyline queries do not require users to specify weights to each criterion and provide them with a set of interesting non-dominated options for different trade-offs between the competing criteria. Based on this, we first investigated (k, r)-Diverse Nearest Neighbors ((k, r)-DNN) Queries, and then we explored In-Route Task Selection (IRTS) queries, in the context of spatial crowdsourcing, considering both offline and online settings.

(k, r)-DNN queries are a novel variation of k-NN queries which return sets containing k elements that are as close as possible to the query point, while, at the same time, being as diverse as possible. We assume that the user specifies the maximum distance he/she is willing to travel in order to visit a POI, i.e., the query radius r. We have considered three different notions of diversity, namely spatial, angular and categorical. Previously proposed solutions are approximate and require the user to determine a specific weight for each type of diversity. Our proposed approaches based on linear skylines find *all* results that are *optimal* under *any* arbitrary linear combination of closeness and diversity.

We proposed two solutions to (k, r)-DNN queries using the notion of skylines. *Recursive Range Filtering (RRF)* strives to reduce the number of combinations generated by recursively combining a partial set with candidate points that are diverse enough to be part of a non-dominated solution. *Pair Graph (PG)* works by pruning subsets that contain pairs of points that have low diversity and could not be together in a non-dominated solution. Experiments varying a number of parameters showed that (1) both *RRF* and *PG* are orders of magnitude faster than a straightforward solution, and (2) *PG* outperforms *RRF* when there is a greater number of candidate points that can be combined with previously selected points, and *RRF* is more efficient otherwise. Also, we confirmed that our approaches are more effective than a previously proposed approximate approach, named *GNE*, by virtue of guaranteeing optimal results, as well as being more generic since they do not require the user to set any particular trade-off.

In the second part of this thesis we presented two variants of the offline IRTS problem, which is defined in the context of spatial crowdsourcing. The IRTS-SP problem considers that the worker's initial route is the shortest path connecting his/her starting point and destination, whereas the IRTS-PP problem considers that the worker has an arbitrary preferred path instead. In both cases the worker is willing to consider the trade-off between a limited detour and rewards collected by completing tasks during such detour. Given the competing nature of those two criteria, we investigated this problem using the skyline paradigm, and after proving the NP-hardness of the problems, we proposed a few heuristic approaches and also analysed their complexity.

For the case of IRTS-SP, our experimental results, using real datasets at the city scale, showed that *kGH-SP*, which expands a graph that connects a task to its k closest tasks, is the best alternative among the heuristics. It produces consistently better results than the greedy heuristics, while being only slightly slower. In the case of IRTS-PP, the results showed that the heuristic guided by the detour's cost (DOH) can obtain solutions with low relative error rates. However, even though it is at least one order of magnitude faster than the exact approach $EXCT\text{-}PP$, its processing time can increase rapidly with the number of tasks and when tasks take a shorter time to be completed. In such cases, $kGH\text{-}PP$ is a better alternative since it can produce results closer to the ones produced by DOH, while still being slightly faster than the greedy heuristics.

Finally, as our last contribution in this thesis, we investigated the online version of the IRTS problem, i.e., the one in which tasks appear dynamically. Online-IRTS provides the worker with a set of non-dominated paths in an online manner using the notion of skyline queries. As new tasks become available, and according to the selections made by the worker in previous iterations, the skyline set is updated so that the worker can contemplate new interesting paths yielding different trade-offs between detour and reward.

In order to solve the Online-IRTS query we proposed two heuristic approaches that build a feasible path incrementally, i.e., task after task, with the goal of ultimately approximating a path in the offline optimal skyline set. Our first proposed heuristic, named Local Optimum Heuristic (LOH), builds the local exact skyline in each iteration considering all available tasks, while the Incremental Heuristic (IH) expands non-dominated paths in order to build an approximate local skyline. Our experimental results using realistic datasets showed that LOH produces paths that yield a relative error up to 36% smaller than that of the solutions produced by IH, while IH is up to 20 times faster than LOH.

A direction for future work is to develop approximate approaches to the (k, r)-DNN query in order to produce good results faster than the exact approaches proposed in this thesis. Regarding the Online-IRTS problem, we envision two interesting directions for future work. The skylines sets obtained at each iteration can be rather large, investigating how to reduce them, for instance using linear skylines [43], could be of practical value to the worker. When investigating the offline IRTS query [6], we noted that the problem was more complex if instead of the shortest path between s and d the worker wanted to consider a preferred arbitrary path. We believe this extra complexity would apply for the Online-IRTS query as well and thus warrant further research. Finally, another interesting direction for future work is to investigate the queries proposed in this thesis within the context of time-dependent road networks, which considers the more realistic scenario where the travel time between locations depends on the departure time and traffic conditions.

References

[1] S. Abbar, S. Amer-Yahia, P. Indyk, S. Mahabadi, and K. R. Varadarajan, "Diverse near neighbor problem," in *Proceedings of the 29th Symposium on Computational Geometry*, 2013, pp. 207–214.

[2] E. Ahmadi, C. F. Costa, and M. A. Nascimento, "Best-compromise in-route nearest neighbor queries," in *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2017, 41:1–41:10.

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

[4] Bellhops, `https://www.getbellhops.com/`.

[5] S. Borzsony, D. Kossmann, and K. Stocker, "The skyline operator," in *Proceedings of 17th International Conference on Data Engineering*, 2001, pp. 421–430.

[6] Camila F. Costa and Mario A. Nascimento, "In-route task selection in spatial crowdsourcing," *ACM Transactions on Spatial Algorithms and Systems*, vol. 6, no. 2, 7:1–7:45, 2020.

[7] J. Carbonell and J. Goldstein, "The use of MMR, diversity-based reranking for reordering documents and producing summaries," in *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1998, pp. 335–336.

[8] B. Carterette, "An analysis of np-completeness in novelty and diversity ranking," *Information Retrieval*, pp. 89–106, 2011.

[9] P. Cheng, X. Lian, L. Chen, J. Han, and J. Zhao, "Task assignment on multi-skill oriented spatial crowdsourcing," *IEEE Transactions on Knowledge and Data Engineering*, pp. 2201–2215, 2016.

[10] P. Cheng, X. Lian, Z. Chen, R. Fu, L. Chen, J. Han, and J. Zhao, "Reliable diversity-based spatial crowdsourcing by moving workers," *Proceedings of the VLDB Endowment*, vol. 8, no. 10, pp. 1022–1033, 2015.

[11] C. L. Clarke, M. Kolla, G. V. Cormack, O. Vechtomova, A. Ashkan, S. Büttcher, and I. MacKinnon, "Novelty and diversity in information retrieval evaluation," in *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2008, pp. 659–666.

[12] 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.

[13] C. F. Costa and M. A. Nascimento, "Towards spatially- and category-wise k-diverse nearest neighbors queries," in *International Symposium on Spatial and Temporal Databases*, 2017, pp. 163–181.

[14] 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.

[15] C. F. Costa, T. Chondrogiannis, M. A. Nascimento, and P. Bouros, "RRAMEN: an interactive tool for evaluating choices and changes in transportation networks," in *Proceedings of the 23nd International Conference on Extending Database Technology*, 2020, pp. 599–602.

[16] C. F. Costa and M. A. Nascimento, "IDA 2016 industrial challenge: Using machine learning for predicting failures," in *Proceedings of the 15th International Symposium on Advances in Intelligent Data Analysis XV*, vol. 9897, 2016, pp. 381–386.

[17] K.-H. Dang and K.-T. Cao, "Towards reward-based spatial crowdsourcing," in *2013 International Conference on Control, Automation and Information Sciences*, 2013, pp. 363–368.

[18] D. Deng, C. Shahabi, and U. Demiryurek, "Maximizing the number of worker's self-selected tasks in spatial crowdsourcing," in *Proceedings of the 21st ACM SIGSPATIAL Iternational Conference on Advances in Geographic Information Systems*, 2013, pp. 324–333.

[19] D. Deng, C. Shahabi, U. Demiryurek, and L. Zhu, "Task selection in spatial crowdsourcing from worker's perspective," *GeoInformatica*, vol. 20, no. 3, pp. 529–568, 2016.

[20] D. Deng, C. Shahabi, and L. Zhu, "Task matching and scheduling for multiple workers in spatial crowdsourcing," in *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2015, p. 21.

[21] Dolly, `https://www.dolly.com/`.

[22] DoorDash, `https://www.doordash.com/`.

[23] B. L. Golden, L. Levy, and R. Vohra, "The orienteering problem," *Naval research logistics*, pp. 307–318, 1987.

[24] Y. Gu, G. Liu, J. Qi, H. Xu, G. Yu, and R. Zhang, "The moving k diversified nearest neighbor query," *IEEE Transactions on Knowledge and Data Engineering*, pp. 2778–2792, 2016.

[25] J. Handl and J. Knowles, "Cluster generators for large high-dimensional data sets with large numbers of clusters," *http://dbkgroup.org/handl/generators*, 2005.

[26] Handy, `https://www.handy.com/`.

[27] HelloTech, `https://www.hellotech.com/`.

[28] X. Huang and C. S. Jensen, "In-route skyline querying for location-based services," in *International Workshop on Web and Wireless Geographical Information Systems*, 2004, pp. 120–135.

[29] Z. Huang *et al.*, "A clustering based approach for skyline diversity," *Expert Systems with Applications*, pp. 7984–7993, 2011.

[30] Instacart, `https://www.instacart.com/`.

[31] A. Jain, P. Sarda, and J. R. Haritsa, "Providing diversity in k-nearest neighbor query results," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2004, pp. 404–413.

[32] L. Kazemi and C. Shahabi, "Geocrowd: Enabling query answering with spatial crowdsourcing," in *Proceedings of the 20th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2012, pp. 189–198.

[33] O. Kucuktunc and H. Ferhatosmanoglu, "λ-diverse nearest neighbors browsing for multidimensional data," *IEEE Transactions on Knowledge and Data Engineering*, pp. 481–493, 2013.

[34] K. C. Lee, W.-C. Lee, and H. V. Leong, "Nearest surrounder queries," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1444–1458, 2010.

[35] Y. Li, M. L. Yiu, and W. Xu, "Oriented online route recommendation for spatial crowdsourcing task workers," in *SSTD*, Springer, 2015, pp. 137–156.

[36] C. Long, R. C.-W. Wong, P. S. Yu, and M. Jiang, "On optimal worst-case matching," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 845–856.

[37] L. Love, `https://www.lawnlove.com/`.

[38] Lyft, `https://www.lyft.com/`.

[39] D. Rafiei, K. Bharat, and A. Shukla, in *Proceedings of the 19th International Conference on World Wide Web*, 2010, pp. 781–790.

[40] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in *ACM SIGMOD Record*, 1995, pp. 71–79.

[41] Rover, `https://www.rover.com/`.

[42] J. She, Y. Tong, L. Chen, and C. C. Cao, "Conflict-aware event-participant arrangement and its variant for online setting," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 9, pp. 2281–2295, 2016.

[43] M. Shekelyan, G. Jossé, M. Schubert, and H.-P. Kriegel, "Linear path skyline computation in bicriteria networks," in *International Conference on Database Systems for Advanced Applications*, 2014, pp. 173–187.

[44] S. Shekhar and J. S. Yoo, "Processing in-route nearest neighbor queries: A comparison of alternative approaches," in *Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems*, 2003, pp. 9–16.

[45] T. Song, Y. Tong, L. Wang, J. She, B. Yao, L. Chen, and K. Xu, "Trichromatic online matching in real-time spatial crowdsourcing," in *IEEE 33rd International Conference on Data Engineering*, 2017, pp. 1009–1020.

[46] Y. Tao, "Diversity in skylines," *IEEE Data Engineering Bulletin*, pp. 65–72, 2009.

[47] TaskRabbit, `https://www.taskrabbit.com/`.

[48] H. To, L. Fan, L. Tran, and C. Shahabi, "Real-time task assignment in hyperlocal spatial crowdsourcing under budget constraints," in *IEEE International Conference on Pervasive Computing and Communications*, 2016, pp. 1–8.

[49] Y. Tong, L. Chen, and C. Shahabi, "Spatial crowdsourcing: Challenges, techniques, and applications," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1988–1991, 2017.

[50] Y. Tong, J. She, B. Ding, L. Wang, and L. Chen, "Online mobile micro-task allocation in spatial crowdsourcing," in *IEEE 32nd International Conference on Data Engineering*, 2016, pp. 49–60.

[51] Uber, `https://www.uber.com/`.

[52] G. Valkanas, A. N. Papadopoulos, and D. Gunopulos, "Skydiver: A framework for skyline diversification," in *Proceedings of the 16th International Conference on Extending Database Technology*, 2013, pp. 406–417.

[53] M. R. Vieira, H. L. Razente, M. C. Barioni, M. Hadjieleftheriou, D. Srivastava, C. Traina, and V. J. Tsotras, "On query result diversification," in *IEEE 27th International Conference on Data Engineering*, 2011, pp. 1163–1174.

[54] Wag, `https://www.wagwalking.com/`.

[55] J. Xin, Z. Wang, M. Bai, L. Ding, and G. Wang, "Energy-efficient β-approximate skylines processing in wireless sensor networks," *Mathematical Problems in Engineering*, 2015.

[56] M. L. Yiu, K. Mouratidis, N. Mamoulis, *et al.*, “Capacity constrained assignment in spatial databases,” in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008, pp. 15–28.

[57] YourMechanic, `https://www.yourmechanic.com/`.

[58] C. Yu, L. Lakshmanan, and S. Amer-Yahia, “It takes variety to make a world: Diversification in recommender systems,” in *Proceedings of the 12th International Conference on Extending Database Technology*, 2009, pp. 368–378.

[59] C. Zhang, Y. Zhang, W. Zhang, X. Lin, M. A. Cheema, and X. Wang, “Diversified spatial keyword search on road networks.,” in *Proceedings of the 17th International Conference on Extending Database Technology*, 2014, pp. 367–378.

[60] L. Zheng and L. Chen, “Mutual benefit aware task assignment in a bipartite labor market,” in *IEEE 32nd International Conference on Data Engineering*, 2016, pp. 73–84.