ON BOOSTING SPATIAL COMPUTATIONS FOR LOCATION-BASED SERVICES

by

CHENG LONG

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy
in Computer Science and Engineering

June 2015, Hong Kong

Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

CHENG LONG

23 June 2015

ON BOOSTING SPATIAL COMPUTATIONS FOR LOCATION-BASED SERVICES

by

CHENG LONG

This is to certify that I have examined the above Ph.D. thesis

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by

the thesis examination committee have been made.

DR. RAYMOND CHI-WING WONG, THESIS SUPERVISOR

PROF. QIANG YANG, HEAD OF DEPARTMENT

Department of Computer Science and Engineering

23 June 2015

ACKNOWLEDGMENTS

I would like to thank my supervisor Prof. Raymond Chi-Wing Wong for giving me all kinds of supports throughout my five years of PhD study. Specifically, I am always grateful to Raymond for introducing me to research, training me the fundamental skills of presentation, writing and problem solving, having weekly meetings with me (e.g., for discussing with me, giving me feedbacks and suggestions, and brainstorming), polishing my drafts, giving me supports to attend international conferences and visit some other universities, and encouraging me when I got stuck by some research problems. Without these help from Raymond, this thesis is not possible.

I would also like to thank Prof. Cyrus Shahabi and Prof. H. V. Jagadesh for hosting my research visits at the University of Southern California (USC) from Feb 2014 to May 2014 and at the University of Michigan (UM) from Oct 2014 to Apr 2015, respectively, during my PhD study. I deeply appreciate their supervision and their valuable advice to me.

I would like to give my deep gratitude to Prof. Ada Wai-Chee Fu, Prof. Philip S. Yu, and Prof. Ke Wang for their invaluable comments on my research, and to Prof. Yubao Liu, Mr. Qi Wang, Dr. Manu Kaul, Mr. Jinsong Lan, Miss Youyang Chen, Prof. Yanjie Fu, Prof. Danhuai Guo, Prof. Shuguang Liu, Prof. Yong Ge, Prof. Yuanchun Zhou, and Prof. Jianhui Li for the collaborations.

My thanks also go to the members of my Thesis Defense Committee, Prof. Dimitris Papadias, Prof. Wilfred Siu Hung Ng, Prof. Xueqing Zhang, and Prof. Mohamed F. Mokbel for reading through my thesis and sitting on this committee. I would also like to thank Prof. Qiong Luo and Prof. Ke Yi for being members of my PhD Thesis Proposal defense in April 2015 and Prof. Dik Lun Lee for being one of the committee members of my PhD Qualifying Examination in January 2012.

I want to give my thanks to my fellow colleagues at HKUST, Yu Peng, Lian Liu, Bin Zhang, Liangliang Ye, Peng Peng, Minhao Jiang, Min Xie, Junqiu Wei, Pakawadee Pengcharoen, Chengxi Yang, Kwok Wai Wong, Kai Ho Chan, Kin Long Ho, Ya Gao, Anhua Chen, Tao Zhong, Coleman Yu, to my friends at USC, Tobias Emrich, Ying Lu, Dingxiong Deng, Huy Pham, and Hien To, and to my friends at UM, Fei Li, Bernardo Gonalves, Zhongjun Jin, Yuanchao Shu, Peng Sun, Xiaoyan Zhu, Yili Gong, Seowoo Jang, Nikita Bhutani, and Dana Wilson. It was enjoyable to have meetings, lunches, dinners and outdoor activities together with them.

My special thanks go to Jiaying, for her accompany, care and support.

Last, I want to thank my dear parents, my brother, my sister-in-law and my cute nephew and little niece. To them, I dedicate the thesis.

Thank you all.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ON BOOSTING SPATIAL COMPUTATIONS FOR LOCATION-BASED SERVICES

by

CHENG LONG

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

ABSTRACT

Nowadays, *location-based services* (LBSs), which refer to those services that are based on *location* (or *spatial*) data, are broadly used in our daily life. Some popular types of LBS include "search-nearby" which searches objects (e.g., restaurants, hotels and shops) near a location, "spatial crowdsourcing" which allows people to post tasks to be performed at a location (these people are called "requesters") and people to pick some tasks to perform (these people are called "workers"), and "trace tracking" which records the trace of a movement (e.g., the moving trace of a hiker). Each type of LBS usually relies on some computation based on spatial data (which is termed as *spatial computation*). For example, the "search-nearby" service relies on *spatial keyword query* to find all objects that are near a given query location and contain a given query keyword, the "spatial crowdsourcing" service relies on *spatial matching* to match between tasks and workers, and the "trace tracking" service relies on *trajectory data*

management.

In this thesis, we introduce three techniques for boosting the spatial computations that are central to LBSs, namely the *collective spatial keyword query* which is one type of spatial keyword query and finds a set of spatial objects that cover all the given query keywords and have the smallest distance from the query location, *worst-case optimized spatial matching* which matches two sets of spatial objects with the smallest worst-case cost, and *direction-preserving trajectory* which simplifies the trajectory while preserving the direction information embedded in the trajectory data.

CHAPTER 1

INTRODUCTION

Nowadays, *location-based services* (LBSs), which refer to those services that are based on *location* (or *spatial*) data, are broadly used in our daily life. Some types of LBSs include the "search-nearby" service [25, 80, 66, 81] which searches some places (e.g., restaurants, hotels and shops) near a location, the "spatial crowdsourcing" service [52, 53, 29, 88] which allows people to post tasks to be performed at a location, and the "trace tracking" service [107, 102] which records the trace of a movement.

Many apps for smart phones provide some types of LBS. Some examples that provide the "search-nearby" service include Yelp, Gowalla, Google Places, NearbyFeed, Sonar, OpenTable, and WeChat. Some examples that provide the "spatial crowdsourcing" service include LocalHands, Airtasker, gMission and iRain. Some examples that provide the "trace tracking" service include GeoFlyer, Maps+, Nike+ and Run Keeper. According to *appcrawlr.com*, there are more than 500 apps that provide LBSs.

Each type of LBS relies on some computation based on spatial data which is called *spatial computation*. For example, one type of spatial computation that is the core of the "search-nearby" service is *spatial keyword query*, which finds the nearest objects that contain a given query keyword to a query location, one for the "spatial crowdsourcing" service is *spatial matching*, which finds a matching between two sets of spatial objects (since spatial crowdsourcing usually involves a procedure of matching the tasks with the workers), and one for the "trace tacking" service is *trajectory data management* (since the trace of a movement is usually represented by a trajectory).

In this thesis, we introduce three techniques, namely the *Collective Spatial Key-*

word Query which is one type of spatial keyword queries, *Worst-Case Optimized Spatial Matching* which is one type of spatial matching and *Direction-Preserving Trajectory Simplification* which could be used for trajectory data management.

1.1 Collective Spatial Keyword Query

Recently, spatial keyword queries become a hot topic in the literature. One example of these queries is the *collective spatial keyword query* (CoSKQ) which is to find a set of objects in the database such that it *covers* a set of given keywords collectively and has the smallest *cost*. Unfortunately, existing exact algorithms have severe scalability problems and existing approximate algorithms, though scalable, cannot guarantee near-to-optimal solutions. Motivated by this, we study the CoSKQ problem and address the above issues. Firstly, we consider the CoSKQ problem using an existing cost measurement called the *maximum sum cost*. This problem is called MaxSum-CoSKQ and is known to be NP-hard. We observe that the maximum sum cost of a set of objects is dominated by at most *three* objects which we call the *distance owners* of the set. Motivated by this, we propose a distance owner-driven approach which involves two algorithms: one is an exact algorithm which runs faster than the best-known existing algorithm by several orders of magnitude and the other is an approximate algorithm which improves the best-known constant approximation factor from 2 to 1.375. Secondly, we propose a new cost measurement called the *diameter cost* and CoSKQ with this measurement is called Dia-CoSKQ. We prove that Dia-CoSKQ is NP-hard. With the same distance owner-driven approach, we design two algorithms for Dia-CoSKQ: one is an exact algorithm which is efficient and scalable and the other is an approximate algorithm which gives a $\sqrt{3}$-factor approximation. We conducted extensive experiments on real datasets which verified that the proposed exact algorithms are scalable and the proposed approximate algorithms return near-to-optimal solutions.

1.2 Worst-Case Optimized Spatial Matching.

Bichromatic reverse nearest neighbor (BRNN) queries have been studied extensively in the literature of spatial databases. Given a set P of service-providers and a set O of customers, a BRNN query is to find which customers in O are "interested" in a given service-provider in P. Recently, it has been found that this kind of queries lacks the consideration of the *capacities* of service-providers and the *demands* of customers. In order to address this issue, some spatial matching problems have been proposed, which, however, cannot be used for some real-life applications like emergency facility allocation where the maximum matching cost (or distance) should be minimized. Motivated by this, we propose a new problem called *SPatial Matching for Minimizing the Maximum matching distance* (SPM-MM). Then, we design two algorithms for SPM-MM, *Threshold-Adapt* and *Swap-Chain*. *Threshold-Adapt* is simple and easy to understand but not scalable to large datasets due to its relatively high time/space complexity. *Swap-Chain*, which follows a fundamentally different idea from *Threshold-Adapt*, runs faster than *Threshold-Adapt* by orders of magnitude and uses significantly less memory. We conducted extensive empirical studies which verified the efficiency and scalability of *Swap-Chain*.

1.3 Direction-Preserving Trajectory Simplification

Trajectories of moving objects are collected in many applications. Raw trajectory data is typically very large, and has to be simplified before use. Motivated by this, we introduce the notion of *direction-preserving trajectory simplification* (DPTS), and show both analytically and empirically that it can support a broader range of applications than traditional position-preserving trajectory simplification (PPTS).

1.3.1 On Minimizing the Size

Within DPTS, we define a problem called *Min-Size* which accepts as input a trajectory and an *error tolerance* and finds as output the simplified trajectory of the trajectory, which has the (directional) error at most the error tolerance and as few positions as follows. To solve the Min-Size problem, we design two exact algorithms and one approximate algorithm with a quality guarantee. Extensive experimental evaluation with real trajectory data shows the benefit of the new techniques.

1.3.2 On Minimizing the Error

The Min-Size problem within DPTS require users to specify an error tolerance which users might not know how to set properly in some cases (e.g., the error tolerance could only be known at some future time and simply setting one error tolerance does not meet the needs since the simplified trajectories would usually be used in many different applications which accept different error tolerances). In these cases, a better solution is to minimize the error while achieving a pre-defined simplification size. Motivated by this, we define another problem for DPTS, called *Min-Error*, which accepts as input a trajectory and a *budget* which is an integer and finds as output the simplified trajectory of the trajectory, which has the number of positions at most the budget and the (directional) error as small as possible. Note that the Min-Error problem is the dual problem of the Min-Size problem. To solve the Min-Error problem, we develop two exact algorithms and one 2-factor approximate algorithm. Extensive experiments on real datasets were conducted and verified our algorithms.

The rest of this thesis is organized as follows. We introduce the collective spatial keyword query, worst-case spatial matching, the Min-Size problem for DPTS, and the Min-Error problem for DPTS in Chapter 2, Chapter 3, Chapter 4, and Chapter 5, respectively. We conclude the thesis in Chapter 6 with some future research directions.

CHAPTER 2

COLLECTIVE SPATIAL KEYWORD SEARCH

2.1 Introduction

With the proliferation of spatial-textual data such as location-based services and geo-tagged websites, *spatial keyword queries* have been studied extensively recently [34, 25, 97, 14]. Given a set of spatial-textual objects and a query constituted by a location and a set of keywords, a typical spatial keyword query finds the object that *best* matches the arguments in the query. One example is to find the object closest to the query location among all objects that cover all the keywords specified in the query [97].

In some applications, users' needs (expressed as keywords) are satisfied by multiple objects *collectively* instead of a *single* object [15]. For instance, a tourist wants to have site-seeing, shopping and dining which could only be satisfied by *multiple* objects, e.g., tourist attractions, shopping malls and restaurants. Another example is that a user would like to set up a project consortium of partners within a certain region that combine to offer the capabilities required for the successful execution of the whole project. Finding multiple objects collectively to satisfy users' needs can be addressed by *Collective Spatial Keyword Query* (CoSKQ) [15].

Specifically, CoSKQ is described as follows. Let $\mathcal{O}$ be a set of objects. Each object $o \in \mathcal{O}$ is associated with a spatial location, denoted by $o.\lambda$, and a set of keywords, denoted by $o.\psi$. Given a query q with a location $q.\lambda$ and a set of keywords $q.\psi$, CoSKQ is to find a set S of objects such that S covers $q.\psi$, i.e., $q.\psi \subseteq \cup_{o \in S} o.\psi$, and the *cost* of S, denoted by $cost(S)$, is minimized.

There are different cost functions for $cost(S)$. One cost function is called the *maximum sum cost function*, denoted by $cost_{MaxSum}(S)$, and was studied in [15]. It is the linear combination of two *max* components: the maximum distance between q and an object in S and the maximum distance between two objects within S. CoSKQ adopting this cost function is called ***MaxSum-CoSKQ***. The other cost function is called the *diameter cost function*, denoted by $cost_{Dia}(S)$. It is defined to be the *diameter* of $S \cup \{q\}$. In fact, diameter-related cost functions have been commonly adopted in graph databases [4, 61, 6, 68] and spatial databases [103, 104, 105]. To the best of our knowledge, we are the first to study this cost function for CoSKQ. CoSKQ adopting this cost function is called ***Dia-CoSKQ***.

Given a query q, an object o is said to be *relevant* (to q) if o contains at least one keyword in $q.\psi$. We denote by $\mathcal{O}_q$ the set of all relevant objects to q. It is sufficient to focus on $\mathcal{O}_q$ only for a specific query q. Given a set S of objects, S is said to be *feasible* if S covers $q.\psi$. Thus, the optimal solution of CoSKQ is a feasible set with the smallest cost.

Although MaxSum-CoSKQ (which is proved to be NP-hard) has been studied by Cao et al. [15], the best-known exact algorithm which we call Cao-Exact is not scalable to large datasets and the two existing approximate algorithms which we call Cao-Appro1 and Cao-Appro2 do not have a very good theoretical guarantee. Specifically, Cao-Exact is a best-first search method based on the feasible set space whose size is $O(|\mathcal{O}_q|^{|q.\psi|})$. Though equipped with some pruning techniques, Cao-Exact is prohibitively expensive when the dataset is large. For example, in our experiments, Cao-Exact took more than 10 days for a query containing 6 keywords on a dataset with 8M objects.

In this thesis, we propose two algorithms for MaxSum-CoSKQ, *MaxSum-Exact* and *MaxSum-Appro*. MaxSum-Exact is an exact algorithm and MaxSum-Appro is a

1.375-approximate algorithm.

MaxSum-Exact is more scalable compared with the best-known algorithm, Cao-Exact. A key observation which is used by MaxSum-Exact is that the number of distinct *costs* of all possible feasible sets is *cubic* (in terms of $|\mathcal{O}_q|$) although the number of all possible *feasible sets* is *exponential* (in terms of $|q.\psi|$). Given a feasible set S, the maximum sum cost function of S is dominated (or determined) by at most *three* objects in S, namely the object with the greatest distance from q and the two objects with the greatest pairwise distance within S. We say that these three objects form the *distance owner group* of S. Thus, the number of distinct costs of all possible feasible sets is bounded by the total number of all possible distance owner groups (which is bounded by $O(|\mathcal{O}_q|^3)$). Motivated by this, we propose a distance-owner driven approach called MaxSum-Exact for MaxSum-CoSK. MaxSum-Exact is a search algorithm based on the search space containing all possible distance owner groups. Besides, it incorporates some search strategies which can prune the search space effectively. Usually, *one* distance owner group corresponds to *many* feasible sets. This is verified by our experiments where MaxSum-Exact ran faster than Cao-Exact by 1-3 orders of magnitude.

MaxSum-Appro, the proposed approximate algorithm, improves the best-known constant approximation factor from 2 to 1.375 without incurring a higher worst-case time complexity.

Furthermore, we consider Dia-CoSKQ which has not been studied in the literature. In this thesis, we prove that Dia-CoSKQ is NP-hard. We also adapt Cao-Exact, Cao-Appro1 and Cao-Appro2 for Dia-CoSKQ. However, these adapted algorithms suffer from the same drawbacks in MaxSum-CoSKQ.

Motivated by this, we propose two algorithms, namely *Dia-Exact* and *Dia-Appro*. Dia-Exact is an exact algorithm which is also a search algorithm based on the search space containing all possible *distance owner groups* and thus it is scalable to large

datasets. Dia-Appro gives a $\sqrt{3}$-factor approximation for Dia-CoSKQ.

We summarize our main contributions as follows.

- Firstly, for MaxSum-CoSKQ, we design two algorithms, MaxSum-Exact and MaxSum-Appro. MaxSum-Exact is more scalable than the best-known exact algorithm, Cao-Exact. MaxSum-Appro improves the best-known constant approximation factor from 2 to 1.375 without incurring a higher worst-case time complexity.
- Secondly, for Dia-CoSKQ, which is new, we prove its NP-hardness and develop two algorithms, Dia-Exact and Dia-Appro. Dia-Exact significantly outperforms the adaptation of Cao-Exact, and Dia-Appro gives a $\sqrt{3}$-factor approximation.
- Thirdly, we conducted extensive experiments on both real and synthetic datasets, which verified our theoretical results and the efficiency of our algorithms.

The rest of this chapter is organized as follows. Section 2.2 gives the definition of the CoSKQ problem and its existing solutions. Section 2.4 and Section 2.5 study MaxSum-CoSKQ and Dia-CoSKQ, respectively. Section 2.6 gives the empirical study and Section 2.3 reviews the related work. Section 2.7 concludes this chapter.

2.2 Background

2.2.1 Problem Definition

Let $\mathcal{O}$ be a set of objects. Each object $o \in \mathcal{O}$ is associated with a location denoted by $o.\lambda$ and a set of keywords denoted by $o.\psi$. Given two objects o and o', we denote by $d(o, o')$ the Euclidean distance between $o.\lambda$ and $o'.\lambda$. Given a query q which consists of a location $q.\lambda$ and a set of keywords $q.\psi$, we denote by $\mathcal{O}_q$ the set of **relevant** objects each of which contains at least one keyword in $q.\psi$, and say that a set of objects is

feasible if it covers $q.\psi$. Besides, we introduce a fictitious object o_q in $\mathcal{O}$ with $o_q.\lambda$ = $q.\lambda$ and $o_q.\psi = \emptyset$. For simplicity, we shall also refer to object o_q as q.

Problem Definition [15]. Given a query $q = (q.\lambda, q.\psi)$, the *Collective Spatial Keyword Query* (CoSKQ) problem is to find a set S of objects in $\mathcal{O}$ such that S covers $q.\psi$ and the *cost* of S is minimized.

In this thesis, we consider two cost functions, the *maximum sum cost* and the *diameter cost*.

Given a set S of objects, the *maximum sum cost* of S, denoted by $cost_{MaxSum}(S)$, is equal to the linear combination of the maximum distance between q and an object in S and the maximum distance between two objects in S. That is,

$$cost_{MaxSum}(S) = \alpha \cdot \max_{o \in S} d(o, q) + (1 - \alpha) \cdot \max_{o_1, o_2 \in S} d(o_1, o_2) \tag{2.1}$$

where $\alpha \in [0, 1]$ is a user parameter. Same as [15], for ease of exposition, we consider the case where $\alpha = 0.5$ only. In this case, we can safely assume that

$$cost_{MaxSum}(S) = \max_{o \in S} d(o, q) + \max_{o_1, o_2 \in S} d(o_1, o_2) \tag{2.2}$$

In fact, the applicability of all of our algorithms does not rely on the setting of α. The only part that is affected is the approximation factor of our approximate algorithm which is *bounded* by $(2 - \sqrt{2}/2 \cdot \alpha)$ (e.g., when $\alpha = 0.5$, the approximation factor of our approximate algorithm is 1.375 which is bounded by $(2 - \sqrt{2}/2 \cdot \alpha) \approx 1.65$). More details in Section A.1 of the Appendix. The CoSKQ problem using this cost is called *MaxSum-CoSKQ*.

As could be noticed, parameter α in the maximum sum cost function is used to balance the two *max* components, namely $\max_{o \in S} d(o, q)$ and $\max_{o_1, o_2 \in S} d(o_1, o_2)$. Sometimes, however, people may not have a concrete idea of how to specify α. To

ease this situation, we define an alternative cost function called *diameter cost* on a set S of objects, denoted by $cost_{Dia}(S)$, which is defined to be the larger of these two *max* components. That is,

$$cost_{Dia}(S) = \max_{o_1,o_2 \in S \cup \{o_q\}} d(o_1, o_2) \tag{2.3}$$

The CoSKQ problem using this cost is called *Dia-CoSKQ*.

Intractability. It has been proved in [15] that MaxSum-CoSKQ is NP-hard. In this thesis, we prove that Dia-CoSKQ is also NP-hard.

Lemma 2.2.1 *Dia-CoSKQ is NP-hard.* □

Proof. We first give the decision problem of Dia-CoSKQ. Let $\mathcal{O}$ be a set of spatial objects each of which is associated with a set of keywords. Given a number C and a query q consisting of a location $q.\lambda$ and a set of keywords $q.\psi$, the problem is to determine whether there exists a set S of objects such that S covers $q.\psi$ and $cost_{Dia}(S) < C$. For simplicity, we denote this decision problem by Dia-CoSKQ.

We then utilize a well-known NP-C problem *3-SAT* for proving that Dia-CoSKQ is NP-hard. 3-SAT is described as follows. Let U be a set of literals (binary variables) $\{e_1, \overline{e_1}, ..., e_n, \overline{e_n}\}$. Note that $\overline{e_i}$ corresponds to the negation of e_i. Given an expression $E = C_1 \wedge C_2 \wedge ... \wedge C_m$ where C_i has its from of $x_i \vee y_i \vee z_i$ and $x_i, y_i, z_i \in U$ for $1 \leq i \leq m$, it determines whether there exists a truth assignment for e_i for $1 \leq i \leq n$ such that E is true.

Given a 3-SAT problem instance, we construct a Dia-CoSKQ problem instance as follows. We arbitrarily select a location as $q.\lambda$ and for each clause C_i in the expression E, we include C_i in $q.\psi$ as a keyword. Thus, q contains m keywords. To construct $\mathcal{O}$, we consider the circle Cir with its center at $q.\lambda$ and its radius equal to 1. For each pair of literals e_i and $\overline{e_i}$ in U, we create two objects o_i and o_i' on the boundary of Cir

such that $d(o_i, o_i') = 2$, i.e., the line segment between o_i and o_i' is a diameter of Cir. We guarantee that no two objects in $\mathcal{O}$ share the same location. Besides, we set $o_i.\lambda$ ($o_i'.\lambda$) to be the set of clauses that contain e_i ($\overline{e_i}$). We set C to be 2. Clearly, the above construction process could be finished in polynomial time.

We proceed to show that the above constructed Dia-CoSKQ problem instance is equivalent to its corresponding 3-SAT problem instance. Assume that the answer to 3-SAT is "yes", i.e., there exists a truth assignment A for the literals in U such that E is true. Then, we construct a set S of objects in $\mathcal{O}$ as follows. For each positive literal $e_i \in U$, we include o_i in S if e_i is true in A. For each negative literal $\overline{e_i} \in U$, we include o_i' in S if e_i is false in A. Clearly, o_i and o_i' cannot be included in S simultaneously. Consider S. First, S covers $q.\psi$ which could be verified by contradiction. Assume that there exists a keyword C_i in $q.\psi$ which is not covered by S. Since E is true, C_i must also be true. Consequently, C_i contains at least one literal which is true. Case 1: this literal is a positive literal e_i. In this case, o_i is included in S and thus S covers C_i, which leads to a contradiction. Case 2: this literal is a negative literal $\overline{e_i}$. In this case, e_i is false and thus o_i' is included in S. As a result, S covers C_i, which, again, leads to a contradiction. Second, we know $cost_{Dia}(S) < C = 2$. This is because $\max_{o \in S} d(o) = 1$ and $\max_{o_1, o_2 \in S} d(o_1, o_2) < 2$ (there exist no pairs of two objects such that the segment between them forms a diameter).

Consider the other direction. Assume that the answer to Dia-CoSKQ is "yes", i.e., there exist a set S of objects such that S covers $q.\psi$ and $cost_{Dia}(S) < C$. Then, we construct a truth assignment A for the literals in U as follows. For each object $o_i \in S$, we set e_i to be true and consequently $\overline{e_i}$ is false. For each object $o_i' \in S$, we set e_i to be false and consequently $\overline{e_i}$ is true. For the remaining literals to which no truth values have been assigned, we set their truth values arbitrarily with the constraint that e_i and $\overline{e_i}$ have different truth values. First, we show that E is a *valid* assignment, i.e.,

there exist no pairs of two literals e_i and $\overline{e_i}$ such that they have a common truth value. This could be verified by contradiction. Assume that there exist two literals e_i and $\overline{e_i}$ such that e_i and $\overline{e_i}$ have the same truth value. It follows that o_i and o_i' co-exist in S which contradicts the assumption that $cost_{Dia}(S) < C = 2$. Second, we show that E is true with the truth assignment A. Again, this could be verified by contradiction. Assume that E is false. Then, there exists a clause C_i which is false. It follows that each positive literal e_i in C_i (if any) is false and thus o_i is not included in S. Each negative literal $\overline{e_i}$ in C_i (if any) is false and thus o_i' is not included in S. As a result, S does not cover C_i which contradicts that S covers $q.\psi$. Therefore, E is true and thus the answer to the 3-SAT problem is "yes".

Therefore, we know that Dia-CoSKQ is NP-hard. ■

2.2.2 Existing Solutions

Cao et el. [15] proposed one exact algorithm, Cao-Exact, and two approximate algorithms, Cao-Appro1 and Cao-Appro2, for MaxSum-CoSKQ.

Cao-Exact. Cao-Exact is a best-first search method using an index called *IR-tree* [25]. An IR-tree is an R-tree in which each node is augmented with an *Inverted File* (IF). Consider a leaf node N. For each keyword t, we construct an *inverted list* which is a list of all objects in node N containing t. All inverted lists in this leaf node N form the IF of N. Consider a non-leaf node N'. For each keyword t, we construct an *inverted list* which is a list of all child nodes in N' *covering* t. Given a keyword t, a node N'' is said to cover t if there exists an object in the subtree rooted at N'' containing t. All inverted lists in this non-leaf node N' form the IF of N'.

Cao-Exact is basically an exhaustive search on the object space with some pruning strategies in the IR-tree. The worst-case time complexity of Cao-Exact is $O(|\mathcal{O}|^{|q.\psi|})$, which corresponds to the size of the set containing all possible feasible sets.

Cao-Appro1. Cao-Appro1 gives a 3-factor approximation for MaxSum-CoSKQ. Specifically, Cao-Appro1 finds for each $t \in q.\psi$, q's nearest neighbor (NN) in $\mathcal{O}$ containing t and returns the set containing all these NNs as the approximate solution. Since Cao-Appro1 issues NN queries at most $|q.\psi|$ times and each NN query takes $O(\log |\mathcal{O}|)$ time [17, 28, 75], the time complexity of Cao-Appro1 is $O(|q.\psi| \cdot \log |\mathcal{O}|)$.

Cao-Appro2. Cao-Appro2 gives a 2-factor approximation for MaxSum-CoSKQ. Specifically, Cao-Appro2 enhances Cao-Appro1 as follows. First, Cao-Appro2 invokes Cao-Appro1 and obtains an approximate solution denoted by S_1. Let o_f be the farthest object from q in S_1 and t_f be a keyword contained by o_f but not contained by any other *closer* object from q in $\mathcal{O}$. Then, for each object o in $\mathcal{O}$ containing t_f, it finds for each keyword t in $q.\psi$, o's nearest object that contains t in $\mathcal{O}$ and obtains a corresponding approximate solution containing all these NNs. Among all these approximate solutions as well as S_1, it returns the one with the smallest cost. Thus, the approximate solution returned by Cao-Appro2 is no worse than that returned by Cao-Appro1. Since there are at most $|\mathcal{O}_q|$ objects containing t_f and the cost for each such object is simply $O(|q.\psi| \cdot \log |\mathcal{O}|)$, the worst-case time complexity of Cao-Appro2 is $O(|\mathcal{O}_q| \cdot |q.\psi| \cdot \log |\mathcal{O}|)$.

2.3 Related Work

Many types of spatial keyword query have been proposed in the literature. Most of them are different from CoSKQ studied in this thesis since they use a single object to cover all keywords specified in the query but CoSKQ uses multiple objects collectively for the same purpose. We review these spatial keyword queries as follows.

A *spatial keyword top-k query* [25] finds top-k objects where the ranking function takes both the spatial proximity and the textual relevance of the objects into consideration. This branch includes [25, 80, 66] (Euclidean space), [81] (road networks),

[83, 26] (trajectory databases), and [98] (moving objects). A common technique shared by these studies is to design a hybrid indexing structure, which captures both the spatial proximity and the textual information of the objects. The IR-tree adopted by us for NN queries and range queries was proposed in [25].

A *spatial keyword k-NN query* [34] finds the k-NNs from the query location, each of which contains the set of keywords specified in the query. That is, unlike the keywords in the spatial keyword top-k queries, which are used as a *soft* constraint, the keywords in the spatial keyword k-NN queries are used as a *hard* constraint. This branch includes [34, 16, 97].

A *spatial keyword range query* [92, 109, 24] takes a region and a set of keywords as input and finds the objects each of which falls in the region and contains the set of keywords. Same as the spatial keyword k-NN queries, the keywords are used as a hard constraint. Usually, they combine a spatial index (e.g., R-tree and Space Filling Curve (SFC)) and a textual index (e.g., inverted file) for query processing.

A *spatial keyword reverse top-k query* [70] finds the set of objects whose spatial keyword top-k query results include the query. Note that in this case, an object which consists of a location and a set of keywords could be regarded as a query which also consists of a location and a set of keywords and vice versa.

An *mCK query* [103, 104] is a spatial keyword query that is very similar to CoSKQ. An mCK query takes m keywords as input and finds m objects with the smallest *diameter* that cover the m keywords specified in the query. Though both the mCK query and CoSKQ use a set of objects for covering a set of keywords collectively, they are different. In the context of an mCK query, it is assumed that each object is associated with a single keyword while in the context of CoSKQ, each object is associated with a set of multiple keywords. Besides, an mCK query only takes a set of keywords as input while our CoSKQ query takes not only a set of keywords but also a query location as

an input.

In addition to the spatial keyword queries, *trip planning* [65, 84, 20, 13] is another branch of related work. In [65], the authors studied a query called "Trip Planning Query" (TPQ). Given a metric graph consisting of a set of points as vertices each of which belongs to a category, a starting point, an ending point, and a set of query categories, the query is to find a route from the starting point to the ending point on the graph such that for each query category, the route traverses a point belonging to this category and the cost of the route is the smallest. In [84], the authors studied a query called "Optimal Sequenced Route" (OSR). Given a set of points each of which belongs to a category, a starting point and a sequence of query categories, the query is to find a route which corresponds a sequence of points such that the route *follows* the sequence of query categories meaning that the sequence of the corresponding categories of the points in the route is exactly the same as the sequence of query categories and the sum of the cost from the starting point to the first point in the route and the cost of the route is minimized. In [20], the authors studied a query called "Multi-Rule Partial Sequenced Route" (MRPSR) which unified TPQ an OSR. In [13], the authors studied a query called "Keyword-aware Optimal Route (KOR)" which corresponds to a variant of TPQ by considering an additional budget constraint. All of these are different from the CoSKQ queries studied in this thesis simply because they output a route while CoSKQ outputs a set.

2.4 Algorithms for MaxSum-CoSKQ

In this section, we propose two algorithms, MaxSum-Exact (Section 2.4.1) and MaxSum-Appro (Section 2.4.2), for MaxSum-CoSKQ. For clarity, we simply write $cost_{MaxSum}(\cdot)$ as $cost(\cdot)$ if the context of the cost function is clear.

Given a query q and a non-negative real number r, we denote the *circle* or the *disk*

centered at $q.\lambda$ with radius r by $D(q, r)$. Given a disk D, we denote the radius of D by $radius(D)$. Given a query q, a disk centered at $q.\lambda$ is called a *q-disk*. Given a q-disk D and an object o in D, o is said to be the *boundary object* of D if there does not exist other objects o' in D such that $d(o', q) > d(o, q)$. Note that in some cases, a boundary object of a disk is along the boundary of a disk and in some other cases, it is inside the disk without touching the boundary of the disk.

2.4.1 Exact Algorithm

In this section, we propose an exact algorithm called *MaxSum-Exact*. The key to the efficiency of MaxSum-Exact is based on the splitting property of the maximum sum cost function.

Splitting Property

Let S' be a feasible set. The maximum sum cost of S' can be split into two parts, namely the *query distance cost* which is $\max_{o \in S'} d(o, q)$ and the *pairwise distance cost* which is $\max_{o_1, o_2 \in S'} d(o_1, o_2)$. We define the **query distance owner** of S' to be o where $o = \arg\max_{o \in S'} d(o, q)$. We also define the **pairwise distance owners** of S' to be o_1 and o_2 where $(o_1, o_2) = \arg\max_{(o'_1, o'_2) \in S' \times S'} d(o'_1, o'_2)$.

Consider Figure 2.1 containing a query location q and 5 objects, namely o_1, o_2, o_3, o_4 and o_5. The set of keywords associated with each object can be found in the figure. Suppose that $q.\psi = \{t_1, t_2, t_3\}$. We know that a set $S' = \{o_1, o_2, o_3\}$ is feasible. The query distance owner of S' is o_1 and the pairwise distance owners of S' are o_2 and o_3.

According to the above splitting property, the cost of a set S' can be dominated (or determined) by exactly three objects in S', namely the query distance owner of S' (i.e., o) and the two pairwise distance owners of S' (i.e., o_1 and o_2). In other words, we can

simply write the cost of S' as follows.

$$cost(S') = d(o, q) + d(o_1, o_2)$$

where o is the distance owner of S', and o_1 and o_2 are the two pairwise distance owners of S'. We say that o, o_1 and o_2 forms a **distance owner group**. Any feasible set with its query distance owner as o and its pairwise distance owners as o_1 and o_2 is said to be (o, o_1, o_2)**-owner consistent**. Note that each feasible set that is (o, o_1, o_2)-owner consistent has the same cost equal to $d(o, q) + d(o_1, o_2)$.

Distance Owner-Driven Approach

Based on the splitting property, we propose a *distance owner-driven approach* as follows. This approach maintains a variable S storing the best feasible set found so far. Initially, S is set to a feasible set (We will describe how we find this feasible set later). Then, it has four major steps.

- *Step 1 (Query Distance Owner Finding):* Select one object o in $\mathcal{O}_q$ to take the role of the query distance owner of a set S' *to be found*.
- *Step 2 (Pairwise Distance Owner Finding):* Select two objects, o_1 and o_2, in $\mathcal{O}_q$ to take the roles of the pairwise distance owners of the set S' (to be found). Note that o, o_1 and o_2 form a distance owner group.
- *Step 3 (Sub-Optimal Feasible Set Finding):* Find the set S' which is (o, o_1, o_2)-owner consistent (if any), and update S with S' if $cost(S') < cost(S)$.
- *Step 4 (Iterative Step):* Repeat Step 1 and Step 2 which find another *distance owner group*, and continue with Step 3 until all distance owner groups are traversed.

The above approach gives a search strategy based on the set of all possible distance owner groups. However, a straightforward implementation of this approach would

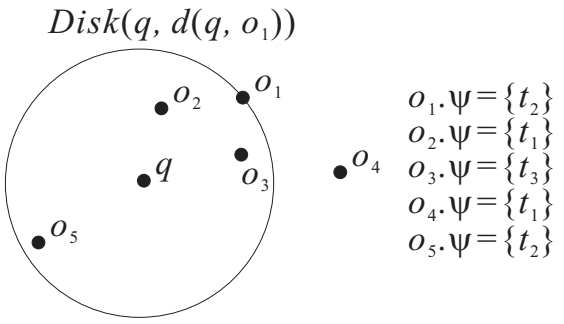


Figure 2.1: An example (CoSKQ)

enumerate all $|\mathcal{O}_q|^3$ distance owner groups, which is prohibitively expensive in practice. Thus, we need a careful design in order to prune the search space effectively. In the following, we elaborate the pruning features enjoyed by this distance owner-driven approach, which cannot be found in the best-known algorithm, Cao-Exact.

Firstly, some objects in $\mathcal{O}_q$ need not be considered in Step 2 after we select an object in Step 1. To illustrate this, consider Figure 2.1. Suppose that we pick o_1 as the query distance owner in Step 1. We do not need to consider o_4 as objects in Step 2. This is because $d(o_4, q)$ is larger than $d(o_1, q)$, which violates the property that o_1 takes the role of the query distance owner of the set S' to be found if S' contains o_1 and o_4. We formalize this pruning feature as follows.

Property 1 (Pruning) *Let S' be a feasible set. If o is the query distance owner of S', then the two pairwise distance owners of S' are inside $D(q, d(o, q))$.* □

Proof. Any object $o' \in S'$ has $d(o', q) \leq d(o, q)$ and thus o' is inside $D(q, d(o, q))$.

■

Secondly, most of the objects in $\mathcal{O}_q$ need not be considered to form a set S' to be found in Step 3. To illustrate this, consider Figure 2.1 again. Suppose that we pick o_1 as the query distance owner in Step 1, and o_2 and o_3 as the pairwise distance owners in Step 2. Similarly, we still do not need to consider o_4 as one of the objects to form the set S' since including o_4 violates the query distance owner property. Besides, we do not need to consider o_5 to form the set S' to be found. This is because $d(o_2, o_5) > d(o_2, o_3)$

which violates the property that o_2 and o_3 take the roles of the pairwise distance owners. Similarly, we formalize this pruning feature as follows.

Property 2 (Pruning) *Let S' be a feasible set. If o is the query distance owner of S', and o_1 and o_2 are two pairwise distance owners of S', then all objects in S' are inside $\mathcal{R}$ where $\mathcal{R} = D(q, d(o, q)) \cap D(o_1, d(o_1, o_2)) \cap D(o_2, d(o_1, o_2))$.* □

Proof. For each $o' \in S'$, we have $d(o', q) \leq d(o, q)$ which implies that o' is inside $D(q, d(o, q))$. For each $o' \in S'$, we have $d(o', o_1) \leq d(o_1, o_2)$ which implies that o' is inside $D(o_1, d(o_1, o_2))$, and $d(o', o_2) \leq d(o_1, o_2)$ which implies that o' is inside $D(o_2, d(o_1, o_2))$. ■

The above pruning features look promising for improving the efficiency of the proposed approach. Moreover, since objects *near* to q usually form the optimal set, we propose to consider the objects in Step 1 iteratively, taking the role of the query distance owner of the set to be found, in ascending order of their distances to q in order to further improve the efficiency of the proposed approach.

Usually, the NN of q in $\mathcal{O}_q$ is not the query distance owner of the set S' to be found. In Figure 2.1, consider the query q with its keyword set to be $\{t_1, t_2, t_3\}$. The NN of q is o_2. Suppose that o_2 is the query distance owner of S'. According to Property 2, all objects in S' fall in $D(q, d(o_2, q))$ and they together cover $q.\psi$. But, in the figure, no object in $D(q, d(o_2, q))$ contains t_2, which implies that we cannot find a feasible set S' with o_2 as its query distance owner.

Based on this observation, we propose to find the *closest possible query distance owner*, say o, of the set S' to be found such that there exists a feasible set in the q-disk $D(q, d(o, q))$. In addition, we do not want to pick any object which is far away from q. Thus, we also propose to find the *farthest possible query distance owner* of S' to be found that we need to consider.

Closest/Farthest Possible Query Dist. Owner

The following two lemmas show how to find the closest and farthest possible query distance owners.

Before we present the first lemma about the closest possible query distance owner, we introduce some notations. Given a query q and a keyword t, the **t-keyword nearest neighbor** of q, denoted by $NN(q, t)$, is defined to be the NN of q containing keyword t. We have a similar definition on $NN(o, t)$ for an object o. We define the **nearest neighbor set** of q, denoted by $N(q)$, to be the set containing q's t-keyword nearest neighbor for each $t \in q.\psi$, i.e., $N(q)$ is $\cup_{t \in q.\psi} NN(q, t)$. Note that $N(q)$ is a feasible set.

Lemma 2.4.1 (Closest Poss. Query Dist. Owner) *Let* $r_{min} = \max_{o \in N(q)} d(o, q)$. *There exists a feasible set in a q-disk D if and only if* $radius(D) \geq r_{min}$. □

Proof. The proof for the "if" part is trivial since for any q-disk D with $radius(D) \geq r_{min}$, $N(q)$ is a feasible set in D. We prove the "only if" part by contradiction. Assume $radius(D) < r_{min}$ and there exists a feasible set S in D. Let o_f be the farthest object from q in $N(q)$, i.e., $r_{min} = d(q, o_f)$. There exists a keyword $t_f \in o_f.\psi \cap q.\psi$ such that t_f is not contained by any object that is closer to q than o_f since otherwise $o_f \notin N(q)$. Since S is feasible, there exists an object $o \in S$ that contains keyword t_f. As a result, we have $d(o, q) \leq radius(D) < r_{min} = d(q, o_f)$, which leads to a contradiction. ■

The above lemma suggests that there is no feasible set in a q-disk D if $radius(D) < r_{min}$. Thus, the disk with its radius equal to r_{min} is the "smallest" disk we need to consider. The boundary object of this disk is the closest possible query distance owner. Note that this object is along the boundary of this disk.

The following lemma gives the "largest" disk we need to consider. Besides, the boundary object of this disk corresponds to the farthest possible query distance owner.

Note that this object might or might not be along the boundary of this disk.

Lemma 2.4.2 (Farthest Poss. Query Dist. Owner) *Let S be a feasible set and $r_{max} = cost(S)$. Let D be a q-disk with $radius(D) > r_{max}$. Then, for any feasible set S' containing at least one object outside D, $cost(S') > cost(S)$.* □

Proof. $cost(S') \geq max_{o \in S'} d(o, q) > radius(D) > r_{max} = cost(S)$. ■

The above lemma suggests that when we have known a feasible set S, there is no need to consider the objects outside $D(q, r_{max})$ where $r_{max} = cost(S)$.

The above two lemmas suggest the "smallest" disk and the "largest" disk we need to consider. Specifically, the object o which takes the role of the query distance owner of S' to be found must be in the *ring* which is roughly equal to the "largest" disk minus the "smallest" disk. Let S be a feasible set. Let $r_{min} = \max_{o \in N(q)} d(o, q)$ and $r_{max} = cost(S)$. We define the **ring** for S, denoted by $R(S)$, to be $D(q, r_{max}) - D(q, r_{min} - \delta)$, where δ is a very small positive real number near to 0.

Lemma 2.4.3 (Ring Candidate) *Let S be a feasible set and S_o be the optimal set for the MaxSum-CoSKQ problem. The query distance owner of S_o is inside $R(S)$.* □

Proof. Let o be the query distance owner of S_o. First, according to Lemma 2.4.2, o cannot be outside $D(q, r_{max})$ since otherwise $cost(S_o) > cost(S)$ which leads to a contradiction. Second, according to Lemma 2.4.1, there exist no feasible sets in $D(q, r_{min} - \delta)$. Thus, o is not inside $D(q, r_{min} - \delta)$ since otherwise S_o which is feasible is inside $D(q, r_{min} - \delta)$ which also leads to a contradiction. Therefore, o is inside $R(S)$. ■

It is easy to verify that the region occupied by $R(S)$ becomes smaller when $cost(S)$ is smaller since the radius of the outer disk of $R(S)$ is equal to $cost(S)$.

The MaxSum-Exact Algorithm

Based on the discussion in the previous subsection, we design *MaxSum-Exact* as shown in Algorithm 1. Specifically, we maintain S for storing the best-known solution found so far, which is initialized to $N(q)$. Then, we perform an iterative process as follows. Consider an iteration. We want to check whether there exists a relevant object in $R(S)$ that has not been processed. If yes, we pick the nearest relevant object o from $R(S)$ that has not been processed to take the role of the query distance owner of the set S' to be found (Step 1). This object is said to be the *query distance owner* for this iteration. We process it as follows. Firstly, we form the q-disk D with its radius equal to $d(o, q)$ and find a set P of all pairs (o_1, o_2) where o_1 and o_2 are in D for taking the roles of the pairwise distance owners (Step 2). Secondly, for each pair (o_1, o_2) in P which is processed in ascending order of $d(o_1, o_2)$, we check whether there exists a feasible set S' which is (o, o_1, o_2)-owner consistent. Case 1: yes. We do the following. Firstly, if $cost(S') < cost(S)$, then we update S by S'. Secondly, we terminate to search the remaining pairs in P since the cost of a final set whose pairwise distance owners corresponds to one of the remaining pairs must be at least the cost of the current set S' whose pairwise distance owners are (o_1, o_2), the current processed pair. Case 2: no. We continue to consider the next pair in P until Case 1 is reached or all the pairs in P have been processed. We continue the above iteration with the next relevant object from $R(S)$ that has not been processed until all objects in $R(S)$ have been processed (Step 4).

We verify the correctness of MaxSum-Exact via Theorem 2.4.1.

Theorem 2.4.1 *MaxSum-Exact returns a feasible set with the smallest cost for MaxSum-CoSKQ.* □

Proof. Let S_o be one of the feasible sets with the smallest cost for MaxSum-CoSKQ.

Algorithm 1 Algorithm *MaxSum-Exact*

Require: query q and a set $\mathcal{O}$ of objects
1: $S \leftarrow N(q)$
2: **while** there is an "un-processed" relevant object o in $R(S)$ **do**
3: // Step 1 (Query Distance Owner Finding)
4: $o \leftarrow$ the nearest "un-processed" relevant object in $R(S)$
5: // Step 2 (Pairwise Distance Owner Finding)
6: $D \leftarrow$ the q-disk with its radius equal to $d(o, q)$
7: $P \leftarrow$ a set of all pairs (o_1, o_2) where o_1 and o_2 are in D
8: // Step 3 (Sub-optimal Feasible Set Finding)
9: **for** each $(o_1, o_2) \in P$ in ascending order of $d(o_1, o_2)$ **do**
10: **if** there exists a feasible set S' in D which is (o, o_1, o_2)-owner consistent **then**
11: **if** $cost(S') < cost(S)$ **then**
12: $S \leftarrow S'$; **break**
13: // Step 4 (Iterative Process)
14: mark o as "processed"
15: **return** S

Suppose that o is the query distance owner of S_o, and o_1 and o_2 are two pairwise distance owners of S_o. According to Lemma 2.4.3, o is inside $R(S)$, where S is the solution maintained in MaxSum-Exact. Thus, o must have been processed in MaxSum-Exact (Step 1). When o is processed, pair (o_1, o_2) is included in P (Step 2) since o_1 and o_2 are inside $D(q, d(o, q))$ (Property 1). As a result, any feasible set which is (o, o_1, o_2)-owner consistent is retrieved (Step 3) and used to update S (there must exist some since S_o is (o, o_1, o_2)-owner consistent). The resulting S will not be updated anymore since it has the same cost as $cost(S_o)$ which is the smallest, and thus S is the final output. ■

Algorithm 1 looks straightforward but how to execute this algorithm *efficiently* needs more careful design. We propose two computation strategies in the algorithm, namely the *self-iteration computation strategy* and the *cross-iteration computation strategy*, to execute this algorithm efficiently. The self-iteration computation strategy is to speed up the operations within an iteration and the cross-iteration computation strategy is to speed up the operations across different iterations.

Self-Iteration Computation Strategy. Consider an iteration in the algorithm whose query distance owner is o. Step 1 (lines 3-4) is straightforward. In Step 2 (lines 5-7),

there is a step of finding a set P of all pairs (o_1, o_2) where o_1 and o_2 are in D. There is no need to keep all pairs (o_1, o_2) in P and some pairs can be pruned. The following two lemmas give some hints for pruning. The first lemma (Lemma 2.4.4) is based on the triangle inequality and the second lemma (Lemma 2.4.5) is based on the best-known set S found so far.

Lemma 2.4.4 (Triangle Inequality) *Let S' be a feasible solution whose query distance owner is o, and pairwise distance owners are o_1 and o_2. Then, $d(o_1, o_2) \geq d(o, q) - \min\{d(o_1, q), d(o_2, q)\}$.* □

Proof. Note that $d(o_1, o_2) \geq d(o_1, o)$ and $d(o_1, o_2) \geq d(o_2, o)$. By the triangle inequality, we know $d(o_1, o) \geq d(o, q) - d(o_1, q)$ and $d(o_2, o) \geq d(o, q) - d(o_2, q)$. Thus, we have $d(o_1, o_2) \geq d(o, q) - \min\{d(o_1, q), d(o_2, q)\}$. ■

The above lemma suggests that the pair (o_1, o_2) in P can be pruned if $d(o_1, o_2) < d(o, q) - \min\{d(o_1, q), d(o_2, q)\}$. Let $d_{min} = d(o, q) - \min\{d(o_1, q), d(o_2, q)\}$. Thus, d_{min} corresponds to the smallest distance threshold for a pair (o_1, o_2).

Lemma 2.4.5 (Best Known Set) *Let S' be a feasible solution whose query distance owner is o and pairwise distance owners are o_1 and o_2. Let S be another feasible solution. $cost(S') \leq cost(S)$ if and only if $d(o_1, o_2) \leq cost(S) - d(o, q)$.* □

Proof. $cost(S') \leq cost(S)$ deduces $d(o, q) + d(o_1, o_2) \leq cost(S)$ which is exactly $d(o, q) \leq cost(S) - d(o_1, o_2)$. ■

Let S be the feasible set found so far in the algorithm. The above lemma suggests that the pair (o_1, o_2) in P can be pruned if $d(o_1, o_2) > cost(S) - d(o, q)$. Let $d_{max} = cost(S) - d(o, q)$. Thus, d_{max} is the largest distance threshold for a pair (o_1, o_2).

According to Lemma 2.4.4 and Lemma 2.4.5, we only need to maintain those pairs with their distances between d_{min} and d_{max} in P.

Consider Step 3 (lines 8-12). Here, we need to process each pair (o_1, o_2) in P. The most time-consuming operation is to check whether there exists a feasible set S' which is (o, o_1, o_2)-owner consistent. Algorithm 2 presents an algorithm for this task. If it succeeds, it outputs S'; otherwise, it outputs $\emptyset$. First, it checks whether $d(o_1, o_2) < \max\{d(o_1, o), d(o_2, o)\}$. If yes, we conclude that there exist no feasible set that is (o, o_1, o_2)-owner consistent since it violates the condition that o_1 and o_2 are the pairwise distance owners (i.e., $d(o_1, o_2) \geq \max\{d(o_1, o), d(o_2, o)\}$). If no, it initializes S' to be $\{o, o_1, o_2\}$. It also maintains a variable ψ, denoting the set of keywords not covered by S' yet, which is initialized as $q.\psi - (o.\psi \cup o_1.\psi \cup o_2.\psi)$. If $\psi = \emptyset$, it returns S' immediately. Otherwise, it proceeds to augment S' with some other objects. According to Property 2, we can safely focus on the region $\mathcal{R} = D(o, d(o, q)) \cap D(o_1, d(o_1, o_2)) \cap D(o_2, d(o_1, o_2))$. Therefore, it retrieves the set $\mathcal{O}'$ of all relevant objects in $\mathcal{R}$. If $\mathcal{O}'$ does not cover ψ, it returns $\emptyset$. Otherwise, it enumerates each possible subset S'' of $\mathcal{O}'$ that covers ψ (by utilizing the *inverted lists* maintained for each keyword in ψ), augment S' by S'' (thus S' becomes feasible) and checks whether S' is (o, o_1, o_2)-owner consistent which is equivalent to checking whether o_1 and o_2 are still the pairwise distance owners of S'. If yes, it outputs S'. Otherwise, it restores S' and checks the next subset of $\mathcal{O}'$. When all subsets of $\mathcal{O}'$ that cover ψ have been traversed and still no feasible set S' which is (o, o_1, o_2)-owner consistent has been found, it returns $\emptyset$.

Cross-Iteration Computation Strategy. We reuse the information computed in the previous iterations for the current iteration.

Consider an iteration where the query distance owner for this iteration is o. With respect to o, we create a q-disk D and also construct set P (line 7 in Algorithm 1). Consider the next iteration where the query distance owner for this iteration is o'. Although we can construct set P' with respect to o' from scratch by applying the procedure of

Algorithm 2 Algorithm for checking whether there exists a feasible set S' which is (o, o_1, o_2)-owner consistent

Require: three objects o, o_1 and o_2
Ensure: a feasible set which is (o, o_1, o_2)-owner consistent if any and $\emptyset$ otherwise
1: **if** $d(o_1, o_2) < \max\{d(o_1, o), d(o_2, o)\}$ **then return** $\emptyset$
2: $S' \leftarrow \{o, o_1, o_2\}$
3: $\psi \leftarrow q.\psi - (o.\psi \cup o_1.\psi \cup o_2.\psi)$
4: **if** $\psi = \emptyset$ **then return** S'
5: $\mathcal{R} \leftarrow D(q, d(o, q)) \cap D(o_1, d(o_1, o_2)) \cap D(o_2, d(o_1, o_2))$
6: $\mathcal{O}' \leftarrow$ a set of all relevant objects in $\mathcal{R}$
7: **if** $\mathcal{O}'$ does not cover ψ **then return** $\emptyset$
8: **for** each subset S'' of $\mathcal{O}'$ that covers ψ **do**
9: $\quad S' \leftarrow S' \cup S''$
10: $\quad$ **if** S' is (o, o_1, o_2)-owner consistent **then return** S'
11: $\quad S' \leftarrow S' - S''$
12: **return** $\emptyset$

generating set P, a much better approach is to construct set P' by using the current content of P because $P \subseteq P'$. Specifically, when we consider the next iteration, we first construct another set Q to be the set of additional pairs in P' compared with P (i.e., $Q = \{(o'', o') | o'' \in D(q, d(q, o))\}$) and then set P' to be $P \cup Q$. Note that $P \cap Q = \emptyset$.

The pruning in P mentioned in the self-iteration computation strategy is still valid even when we construct P' in the above way. Specifically, the pairs pruned previously in P still do not need to be considered in P' at the next iteration. This is because d_{min} is *monotonically increasing* and d_{max} is *monotonically decreasing* with more iterations. To illustrate, consider a pair (o_1, o_2) in P at the previous iteration. Note that $d_{min} = d(o, q) - \min\{d(o_1, q), d(o_2, q)\}$ and $d_{max} = cost(S) - d(o, q)$. At the next iteration, o will become o', which is at least as far as o from q (i.e., $d(o', q) \geq d(o, q)$). Thus, at the next iteration, d_{min} will remain the same or will increase. In addition, the cost of the solution S maintained at the next iteration is at most the cost of that maintained at the previous iteration. Thus, at the next iteration, d_{max} will remain the same or will decrease.

Implementation and Time Complexity

We adopt the IR-tree built on $\mathcal{O}$ to support both the NN query (line 1 of Algorithm 1) and the range query (line 7 of Algorithm 1 and line 6 of Algorithm 2). For the NN query, we adopt the best-first search method [45] and for the range query, we perform a simple breadth-first traversal with the constraint of the range. Besides, given a query q, since we only focus on the set of relevant objects, when performing NN queries and range queries, we can utilize the IF information maintained in the IR-tree for pruning.

Since the pairs in P are processed in ascending order of their distances and P is maintained dynamically (because of the Cross-Iteration Computation Strategy), we adopt a *binary search tree* for maintaining P, which allows efficient sorting and update.

Let n_1 be the number of iterations (lines 2-14) in MaxSum-Exact (Algorithm 1). Note that $n_1 << |\mathcal{O}_q|$ since n_1 corresponds to the number of relevant objects we process in $R(S)$ and the area occupied by $R(S)$ is typically small. Let $|P|$ be the size of the set P we use in the algorithm. Similarly, we know that $|P| << |\mathcal{O}_q|^2$. Let β be the cost of Algorithm 2. It is easy to verify that the time complexity of MaxSum-Exact is $O(n_1 \cdot |P| \cdot \beta)$.

Next, we analyze β. The cost of lines 1-4 (Algorithm 2) is dominated by those of other parts in the algorithm. The cost of lines 5-6 is simply $O(\log |\mathcal{O}| + |\mathcal{O}_q|)$ since we can issue three range queries and then perform an intersection on the query results. The cost of line 7 is $O(|\psi| \cdot |\mathcal{O}_q|)$. The cost of lines 8-11 is $O(|\mathcal{O}'|^{|\psi|} \cdot |\psi|^2)$ since it enumerates at most $O(|\mathcal{O}'|^{|\psi|})$ subsets S'' that cover ψ and each subset incurs a checking operation (line 10) whose cost is $O(|\psi|^2)$ (since $|S''| = O(|\psi|)$ and we can try all pairwise distances within S'' to do the checking). Thus, β is $O(\log |\mathcal{O}| + |\mathcal{O}_q| + |\psi| \cdot |\mathcal{O}_q| + |\mathcal{O}'|^{|\psi|} \cdot |\psi|^2)$. Note that $|\mathcal{O}'| << |\mathcal{O}_q|$ (since $\mathcal{O}'$ corresponds to a set of relevant objects in a small region), $|\mathcal{O}_q| < |\mathcal{O}|$ and $|\psi| \leq |q.\psi| - 1$.

In conclusion, the time complexity of MaxSum-Exact is $O(n_1 \cdot |P| \cdot (\log |\mathcal{O}| +$

$|\mathcal{O}_q| + |\psi| \cdot |\mathcal{O}_q| + |\mathcal{O}'|^{|\psi|} \cdot |\psi|^2))$.

2.4.2 Approximate Algorithm

In this section, we propose a 1.375-factor approximate algorithm called *MaxSum-Appro* which is better than the best-known 2-factor approximate algorithm, Cao-Appro2.

Before we present MaxSum-Appro, we introduce the concept of "o-neighborhood feasible set". Given a query q and an object $o \in \mathcal{O}$, the **o-neighborhood feasible set** is defined to be the set containing o and all other objects each of which is the t-keyword nearest neighbor of o in $D(q, d(o, q))$ for each $t \in q.\psi - o.\psi$. For example, consider Figure 2.1. Suppose that the query $q.\psi$ is $\{t_1, t_2, t_3\}$. Then, the o_1-neighborhood feasible set is $\{o_1, o_2, o_3\}$ since $q.\psi - o_1.\psi = \{t_1, t_3\}$, o_1's t_1-keyword nearest neighbor in $D(q, d(o_1, q))$ is o_2 and o_1's t_3-keyword nearest neighbor in $D(q, d(o_1, q))$ is o_3. It could be easily verified that the o-neighborhood feasible set exists iff o is outside $D(q, r_{min} - \delta)$ since an o-neighborhood feasible set is a feasible set.

In MaxSum-Appro, we only consider the o-neighborhood feasible sets for those objects o that are inside $R(S)$ where S is a feasible set, and thus they always exist.

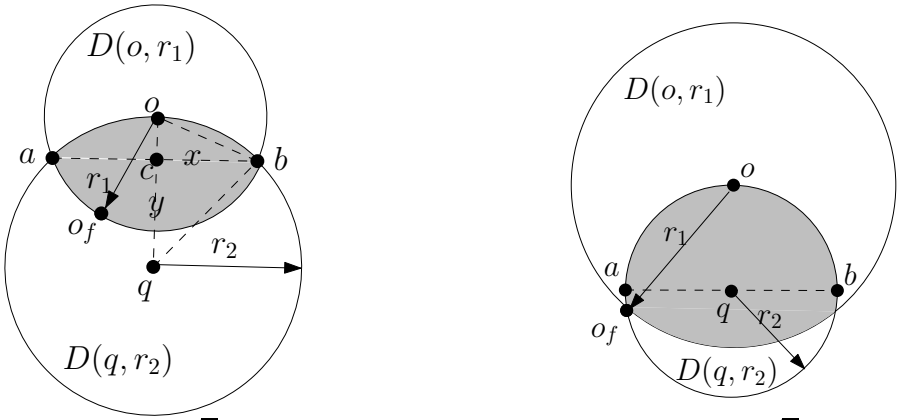
We present MaxSum-Appro in Algorithm 3. MaxSum-Appro is exactly Algorithm 1 by replacing Step 2 and Step 3 which are relatively expensive with the new efficient operation of finding the o-neighborhood feasible set which could be finished by issuing $|q.\psi - o.\psi|$ NN queries.

Theoretical Analysis. Although the set S returned by the MaxSum-Appro algorithm might have a larger cost than the optimal set S_o, the difference is bounded.

Theorem 2.4.2 *MaxSum-Appro gives a 1.375-factor approximation for the MaxSum-CoSKQ problem.* □

Algorithm 3 Algorithm *MaxSum-Appro*

Require: query q and a set $\mathcal{O}$ of objects
1: $S \leftarrow N(q)$
2: **while** there is an "un-processed" relevant object o in $R(S)$ **do**
3: // Step 1 (Query Distance Owner Finding)
4: $o \leftarrow$ the nearest "un-processed" relevant object in $R(S)$
5: // Step 2 (o-Neighborhood Feasible Set Finding)
6: $S' \leftarrow$ the o-neighborhood feasible set
7: **if** $cost(S') < cost(S)$ **then**
8: $S \leftarrow S'$
9: // Step 3 (Iterative Process)
10: mark o as "processed"
11: **return** S



(a) $r_1 \leq \sqrt{2}r_2$ (b) $r_1 > \sqrt{2}r_2$

Figure 2.2: Illustration of the proof of Theorem 2.4.2

Proof. Let S_o be the optimal solution and S be the solution returned by MaxSum-Appro. Let o be the query distance owner of S_o. By Lemma 2.4.3, we know that o is in $R(S)$. Besides, we can safely assume that o is a relevant object. Thus, there exists an iteration in MaxSum-Appro such that we process o (line 3) and thus we find its o-neighborhood feasible set denoted by S'.

Since S is the final solution returned by MaxSum-Appro, we know that $cost(S) \leq cost(S')$. The remaining part of the proof shows that $cost(S') \leq 1.375 \cdot cost(S_o)$.

Let o_f be the object in S' that is the farthest from o and $r_1 = d(o_f, o)$. Then, all objects in S' fall in $D(o, r_1)$. Let $r_2 = d(o, q)$. Since o is the query distance owner of S', we know that all objects in S' fall in $D(q, r_2)$. In summary, all objects in S' fall in

$D(o, r_1) \cap D(q, r_2)$.

Consider $cost(S_o)$. It could be verified by using a similar method for proving Lemma 2.4.1 that $\max_{o_1,o_2 \in S_o} d(o_1, o_2) \geq d(o, o_f)$. Thus, we have $cost(S_o) \geq r_2 + r_1$.

In the following, we consider two cases on r_1 according to whether there exists a line segment linking two points at the boundary of $D(q, r_2)$ such that it has its length equal to $2r_2$ (i.e., the diameter of $D(q, r_2)$) and falls in $D(o, r_1) \cap D(q, r_2)$. Note that the boundary case happens when $r_1 = \sqrt{2}r_2$ and there exists *exactly* one such segment.

Case 1: $r_1 \leq \sqrt{2}r_2$. We denote the intersection points between the boundaries of $D(o, r_1)$ and $D(q, r_2)$ by a and b, as shown in Figure 2.2(a). Let c be the intersection point between segment $\overline{qo}$ and segment $\overline{ab}$. Let $x = d(a, c) = d(b, c)$ and $y = d(c, q)$. Since $\triangle_{ocb}$ and $\triangle_{qcb}$ are right-angled triangles, we know $x^2 + (r_2 - y)^2 = r_1^2$ and $y^2 + x^2 = r_2^2$ by the *hypothesis theorem*. By solving these two equations, we obtain $x = \sqrt{r_1^2 - r_1^4/4r_2^2}$ and thus $d(a, b) = 2x = 2\sqrt{r_1^2 - r_1^4/4r_2^2}$. In this case, it can be verified that $\max_{o_1,o_2 \in S'} d(o_1, o_2) \leq d(a, b)$ (since all objects in S' are in $D(o, r_1) \cap D(q, r_2)$, as shown in the shaded area of Figure 2.2(a)) and hence $cost(S') \leq r_2 + 2\sqrt{r_1^2 - r_1^4/4r_2^2}$. Therefore,

$$\frac{cost(S')}{cost(S_o)} \leq \frac{r_2 + 2\sqrt{r_1^2 - r_1^4/4r_2^2}}{r_2 + r_1} = 1 + \frac{2\sqrt{1 - r_1^2/4r_2^2} - 1}{r_2/r_1 + 1}$$

Let $z = r_1/r_2$. Thus, $\frac{cost(S')}{cost(S_o)} \leq 1 + \frac{2\sqrt{1-z^2/4}-1}{1/z+1}$. Since $r_1 \leq \sqrt{2}r_2$, we have $z \in (0, \sqrt{2}]$ [1]. We define $f(z) = 1 + \frac{2\sqrt{1-z^2/4}-1}{1/z+1}$ on $\{z | z \in (0, \sqrt{2}]\}$. It could be verified that $f(z)$ is monotonically increasing on $(0, 0.875)$ and is monotonically decreasing on $(0.875, \sqrt{2}]$. Thus, $f(z) \leq f(0.875) < 1.375$. Therefore,

$$\frac{cost(S')}{cost(S_o)} \leq f(z) \leq 1.375$$

[1]The interval $(0, \sqrt{2}]$ does not include the boundary case where $z = 0$ (i.e., $r_1 = 0$). In this case, we have $cost(S')/cost(S_o) = 1$.

Case 2: $r_1 > \sqrt{2}r_2$. Let $\overline{ab}$ be any segment linking two points at the boundary of $D(q, r_2)$ which has its length equal to $2r_2$ and falls in $D(o, r_1) \cap D(q, r_2)$. For illustration, consider Figure 2.2(b). That is, $d(a, b) = 2r_2$. Similar to Case 1, it could be verified that $\max_{o_1,o_2 \in S'} d(o_1, o_2) \leq d(a, b) = 2r_2$. Thus, $cost(S') \leq r_2 + 2r_2$. Therefore,

$$\frac{cost(S')}{cost(S_o)} \leq \frac{r_2 + 2r_2}{r_2 + r_1} = \frac{1+2}{1 + r_1/r_2} \leq \frac{1+2}{1+\sqrt{2}} < 1.25$$

Thus, by combining Case 1 and Case 2, we have $cost(S') \leq 1.375 \cdot cost(S_o)$, which completes the proof. ■

Implementation and Time Complexity. We also adopt the IR-tree built on $\mathcal{O}$ to support the NN query and the range query.

Let n_1 be the number of iterations in MaxSum-Appro (lines 2-10 in Algorithm 3) and γ be the cost of executing an iteration. Then, the time complexity of MaxSum-Appro is $O(n_1 \cdot \gamma)$. Note that γ is dominated by the step of finding the o-neighborhood feasible set (line 6) whose cost is bounded by $O(|q.\psi| \cdot \log |\mathcal{O}|)$ (it issues at most $|q.\psi - o.\psi|$ NN queries each of which takes $O(\log |\mathcal{O}|)$ time). Thus, $\gamma = O(|q.\psi| \cdot \log |\mathcal{O}|)$. Therefore, the time complexity of MaxSum-Appro is $O(n_1 \cdot |q.\psi| \cdot \log |\mathcal{O}|)$ where $n_1 << |\mathcal{O}_q|$. Note that the worst-case time complexity of MaxSum-Appro is $O(|\mathcal{O}_q| \cdot |q.\psi| \cdot \log |\mathcal{O}|)$, which is the same as that of Cao-Appro2.

2.5 Algorithms for Dia-CoSKQ

In this section, we propose two algorithms, Dia-Exact and Dia-Appro, for Dia-CoSKQ. Similarly, in this section, for clarity, we simply write $cost_{Dia}(\cdot)$ as $cost(\cdot)$ if the context of the cost function is clear.

2.5.1 Exact Algorithm

Interestingly, we can adopt the same MaxSum-Exact algorithm (Algorithm 1) by replacing the cost measurement from the maximum sum cost to the diameter cost. We call this algorithm *Dia-Exact*. The reason is that we can still use the query distance owner and the pairwise distance owners of a set S' to be found to find the optimal solution for Dia-CoSKQ. Next, we explain the reason in detail.

Consider the diameter cost. Given a set S' of objects in $\mathcal{O}$, we have $cost(S') = \max_{o',o'' \in S' \cup \{o_q\}} d(o', o'')$. Clearly, the (diameter) cost of a set S' can be dominated (or determined) by two pairwise distance owners of $S' \cup \{o_q\}$ (not S' used in the maximum sum cost), which form a distance owner group (for Dia-CoSKQ). It is similar to the maximum sum cost of a set S' which is dominated by the query distance owner of S' and two pairwise distance owners of S'. But, there are two differences. The first difference is that the diameter cost is dominated by the pairwise distance owners only (without the query distance owner). The second difference is that the pairwise distance owners used for the diameter cost are based on the set $S' \cup \{o_q\}$ instead of S'.

Based on the above observations, we *directly* adapt the distance owner-driven approach as follows. This approach maintains a variable S storing the best feasible set found so far. Initially, S is set to a feasible set. This involves three major steps.

- *Step 1 (Pairwise Distance Owner Finding):* Select two objects, o' and o'', in $\mathcal{O}_q \cup \{o_q\}$ to take the roles of the pairwise distance owners of the set $S' \cup \{o_q\}$ where S' is to be found. Note that o' and o'' form a distance owner group.
- *Step 2 (Sub-Optimal Feasible Set Finding):* Find a set S' of objects in $\mathcal{O}_q$ such that the pairwise distance owners of $S' \cup \{o_q\}$ are o' and o'' (if any), and update S with S' if $cost(S') < cost(S)$.
- *Step 3 (Iterative Step):* Repeat Step 1 which finds another distance group, and continue with Step 2 until all distance owner groups have been traversed.

Interestingly, Step 1 which *originally* finds two objects to take the roles of the two pairwise distance owners *based on* $S' \cup \{o_q\}$ can be *refined* to a number of sub-steps of finding two objects to take the roles of the two pairwise distance owners *based on* S' simply (not $S' \cup \{o_q\}$) and finding an object to take the role of the query distance owner *based on* S'. This refinement can be explained by the following observation.

Observation 1 *Let S' be the feasible set. The pairwise distance owners of $S' \cup \{o_q\}$ are either (1) o_q and the query distance owner of S' or (2) the pairwise distance owners of S'.* □

Suppose that o takes the role of the query distance owner of S' to be found, and o_1 and o_2 take the roles of the two pairwise distance owners of S'.

Observation 1 involves two cases. In Case (1) of Observation 1, we know that the pairwise distance owners of $S' \cup \{o_q\}$ are o_q and the query distance owner o of S'. In this case, we deduce that $d(o_1, o_2) \leq d(o, q) (= d(o, o_q))$.

In Case (2) of Observation 1, we know that the pairwise distance owners of $S' \cup \{o_q\}$ are the pairwise distance owners of S', say o_1 and o_2. In this case, $d(o, q) \leq d(o_1, o_2)$.

In conclusion, if we know that $d(o_1, o_2) \leq d(o, q)$, then o_q and o are the pairwise distance owners of $S' \cup \{o_q\}$. Otherwise, o_1 and o_2 are the pairwise distance owners of $S' \cup \{o_q\}$.

Thus, Step 1 can be refined with the following three sub-steps.

- *Step 1(a) (Query Distance Owner Finding):* Select an object o in $\mathcal{O}_q$ to take the role of the query distance owner of a set S' to be found.
- *Step 1(b) (Pairwise Distance Owner Finding):* Select two objects o_1 and o_2 in $D(q, d(o, q))$ to take the roles of the pairwise distance owners *of the set* S' to be found.

- *Step 1(c) (Pairwise Distance Owner Determination):* If $d(o, q) \geq d(o_1, o_2)$, assign to o and o_q the roles of pairwise distance owners of $S' \cup \{o_q\}$; otherwise, assign the roles to o_1 and o_2.

With this refinement, the distance owner-driven approach still has its similar pruning features under the diameter cost. Specifically, Property 1 and Property 2 used for MaxSum-CoSKQ have their counterparts used for Dia-CoSKQ as Property 3 and Property 4, respectively.

Property 3 (Pruning) *Let S' be a feasible set. If o is the query distance owner of S', then the two pairwise distance owners of $S' \cup \{o_q\}$ are inside $D(q, d(o, q))$.* □

Property 4 (Pruning) *Let S' be a feasible set, o be the query distance owner of S', and o_1 and o_2 be the two pairwise distance owners of $S' \cup \{o_q\}$. Then all objects in S' fall in $D(q, d(o, q)) \cap D(o_1, d(o_1, o_2)) \cap D(o_2, d(o_1, o_2))$.* □

Similar to the maximum sum cost, when the diameter cost is used, the object to be found in Step 1(a) is fetched based on the proximity to the query point q. The proximity is also related to the closest possible query distance owner (Lemma 2.4.1) and the farthest possible query distance owner (Lemma 2.4.2). It is easy to verify that Lemma 2.4.1 and Lemma 2.4.2 still hold when the cost measurement is changed from the maximum sum cost to the diameter cost. Thus, Lemma 2.4.3, which states that the *ring* is the region containing the query distance owners to be considered, still holds.

In summary, we present the algorithm for finding the optimal solution of Dia-CoSKQ in Algorithm 4 (which is quite similar to Algorithm 1) except that we need to determine the pairwise distance owner of $S' \cup \{o_q\}$ (in Step 1(c)) which cannot be found in MaxSum-CoSKQ.

Algorithm 4 Algorithm *Dia-Exact*

Require: query q and a set $\mathcal{O}$ of objects

$S \leftarrow N(q)$
while there is an "un-processed" relevant object o in $R(S)$ **do**
  // Step 1(a) (Query Distance Owner Finding)
  $o \leftarrow$ the nearest "un-processed" relevant object in $R(S)$
  // Step 1(b) (Pairwise Distance Owner Finding)
  $D \leftarrow$ the q-disk with its radius equal to $d(o, q)$
  $P \leftarrow$ a set of all pairs (o_1, o_2) where o_1 and o_2 are in D
  for each $(o_1, o_2) \in P$ in ascending order of $d(o_1, o_2)$ **do**
    // Step 1(c) (Pairwise Distance Owner Determination)
    if $d(o, q) \geq d(o_1, o_2)$ **then** $o' \leftarrow o; o'' \leftarrow o_q$
    else $o' \leftarrow o_1; o'' \leftarrow o_2$
    // Step 2 (Sub-Optimal Feasible Set Finding)
    if there exists a feasible set S' in D which is (o, o', o'')-owner consistent **then**
      if $cost(S') < cost(S)$ **then**
        $S \leftarrow S'$; **break**
  // Step 3 (Iterative Process)
  mark o as "processed"
return S

Theorem 2.5.1 *Dia-Exact returns a feasible set with the smallest cost for the Dia-CoSKQ problem.* □

Proof. Let S_o be one of the feasible set with the smallest cost. Let o be the query distance owner of S_o, and let o_1 and o_2 be the two pairwise distance owners of S_o. First, o is inside $R(S)$ (Lemma 2.4.3). Thus, there exists an iteration where o is processed. When o is processed, pair (o_1, o_2) must be included in P (Property 3). There are two cases. Case 1: $d(o, q) \geq d(o_1, o_2)$. In this case, any feasible set S' that is (o, o, o_q)-owner consistent is retrieved and used to update S (there must exist some since S_o is (o, o, o_q)-owner consistent). Thus, the resulting S has its cost equal to $d(o, q) = cost(S_o)$. Case 2: $d(o, q) < d(o_1, o_2)$. In this case, any feasible set S' that is (o, o_1, o_2)-owner consistent is retrieved and used to update S (there must exist some since S_o is (o, o_1, o_2)-owner consistent). Thus, the resulting S has its cost equal to $d(o_1, o_2) = cost(S_o)$. In either case, S will not be updated anymore since it has the smallest cost (i.e., $cost(S_o)$) and thus it is the final output. ■

Same as Section 2.4.1, in Dia-Exact, we have the self-iteration computation strat-

egy and the cross-iteration computation strategy.

Self-Iteration Computation Strategy: Consider an iteration where the query distance owner for this iteration is o. We can use the same mechanism described in Section 2.4.1 after d_{min} and d_{max} are updated from $d(o,q) - \min\{d(o_1,q), d(o_2,q)\}$ and $cost(S) - d(o,q)$ to $d(o,q)$ and $cost(S)$, respectively. All pruning properties still hold.

Note that d_{min} (which is originally set to $d(o,q) - \min\{d(o_1,q), d(o_2,q)\}$ in MaxSum-CoSKQ) is based on the triangle inequality (Lemma 2.4.4), which means that it can be used for pruning in both MaxSum-CoSKQ and Dia-CoSKQ. However, in Dia-CoSKQ, d_{min} can be updated to a tighter value as $d(o,q)$ since all pairs with their pairwise distances smaller than $d(o,q)$ cannot take the roles of the pairwise distance owners of $S' \cup \{o_q\}$.

Cross-Iteration Computation Strategy: We use the same information reuse techniques as in Section 2.4.1 for Dia-Exact since the updated d_{min} (i.e., $d(o,q)$) is monotonically increasing and the updated d_{max} (i.e., $cost(S)$) is monotonically decreasing with more iterations. Thus, the pairs pruned in P at the previous iterations need not be considered in the later iterations.

Time Complexity. It could be verified that the time complexity of Dia-Exact is the same as that of MaxSum-Exact.

2.5.2 Approximate Algorithm

In this section, we propose a $\sqrt{3}$-factor approximate algorithm which is exactly the same as Algorithm 3 but the cost measurement used is the diameter cost. This algorithm is called *Dia-Appro*.

Theoretical Analysis. Although the set S returned by *Dia-Appro* may have a larger cost than the optimal set S_o, it has an approximate factor of $\sqrt{3}$.

Theorem 2.5.2 *Dia-Appro gives a $\sqrt{3}$-factor approximation for the Dia-CoSKQ problem.* □

Proof. We use the same notations as defined in the proof of Theorem 2.4.2.

Consider $cost(S_o)$. Similar to the proof of Theorem 2.4.2, we have $\max_{o'_1,o'_2 \in S_o} d(o'_1, o'_2) \geq d(o, o_f) = r_1$. Recall that $\max_{o' \in S_o} d(o', q) = d(o, q) = r_2$. As a result, we have $cost(S_o) = \max\{\max_{o' \in S_o} d(o', q), \max_{o'_1,o'_2 \in S_o} d(o'_1, o'_2)\} \geq \max\{r_2, r_1\}$.

According to the Dia-Appro algorithm, we have $cost(S) \leq cost(S')$. The remaining part of the proof shows that $cost(S') \leq \sqrt{3} \cdot cost(S_o)$ which further implies $cost(S) \leq \sqrt{3} \cdot cost(S_o)$.

Same as the proof of Theorem 2.4.2, we consider two cases of r_1.

Case 1: $r_1 \leq \sqrt{2} r_2$. This case corresponds to Figure 2.2(a). It can be verified that $\max_{o'_1,o'_2 \in S'} d(o'_1, o'_2) \leq d(a, b) = 2\sqrt{r_1^2 - r_1^4/4r_2^2}$ (since all objects in S' fall in $D(o, r_1) \cap D(q, r_2)$ as shown by the shaded area). Recall $\max_{o' \in S'} d(o, q) = r_2$. As a result, we have $cost(S') \leq \max\{r_2, 2\sqrt{r_1^2 - r_1^4/4r_2^2}\}$.

We further consider three sub-cases under Case 1 based on the relationship among r_1, r_2 and $2\sqrt{r_1^2 - r_1^4/4r_2^2}$.

Case 1(a): $r_1 \leq \sqrt{2 - \sqrt{3}} r_2$. In this case, we have $r_2 > r_1$ and $r_2 \geq 2\sqrt{r_1^2 - r_1^4/4r_2^2}$. Thus, $cost(S_o) \geq \max\{r_2, r_1\} = r_2$ and $cost(S') \leq \max\{r_2, 2\sqrt{r_1^2 - r_1^4/4r_2^2}\} = r_2$ Therefore,

$$\frac{cost(S')}{cost(S_o)} \leq \frac{r_2}{r_2} = 1$$

Case 1(b): $\sqrt{2 - \sqrt{3}} r_2 < r_1 \leq r_2$. In this case, we have $r_2 \geq r_1$ and

$2\sqrt{r_1^2 - r_1^4/4r_2^2} > r_2$. Thus, $cost(S_o) \geq r_2$ and $cost(S') \leq 2\sqrt{r_1^2 - r_1^4/4r_2^2}$. Therefore,

$$\frac{cost(S')}{cost(S_o)} \leq \frac{2\sqrt{r_1^2 - r_1^4/4r_2^2}}{r_2} = \sqrt{4(\frac{r_1}{r_2})^2 - (\frac{r_1}{r_2})^4} \tag{2.4}$$

Note that function $f(z) = \sqrt{4z^2 - z^4}$ is monotonically increasing on $(\sqrt{2-\sqrt{3}}, 1]$. Since $\frac{r_1}{r_2} \in (\sqrt{2-\sqrt{3}}, 1]$, Thus, we have

$$\frac{cost(S')}{cost(S_o)} \leq \sqrt{4(1)^2 - (1)^4} = \sqrt{3}$$

Case 1(c): $r_2 < r_1 \leq \sqrt{2}r_2$. In this case, we have $r_2 < r_1$ and $2\sqrt{r_1^2 - r_1^4/4r_2^2} > r_2$. Thus, $cost(S_o) \geq r_1$ and $cost(S') \leq 2\sqrt{r_1^2 - r_1^4/4r_2^2}$. Therefore,

$$\frac{cost(S')}{cost(S_o)} \leq \frac{2\sqrt{r_1^2 - r_1^4/4r_2^2}}{r_1} = \sqrt{4 - (r_1/r_2)^2} < \sqrt{3}$$

Case 2: $r_1 > \sqrt{2}r_2$. This case corresponds to Figure 2.2 (b). In this case, $d(a,b) = 2r_2$. Similar to Case 1, we have $\max_{o_1,o_2 \in S'} d(o_1, o_2) \leq d(a,b) = 2r_2$. Therefore,

$$\frac{cost(S')}{cost(S_o)} \leq \frac{\max\{r_2, 2r_2\}}{\max\{r_1, r_2\}} = \frac{2r_2}{r_1} < \frac{2r_2}{\sqrt{2}r_2} = \sqrt{2}$$

In view of above discussion, we know that $cost(S')/cost(S_o) \leq \sqrt{3}$, which completes the proof. ■

Time Complexity. Since Dia-Appro is identical to MaxSum-Appro except that Dia-Appro adopts a different cost measurement, Dia-Appro has the same complexity as MaxSum-Appro.

2.5.3 Adaptions of Existing Solutions

In this section, we adapt the existing solutions in [15], which are originally designed for MaxSum-CoSKQ, for Dia-CoSKQ.

Cao-Exact. Cao-Exact is a best-first search method based on the object space and thus its applicability is independent of the cost measurement used in the CoSKQ problem. Therefore, Cao-Exact can be directly applied to Dia-CoSKQ by replacing the cost measurement with the diameter cost. However, due to its prohibitively huge search space, Cao-Exact is not scalable to large datasets.

Cao-Appro1 & Cao-Appro2. We can directly adopt Cao-Appro1 and Cao-Appro2 for Dia-CoSKQ by replacing the maximum sum cost with the diameter cost.

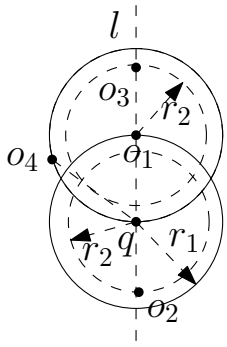
According to [15], the approximation factors of Cao-Appro1 and Cao-Appro2 are 3 and 2, respectively, for MaxSum-CoSKQ. In the following, we prove that both Cao-Appro1 and Cao-Appro2 give 2-factor approximations for Dia-CoSKQ.

Lemma 2.5.1 *Cao-Appro1 and Cao-Appro2 give 2-factor approximations for Dia-CoSKQ.* □

Proof. First, we prove that the approximation ratio of Cao-Appro1 is 2.

Let S be the set returned by Cao-Appro1 and S_o be the optimal set. Let o_f be the object in S that is the farthest from q, i.e., $d(o_f, q) = \max_{o \in S} d(o, q)$. First, we have $cost(S_o) \geq d(o_f, q)$. Second, for any two objects o_1 and o_2 in S, we have $d(o_1, o_2) \leq d(o_1, q) + d(o_2, q) \leq 2 \cdot d(o_f, q)$ by the *triangle inequality*. Therefore, $cost(S) \leq \max\{d(o_f, q), 2 \cdot d(o_f, q)\} = 2 \cdot d(o_f, q)$. As a result, we know $cost(S)/cost(S_o) \leq 2$.

Since the solution returned by Cao-Appro2 is no worse than that returned by Cao-Appro1, the approximation ratio of Cao-Appro2 is also bounded by 2.



(a) Spatial layout

object	keywords
q	t_1, t_2
o_1	t_1
o_2	t_2
o_3	t_2
o_4	t_2

(b) Keyword information

Figure 2.3: A problem instance for Cao-Appro2

Furthermore, we show that Cao-Appro2 cannot provide better error guarantees by constructing a problem instance where the approximation ratio of Cao-Appro2 is infinitely close to 2.

The problem instance is shown in Figure 2.3. In Figure 2.3(a), we have four objects, o_1, o_2, o_3 and o_4. o_1, o_2 and o_3 are located at the boundary of D(q, r_1), D(q, r_2) and D(o_1, r_2), respectively. $r_2 = r_1 - \delta$ ($\delta > 0$). Besides, q, o_1, o_2 and o_3 are on the same vertical line l. o_4 is on the boundary of D(o_1, r_1) and outside D(q, r_1). In addition, $d(o_4, q) = r_1 + \delta$. The keyword information of these objects and the query are shown in Figure 2.3(b).

Given the above problem instance, Cao-Appro2 works as follows. First, it invokes the Cao-Appro1 algorithm which returns $S_1 = \{o_1, o_2\}$ as the first candidate of the approximate solution whose cost is equal to $d(o_1, o_2) = r_1 + r_2$. Thus, o_1 is the farthest object in S_1 from q and t_1 is the keyword that is covered by o_1 but not by any other objects in S_1. As a result, Cao-Appro2 would invoke Cao-Appro1 for each object containing t_1. In this problem instance, since only o_1 contains t_1, Cao-Appro2 invokes Cao-Appro1 at o_1 only and $S_2 = \{o_1, o_3\}$ would be returned as the second candidate of the approximate solution whose cost is equal to $d(o_3, q) = r_1 + r_2$. Since Cao-Appro2 obtains two candidates S_1 and S_2 with the same cost, it returns any of them, say S_2, as the final approximate solution S. However, the optimal solution S_o of this problem

instance is $\{o_1, o_4\}$ with its cost equal to $d(o_4, q) = r_1 + \delta$. Therefore,

$$\frac{cost(S_2)}{cost(S_o)} = \frac{r_1 + r_2}{r_1 + \delta} = 2 - \frac{3}{r_1/\delta + 1}$$

When δ approaches 0, the ratio approaches 2. ■

Thus, among all known approximate algorithms for Dia-CoSKQ, our Dia-Appro provides the best constant-factor approximation.

2.6 Empirical Studies

2.6.1 Experimental Set-up

Datasets. We used the real datasets adopted in [15], namely Hotel, Web and GN. Dataset Hotel corresponds to a set of hotels in the U.S. (www.allstays.com), each of which is associated with its location and a set of words that describe the hotel (e.g., restaurant and pool). Dataset Web was created from two real datasets. The first one, named WEBSPAMUK2007[2], corresponds to a set of web documents. The second one is a set of spatial objects, named TigerCensusBlock[3], which corresponds to a set of census blocks in Iowa, Kansas, Missouri and Nebraska. Specifically, Web consists of the spatial objects in TigerCensusBlock, each of which is associated with a document randomly selected from WEBSPAMUK2007. Dataset GN was collected from the U.S. Board on Geographic Names (geonames.usgs.gov). Each object in GN is a 2D location which is associated with a set of keywords describing it (e.g., a geographic name like valley).

Query Generation. Given a dataset $\mathcal{O}$ and a positive integer k, we generated a query q with the size of its keyword set equal to k as in [15]. For the $q.\lambda$ part, we randomly

[2]http://barcelona.research.yahoo.net/webspam/datasets/uk2007

[3]http://www.rtreeportal.org

Statistics	GN	Web	Hotel
Number of objects	1,868,821	579,727	20,790
Number of unique words	222,409	2,899,175	602
Number of words	18,374,228	249,132,883	80,845

Table 2.1: Real datasets (CoSKQ)

picked a location from the data space of $\mathcal{O}$. For the $q.\psi$ part, we first sorted all the keywords that are associated with the objects in $\mathcal{O}$ in descending order of their frequencies and then randomly picked k keywords among all keywords each of which has its *percentile rank* within range $[10, 40]$ by default. Note that in this way, each of the keywords in $q.\psi$ has a relatively high frequency.

Algorithms. For MaxSum-CoSKQ, we consider 2 exact algorithms, namely MaxSum-Exact and Cao-Exact, and 3 approximate algorithms, namely MaxSum-Appro, Cao-Appro1 and Cao-Appro2. For Dia-CoSKQ, we consider 2 exact algorithms, namely Dia-Exact and Cao-Exact (the adaption), and 3 approximate algorithms, namely Dia-Appro, Cao-Appro1 and Cao-Appro2. All algorithms were implemented in C/C++.

Our experiments were conducted on a Linux platform with a 2.66GHz machine and 4GB RAM.

2.6.2 Experimental Results

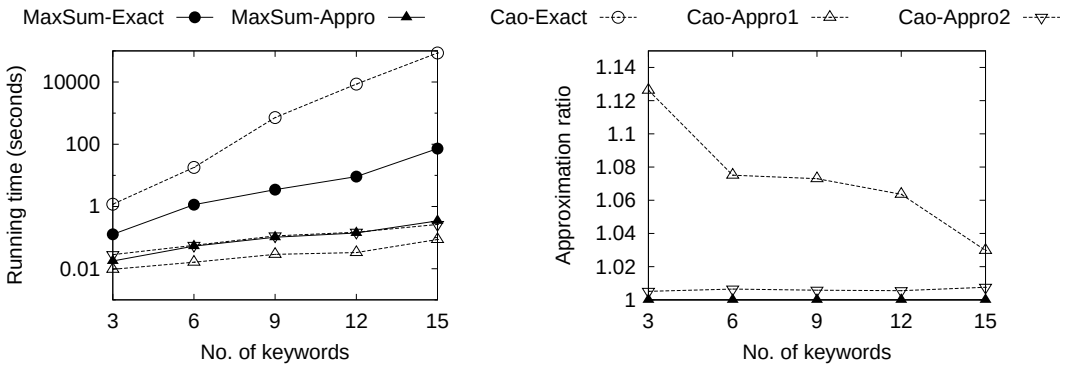
We consider 2 measurements, the running time and the approximation ratio (for approximate algorithms only). For each set of settings, we generated 50 queries, ran the algorithms with each of these 50 queries, and averaged the experimental measurements.

Experiments for MaxSum-CoSKQ

Effect of $|q.\psi|$. We generated 5 types of queries with different values of $|q.\psi|$. The values we used are 3, 6, 9, 12 and 15. The results on the dataset GN are shown in

Figure 2.4. According to Figure 2.4(a), our MaxSum-Exact is faster than Cao-Exact by 1-3 orders of magnitude. When $|q.\psi|$ increases, the running time gap between MaxSum-Exact and Cao-Exact increases. Besides, MaxSum-Appro and Cao-Appro2 have comparable running time, which verified our theoretical analysis that MaxSum-Appro and Cao-Appro2 have the same worst-case time complexity. Cao-Appro1 runs the fastest due to its simplicity. According to Figure 2.4(b), the approximation ratio of our MaxSum-Appro algorithm is near to 1, which shows that the accuracy of MaxSum-Appro is extremely high in practical. We note here that the approximation ratio in the figure corresponds to the average over 50 queries, among which, the approximation ratio of MaxSum-Appro is exactly 1 for most queries (e.g., more than 45). As a result, the approximation ratio of MaxSum-Appro in the figures is always near to 1. Consistent to our theoretical results, the approximation ratios of Cao-Appro1 and Cao-Appro2 are larger than that of MaxSum-Appro.

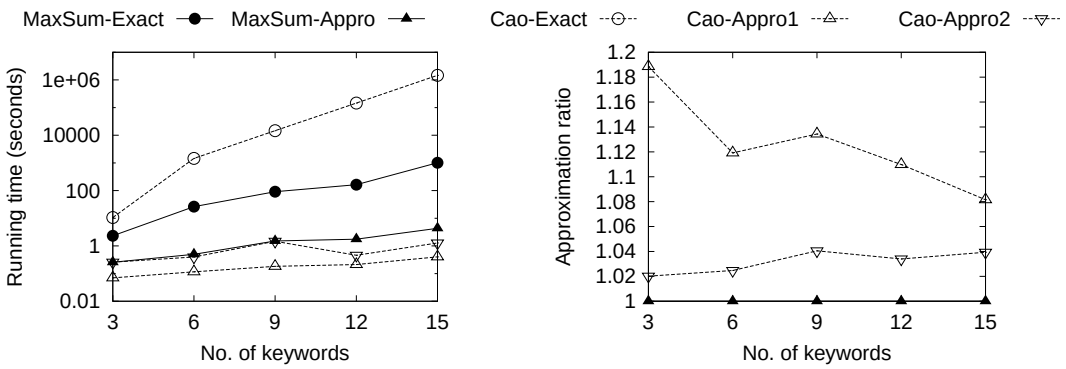
We have similar results on Web (Figure 2.5) and Hotel (Figure 2.6).



(a) Running time (b) Appro. ratio

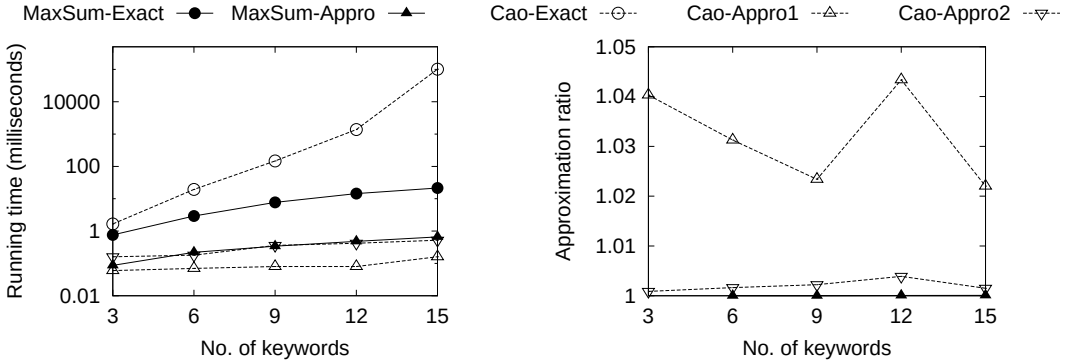
Figure 2.4: Effect of $|q.\psi|$ (GN, MaxSum-CoSKQ)

Effect of average $|o.\psi|$. Our experiments were based on dataset Hotel whose average size of a keyword set of an object ($|o.\psi|$) is nearly 4 (i.e., 80,845/20,790). We generated a set of several datasets based on dataset Hotel such that the average sizes (i.e., average $|o.\psi|$'s) are equal to $4 \cdot i$ for some integers i. To generate a dataset with its average $|o.\psi|$ equal to $4 \cdot i$, we proceed with $i - 1$ rounds. At each round, for each object o in



(a) Running time (b) Appro. ratio

Figure 2.5: Effect of $|q.\psi|$ (Web, MaxSum-CoSKQ)



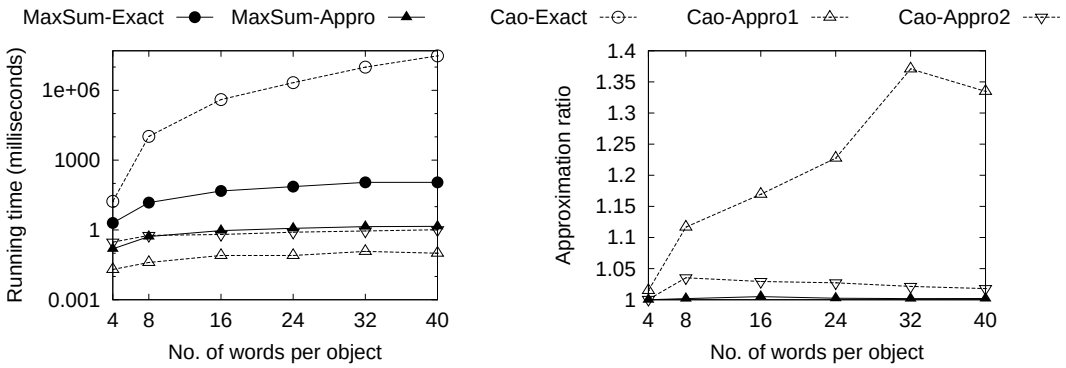
(a) Running time (b) Appro. ratio

Figure 2.6: Effect of $|q.\psi|$ (Hotel, MaxSum-CoSKQ)

dataset Hotel, we randomly pick another object o' and update $o.\psi$ to be $o.\psi \cup o'.\psi$. It could be verified that the average $|o.\psi|$ of the resulting dataset is nearly $4 \cdot i$. In our experiments, we vary i by choosing one of the values in $\{1, 2, 4, 6, 8, 10\}$. Note that $i = 1$ means that the resulting dataset is exactly dataset Hotel.

The results are shown in Figure 2.7. According to Figure 2.7(a), the running times of all algorithms increase when the average $|o.\psi|$ increases. The reason is that when the average $|o.\psi|$ increases, the number of relevant objects ($|\mathcal{O}_q|$) in the dataset would probably increase, which further affects the running times of the algorihtms. Since all algorithms except for Cao-Appro1 have their time complexities involving $|\mathcal{O}_q|$. Cao-Appro1, though has its time complexity independent of $|\mathcal{O}_q|$, has its NN queries affected by $|\mathcal{O}_q|$: the larger $|\mathcal{O}_q|$ is, the more expensive the NN query would probably be. Besides, it is worth mentioning that when the average $|o.\psi|$ increases, the increase

rate of the running time of Cao-Exact is significantly larger than those of the other algorithms including MaxSum-Exact. This is because Cao-Exact is based on the search space of the set of all possible feasible sets whose size increases rapidly with $|\mathcal{O}_q|$ ($|\mathcal{O}_q|^{|q.\psi|}$). Thus, Cao-Exact is not scalable on datasets with a large average $|o.\psi|$. According to Figure 2.7(b), the average $|o.\psi|$ has no obvious trend on the approximation ratios of the approximate algorithms. Besides, MaxSum-Appro with its approximation ratio near to 1 always keeps its accuracy superiority over other approximate algorithms.



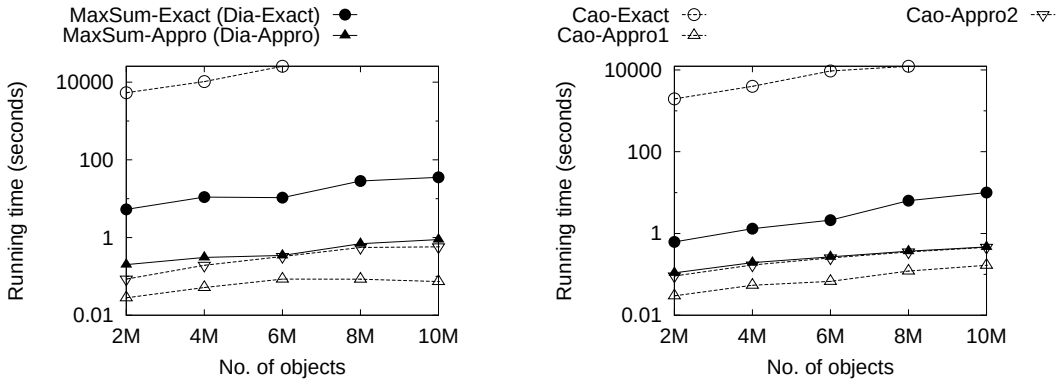
(a) Running time (b) Appro. ratio

Figure 2.7: Effect of average $|o.\psi|$ (MaxSum-CoSKQ)

Scalability Test. We conducted a scalability test on the algorithms with 5 synthetic datasets with their sizes varying from 2M to 10M. The synthetic datasets were generated from a smaller dataset GN. To generate a dataset $\mathcal{O}$ with its size equal to n, we first inserted all the objects from dataset GN into $\mathcal{O}$ and then repeatedly created objects in $\mathcal{O}$ such that $\mathcal{O}$ has a similar spatial distribution as dataset GN until $|\mathcal{O}| = n$. For each newly created object o in $\mathcal{O}$, we randomly pick a document from WEBSPAMUK2007 and use it as $o.\psi$.

The results are shown in Figure 2.8(a), where we do not show the running time of the algorithm if it runs more than 10 days or out of memory. According to these results, both our exact algorithm (MaxSum-Exact) and our approximate algorithm (MaxSum-Appro) are scalable to large datasets with millions of objects. For example, in a dataset with size equal to 10M, MaxSum-Exact ran less than 100s and MaxSum-Appro ran in

real-time. In contrast, Cao-Exact is not scalable. In particular, in our experiments, Cao-Exact took more than 1 day on a dataset with size equal to 6M and it took more than 10 days on a dataset with size equal to 8M.



(a) MaxSum-CoSKQ (b) Dia-CoSKQ

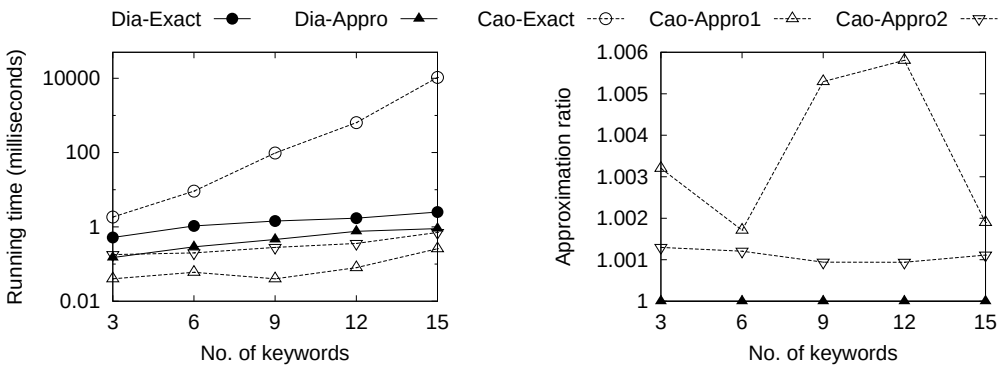
Figure 2.8: Scalability Test (CoSKQ)

Experiments for Dia-CoSKQ

Effect of $|q.\psi|$. The results on dataset GN is shown in Figure 2.9. According to Figure 2.9(a), our Dia-Exact is faster than Cao-Exact by 1-4 orders of magnitude. When $|q.\psi|$ increases, the running time gap between Dia-Exact and Cao-Exact increases. Besides, Dia-Appro and Cao-Appro2 have comparable running times. According to Figure 2.9(b), similar to MaxSum-Appro (for MaxSum-CoSkQ), the approximation ratio of Dia-Appro is near to 1 (for Dia-CoSKQ), which is better than those of Cao-Appro1 and Cao-Appro2.

The results on datasets Web and Hotel are similar and thus they are omitted here due to the page limit.

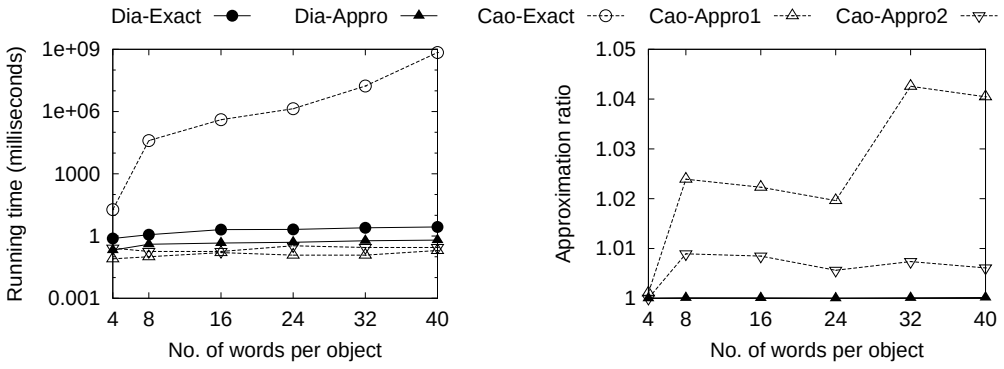
Effect of Average $|o.\psi|$. Similar to the experiments for MaxSum-CoSKQ, we generated a set of datasets by varying their average $|o.\psi|$ values. The results are shown in Figure 2.10. According to Figure 2.10(a), when the average $|o.\psi|$ increases, the running time of Cao-Exact increases significantly while the running times of other al-



(a) Running time (b) Appro. ratio

Figure 2.9: Effect of $|q.\psi|$ (GN, Dia-CoSKQ)

gorithms are only slightly affected. This is similar to the case for MaxSum-CoSKQ and the explanation for MaxSum-CoSKQ as we discussed previously could be applied here for Dia-CoSKQ. According to Figure 2.10(b), the average $|o.\psi|$ value has no obvious trend on the accuracy of the approximate algorithms.



(a) Running time (b) Appro. ratio

Figure 2.10: Effect of average $|o.\psi|$ (Dia-CoSKQ)

Scalability Test. We conducted a scalability test on the algorithms for Dia-CosKQ with the same synthetic datasets used in the scalability test for MaxSum-CoSKQ.

The results are shown in Figure 2.8(b), where we do not show the running time of the algorithm if it runs more than 10 days or out of memory. According to these results, both our exact algorithm (Dia-Exact) and our approximate algorithm (Dia-Appro) are scalable to large datasets with millions of objects. In contrast, Cao-Exact is not scalable to large datasets.

Conclusion: MaxSum-Exact (Dia-Exact) runs faster than Cao-Exact by several orders of magnitude for MaxSum-CoSKQ (Dia-CoSKQ). Besides, MaxSum-Exact (Dia-Exact) is scalable in terms of $|\mathcal{O}|$ as well as the average $|o.\psi|$ but Cao-Exact is not. Our MaxSum-Appro (Dia-Appro) has a better accuracy while having comparable running time as those existing approximate algorithms.

2.7 Conclusion

In this chapter, we study two types of the CoSKQ problem, namely MaxSum-CoSKQ and Dia-CoSKQ. MaxSum-CoSKQ is a CoSKQ problem using the existing maximum sum cost, which is NP-hard. We design two algorithms for MaxSum-CoSKQ, MaxSum-Exact and MaxSum-Appro. MaxSum-Exact is an exact algorithm which significantly outperforms its existing competitor in terms of both efficiency and scalability and MaxSum-Appro is an approximate algorithm which improves the best-known constant approximation factor from 2 to 1.375. We also propose a new cost function and the CoSKQ problem using this function is Dia-CoSKQ. We design two algorithms for Dia-CoSKQ, Dia-Exact and Dia-Appro. Dia-Exact is an exact algorithm while Dia-Appro is a $\sqrt{3}$-factor approximate algorithm. Extensive experiments were conducted which verified our theoretical findings and algorithms.

CHAPTER 3

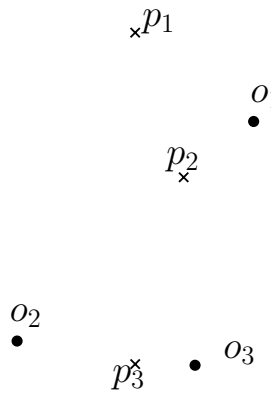
WORST-CASE OPTIMIZED SPATIAL MATCHING

3.1 Introduction

Bichromatic reverse nearest neighbor (BRNN) queries have been studied extensively [51, 57, 95]. Let P be a set of service-providers and O be a set of customers. A BRNN query is to find which customers in O are "interested" in a given service-provider in P. However, BRNN queries lack the consideration of the *capacities* of service-providers and the *demands* of customers. In order to address this issue, some *spatial matching problems* [96, 91, 90] have been proposed which assign service-providers to customers with the above consideration.

In some real-life applications like hospital allocation, a common goal is to minimize the *maximum* distance (or cost) between a hospital and a residential estate served by this hospital. For example, in the Hong Kong ambulance service, the minimized maximum cost is about 12 minutes (driving distance) [1].

To illustrate, we go through a toy example as shown in Figure 3.1. In Figure 3.1(a), P contains three hospitals p_1, p_2 and p_3 and O contains three residential estates o_1, o_2 and o_3. Figure 3.1(b) shows all pairwise distances between P and O. For the sake of illustration, suppose that the *capacity* of each hospital p in P is 1, which means that the greatest amount of the service given by p is 1, and the *demand* of each residential estate o in O is also 1, which means that the amount of the service requested by o is 1. In this case, each hospital can serve at most one residential estate. In order to minimize the maximum distance between a hospital and the residential estate served



(a) Spatial layout

$dist$	o_1	o_2	o_3
p_1	5	10	11
p_2	3	7	6
p_3	9	4	2

(b) Pairwise distances

Figure 3.1: A running example (SPM-MM)

by this hospital, we form an *assignment* between P and O as shown in Figure 3.2(a). In this assignment, p_1, p_2 and p_3 serve o_1, o_3 and o_2, respectively. If p serves o, we draw a line between p and o in the figure. The number next to the line is called the *matching distance* between p and o which corresponds to the Euclidean distance between p and o. In this assignment, the *maximum matching distance* (*mmd*) is equal to 6. Besides, we cannot find any other assignment which satisfies the service demand of each customer and has its *mmd* smaller than 6. Thus, 6 is the *optimal mmd*.

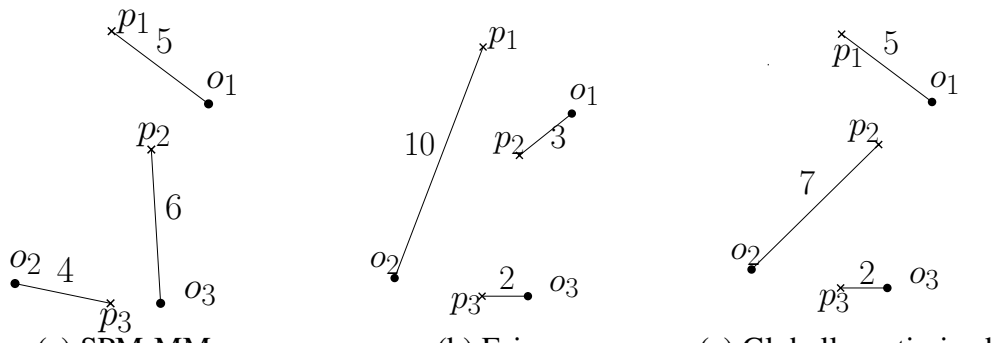
In this thesis, we propose a new problem called *SPatial Matching for Minimizing Maximum matching distance* (SPM-MM). Given a set P of service-providers each of which has a capacity and a set O of customers each of which has a demand, the SPM-MM problem is to assign the service-providers in P to the customers in O with the consideration of the capacities of the service-providers such that the demand of each customer in O is satisfied and the maximum matching distance (i.e. *mmd*) is minimized.

SPM-MM has extensive applications in matching between two sets of objects where the *worst-case* cost should be minimized. The notions of "service-provider" and "customer" in SPM-MM are general and can have alternative semantics in different (even non-geographic) applications. One such application is the allocation problem

between emergency facilities and users. Hospitals, fire stations and police stations are some examples of emergency facilities and residential estates and commercial areas are some examples of users. Logistics, data warehouse allocation and mail delivery are some applications with non-emergency facilities. *Profile matching* [96] is another application where we want to match "items" (regarded as service-providers) with "customers" such that the worst-case dissatisfactory rate among all customers is minimized.

It turns out that SPM-MM reduces to be a classical problem in computer science, *Bottleneck Matching Problem* (BMP) [41], when each service-provider has its capacity equal to 1 and each customer has its demand equal to 1 as well. Given two sets of n objects, A and B, and the cost of matching each object in A with each object in B, the BMP problem is to find the *perfect matching* with the smallest *cost* among all perfect matchings between A and B where the *cost* of a perfect matching M is defined to be the greatest cost of matching an object from A and an object from B in M. It can be verified that SPM-MM becomes BMP when $|P| = |O|$, each service-provider $p \in P$ (customer $o \in O$) has its capacity (demand) equal to 1, and the distance between p and o is used as the cost of matching p with o for each $p \in P$ and each $o \in O$. [11] provides a comprehensive study on existing solutions of BMP, among which, the *Threshold* algorithm is the fastest.

No existing algorithms can be used to solve the SPM-MM problem. Firstly, the algorithms for BMP cannot be used *directly* for SPM-MM since in SPM-MM, the capacities/demands could be arbitrary positive integers. Besides, we will show that an *adapted* version of the *Threshold* algorithm, which is originally designed for BMP, is not scalable for SPM-MM. Secondly, the solutions for all existing spatial matching problems cannot be used for SPM-MM. To illustrate this, we first give a brief background of these problems. Two major types of spatial matching problems have been studied. The first one [96] aims to find the *fair* assignment between P and O which



(a) SPM-MM (b) Fair (c) Globally-optimized

Figure 3.2: Spatial matching problems

is to assign to each customer the nearest service-provider that has not been exhausted of serving other *closer* customers. Figure 3.2(b) shows the fair assignment between P and O whose *mmd* is equal to 10 (> 6). The second one is to find the *globally optimized* assignment between P and O which guarantees that each customer's service demand is satisfied and the *overall* matching cost is minimized. Figure 3.2(c) shows the globally optimized assignment between P and O whose *mmd* is equal to 7 (> 6).

In this thesis, we design two algorithms for the SPM-MM problem. The first one is called *Threshold-Adapt* and the second one is called *Swap-Chain*. *Threshold-Adapt* is an algorithm which shares a similar idea as *Threshold* which is originally designed for BMP. Unfortunately, *Threshold-Adapt* is not scalable to large datasets due to its high time/space complexity. *Swap-Chain* is an algorithm which is scalable and runs faster than *Threshold-Adapt* by orders of magnitude by using the concept of finding a series of elements where every two adjacent elements are "close" to each other for re-matching. The operation of finding a "close" element from another element can be implemented efficiently by spatial queries.

It is worth mentioning that our proposed algorithms are not limited to the Euclidean space. In fact, they can also be adapted to non-metric space (with a certain sacrifice of efficiency). For example, our algorithms can be adapted to settle the SPM-MM problem in Figure 3.1, even if the distance between a hospital and a residential estate is their road-network distance. The discussion on how to adapt our techniques to non-

metric space is given in Section 3.6.

We summarize our main contributions as follows. Firstly, to the best of our knowledge, we are the first to propose the SPM-MM problem, which has extensive real-life applications. Secondly, to solve SPM-MM, we design our first algorithm, *Threshold-Adapt*, based on an idea of one popular solution of BMP, *Threshold*. *Threshold-Adapt* is not scalable for large datasets due to its high time/space complexity. Therefore, we develop another novel algorithm, *Swap-Chain*, which runs faster than *Threshold-Adapt* by orders of magnitude and is scalable to very large datasets (in millions). Finally, we conducted extensive empirical studies on these two solutions.

The rest of this chapter is organized as follows. Section 3.2 defines the SPM-MM problem, and Section 3.3 provides the related work of SPM-MM. Section 3.4 and Section 3.5 introduce two algorithms, *Threshold-Adapt* and *Swap-Chain*, respectively. Section 3.6 gives some discussions. Section 3.7 includes the empirical studies and Section 3.8 concludes the chapter.

3.2 The SPM-MM Problem

Let P be a set of service-providers and O be a set of customers. Each service-provider p (customer o) has a service capacity (demand), denoted by $p.w$ ($o.w$). We represent the Euclidean distance between o and p with $d(o, p)$.

Let $W_O = \sum_{o \in O} o.w$ and $W_P = \sum_{p \in P} p.w$. We assume that the service demands of all customers in O can be satisfied by the service-providers in P, i.e. $W_P \geq W_O$. Under this assumption, it is possible that some service-providers are not matched with customers. In case that $W_P < W_O$, we swap the roles of P and O and thus this assumption still holds.

Problem 1 (SPM-MM) *SPM-MM generates the assignment A denoting a set con-*

taining the elements in the form of triplets (o, p, w)*, where* (o, p, w) *is called a* match *between* o *and* p *and denotes that* p *provides the service with the amount of* w *to* o*. Furthermore, the following three conditions hold.*

- *No service-provider provides its service of the amount greater than its capacity, i.e.,* $\forall p \in P$, $\sum_{(o,p,w)\in A} w \leq p.w$.
- *Each customer's service demand is satisfied, i.e.,* $\forall o \in O$, $\sum_{(o,p,w)} w = o.w$.
- *The* mmd *of* A *is minimized, i.e.,* $\max\{d(o, p)|(o, p, w) \in A\}$ *is minimized.*

Note that in the following, for clarity, the match (o, p, w) *is simply denoted as* (o, p) *when* $w = 1$. □

In order to ease our discussion, we say that an assignment is *full* if it satisfies the Capacity Constraint and the Demand Constraint defined above. Note that there are an exponential number of full assignments. To illustrate, consider the case where $|P| = |O| = n$ and the capacity (demand) of each service-provider (customer) is equal to 1. In this case, there exist $n!$ possible full assignments.

3.3 Related Work

We classify the related work into three branches.

The BMP Problem: The first branch is the *Bottleneck Matching* problem [41, 11, 32, 36] (BMP). BMP was first proposed by Gross in [41]. Given two sets of n objects, $A = \{a_1, a_2, ..., a_n\}$ and $B = \{b_1, b_2, ..., b_n\}$, and the cost matrix $C_{n\times n}$ (c_{ij} represents the cost of matching a_i with b_j for $1 \leq i, j \leq n$), BMP is to find the perfect matching between A and B, which minimizes the maximum matching cost.

One may come up with the following straightforward solution to solve our SPM-MM problem by using the existing solutions for BMP. Specifically, we duplicate each o

in O $o.w$ times and each p in P $p.w$ times. Then, we can use the existing algorithm originally designed for BMP to find the solution for our SPM-MM problem. However, this duplication is cumbersome and undesirable (especially when the capacities/demands are very large), because the resulting datasets would be prohibitively large.

Next, we describe the most popular solution for BMP. [11] provides a comprehensive study of the solutions of BMP, among which, the *Threshold* method has the lowest time complexity. The best-known algorithm for BMP is due to Gabow and Tarjan in [36], which is based on *Threshold*. *Threshold* is based on the property that the minimized *maximum matching cost* (i.e., the optimal *mmd*) must reside in the cost matrix $C_{n\times n}$. Therefore, it maintains a set X containing the candidates of the optimal *mmd*, which is initialized to be $\emptyset$. For each cost entry c in $C_{n\times n}$, it first constructs a *bipartite graph* between A and B containing the edges each of which is a pair (a_i, b_j) whose matching cost is *at most* c. Then, it checks whether there exists a perfect matching in this bipartite graph. If yes, it includes c in X. Finally, it returns the smallest cost in X, which is shown to be the optimal *mmd*. The above checking operation could be accomplished with a *maximum cardinality matching* procedure [49] on the corresponding bipartite graph which finds the greatest number of matches in the graph. However, *Threshold* incurs an expensive space cost of $O(n^2)$ since it has to maintain the cost matrix $C_{n\times n}$. Thus, it is not scalable to large datasets.

There is an existing study [32] for BMP in the context of spatial databases where the matching distance between two objects is their Euclidean distance. The method in [32] is exactly the *Threshold* algorithm except that the *maximum cardinality matching* procedure [49] is improved. However, this method cannot be used directly for SPM-MM where the capacities/demands are any positive integers. Besides, the techniques in [32] originally designed for improving the maximum cardinality matching procedure in *Threshold* cannot be adopted for our *Threshold-Adapt* algorithm (which

will be introduced in Section 3.4) since *Threshold-Adapt* involves no maximum cardinality matching procedure.

A *monochromatic* version of BMP (i.e., only one set of data) is considered in [18, 33]. But, these studies are different from ours which uses a *bichromatic* setting where two sets of data (i.e., P and O) are considered for matching.

Some recent papers [30, 100] in the field of operations research also studied the bottleneck problem and its variations, but they do not focus on the efficiency issue. Specifically, a common technique in this field [30, 100] is constrained optimization/programming, which is known to be slow for large datasets. Besides, [30, 100] only studied the problems in the context of graphs instead of spatial databases.

Spatial Matching Problems: The second branch is the existing spatial matching problems [96, 91, 90]. [96] proposed the *SPatial Matching* problem (SPM), which generates a *fair* assignment between P and O. [91] proposed the *Capacity Constrained Assignment* problem (CCA), which returns the *globally optimized* assignment. Recently, a continuous version of CCA [90] was proposed where customers move *dynamically*.

Since SPM and CCA have different optimization criteria from SPM-MM, the existing solutions developed for SPM and CCA cannot be applied here. In fact, as will be verified in our empirical study, the mmd's of the assignments of SPM and CCA are much larger than the mmd of the SPM-MM assignment.

Problems with Minimum Maximum Distance: The third branch is related to some other problems [46, 8] using the *minimum maximum distance* as a measurement. Given n cities, the k-*center* problem [46], one of the traditional computer science problems, is to build k warehouses at different cities ($k \leq n$) such that the maximum distance from a city to its *nearest* warehouse is minimized. The goal of k-center which is to *select* k cities out of n cities is different from that of SPM-MM which is to *match* service-providers and customers. [8] studied an assignment problem between servers

and clients. The matching distance between a server and a client depends on both the physical distance and the *load* of the server where the load of a server corresponds to the number of clients served by this server. In other words, the matching distance between a server and a client defined in [8] in an assignment can be different from the one in another assignment.

3.4 The Threshold-Adapt Algorithm

3.4.1 Theoretical Properties

Given a set P of service-providers and a set O of customers, let d_o be the optimal *mmd* for the SPM-MM problem. Intrinsically, d_o is a pairwise distance between a service-provider p in P and a customer o in O. It follows that $d_o \in S$, where S is the set of all possible pairwise distances between P and O, i.e., $S = \{d(p, o) | p \in P, o \in O\}$. Note that $|S| = |P| \cdot |O|$. We present this property in the following Lemma 3.4.1.

Lemma 3.4.1 (Search Space) *Let d_o be the optimal* mmd *for the SPM-MM problem. d_o is in S.* □

Proof. d_o is a pairwise distance ($d_o \in S$). ■

According to Lemma 3.4.1, one straightforward method of finding d_o is to determine whether each value in S is *feasible* for the SPM-MM problem, insert all feasible values into a set X, and find the minimum value in X as d_o. The definition of "feasibility" is defined next.

Definition 3.4.1 (Feasibility) *Given a positive real number d, d is* feasible *if and only if there exists a full assignment A between P and O such that its* mmd *is at most d.* □

Lemma 3.4.2 (Feasibility) *Let d_o be the optimal* mmd *for the SPM-MM problem. d_o is feasible.* □

Proof. In the SPM-MM assignment which is full, all matching distances are at most d_o. ■

3.4.2 Algorithm

We develop our *Threshold-Adapt* algorithm by using the search space S and the feasibility property described in Lemma 3.4.2. Specifically, *Threshold-Adapt* checks the feasibility of each distance in S and returns the smallest feasible distance.

Theorem 3.4.1 *The* Threshold-Adapt *algorithm returns the optimal assignment for the SPM-MM problem.* □

Proof. The correctness follows from Lemma 3.4.1 and 3.4.2. ■

Let α be the cost of checking the feasibility of a given value d. One straightforward implementation of *Threshold-Adapt* has the time complexity equal to $O(|S| \cdot \alpha) = O(|P| \cdot |O| \cdot \alpha)$. In the following, we consider two issues of *Threshold-Adapt*.

The first issue is to further reduce the size of the search space from $|P| \cdot |O|$ to $O(\log(\max\{|P|, |O|\}))$ based on the following *monotonicity* property.

Lemma 3.4.3 (Monotonicity) *Let d and d' be two positive real numbers where $d < d'$. If d is feasible, then d' is feasible.* □

Proof. Since d is feasible, there exists a full assignment A such that A's *mmd* is at most d and thus A's *mmd* is smaller than d'. Therefore, d' is feasible. ■

According to the above lemma, if we know that a value d' in S is not feasible, then any value d in S smaller than d' must not be feasible. This gives hints for a further reduction of the search space.

Specifically, we sort all values in S in ascending order and store the sorted values in a list L. Then, we adopt *binary search* to find the smallest feasible value in L (which corresponds to the optimal *mmd*). This method checks $O(\log |L|)$ pairwise distances in S. Note that $|L| = |P| \cdot |O|$. Thus, the size of the search space becomes $O(\log(|P| \cdot |O|)) = O(\log(\max\{|P|, |O|\}))$ which is significantly smaller than the original size of $|P| \cdot |O|$.

The second issue is to propose an efficient method to perform the judging task to determine whether a given value d is feasible or not. We propose the following three-step algorithm.

Step 1: Construction of a Flow Network wrt d. We construct a *flow network* $G_d(V_d, E_d)$ wrt d as follows. We create a source vertex s and a sink vertex t, and V_d is constructed to be $P \cup O \cup \{s, t\}$. For each pair $(o, p) \in O \times P$ with $d(o, p) \leq d$, we create an edge (p, o) in E_d and set its capacity to be $\min\{p.w, o.w\}$. For each p in P (o in O), we create an edge (s, p) ((o, t)) in E_d and set its capacity to be $p.w$ ($o.w$).

Step 2: Construction of a Maximum-Flowed Network. We perform a *maximum-flow algorithm* [5], denoted by $A_{max-flow}$, on the flow network G_d and obtain the *maximum flow* from s to t in G_d. We denote the amount of this maximum flow by mf. The *maximum-flowed network* is the flow network G_d, where each edge is associated with its flow in the resulting maximum flow. We denote by $e.f$ the flow associated with the edge e.

Step 3: Feasibility Checking on d. We compare mf with W_O. If $mf = W_O$, we conclude that d is feasible; otherwise, we conclude that d is not feasible. In the former case, we construct an assignment, denoted by A_d, based on the maximum-flowed network at Step 2. We initialize A_d to $\emptyset$. Then, for each edge e in the form of (p, o) in the maximum-flowed network with $e.f > 0$, we create a match $(o, p, e.f)$ in A_d.

The correctness of the above three-step algorithm is verified by the following lemma.

Lemma 3.4.4 *The three-step algorithm returns a full assignment A_d with its* mmd *at most d if and only if d is feasible.* □

Proof. We consider two cases. Case 1: $mf = W_O$. In this case, the three-step algorithm returns A_d. We prove that A_d is a full assignment with its *mmd* at most d. First, we know that A_d satisfies the Capacity Constraint since for each $p \in P$,

$$\sum_{(o,p,w)\in A_d} w = \sum_{e=(p,o)\in E} e.f = (s,p).f \leq p.w$$

Second, we show that A_d satisfies the Demand Constraint by contradiction. Assume there exists a customer $o' \in O$ whose demand is not satisfied in A_d, i.e., $\sum_{e=(p,o')\in E} e.f < o'.w$. We have

$$\begin{aligned} mf &= \sum_{e=(p,o)\in E} e.f = \sum_{o''\in O} \sum_{e=(p,o'')\in E} e.f \\ &< \sum_{o''\in O} o''.w = W_O \end{aligned}$$

which contradicts $mf = W_O$. Third, it is easy to verify that A_d's *mmd* is at most d since for each edge in the form of (p, o) in E, we have $d(p, o) \leq d$ (this is guaranteed by Step 1 of the three-step algorithm). In conclusion, we know that A_d is a full assignment with its *mmd* at most d which further implies that d is feasible. Case 2: $mf < W_O$. In this case, it could be verified that there exists no full assignment which has its *mmd* at most d by contradiction (note that a full assignment implies a flow with its amount equal to W_O which contradicts $mf < W_O$). That is, d is not feasible. ■

Time Complexity. After we address the first issue and the second issue, we know that the *Threshold-Adapt* algorithm triggers $O(\log(\max\{|P|, |O|\}))$ times of run-

ning the maximum-flow algorithm. Thus, the time complexity of *Threshold-Adapt* is $O(\log(\max\{|P|, |O|\}) \cdot \alpha)$ where α is the cost of a maximum-flow algorithm (e.g., $\alpha = O(n^2m)$ on a flow network with n vertices and m edges if the recently proposed *IBFS* algorithm [40] is adopted). We will test different maximum-flow algorithms in our experiments for optimizing the performance of *Threshold-Adapt*.

We note here that *Threshold-Adapt* suffers from two intrinsic space problems which limit the application scope of *Threshold-Adapt* to small/medium-sized datasets only. First, it relies on a search space S whose size is $|P| \cdot |O|$. This is prohibitively large when the datasets are large (e.g., S simply occupies about $7.45GB$ space when $|O| = 100k$ and $|P| = 10k$). Second, it has to maintain a flow network $G_d(V_d, E_d)$ which has its worst-case space complexity of $O(|P| \cdot |O|)$. Motivated by the above space issues of *Threshold-Adapt*, we design another algorithm called *Swap-Chain* in the next section, which not only avoids these issues by adopting a fundamentally different idea, but also runs faster by orders of magnitude.

3.5 The Swap-Chain Algorithm

In Section 3.5.1, we give an overview of the *Swap-Chain* algorithm. We then present it in Section 3.5.2, and discuss some issues of *Swap-Chain* and its theoretical results in Section 3.5.3.

3.5.1 Overview

Swap-Chain has the following three steps.

- **Step 1 (Assignment Initialization):** It first initializes a full assignment A using a given strategy. We will discuss different strategies for this step in Section 3.5.3. One strategy is finding a fair assignment (which is full) by an existing

algorithm [96].

- **Step 2 (Assignment Adjustment):** It re-assigns some matches in A to form another full assignment A' such that the *mmd* of A' is smaller than that of A.
- **Step 3 (Iterative Step):** It repeats Step 2 until it is not possible to perform the assignment adjustment step.

In Step 2, the algorithm reduces the *mmd* of an assignment A by re-assigning some matches in the assignment. Note that the *mmd* of an assignment denotes the maximum matching distance of a match in the assignment and this match is called an *extreme match*. Specifically, the main idea of Step 2 is to find an extreme match in the assignment, break this match and *some* other matches, and re-assign these matches such that the *mmd* of the resulting assignment is smaller.

3.5.2 Algorithm

Concepts and Algorithm

Before introducing the *Swap-Chain* algorithm, we introduce some concepts and lemmas related to the algorithm.

Let A be an assignment. Given a customer $o \in O$, the *deficient demand* of o in A is defined to be $o.w - \sum_{(o,p,w)\in A} w$. o is said to have his/her deficient demand in A if the deficient demand of o in A is non-zero. Otherwise, o is said to have no deficient demand in A. Given a service-provider $p \in P$, the *free capacity* of p in A is defined to be $p.w - \sum_{(o,p,w)\in A} w$. Similarly, p is said to have its free capacity or have no free capacity in A according to different cases. A service-provider p is said to be *available* in A if it has its free capacity in A. Otherwise, it is said to be *occupied* in A.

Definition 3.5.1 (d-Available/Occupied service-provider) *Given a non-negative real number d and a customer o, a service-provider $p \in P$ is said to be a* d-available

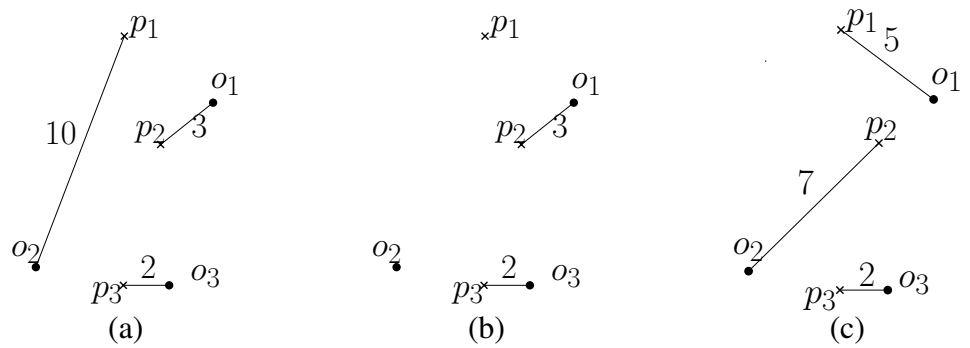


Figure 3.3: The process of *Swap-Chain*

service-provider *(d-occupied* service-provider*) for o in A if and only if p is available (occupied) in A and $d(o, p) < d$.* □

Example 1 [d-Available/Occupied service-provider] Consider Figure 3.3(b). For the ease of illustration, we assume that the capacity (demand) of each service-provider (customer) is 1 in the figure. Suppose that we have a (non-full) assignment A equal to $\{(o_1, p_2), (o_3, p_3)\}$. p_1 is an available service-provider in A but both p_2 and p_3 are occupied service-providers in A. Let $d = 10$. Since $d(o_1, p_1) = 5 < d$, p_1 is a d-available service-provider for o_1 in A. Besides, since $d(o_2, p_2) = 7 < d$ and $d(o_2, p_3) = 4 < d$, both p_2 and p_3 are two d-occupied service-providers for o_2 in A. However, since $d(o_2, p_1) = 10$ which is *exactly* equal to d, p_1 is not a d-available service-provider for o_2 in A. Note that there does not exist any d-available service-provider for o_2 in A. □

Definition 3.5.2 (d-satisfiability) *Given a non-negative real number d and a customer o, o is said to be* d-satisfiable *in A if and only if one of the following conditions is satisfied.*

- ***Availability Condition:*** *There exists a d-available service-provider for o in A, or*
- ***Non-Availability Condition:*** *There does not exist any d-available service-provider for o in A and there exists a d-occupied service-provider p' for o in*

A such that p' is matched with another customer o' in A and o' is d-satisfiable in A. In this case, (p', o') is said to be a d-substitute pair *for o in A.* □

Note that "d-satisfiability" is a recursive definition. The *availability condition* corresponds to the base condition in the recursive definition while the *non-availability condition* corresponds to the recursive condition.

Example 2 [d-satisfiability] Consider Example 1. Suppose that the assignment A is still $\{(o_1, p_2), (o_3, p_3)\}$. Let $d = 10$. o_1 is d-satisfiable since there exists a d-available service-provider for o_1 in A (i.e., p_1). o_2 is also d-satisfiable because there does not exist any d-available service-provider for o_2 in A and there exists a d-occupied service-provider for o_2 in A, namely p_2, such that p_2 is matched with another customer o_1 and o_1 is d-satisfiable in A. Thus, (p_2, o_1) is a d-substitute pair for o_2 in A. □

The following lemma shows the relationship between "d-satisfiability" and the optimal assignment for SPM-MM.

Lemma 3.5.1 (Optimal Assignment) *Let A be an assignment. If there does not exist any extreme match m in A such that the customer originally matched in m is d-satisfiable in $A - \{m\}$ where d is the matching distance of m in A, then A is the optimal assignment for the SPM-MM problem.* □

Proof.

We consider two cases. The first case is that for each p and each o, $p.w = 1$ and $o.w = 1$. Thus, each match (o', p', w') can be expressed as (o', p') (since $w' = 1$). The second case is a general case that for each p and each o, $p.w$ and $o.w$ can be equal to any positive integer.

Consider the first case. We prove by contradiction. Let A be the assignment such that there does not exist any extreme match m in A such that the customer o originally

matched in m is d-satisfiable in $A - \{m\}$ and d is the matching distance of m in A. Suppose that A is not the optimal assignment for the SPM-MM problem. That is, there exists another assignment A_o such that the *mmd* of A_o, denoted by d_o, is smaller than the *mmd* of A, denoted by d. Let $A' = A - \{m\}$.

Consider A_o. We know that for each match $(o', p') \in A_o$, $d(o', p') \leq d_o$. Since $d_o < d$, we have the following. For each match $(o', p') \in A_o$,

$$d(o', p') < d \tag{3.1}$$

There exists a service-provider p_1 such that $(o, p_1) \in A_o$. We conclude that

$$d(o, p_1) < d \tag{3.2}$$

Consider A'. We know that o (from match m) is not d-satisfiable in A'. Thus, we deduce that there does not exist any d-*available* service provider for o in A'. We further consider two sub-cases.

Case (a): There does not exist any d-*occupied* service-provider for o in A'. We deduce that there does not exist any service-provider p such that $d(o, p) < d$. This contradicts to that $d(o, p_1) < d$ (in Inequality (3.2)).

Case (b): There exists a d-occupied service-provider for o in A'. According to Inequality (3.2), we deduce that p_1 is a d-occupied service-provider for o in A'. Thus, there exists a customer o_1 which is matched with p_1 in A'.

In the following, we will show that o_1 is d-satisfiable in A'. After we obtain this result, we can conclude that o is d-satisfiable in A', which leads to a contradiction.

We first construct an undirected graph G (which will be used later in the proof) as follows. Firstly, we construct an assignment $A_c = (A_o \cup A') - (A_o \cap A')$. As a result, all customers in $O - o$ are involved in either zero matches in A_c or *exactly* two distinct matches in A_c. We construct a set V of vertices to be $P \cup O$. For each $(o', p') \in A_c$, we

create an edge (o', p'). All edges created form a set E. The graph G is defined based on V and E.

It is easy to verify that in this graph G, any path starting from o is a list containing interleaved customers and service-providers in the form of $(o, p_1, o_1, p_2, o_2, ...)$ such that the following three rules hold: (R1) o is matched with p_1 in A_o, (R2) o_i is matched with p_{i+1} in A_o for $i = 1, 2, ...$, and (R3) p_i is matched with o_i in A' for $i = 1, 2, ...$

According to the three rules, we deduce the following two statements: (1) any path from o to a service-provider in G is non-cyclic, and (2) there exists a path from o to a service-provider point/vertice p_n such that p_n is the first service-provider with its free capacity in A' along the path. The correctness of Statement (1) can be shown since there is only one edge involving o in E and each vertice in $V - \{o\}$ is involved at most two edges in E. Statement (2) can be proved as follows. Since the total number of vertices is bounded by $|V|$ and any path $\mathcal{P}$ from o to a service-provider is *non-cyclic* (by Statement (1)), the length of $\mathcal{P}$ is *bounded*. Consider a customer o' (not o) along the path $\mathcal{P}$ from o. Since o' is involved in exactly two edges in E (it is not possible that o' is involved in zero edges in E since o' is along $\mathcal{P}$ from o), we know that o' is matched in both A_o and A', and thus the path from o can be prolonged at o'. Consider a service-provider p' along the path $\mathcal{P}$ from o. If p' is involved in exactly two edges in E, similarly, it is matched in both A_o and A', and thus the path from o can be prolonged at p'. If it is involved in exactly one edge in E, it means that it is matched in A_o only (but not A') (by R2) and thus the path from o cannot be prolonged at p'. In this case, p' has its free capacity in A'. This completes the proof when we set $p_n = p'$ in this case.

Based on the above two statements, we conclude that the path is of the non-cyclic form of $(o, p_1, o_1, p_2, o_2, ..., p_{n-1}, o_{n-1}, p_n)$ where p_n is the first service-provider with its free capacity in A' along the path.

Next, we prove that o_i is d-satisfiable in A' for $i = 1, 2, .., n-1$. We prove by

induction starting from proving the d-satisfiability of o_{n-1} as a base case. This proof can be done easily by the three rules described above and Inequality (3.1).

Consider o_{n-1}. Since o_{n-1} is matched with p_n in A_o (by R2), according to Inequality (3.1), we deduce that $d(o_{n-1}, p_n) < d$. Thus, since p_n has its free capacity in A', p_n is a d-available service-provider for o_{n-1} in A'. We conclude that o_{n-1} is d-satisfiable in A'.

Next, we assume that there exists a positive integer k such that $2 \leq k \leq n-1$ and o_k is d-satisfiable in A'. With this assumption, we want to prove that o_{k-1} is d-satisfiable in A'. Since p_n is the first service-provider with its free capacity in A' along the path, we know that p_k has no free capacity in A' (since $2 \leq k \leq n-1$). Besides, p_k is matched with o_{k-1} in A_o (by R2). According to Inequality (3.1), we have $d(o_{k-1}, p_k) < d$. We deduce the following Statement (*): p_k is a d-occupied service-provider for o_{k-1} in A'.

Consider two cases. *Case (I):* There exists a d-available service-provider for o_{k-1} in A'. In this case, o_{k-1} is d-satisfiable in A' (by Availability Condition).

Case (II): There does not exist any d-available service-provider for o_{k-1} in A'. In this case, note that p_k is matched with o_k in A' (by R3) and o_k is d-satisfiable in A' (by the hypothesis). Since p_k is a d-occupied service-provider for o_{k-1} in A' (by Statement (*)), we deduce that o_{k-1} is d-satisfiable in A' (by Non-Availability Condition).

In Case (I) and Case (II), we have the same conclusion. By induction, we conclude that o_i is d-satisfiable in A' for $i = 1, 2, .., n-1$.

So, o_1 is d-satisfiable in A'. The proof for Case 1 is complete.

Consider Case 2. Case 2 can be easily transformed to Case 1 by duplicating each customer o' (service-provider p') $o'.w$ ($p'.w$) times, and each match (o', p', w') w' times. This completes the proof. ■

Algorithm 5 Algorithm *Swap-Chain*(P, O)

1: initialize a full assignment A between P and O
2: **while** there exists an extreme match m in A which involves a customer o such that o is d-satisfiable in $A - \{m\}$ where d is the matching distance of this extreme match **do**
3: $\quad A \leftarrow$ *Swap*(A, m)
4: **return** A

The above lemma motivates us to design *Swap-Chain* as shown in Algorithm 5. In this algorithm, *Swap* is the *re-matching operation* related to an extreme match m in A. We will describe how we perform this operation next.

The Swap Operation

We first need to introduce a concept called "d-swapping chain" which is used for the *Swap* operation. Roughly speaking, it is a *list* of objects describing which customers and service-providers in the current assignment are involved in the re-matching (or *Swap*) operation such that the new matching distance for each of these customers is smaller than d where d is a non-negative real number.

A *list* is represented in the form of $(x_1, x_2, ..., x_l)$ where x_i is an object (either a customer or a service-provider) for $i \in [1, l]$ and l is the number of objects in the list. Given a list L in the form of $(x_1, x_2, ..., x_l)$, a pair in the form of (x_i, x_{i+1}) is said to be an *even pair* in L if i is divisible by 2. Otherwise, it is said to be an *odd pair* in L. Given two lists L_1 and L_2 where L_1 is $(x_1, x_2, ..., x_l)$ and L_2 is $(y_1, y_2, ..., y_{l'})$, the *list concatenation* of L_1 and L_2, denoted by $L_1 \diamond L_2$, is defined to be $(x_1, x_2, ..., x_l, y_1, y_2, ..., y_{l'})$.

Definition 3.5.3 (d-Swapping Chain) *Let A be an assignment. Suppose that o is d-satisfiable in A. We define a d-swapping chain from o in A, denoted by $C_d(o)$, as follows according to the availability condition and the non-availability condition.*

- *$C_d(o)$ is the list (o, p') if the availability condition is satisfied where p' is a d-available service-provider for o in A, or*

- $C_d(o)$ *is the list* $(o, p') \diamond C_d(o')$ *if the non-availability condition is satisfied where* (p', o') *is a* d*-substitute pair for* o *in* A*.* □

Example 3 [d-Swapping Chain] Consider Example 1. The assignment A is still $\{(o_1, p_2), (o_3, p_3)\}$. Let $d = 10$. A d-swapping chain from o_1 in A, denoted by $C_d(o_1)$, can be (o_1, p_1). Besides, a d-swapping chain from o_2 in A, denoted by $C_d(o_2)$, can be $(o_2, p_2) \diamond C_d(o_1)$ (which is equal to (o_2, p_2, o_1, p_1)) since (p_2, o_1) is a d-substitute pair for o_2 in A. □

Let A be an assignment and d be a non-negative real number *at least* the *mmd* of A. Given a customer o, a d-swapping chain from o in A, denoted by C, has the following properties.

- The total number of objects in C is even.
- C is a list containing interleaved customers and service-providers. The first object in C is a customer. We call it as the *first customer* wrt C. The last object in C is a service-provider p' and the second-to-last object in C is a customer o'. We call p' as the *last service-provider* wrt C. Note that p' is a d-available service-provider for o' in A.
- Each *odd* pair in L is in the form of (o', p') and $d(o', p') < d$.
- Each *even* pair in L is in the form of (p', o') and $d(p', o') \leq d$ (note that $d(p', o')$ is at most the *mmd* while d is at least the *mmd* as has been specified). For each even pair (p', o') in L, there exists a positive integer w' such that $(o', p', w') \in A$ and w' is said to be the *weight* of the even pair (p', o').

Note that given a customer o and a non-negative real number d at least the *mmd* of an assignment A, o is d-satisfiable in A if and only if there exists a d-swapping chain from o in A.

After we describe the d-swapping chain, we are ready to describe how to perform the re-matching operation, *Swap*, based on this chain. For the ease of illustration, we first assume that the capacity (demand) of each service-provider (customer) is 1. We call this assumption the *unit assumption*. After we explain the intuition under the unit assumption, we will relax it. Under the unit assumption, the amount of the service given by a service-provider to a customer in a match is exactly equal to 1. Thus, the weight of each possible even pair in a swapping chain C is equal to 1.

Suppose that we are given a full assignment A. We describe our *Swap* algorithm as follows.

Step (a) (Extreme Match Breaking): We find an extreme match m. Let d be the matching distance of m in A and o be the customer matched in m. We then break this extreme match m in A. That is, we remove m from A and form a new assignment A' (i.e., $A' = A - \{m\}$).

Step (b) (Swapping Chain Finding): We then find a d-swapping chain from o in A', denoted by C.

Step (c) (Chain Breaking): Note that each *even* pair (p', o') in C corresponds to a match in A'. For each even pair (p', o') in C, we break the match (p', o') (or formally (o', p')) in A'. Note that the customer o' in each even pair (p', o') in C has no deficient demand before this step but has his/her deficient demand after this step.

Step (d) (Chain Matching): For each *odd* pair (o', p') in C, we form a match (o', p') in A'. At this moment, the customer o' in each odd pair (o', p') in C has no deficient demand.

Let X be the set of customers involved in the swapping chain C. Note that with the above *Swap* algorithm, the *mmd*, say d', of the resulting assignment involving only the customers in X is smaller than the *mmd*, say d, of the original assignment involving only the customers in X. This is because we make sure that for each odd pair (o', p')

in C (which forms a match in the resulting assignment), the distance between o' and p' is smaller than d.

If the original assignment contains *exactly one* extreme match, it is easy to see that the *mmd* of the resulting assignment involving *all* customers is smaller than the *mmd* of the original assignment involving *all* customers. However, it is possible that multiple extreme matches exist in an assignment A which have the same matching distance d. The *mmd* of the resulting assignment involving *all* customers decreases only after we break *all* of these extreme matches.

Example 4 [Swap] Suppose that the capacity (demand) of each service-provider (customer) is 1. Consider Figure 3.3(a) which shows a full assignment $\{(o_1, p_2), (o_2, p_1), (o_3, p_3)\}$. We denote this assignment by A. (o_2, p_1) is an extreme match in A. Let $d = d(o_2, p_1) = 10$. The *Swap* operation based on A and match (o_2, p_1) works as follows. First, we break the extreme match (o_2, p_1) and the resulting assignment is shown in Figure 3.3(b). Second, we find a d-swapping chain C from o_2 which is (o_2, p_2, o_1, p_1) (Refer Example 3 for illustration). Third, we break the even pairs in C which include (p_2, o_1) only. Forth, for each odd pair in C, we form its corresponding match and thus matches (o_2, p_2) and (o_1, p_1) are formed. Figure 3.3(c) shows the resulting assignment. Clearly, the new assignment is still full, but with a smaller *mmd* (i.e., 7). □

Next, we relax the unit assumption such that the capacity (demand) of each service-provider (customer) could be any positive integer instead of 1. In this case, the weight of an even pair in a swapping chain C can be different from that of another even pair.

The *Swap* algorithm can also be used with this relaxation except the following changes related to the weight of a match.

Step (a) (Extreme Match Breaking): We perform the same operation as before. But,

after the breaking of an extreme match in the form of (o, p, w), resulting an assignment A', we obtain that o has its deficient demand equal to w (instead of 1) while p has its free capacity at least w (instead of 1).

Step (b) (Swapping Chain Finding): Similarly, we perform the same operation.

Step (c) (Chain Breaking): In this step, due to the weights of matches, we have to calculate the weights of matches which are used in this chain breaking operation. Specifically, let W be $W_e(C) \cup \{w_o\} \cup \{w_p\}$ where $W_e(C)$ is the set of the weights of all possible even pairs in C, w_o is the deficient demand of the first customer wrt C in A' and w_p is the free capacity of the last service-provider wrt C in A'. We define the *swapping amount* of the chain C, denoted by $Amount(C)$, to be $\min_{w \in W}\{w\}$. Roughly speaking, $Amount(C)$ corresponds to the greatest possible amount of service in a match along the chain such that Steps (c) and (d) can be executed successfully. Let $w_s = Amount(C)$. Note that w_s is smaller than or equal to the weight of each even pair in C.

We execute Step (c) as follows. For each *even* pair (p', o') in C, we break the match (o', p', w') in A' where w' is a positive integer and form a match $(o', p', w' - w_s)$ in A'. Note that the customer o' in each even pair (p', o') in C has no deficient demand before this step but has his/her deficient demand equal to w_s after this step.

Step (d) (Chain Matching): In Step (d), we perform the matching with the weight w_s. That is, for each *odd* pair (o', p') in C, we form a match (o', p', w_s) in A'. Note that at this moment, the customer o' in each odd pair (o', p') in C except the first odd pair has no deficient demand in A'.

Step (e) (Deficient Demand Checking): This step is new. If the customer o in the first odd pair has no deficient demand in A' (this case happens when $w_s = w_o$), we can return the resulting assignment A' generated from Step (d). If o has his/her deficient demand in A' (this case occurs when $w_s < w_o$), then we continue to execute Step (b)

Algorithm 6 Algorithm *Swap*(A, m)

Require: a full assignment A and an extreme match m in A
// Step (a) (Extreme Match Breaking)
Let m be the match involving customer o and service-provider p with its matching distance equal to d
$A' \leftarrow A - \{m\}$
while there exists a d-swapping chain from o in A' **do**
  // Step (b) (Swapping Chain Finding)
  $C \leftarrow$ a d-swapping chain from o in A'
  // Step (c) (Chain Breaking)
  $w_s \leftarrow Amount(C)$
  for each *even* pair (p', o') in C **do**
    find a match (o', p', w') in A'
    $A' \leftarrow A' - \{(o', p', w')\}$
    if $w' \neq w_s$ **then** $A' \leftarrow A' \cup \{(o', p', w' - w_s)\}$
  // Step (d) (Chain Matching)
  for each *odd* pair (o', p') in C **do**
    $A' \leftarrow A' \cup \{(o', p', w_s)\}$
  // Step (e) (Deficient Demand Checking)
  if o has no deficient demand in A' **then break;**
// Step (f) (Post-Matching)
if o has his/her deficient demand in A' equal to w'' **then**
  $A' \leftarrow A' \cup \{(o, p, w'')\}$
return A'

to Step (d) until o has no deficient demand in A' or o becomes not d-satisfiable in A'. When we stop the above iterative process, if o has no deficient demand in A', similarly, we can return A' as the output. If o is not d-satisfiable in A', we will run an additional step called *Post-Matching* in Step (f).

Step (f) (Post-Matching): This step is also new. It will be executed if o has his/her deficient demand in A', say w'', and is not d-satisfiable in A'. In this case, it is not possible to reduce the matching distance of the extreme match m involving o. Thus, we create the match (o, p, w'') in A' where p is the service-provider involved in m. Finally, we return A'.

The pseudo-code of *Swap* is shown in Algorithm 6.

With Algorithm 6, it is easy to show the correctness of the *Swap-Chain* algorithm (Algorithm 5) as follows.

Theorem 3.5.1 *The* Swap-Chain *algorithm returns the optimal assignment for the*

SPM-MM problem. □

Proof. The correctness follows from Lemma 3.5.1. ■

3.5.3 Remaining Issues & Theoretical Analysis

Remaining Issues. There are two remaining issues in *Swap-Chain*, namely the initialization of a full assignment (line 1 in Algorithm 5) and the Swapping Chain Finding step in the *Swap* algorithm (line 6 in Algorithm 6).

Issue 1: There are many possible ways of initializing a full assignment. In our implementation, we consider the following two methods, namely *Sort* and *Fair*. *Sort* returns an assignment by a two-step approach. First, for each $o \in O$, it maintains a list of all service-providers in ascending order of their distances to o. Second, it processes all $o \in O$ one by one. When processing a specific o, it traverses the service-providers in o's corresponding list sequentially and for the currently traversed p, it assigns the service with the amount equal to $\min\{o.d, p.f\}$ from p to o, where $o.d$ is o's deficient demand and $p.f$ is p's free capacity. The traversing process stops when o's demand has been satisfied. *Fair* denotes the method of generating the fair assignment. Note that we do not adopt the globally optimized assignment in the initialization since the time complexity of the algorithm for finding the globally optimized assignment is much higher than that for finding other assignments like the fair assignment.

Issue 2: For Swapping Chain Finding (i.e., finding a d-swapping chain from a customer o in an assignment A), we design a *Breadth First Search* (BFS) method as follows. It maintains a queue Q and initially inserts o into Q. Then, it processes the elements in Q one by one as follows. It starts processing the first element in Q. If the current element in Q (being processed) is a customer, say o_c, it inserts into Q all service-providers (in any order) that have their distances from o_c *smaller* than d and *have not been inserted* into Q. These service-providers can be found by issuing a

range query on P from o_c. We say that o_c is the *parent* of all these service-providers. If the current element in Q (being processed) is a service-provider, say p_c, consider two cases. Case 1: p_c has no free capacity. In this case, it inserts all the customers matched with p_c in A into Q (in any order) and p_c is said to be the *parent* of all these customers. Case 2: p_c has its free capacity. In this case, it traces all *ancestors* of p_c until the (starting) customer o is reached, and returns the traced list (in this list, the first element is o and the last element is p_c) as a d-swapping chain from o. The above process continues with the next element in Q until either a d-swapping chain is found or all elements in Q have been processed. In the latter case, it means that there does not exist any d-swapping chain from o.

Here, we need to perform range queries on P. Let $\beta(|P|)$ be the cost of a range query on a dataset of size $|P|$. In [19], with the data structure with its size of $O(|P|(\log|P|\log\log|P|)^2)$ and its construction time complexity of $O(|P|\log|P|)$, $\beta(|P|) = O(\log|P| + k)$ where k is the size of the answer of this query. In practice, $k << |P|$ usually holds. Note that in our implementation, instead of the data structure proposed in [19], we adopt an R-tree index built on P for supporting range queries since it is available in commercial databases and is found to be efficient in practice (though it does not have good worst-case asymptotic performance).

Theoretical Properties. We first describe some theoretical properties which will be used to analyze the time complexity of our *Swap-Chain* algorithm.

Given a match (o, p, w) in an assignment, we say that (o, p) is its *match signature*. Given an assignment A, a list of interleaved objects from P and O in the form of $(o_1, p_1, o_2, p_2 \ldots, o_n, p_n)$ is said to be a *match cycle* if each two adjacent objects in the list form a match in A, i.e. o_i is matched with p_i for $1 \leq i \leq n$, o_{i+1} is matched with p_i for $1 \leq i \leq n-1$, and o_1 is matched with p_n in the assignment. The length of a cycle is defined to be the number of elements in the cycle. An assignment A is said to

be *cyclic* if A contains a match cycle.

Interestingly, a non-cyclic assignment has a theoretical bound on the number of matches in the assignment.

Lemma 3.5.2 *Given P and O, the number of matches in a non-cyclic assignment is bounded by $|P| + |O| - 1$.* □

Proof. First, we show that in a non-cyclic assignment A involving no match cycle, there exists an element e (either a service-provider or a customer) such that e is involved in *exactly* one match in A (We can prove by contradiction since if each element is involved in at least two matches, there exists a match cycle in the assignment). We say this match is *critical*. Second, given an assignment A, we iteratively remove each *critical* match from A until no matches exist in A. Since each removal operation makes at least one element unmatched and the last removal operation makes exactly two elements unmatched, we know the number of matches in A is at most $|P|+|O|-1$.

■

Furthermore, given an assignment A with a match cycle C, the following lemma suggests that C could be *destroyed* in A easily such that some conditions in A are still satisfied.

Lemma 3.5.3 *Let A be a cyclic assignment with a match cycle C. We can transform A to another assignment A' such that (1) the* mmd *of A' is at most that of A; (2) the deficient demand (free capacity) of each $o \in O$ ($p \in P$) remains unchanged and (3) A' does not contain C nor any matches with* new *match signatures compared with A. Besides, the cost of this transformation is $O(n)$ where n is the length of C.* □

Proof. Let the cycle C be $(o_1, p_1, o_2, p_2, ..., o_n, p_n)$. Without loss of generality, let (o_1, p_1) be the match along C which has the smallest matching weight, says w_m. We

break each match (o_i, p_i, w_i) with the amount of w_m for each $i \in [1, n]$. Thus, o_i has its deficient demand at least w_m and p_i has its free capacity at least w_m for each $i \in [1, n]$. Next, we create a new match (o_{i+1}, p_i, w_m) for each $i \in [1, n-1]$ and a new match (o_1, p_n, w_m). (Note that for the new match formed (o, p, w_m), if there exists an original match (o, p, w) in the assignment, we just combine these two matches as a single match $(o, p, w_m + w)$). Let A' be the resulting assignment. It can be verified that o_1 and p_1 originally matched in A are not matched in A' and thus A' does not contain cycle C. Besides, it is easy to verify that for each $o \in O$, $\sum_{(o,p,w)\in A} w = \sum_{(o,p,w)\in A'} w$ and for each $p \in P$, $\sum_{(o,p,w)\in A} w = \sum_{(o,p,w)\in A'} w$. Furthermore, the *mmd* of A' is at most that of A and no matches with new match signatures are formed in A'. Clearly, the cost of the above process is simply $O(n)$. ■

Time Complexity. We let $|V| = |P| + |O|$ and $|E| = |P| \cdot |O|$. Suppose that we build an index as introduced in [19] on P to facilitate range queries described before. Let λ be the time complexity of building this index. Let γ be the time complexity of the full assignment initialization (line 1 of Algorithm 5). Let R be the total number of possible extreme matches fetched in *Swap-Chain* (i.e., the number of iterations in lines 2-3 of Algorithm 5). Let I denote the time complexity of the *Swap* algorithm. The time complexity of *Swap-Chain* is $O(\lambda + \gamma + R \cdot I)$.

Consider λ. From [19], we know that $\lambda = O(|P| \log |P|)$.

Consider γ. If *Sort* is adopted, it could be verified that γ is equal to $O(|O| \cdot |P| \log |P|)$. If *Fair* is used, γ is equal to $O((|P| + |O|) \cdot (\log |P| + \log |O|))$ [96]. Besides, we introduce a lemma which will be used later.

Lemma 3.5.4 *The assignment initialized by* Sort *and the assignment initialized by* Fair *are both non-cyclic.* □

Proof. Let A_s be the assignment initialized by *Sort* and A_f be the one initialized

by *Fair*.

We first prove that A_s is non-cyclic. Suppose *Sort* processes O in order of $o_1, o_2, ..., o_m$, where m is the size of O. We denote by A_{o_i} the assignment that is formed immediately after processing o_i (thus, $A_{o_m} = A_s$). We claim that for a specific customer o_i, among all products that are matched with o_i in A_{o_i}, at most one product has its free capacity non-zero. This can be verified by the principle adopted in *Sort* that o_i always exhausts the current product chosen to be matched with o_i before the next product is considered. Now, we show A_s is non-cyclic by contradiction. Assume that there exists in A_s a match cycle $C = (o_{c_1}, p_{c_1}, ..., o_{c_n}, p_{c_n})$. Without loss of generality, among all customers involved in C, we assume o_{c_1} is the first customer processed by *Sort*. Consider $A_{o_{c_1}}$ (the assignment formed immediately after o_{c_1} is processed). Both p_{c_1} and p_{c_n} are matched with o_{c_1} in $A_{o_{c_1}}$ (since p_{c_1} and p_{c_n} are matched with o_{c_1} in A_s) and both of them have their free capacities non-zero (since p_{c_1} (p_{c_n}) is matched with o_{c_2} (o_{c_n}) later on where processing o_{c_2} ($o_c c_n$)). Thus, this leads a contradiction that at most one product matched with o_{c_1} in $A_{o_{c_1}}$ has its free capacity non-zero.

Next, we prove that A_f is non-cyclic by contradiction. This proof can be done by using the fact that no *dangling pair* [96] exists in a fair assignment. We only consider the case where the pairwise distances between P and O are distinct. For the other case where some pairwise distances are identical, we can always perform an infinitesimal perturbation on the locations of some products and/or customers such that ties of distances are broken. Assume that there exists a match cycle $C = (o_1, p_1, ..., o_n, p_n)$ in A_f. Without loss of generality, we assume that match (o_1, p_1, w_1) has the smallest matching distance among all matches involved in C. As a result, we know $d(o_1, p_1) < d(o_2, p_1)$ and $d(o_1, p_1) < d(o_1, p_n)$, which implies that (o_1, p_1) is a *dangling pair* [96]. However, according to [96], there exist no dangling pairs for a fair assignment. There, we know A_f is not a fair assignment, which leads to a contradic-

tion. ■

Consider R. Before we give the bound on R, we give a lemma.

Lemma 3.5.5 *A match with a given match signature can be fetched as an extreme match at most once in* Swap-Chain. □

Proof. This lemma is trivially true if all pairwise distances are distinct. This lemma also holds even if they are not distinct. This is because during the execution of *Swap* (in line 3 of *Swap-Chain*), once the extreme match with a particular match signature is broken, no matches with the same match signature will be formed again except the last Step (f) (i.e., post-processing) which denotes that there is no need to fetch additional extreme matches and the algorithm terminates. ■

Note that there are at most $|E|$ $(= |P| \cdot |O|)$ possible match signatures. By Lemma 3.5.5, we deduce that R is bounded by $|E|$. In practice, $R << |E|$. In our experiments, R is about 500 on average, which is very small compared with $|E|$ which is as large as 250,000,000 in our default setting.

Consider I. According to Algorithm 6, I depends on the cost of the while-loop (lines 6-17) and the total number of while-loops, denoted by t. Consider a while-loop which involves the operation of finding a d-swapping chain (line 6), whose cost is denoted by $\mathcal{C}_1$, the operation of re-matching the elements along the chain (line 7-17), whose cost is denoted by $\mathcal{C}_2$, and an additional operation introduced here which is used to transform the assignment obtained to a non-cyclic assignment and whose cost is denoted by $\mathcal{C}_3$. Thus, I is $t \cdot (\mathcal{C}_1 + \mathcal{C}_2 + \mathcal{C}_3)$.

Consider $\mathcal{C}_1$ which corresponds to the time cost of the BFS implementation. Note that at the beginning of each while-loop, the assignment is non-cyclic. This is because the initialized assignment is non-cyclic (Lemma 3.5.4) and at the end of each while-loop, the additional operation introduced here transforms the assignment to a non-

cyclic one. Thus, it could be verified that $\mathcal{C}_1 = O(|O| \cdot \beta(|P|) + |P|)$ (the BFS method (1) involves at most $|O|$ range queries on P (which incurs the cost of $O(|O| \cdot \beta(|P|))$), and (2) retrieves at most $|P| + |O| - 1$ matches (from service-providers) according to Lemma 3.5.2 and the fact that the assignment is non-cyclic (which incurs the cost of $O(|P| + |O|)$)).

Consider $\mathcal{C}_2$. It is simply $O(|P| + |O|)$ (=$O(|V|)$).

Consider $\mathcal{C}_3$. After the Chain Matching step, the assignment contains $O(|V|)$ matches (since it contains at most $|P| + |O| - 1$ matches at the beginning of the while-loop and the Chain Matching step forms at most $\min\{|P|, |O|\}$ new matches). Clearly, each match cycle in an assignment A corresponds to a cycle in the un-directed graph $G_A(V', E')$, which involves P and O as vertices in V' and all matches as edges in E'. Note that $|E'| = O(|V|)$ (by Lemma 3.5.2). Thus, to find a match cycle in A, we can find a cycle in G_A and this can be easily achieved by a common DFS technique [27], which runs in $O(|V'| + |E'|)$ (=$O(|V|)$) time. According to Lemma 3.5.3, destroying a match cycle incurs $O(|V|)$ (a match cycle has its length at most $|P| + |O|$) and it does not introduce any match with a new match signature. So, we can transform the assignment to a non-cyclic one by iteratively destroying the match cycles until no match cycles exist in the assignment. Thus, $\mathcal{C}_3 = O(c \cdot |V|)$, where c is the number of match cycles formed due to the Chain Matching step. It could be verified that c is bounded by $\min\{|P|, |O|\}$ since the Chain Matching step introduces at most $\min\{|P|, |O|\}$ matches and each such match can form at most one new match cycle. In practice, $c << \min\{|P|, |O|\}$ (e.g., c is about 17 on average in our experiments under the default setting).

Consider t. Recall that t is the number of while-loops in *Swap* needed to re-satisfy the deficient demand of customer o due to the break operation on the extreme match involving o. Clearly, t is bounded by $\overline{w} = \min\{\max_{p \in P} p.w, \max_{o \in O} o.w\}$. Usually,

t is much smaller than this upper bound $\overline{w}$. For example, in our experiments on real datasets, on average, t is 2 (with a maximum of 40) but $\overline{w}$ is in thousands.

In view of the above discussion, we know that $I = O(t \cdot (|V| \cdot \beta(|P|) + c \cdot |V|))$. As a result, the time complexity of *Swap-Chain* is $O(\lambda + \gamma + R \cdot t \cdot (|V| \cdot \beta(|P|) + c \cdot |V|))$, where $R << |E|$, $t << \min\{\max_{p \in P} p.w, \max_{o \in O} o.w\}$, and $c << \min\{|P|, |O|\}$.

3.6 Discussion

Any assignment with its *mmd* equal to the optimal *mmd* is a solution of SPM-MM. Thus, there may exist multiple possible solutions for SPM-MM. In this case, our *Swap-Chain* returns one of them at random. However, SPM-MM can be enriched by considering a secondary objective (e.g., minimizing the sum of the matching distances) for the final solution among these multiple solutions. Furthermore, the bottleneck nature of the SPM-MM objective makes it quite easy to be incorporated with a secondary objective since the optimized mmd, say d_o, can always be used as a *hard* constraint for optimizing the secondary objective. Specifically, matching *any* pair of two objects which has its distance bounded by d_o does not destroy the optimality while matching *any* pair of two objects which has its distance larger than d_o definitely ruins the optimality. Thus, we can adopt a two-step mechanism for the SPM-MM problem with a secondary objective. First, we compute the optimal *mmd*, say d_o, using *Swap-Chain*. Second, we ignore all pairs (o, p) with $d(o, p) > d_o$ for matching when optimizing the secondary objective. For instance, if the secondary objective is to minimize the sum of the matching distances, we can solve this enriched version of SPM-MM easily by first computing the optimal *mmd* d_o and then adopting any popular algorithm for *Minimum Weight Matching* [5] with the constraint that all pairs (o, p) with $d(o, p) > d_o$ cannot be matched (this could be achieved by excluding from the graph used by the algorithm all those edges with the corresponding distances larger than d_o).

Next, we discuss SPM-MM in a more general setting where the pairwise distances between P and O could be *non-metric* or *non-spatial*. Interestingly, our proposed methods can also be adapted to this general setting. *Threshold-Adapt* still works in the general setting (recall that *Threshold-Adapt* is adapted from *Threshold* which is designed for general bipartite graphs). We can also adapt *Swap-Chain* to the general setting with some sacrifice of its time complexity as follows. Two parts involved in *Swap-Chain* rely on the spatial setting, namely the *Fair* method for initializing a full assignment (a fair one) and the BFS method for finding a d-swapping chain in an assignment A. To initialize a fair assignment in the general setting, one can adopt the *Stable Marriage* algorithm which incurs the cost of $O(|P| \cdot |O|)$ [37] (instead of $O((|P| + |O|) \cdot (\log |P| + \log |O|))$ in the spatial setting [96]). To find a d-swapping chain from a customer o in the general setting, one can first *materialize* a *directed* graph $G'(V', E')$ such that (1) $V' = P \cup O$; (2) for each $o \in O$, (o, p) is a directed edge in E' for all $p \in P$ with $d(o, p) < d$; (3) for each $p \in P$, (p, o) is a directed edge in E' for all $o \in O$ that is matched with p in A, and then find a path from o to a service-provider p with its free capacity non-zero using a BFS on G'. The resulting path corresponds to a d-swapping chain from o in A. Clearly, $|V'| = |V|$ and $|E'| \leq |E|$. Thus, the cost of the BFS method is $O(|V| + |E|)$ (instead of $O(|O| \cdot \beta(|P|) + |P|)$ in the spatial setting, where $\beta(|P|)$ is the range query cost on P).

Finally, we would like to note some differences between our d-swapping chain technique and the well-known *augmenting path* techniques. A typical augmenting path technique is used for computing the *maximum flow* whose main idea is to iteratively finding an *augmenting path* and augmenting the flow along this path until no augmenting paths are possible. As could be noticed, the goal of an augmenting path technique is to *increase* the flow iteratively while the goal of our d-swapping chain technique is to *keep* the flow while decreasing the *mmd* of the corresponding matching. Specifically, in our d-swapping chain technique, we find an extreme match (o, p) and *break* it so

	Cardinality	
	Populated Areas (PA)	Fire Stations (FS)
AB	4,999	447
BC	6,609	595
ON	12,474	1,215
QC	12,936	1,259

Table 3.1: Real datasets (SPM-MM)

Factor	Configuration
Cardinality ($\|O\|$)	10k, 30k, **50k**, 70k, 100k
Dim.	2, **3**, 4, 5
Size ratio (r)	5, **10**, 15, 20, 25
Weight ratio (k)	1, 1.5, **2**, 2.5, 3
O's weights	[1, 10)
Scalability ($\|O\|$)	250k, 500k, 750k, 1000k

Table 3.2: Synthetic datasets (SPM-MM)

that we will find a d-swapping chain from o, while in an augmenting path technique, there is no such breaking operation on a chosen match before the augmenting path is to be found.

3.7 Empirical studies

We used four real datasets, namely AB, BC, ON and QC, in our experiments. Each real dataset contains two sets of spatial objects, a set of populated areas (PA) and a set of fire stations (FS). Specifically, dataset AB contains the set of PAs and the set of FSs in Alberta, Canada. Datasets BC, ON and QC contain the same information in the three other provinces in Canada, namely, British Columbia, Ontario and Quebec, respectively. We collected the PAs from Census Canada (http://www12.statcan.gc.ca), each of which corresponds to a dissemination area, and estimated the coordinates of PAs with the help of the Postal Code Conversion File of Canada [86]. The population of each PA ranges from 400 to 700 in most cases [86]. We collected the FSs from Fire-Canada (http://www.firecanada.ca) and estimated the coordinates of FSs via Google Maps. The capacities of FSs range from 5,500 to 10,000. The coordinates are all normalized to range [0,10000]. For each dataset, we adopt the set of PAs as O and the set of FSs as P. The summaries of the real datasets are shown in Table 3.1.

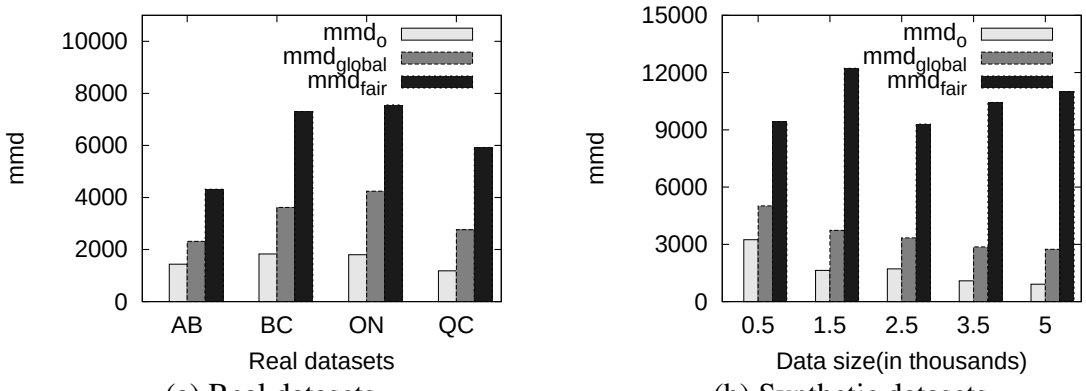
We also used synthetic datasets in our experiments, which are generated as follows. The coordinates of spatial objects follow the Uniform distribution on range [0, 10000]

by default. The demand of each customer in O is set to be [1, 10) randomly. To generate the capacities of the service-providers in P, we define a parameter k, called *weight ratio*, to be the expected ratio between the sum of the service capacities of all service-providers and the sum of the service demands of all customers, i.e., $k = \sum_{p \in P} p.w / \sum_{o \in O} o.w$. Based on the configuration of k, we set the capacities of the service-providers. By default, the capacities are set to be [80, 120) randomly. The parameter configuration of synthetic datasets is shown in Table 3.2 where the default values are shown in bold font.

3.7.1 SPM-MM vs. Existing Spatial Matching Problems

We conducted experiments to compare the optimal mmd, mmd_o, with the mmd's of the fair assignment and the globally optimized assignment, namely mmd_{fair} and mmd_{global}, respectively. In this experiment, we randomly select 10% (5%) in P and 10% (5%) in O for each real (synthetic) dataset. This is because the algorithm (we use the SSPA algorithm in [5]) for computing mmd_{global} is not scalable to large datasets.

Figure 3.4 shows that mmd_{fair} and mmd_{global} are larger than mmd_o. For example, in the real dataset ON (Figure 3.4(a)), the ratio between mmd_{fair} (mmd_{global}) and mmd_o is about 3.5 (2.3). We have similar results on synthetic datasets as shown in Figure 3.4(b).



(a) Real datasets (b) Synthetic datasets

Figure 3.4: Results for the mmd's of different assignments (SPM-MM)

3.7.2 Performance Study

Next, we give the performance study on our proposed algorithms, namely *Threshold-Adapt* and *Swap-Chain*, which include eight instances in total. The details are described as follows.

Recall that *Threshold-Adapt* involves a *maximum-flow* procedure. In the literature, many maximum-flow algorithms have been developed which could be categorized into three branches, namely, *Augmenting-Path* (which mainly includes Dinic, BK and IBFS [40]), *Push-Relabel* (which mainly includes HIPR and PRF), and *Pseudoflow* (which mainly includes HPR). More details about these maximum-flow algorithms could be found in [2] (and the references therein). Besides, according to [2], these maximum-flow algorithms usually favor different applications and it is not always the case that a maximum-flow algorithm with a smaller time complexity runs faster than another with a larger one. Motivated by this, we consider all the above six maximum-flow algorithms, namely, Dinic, BK, IBFS, HIPR, PRF and HPF, for optimizing *Threshold-Adapt*, and the corresponding instances of *Threshold-Adapt* are denoted by *TA-Dinic*, *TA-BK*, *TA-IBFS*, *TA-HIPR*, *TA-PRF* and *TA-HPF*, respectively. Besides, we consider two instances of the *Swap-Chain* algorithm, namely, *Swap-Fair* and *Swap-Sort*, with the initialization methods of *Fair* and *Sort*, respectively.

We evaluated the algorithms mainly in terms of *running time* and *memory*, and study the effects of cardinality, dimensionality, size ratio and weight ratio on the performance of the algorithms. The memory of *Threshold-Adapt* is mainly due to the search space S and the flow network graph, and the memory of *Swap-Chain* is mainly due to the R-tree built on P and the maintained assignment.

We implemented our algorithms in C/C++ and conducted the experiments on a Linux platform with a 2.26GHz CPU and 36GB physical memory.

We present our experimental results as follows.

(1) Effect of Cardinality. We vary $|O|$ and the results are shown in Figure 3.5. We have the following observations. First, there is a clear efficiency gap between the *Swap-Chain* algorithms and the *Threshold-Adapt* algorithms and the gap becomes larger when the data size increases. For example, when $|O| = 100k$, *Swap-Chain* is faster than *TA-IBFS* by more than one order of magnitude. Second, the two *Swap-Chain* algorithms favor different cases. Specifically, *Swap-Sort* runs faster than *Swap-Fair* on relatively small datasets (e.g., $\leq 40k$) while the opposite case becomes true on relatively large datasets. This could be explained by the fact that (1) *Swap-Sort* has no cost of building an R-tree on O while *Swap-Fair* does and (2) *Swap-Sort* has a more expensive initialization procedure (i.e., *Sort*) than *Swap-Fair*. Third, the memory usages of the *Swap-Chain* algorithms are quite low while those of the *Threshold-Adapt* algorithms are dramatically higher (by 2-3 orders of magnitude). For example, when $|O| = 100k$, the *Swap-Chain* algorithms use less than 50MB while each of *Threshold-Adapt* algorithms occupies more than 15GB memory. Forth, among all *Threshold-Adapt* algorithms, *TA-IBFS* runs the fastest and occupies the least memory. For ease of presentation, in the following, we focus on *TA-IBFS* only as the representative of the *Threshold-Adapt* algorithms since it beats all other instances of *Threshold-Adapt* in terms of both time efficiency and space efficiency.

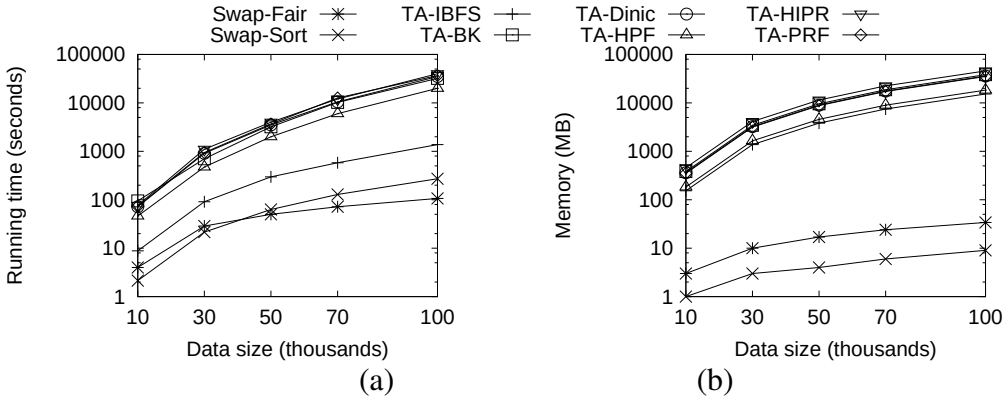


Figure 3.5: Effect of cardinality (synthetic datasets, SPM-MM)

(2) Effect of Dimensionality. Figure 3.6 shows the results of the effect of dimen-

sionality. We observe that the dimensionality only affects the *Swap-Fair* algorithm slightly. Specifically, when the dimensionality increases, the running time of *Swap-Fair* increases slightly. This is because *Swap-Fair* needs to build the R-trees on both P (for searching d-swapping chains) and O (for computing a fair assignment), which cost increases when the dimensionality increases. The dimensionality has negligible effects on *Swap-Sort* and *TA-IBFS*.

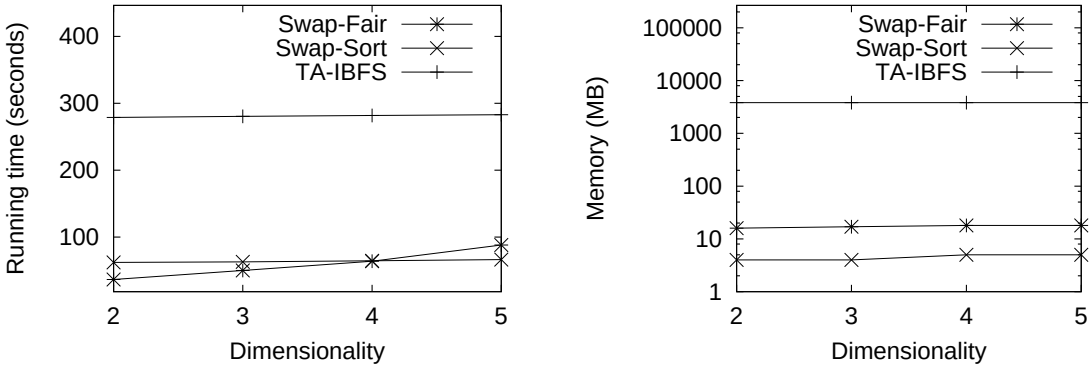


Figure 3.6: Effect of dimensionality (synthetic datasets, SPM-MM)

(3) Effect of Size ratio. We observe some opposite trends on running time and memory when we increase the size ratio r compared with those when we increase the data size. This is reasonable since when the size ratio r increases, $|P|$ decreases (note that $|O|$ is fixed).

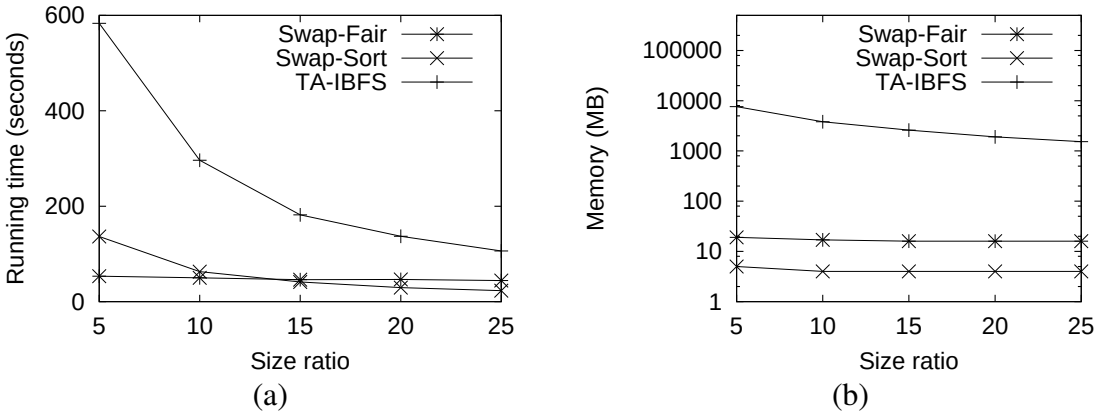


Figure 3.7: Effect of size ratio (synthetic datasets, SPM-MM)

(4) Effect of Weight ratio. Figure 3.8 shows the effect of the weight ratio k. We

observe that the weight ratio has slight effect on *TA-IBFS* only. Specifically, when k increases, the running time of *TA-IBFS* decreases slightly. The reason might be that when k increases (i.e., the total capacities of the service-providers becomes relatively larger), it is more likely that an augmenting path (note that IBFS is an augmenting path algorithm) *carries* more flow and thus the process of computing the maximum-flow could be finished more quickly.

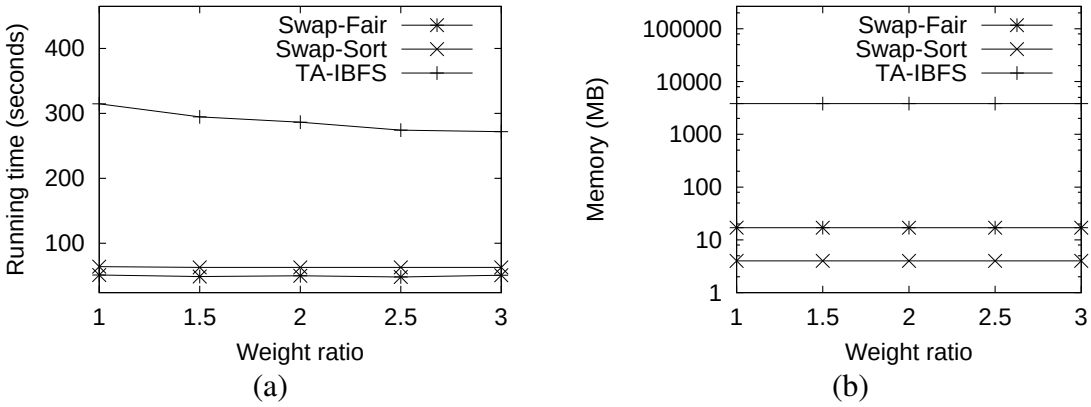


Figure 3.8: Effect of weight ratio k (synthetic datasets, SPM-MM)

(5) Scalability test. Figure 3.9 shows the results of the scalability test for *Swap-Sort* and *Swap-Fair*. Since *Threshold-Adapt* is not scalable, we did not conduct this test for *Threshold-Adapt*. As shown in the figure, the two algorithms are still efficient on large datasets (in millions). Furthermore, *Swap-Fair* is more scalable than *Swap-Sort*. This is because on a large dataset, the initialization process of *Swap-Fair* (i.e., *Fair*) is much faster than that of *Swap-Sort* (i.e., *Sort*).

(6) Experiments on real datasets. Figure 3.10 shows the results for real datasets which are similar to the results for synthetic datasets. Compared with the *Threshold-Adapt* algorithm, our *Swap-Chain* algorithms run faster and use significantly less memory.

(7) Comparison with the *Threashold* algorithm in Euclidean space. We are interested in studying the performance of our proposed algorithms when they are used for

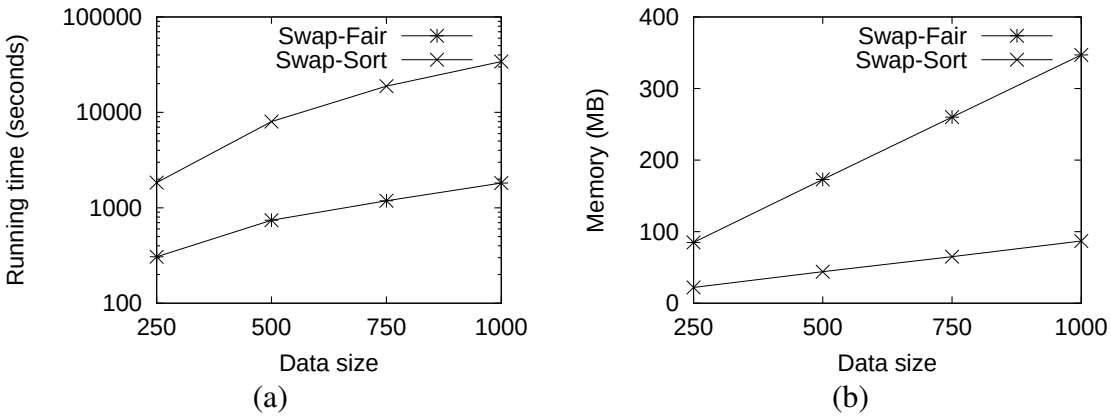


Figure 3.9: Scalability test (synthetic datasets, SPM-MM)

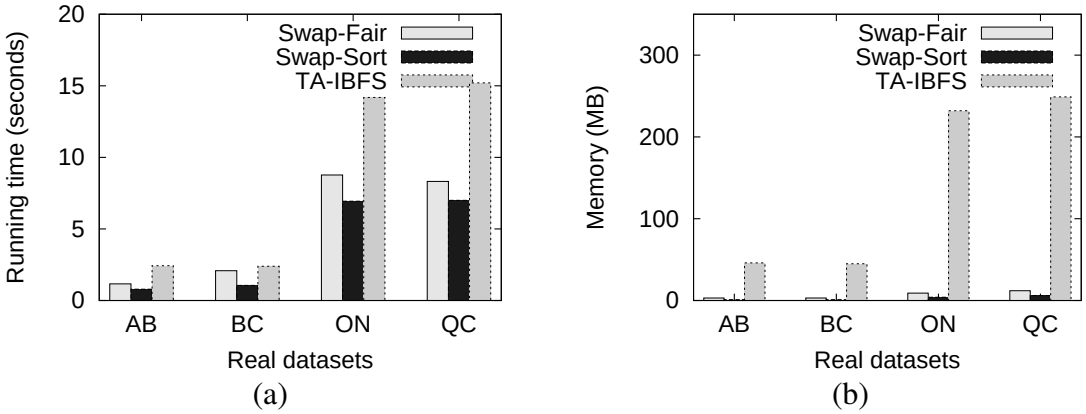


Figure 3.10: Results for real dataset (SPM-MM)

the *un-weighted* version of SPM-MM (i.e., all the capacities/demands are 1's). We compare our algorithm with the state-of-the-art called *Match* [32] which has a theoretical time complexity of $O(n^{1.5} \log n)$. We note here that though *Match* has a smaller time complexity, it has quite a narrow application scope (i.e., for the *un-weighted* version only) and the time complexity is restricted to the 2D space only [32]. The results are shown in Figure 3.11. We observe that our *Swap-Chain* algorithms have comparable running time with the *Match* algorithm and run even faster than *Match* on relatively large datasets. This might be due to the fact that a constant factor which could be large is omitted from the time complexity analysis in [32]. Besides, we found that our *Swap-Chain* algorithms enjoy the superiority of space efficiency over the *Match* algorithm. We also used our real datasets for this experiment by setting the capacities/demands to 1s and observed similar results.

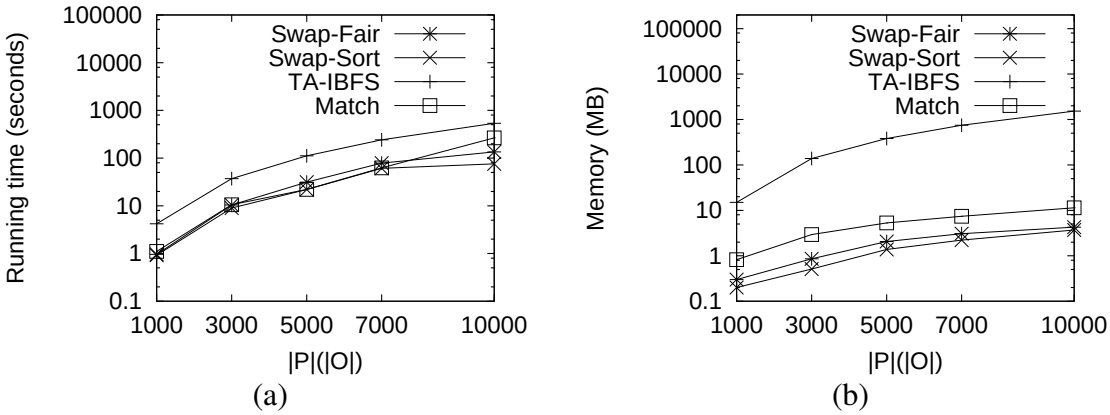


Figure 3.11: *Threshold* vs. *Swap-Chain* (SPM-MM)

(8) Experiments with a Secondary Objective. Besides, we conducted some experiments on the SPM-MM problem with a secondary objective of minimizing the *sum of matching distances* called *sum-md*. Let $A_{mmd}(A_{sum-md})$ be the assignment obtained by optimizing *mmd* (*sum-md*) only. Let $A_{mmd,sum-md}$ be the assignment obtained by optimizing *mmd* first and *sum-md* second. We adopted the SSPA algorithm [5] for optimizing *sum-md*. We conducted our experiments on both synthetic and real datasets where each synthetic/real dataset was sampled first with the sampling rate set to 5% due to the relatively expensive cost of SSPA. The results on the real datasets are shown in Figure 3.12. We observe that on average, compared to A_{sum-md}, $A_{mmd,sum-md}$ can be obtained with a similar time (the cost of optimizing *mmd* is an additional part but the constraint of the optimized *mmd* helps reduce the running time of the process of optimizing *sum-md*) and has the *sum-md* value usually not far away from the *sum-md* value of A_{sum-md} (e.g., within 1.1 factor).

(9) Experiments with non-Euclidean Distances. In addition, we conducted some experiments on our proposed algorithms (*Threshold-Adapt* and *Swap-Chain*) when they are applied to the cases where non-Euclidean distances are used. We used the same real datasets except that the underlying distances between pairs of two objects are measured by the driving time between the two objects. The results are shown in Figure 3.13. We observe that compared with the case of using the Euclidean distances, the efficiency

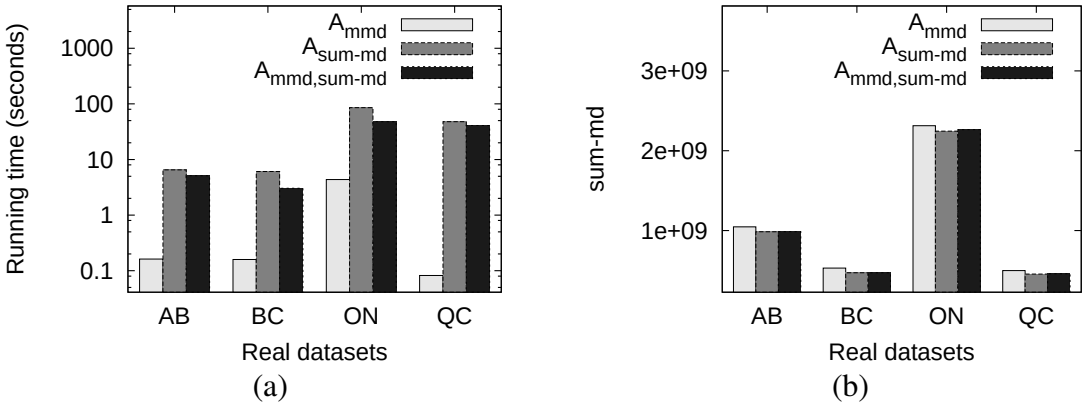


Figure 3.12: Experiments with a secondary objective (SPM-MM)

of *Threshold-Adapt* is similar while the efficiency of the *Swap-Chain* algorithms, especially *Swap-Fair*, degrades to some extent. But, the *Swap-Chain* algorithms still retain the superiority over *Threshold-Adapt* in terms of both time efficiency and space efficiency. For example, on dataset QC, the running time of *Swap-Fair* (*Swap-Sort*) is about 15s (9s) while that of *Threshold-Adapt* is nearly 17s. Thus, compared with the spatial setting, the degrading ratio of *Swap-Fair* (*Swap-Sort*) is about 9/15 (8/9) and that of *Threshold-Adapt* is about 16/17.

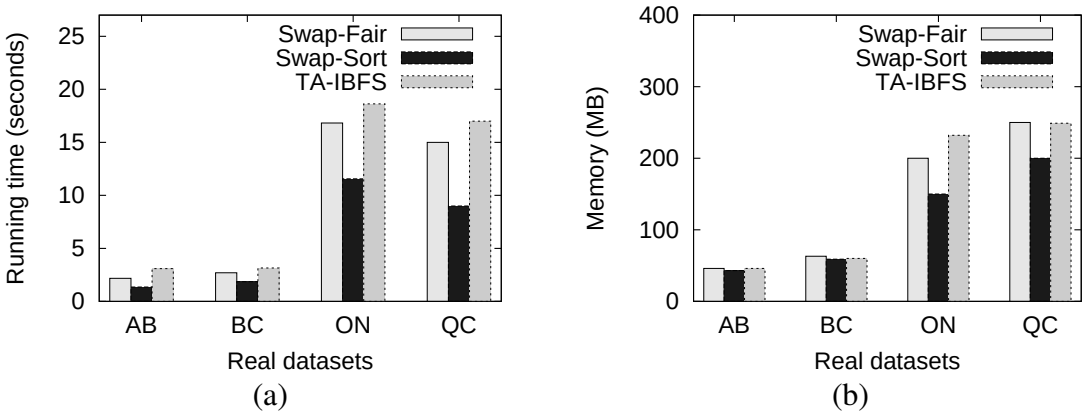


Figure 3.13: Experiments with non-Euclidean distances (SPM-MM)

(10) Comparison with the Augmenting Path Technique. We also conducted experiments on the *Swap-Chain* algorithms with the adaptions of Augmenting Path (AP) techniques [5]. We interpret our d-swapping chains as augmenting paths and denote the resulting *Swap-Chain* algorithms corresponding to *Swap-Fair* and *Swap-Sort* by

AP-Fair and *AP-Sort*, respectively. In our implementations of *AP-Fair* and *AP-Sort*, when finding an augmenting path which corresponds to finding a d-swapping chain in *Swap-Fair* and *Swap-Sort*, respectively, we do a BFS in a graph structure G which contains the edges (o, p) for all pairs of (o, p) with $d(o, p) < d$ and also the edges (p, o) for all matches (o, p, w) in the current assignment. Figure 3.14 shows the results. We observe that there is a clear efficiency gap between *Swap-Fair* (*Swap-Sort*) and *AP-Fair* (*AP-Sort*), and this gap becomes larger when the data size increases. Besides, *AP-Fair* (*AP-Sort*) occupies significantly more memory than *Swap-Fair* (*Swap-Sort*). The reason for the efficiency gap is that each range query on P in *Swap-Fair* (*Swap-Sort*) is performed in $O(\log |P| + k)$ time [19] where k is the size of the answer of the query while its counterpart in *AP-Fair* (*AP-Sort*) is performed by scanning an adjacent list which takes $O(|P|)$ time. The reason for the results of memory usage is that *Swap-Fair* (*Swap-Sort*) maintains no graph structures while *AP-Fair* (*AP-Sort*) does.

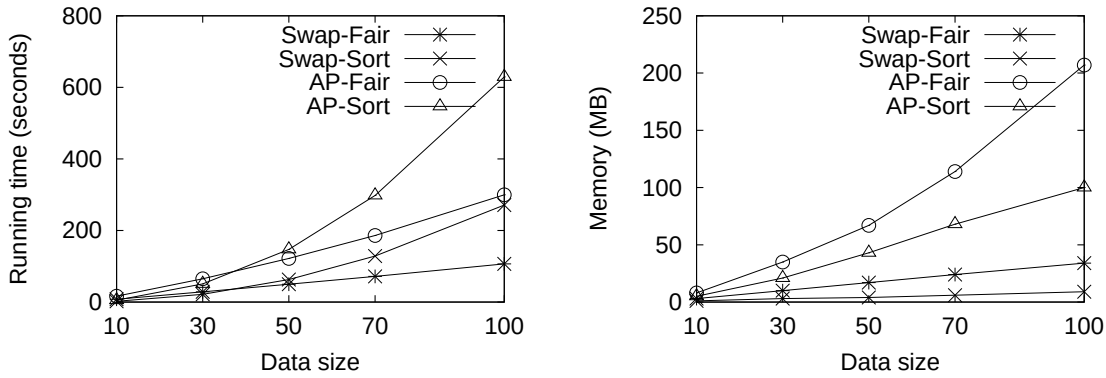


Figure 3.14: d-swapping chain vs. Augmenting path (SPM-MM)

Conclusion: The *Swap-Chain* algorithms, which are efficient and scalable, beat the *Threshold-Adapt* algorithms in terms of both time efficiency and space efficiency. Besides, *Swap-Sort* runs faster than *Swap-Fair* when the datset is relatively small (e.g., $|O| \leq 40k$) while *Swap-Fair* enjoys its superiority over *Swap-Sort* on large datasets.

3.8 Conclusion

In this chapter, we propose a new problem called *SPatial Matching for Minimizing Maximum matching distance* (SPM-MM). We design two algorithms for SPM-MM, namely *Threshold-Adapt* and *Swap-Chain*. *Threshold-Adapt* is simple and easy to understand but not scalable to large datasets. *Swap-Chain* avoids the scalability issues of *Threshold-Adapt* by adopting a novel idea of swapping the matches iteratively and runs faster than *Threshold-Adapt* by orders of magnitudes. We conducted extensive experiments which verified the efficiency and scalability of *Swap-Chain*.

CHAPTER 4

DIRECTION-PRESERVING TRAJECTORY SIMPLIFICATION: MINIMIZING THE SIZE

4.1 Introduction

With the proliferation of GPS-embedded devices (e.g., smart phones and taxis), trajectory data is becoming ubiquitous. Indeed, it has been studied extensively in the past decades in the literature of Moving Objects Databases (MOD) [85, 76], and people use trajectory data for many different purposes, e.g., traffic analysis [69], route recommendation [71, 93], social relationship analysis [78, 99], and user behavior analysis [106, 87]. Trajectory data is usually generated by periodically collecting the position of a moving object with the help of the GPS technologies.

Since the raw trajectory data is usually very large, simplifying trajectory data is important. To appreciate this, consider a city with 10k taxis. Suppose that we track the trajectory of each taxi by sampling its position once every 5 seconds (i.e., the sampling rate is 5s). The size of the collected trajectories for just one day is approximately 4 GB.

Raw trajectory data is large, and hence expensive to store. Even worse, it is expensive to manipulate and to analyze on account of its large size. In fact, most existing query processing and data mining algorithms on trajectory data are memory-resident and thus cannot be used with raw trajectory data that is too large to fit in memory.

A question one may ask is why not just sampling less frequently to reduce the size of the data. The answer is that, in real life, objects have great variance in their velocities. A taxi moving at 40 mph would have moved about 100 yards in 5s, whereas

another taxi stuck at a traffic signal may not have moved at all. Obviously, we need more frequent observations of the former than of the latter. Similarly, we need more observations to capture a taxi that makes a turn and fewer for one that continues straight. Therefore, standard practice is to oversample initially, and then to simplify by eliminating observations that add little information.

In view of this, several algorithms have been developed for simplifying trajectory data [72, 31, 79, 74, 56]. All these algorithms make the natural assumption that the goal should be to simplify trajectories such that the *position information* captured in the simplified trajectories is "similar" to the position information captured in the original trajectories. We can call them *position-preserving trajectory simplification algorithms*. However, as we will soon see, this objective, though natural, is not the best choice in many situations. To illustrate, let us work through a toy example in detail.

Example 5 (Motivating Example) Consider three raw trajectories T_1, T_2 and T_3 as shown in bold lines in Figure 4.1(a)(i), (a)(ii) and (a)(iii), respectively. Each of these trajectories has *four* positions, p_1, p_2, p_3 and p_4. T_1 and T_2 are *similar* to each other, and each of them is *dissimilar* to T_3. Thus, a trajectory clustering algorithm, such as [64], should group T_1 and T_2 in the same cluster and place T_3 by itself in a separate cluster.

Now suppose that these raw trajectories are too large, and so must be simplified to three points each before being further processed. We could use an existing position-preserving trajectory simplification, denoted by $\mathcal{A}_{pos}$, for this simplification. Following existing studies, the first position p_1 and the last position p_4 in each trajectory have to be kept. Therefore, one of position p_2 and position p_3 is to be retained, and the other one dropped.

Consider the simplification process on T_1. It can drop either p_2 or p_3 in the simplified trajectory. Let d_1 (d_2) be p_2's (p_3)'s perpendicular distance to line segment

$\overline{p_1p_3}$ ($\overline{p_2p_4}$). Since $d_1 > d_2$, $\mathcal{A}_{pos}$ drops p_3 and returns the simplified trajectory T_1' as shown in Figure 4.1(b)(i). Similarly, $\mathcal{A}_{pos}$ return T_2' (Figure 4.1(b)(ii)) and T_3' (Figure 4.1(b)(iii)) as the simplified trajectories of T_2 and T_3, respectively. We now see that, though raw trajectories T_1 and T_2 are *similar*, their simplified trajectories T_1' and T_2' generated by $\mathcal{A}_{pos}$ are *dissimilar*. On the other hand, raw trajectories T_1 and T_3 are *dissimilar*, but their simplified trajectories T_1' and T_3' generated by $\mathcal{A}_{pos}$ are *similar*. In consequence, the clustering algorithm on the simplified trajectories T_1', T_2' and T_3', places T_1 and T_3 together into one cluster, and thus fails to produce correct (or expected) clusters.

In contrast, as will be shown later, a *direction-preserving* trajectory simplification method we introduce below, denoted by $\mathcal{A}_{dir}$, would simplify T_1, T_2 and T_3 to T_1'' (Figure 4.1(c)(i)), T_2'' (Figure 4.1(c)(ii)) and T_3'' (Figure 4.1(c)(iii)), respectively. Since T_1'' and T_2'' are similar to each other and each of them is dissimilar to T_3'', the clustering algorithm based on these simplified trajectories would produce the expected clusters.

□

Before we can discuss direction-preserving trajectory simplification in depth, we first have to describe what direction information is, which we do next.

4.1.1 Direction Information

When an object moves from position p to position p', we define the *direction* of this movement to be the angle of an anticlockwise rotation from the positive x-axis to a vector from p to p'. The directions of all movements captured in the trajectory is called the *direction information*, and is used heavily, both directly and indirectly, in a wide range of applications on trajectory data. We list some of them as follows.

- *Map Matching* [9]. Given a digital map of a road network and a trajectory of an object moving on the road network, the map matching problem is to locate the

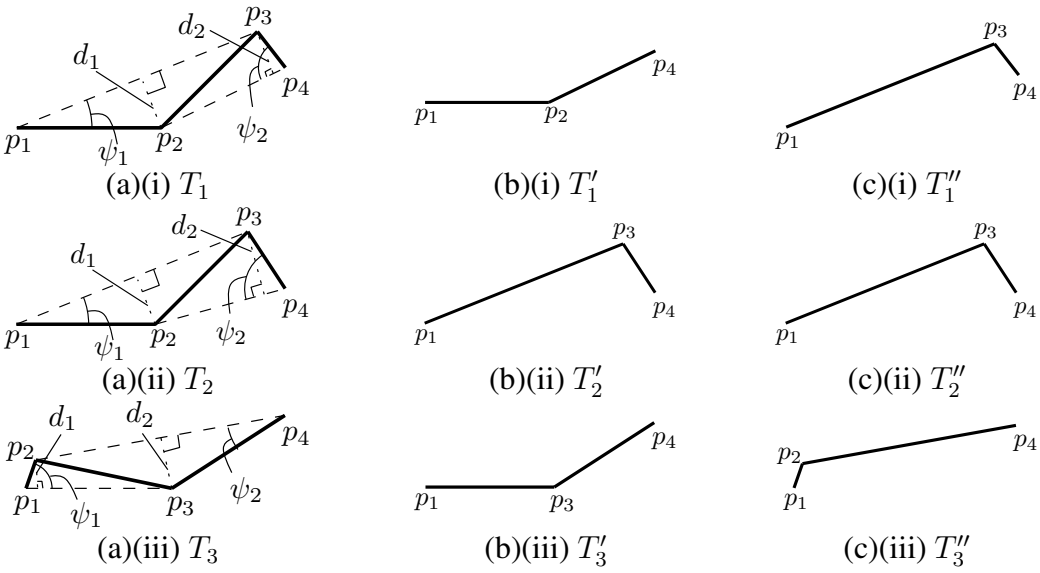


Figure 4.1: A motivating example (for DPTS)

trajectory on the digital map. Since each road segment in the road network has its own *orientation*, restricting the *directions* of the movements in the trajectory, the direction information plays an essential role in most map matching algorithms [9].

- *Knowledge Discovery on Trajectory Data*. As with other types of data, a rich set of knowledge discovery tasks has been proposed on the trajectory data [39]. Among them, many algorithms rely heavily on the direction information, which include [64, 50] for *Clustering*, [62] for *Outlier Detection* and [63] for *Classification*.

- *Direction-based Query Processing*. Sometimes, there are reasons to query trajectory information directly. One example is to find the trajectories moving within a direction range in a given time slot [10]. Another example is to find trajectories *similar* to a given trajectory, where the similarity measurement is based solely on moving direction [77].

In short, there are many situations in which direction preservation is important. Furthermore, as we show analytically in Section 4.3 and empirically in Section 4.6,

direction preservation is stronger than position preservation, in that a simplification that preserves direction information well can be shown to preserve position information also, within some reasonable bounds. However, the converse is not true: position-preserving simplifications can be very bad at direction preservation.

4.1.2 Direction-Preserving Trajectory Simplification (DPTS)

In this thesis, we propose a new trajectory simplification mechanism called *Direction-Preserving Trajectory Simplification* (DPTS) such that the direction information loss due to the simplification process is bounded. Within DPTS, we propose a *direction-based measurement* E_d, which is new and is defined to measure the *error* of a simplified trajectory in terms of the direction information. Let T be a trajectory and T' be a simplification of T. The *error* (or *simplification error*) of T' under E_d, denoted by $\epsilon(T')$, is equal to the maximum *angular difference* between the direction of the movement during each time period in T and the direction of the movement during the same time period in T'. Then, the problem we study within DPTS, called the *Min-Size* problem, is to simplify a given trajectory such that its *size* is minimized and its incurred simplification error (i.e., $\epsilon(T')$) is bounded by a given error tolerance ϵ_t where $\epsilon_t \in [0, \pi)$.

We use the *maximum* angular difference rather than the average angular difference to preserve better the shape of the trajectory. If we used the average one, we could still have a few segments that were completely off, resulting in the types of errors illustrated in Figure 4.1 for position-preserving techniques.

In this thesis, we study the properties of DPTS, develop multiple algorithms to solve the Min-Size problem, both exactly and approximately, and evaluate our algorithms experimentally. Specifically, we make the following contributions.

Contribution. First, we propose a novel notion of direction-preserving trajectory sim-

plification (DPTS), which favors a wide spectrum of applications on trajectory data. Second, we show that DPTS not only preserves direction information, but also preserves position information, thereby supporting a wide range of applications. Third, we adopt a common dynamic programming (DP) technique for Min-Size. Since it is not scalable, we propose a novel exact algorithm called *SP* for Min-Size. *SP* solves Min-Size by first constructing a graph based on the given trajectory, then computing a shortest path in this graph and finally returning the solution for Min-Size according to the shortest path found. The time complexity of *SP* is $O(C \cdot n^2)$, where C is usually a small constant ($C = 1$ if $\epsilon_t \leq \pi/2$). Fourth, since even an $O(n^2)$ running time is likely to be unacceptable for a large n, we propose a scalable approximate algorithm called *Intersect* which runs in $O(n)$ time. We show that *Intersect* provides a certain degree of the quality guarantee in terms of the size of the simplified trajectory returned, in spite of running so fast. Finally, we perform a careful experimental comparison of these algorithms and a baseline using real trajectory data. The baseline is developed by common sense modifications of standard trajectory simplification techniques to address the Min-Size problem.

The rest of this chapter is organized as follows. We define the Min-Size problem in Section 4.2 and review the related work in Section 4.3. We introduce the exact and approximate algorithms for Min-Size in Section 4.4 and Section 4.5, respectively. We give the empirical study in Section 4.6 and conclude this chapter in Section 4.7.

4.2 Problem Definition

A trajectory is represented by a sequence of n triplets in the form of $((x_1, y_1, t_1), (x_2, y_2, t_2), ..., (x_n, y_n, t_n))$, where (x_i, y_i) is the position in the 2D Euclidean space at time stamp t_i. We define *positions* $p_i = (x_i, y_i)$ for each $i \in [1, n]$. Then, T's *trace* is the sequence of ordered positions, i.e., $(p_1, p_2, ..., p_n)$.

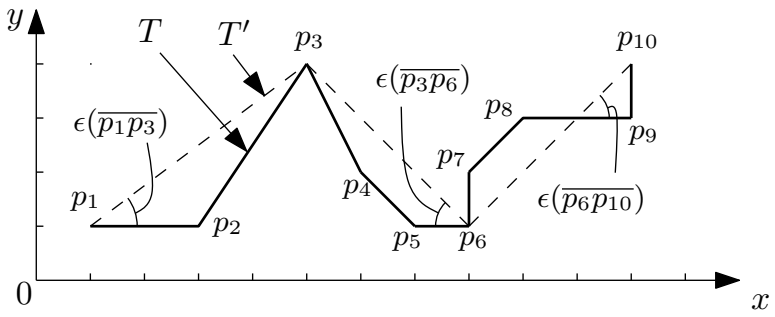


Figure 4.2: A running example (Min-Size)

Since the direction information of a trajectory is captured by its trace only, in the following, following existing studies, we focus on the trace part of the trajectory and use the terms "trajectory" and "trace" interchangeably. Thus, we simply denote T by $(p_1, p_2, ..., p_n)$ by keeping the position information only. The *size* of T, denoted by $|T|$, is defined to be the number of positions in T.

Consider a running example as shown in Figure 4.2. In this figure, the trajectory T is represented in the form of $(p_1, p_2, ..., p_{10})$. The size of this trajectory (i.e., $|T|$) is 10. The start position of T is p_1 and the end position of T is p_{10}.

The straight line linking two positions p_i and p_j in T where $1 \leq i < j \leq n$ is denoted by $\overline{p_i p_j}$. If p_i and p_j are *adjacent* in T (i.e., $j = i + 1$), then $\overline{p_i p_j}$ is said to be a *segment* in T. Thus, a trajectory could also be regarded as a sequence of $n - 1$ segments joining at $n - 2$ positions (in addition to unique start and end positions).

In Figure 4.2, the solid horizontal straight line connecting p_1 and p_2 is denoted by $\overline{p_1 p_2}$. Similarly, the dashed inclined straight line connecting p_1 and p_3 is denoted by $\overline{p_1 p_3}$. Here, $\overline{p_1 p_2}$ is a segment in T but $\overline{p_1 p_3}$ is not a segment in T. All segments in T are shown in solid lines in the figure. In T, there are 9 segments jointing at 8 positions, namely $p_2, p_3, ..., p_9$.

Trajectory T' is said to be a *simplification* of T if T' is of the form of $(p_{s_1}, p_{s_2}, ..., p_{s_m})$ where $m \leq n$ and $1 = s_1 < s_2 < ... < s_m = n$. Note that p_1 and p_n in T must be kept in any simplification of T. There are $m - 1$ segments in

T', and T' is using $m-1$ segments to represent T containing $n-1$ segments. For each $k \in [1, m)$, the segment $\overline{p_{s_k}p_{s_{k+1}}}$ in T' is used to *approximate* the sequence of segments between p_{s_k} and $p_{s_{k+1}}$ in T, namely $\overline{p_{s_k}p_{s_k+1}}$, $\overline{p_{s_k+1}p_{s_k+2}}$, ..., $\overline{p_{s_{k+1}-1}p_{s_{k+1}}}$. In other words, this sequence of segments in T is approximated by a single segment (i.e., $\overline{p_{s_k}p_{s_{k+1}}}$) in T' only.

Consider our running example. Let $T' = (p_1, p_3, p_6, p_{10})$. T' is a simplification of T in Figure 4.2. Here, $s_1 = 1, s_2 = 3, s_3 = 6$ and $s_4 = 10$. Note that the size of T' is 4. All segments in T' are shown in dashed lines in the figure. There are 3 segments in T'. In other words, T' is using 3 segments to approximate 9 segments in T. Consider segment $\overline{p_1p_3}$ in T'. It is used to approximate the sequence of segments between p_1 and p_3 in T, namely $\overline{p_1p_2}$ and $\overline{p_2p_3}$. In other words, $\overline{p_1p_2}$ and $\overline{p_2p_3}$ are approximated by a single segment $\overline{p_1p_3}$. Similarly, trajectory $T'' = (p_1, p_{10})$ is also a simplification of T, which uses only one segment (i.e., $\overline{p_1p_{10}}$) to approximate the whole trajectory T.

Direction-based Error Measurement E_d. Given a segment $\overline{p_ip_{i+1}}$ in T, the *direction* of $\overline{p_ip_{i+1}}$, denoted by $\theta(\overline{p_ip_{i+1}})$, is defined to be the angle of an *anticlockwise* rotation from the positive x-axis to a vector from p_i to p_{i+1}. Thus, each direction falls in $[0, 2\pi)$. Consider our running example (Figure 4.2). $\theta(\overline{p_7p_8})$ is $\pi/4 (= 0.788)$ radian and $\theta(\overline{p_4p_5})$ is $7\pi/4 (= 5.498)$ radian, as illustrated in Figure 4.3(a). It is easy to verify that $\theta(\overline{p_1p_2})$ is equal to 0 radian, $\theta(\overline{p_2p_3})$ is equal to 0.983 radian ($= \tan^{-1} 3/2$) and $\theta(\overline{p_1p_3})$ is equal to 0.644 radian ($= \tan^{-1} 3/4$).

The *angular difference* between two directions θ_1 and θ_2, denoted by $\triangle(\theta_1, \theta_2)$, is defined to be the minimum of the angle of the anticlockwise rotation from θ_1 to θ_2 and that from θ_2 to θ_1, i.e.,

$$\triangle(\theta_1, \theta_2) = \min\{|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|\} \tag{4.1}$$

For illustration, Figure 4.3(b) shows the case where $\triangle(\theta_1, \theta_2) = |\theta_1 - \theta_2|$ and Fig-

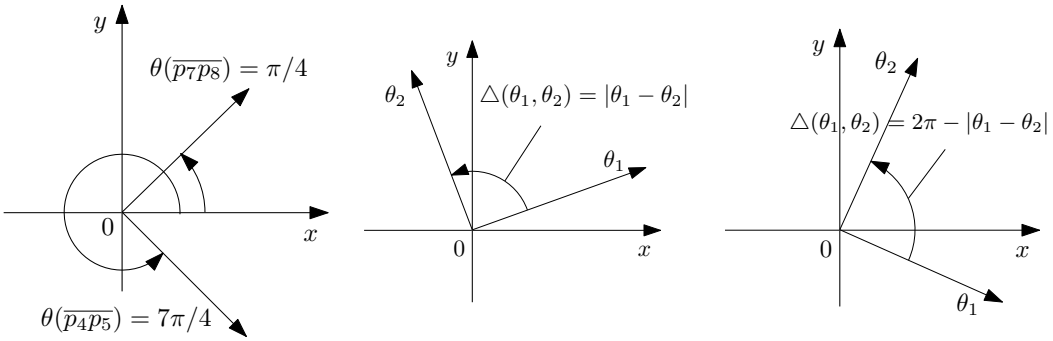


Figure 4.3: Examples illustrating the definition of "direction" and "angular difference"

ure 4.3(c) shows the case where $\triangle(\theta_1, \theta_2) = 2\pi - |\theta_1 - \theta_2|$. Note that the angular difference between any two directions falls in $[0, \pi]$.

Consider our running example. The angular difference between $\theta(\overline{p_1p_2})$ and $\theta(\overline{p_1p_3})$ is $|0 - 0.644| = 0.644$ and that between $\theta(\overline{p_2p_3})$ and $\theta(\overline{p_1p_3})$ is $|0.983 - 0.644| = 0.339$.

Let $T' = (p_{s_1}, p_{s_2}, ..., p_{s_m})$ be a simplification of T The *simplification error* of T' under E_d, denoted by $\epsilon(T')$, is defined as follows. Given a segment $\overline{p_{s_k}p_{s_{k+1}}}$ in T', the *simplification error* of $\overline{p_{s_k}p_{s_{k+1}}}$, denoted by $\epsilon(\overline{p_{s_k}p_{s_{k+1}}})$, is defined to be the *greatest angular difference* between the direction of $\overline{p_{s_k}p_{s_{k+1}}}$ and the direction of a segment in T approximated by $\overline{p_{s_k}p_{s_{k+1}}}$. That is,

$$\epsilon(\overline{p_{s_k}p_{s_{k+1}}}) = \max_{s_k \le h < s_{k+1}} \triangle(\theta(\overline{p_{s_k}p_{s_{k+1}}}), \theta(\overline{p_hp_{h+1}}))$$

Then, the *simplification error* of T' under E_d is defined to be the *greatest* simplification error of a segment in T'. That is,

$$\epsilon(T') = \max_{1 \le k < m} \epsilon(\overline{p_{s_k}p_{s_{k+1}}}) \tag{4.2}$$

Consider back our running example (Figure 4.2). Each segment in T' has its simplification error. Consider the first segment $\overline{p_1p_3}$ in T' which approximates two segments in T, namely $\overline{p_1p_2}$ and $\overline{p_2p_3}$. Recall that $\triangle(\theta(\overline{p_1p_3}), \theta(\overline{p_1p_2})) = 0.644$ and

$\triangle(\theta(\overline{p_1p_3}), \theta(\overline{p_2p_3})) = 0.339$. Thus, the simplification error of $\overline{p_1p_3}$ (i.e., $\epsilon(\overline{p_1p_3})$) is equal to $\max\{0.644, 0.339\} = 0.644$. Similarly, we compute the simplification errors of the second segment $\overline{p_3p_6}$ and the third segment $\overline{p_6p_{10}}$ in T' which are both equal to 0.785. Thus, the simplification error of T' in this example is equal to $\max\{0.644, 0.785, 0.785\} = 0.785$.

In the following, when we write $\epsilon(\overline{p_ip_j})$ $(0 \leq i < j \leq n)$, we mean the simplification error of $\overline{p_ip_j}$ when it is used to approximate the line segments between p_i and p_j in T.

Problem Statement of The Min-Size Problem. Let T be a trajectory and ϵ_t be the error tolerance ($\epsilon_t < \pi$). Trajectory T' is said to be an ϵ_t-*simplification* of T if T' is a simplification of T and $\epsilon(T') \leq \epsilon_t$.

The DPTS problem is formalized as follows.

Problem 2 (Min-Size) *Given a trajectory T and an error tolerance ϵ_t,* ***the Min-Size problem*** *is to find the ϵ_t-simplification of T with the smallest size.* □

Consider our running example. Suppose that we set ϵ_t to 0.785. T' is an ϵ_t-simplification of T since $\epsilon(T') = 0.785 \leq \epsilon_t$. In fact, T' is the ϵ_t-simplification of T with the smallest size (which involves only four remaining positions).

4.3 Related Work

We describe how DPTS relates to existing error measurements (Section 4.3.1) and trajectory simplification techniques (Section 4.3.2).

4.3.1 Existing Error Measurements

In this section, we show that the direction-preserving simplified trajectories give error guarantees in position-related properties, such as length and speed. However, the reverse is not true. That is, the position-preserving simplified trajectories [72, 74, 79, 56, 31] do not give any error guarantee on the direction information.

Before we give our claims/properties, we review some representative existing error measurements.

Existing Error Measurements

Let $T = (p_1, p_2, ..., p_n)$ be a trajectory and $T' = (p_{s_1}, p_{s_2}, ..., p_{s_m})$ be a simplification of T ($m \leq n$). Several *position-based* measurements for evaluating the "simplification error" of T' have been defined in the literature. These measurements for T' are usually defined to be a *distance measure* which takes T and T' as input. For each position p_h of T at the time stamp equal to t_h where $1 \leq h \leq n$, the distance measure defines the *estimated position* of p_h, denoted by p'_h, in T' based on some criteria. Let $d(\cdot, \cdot)$ be the Euclidean distance between two given points. Thus, the distance measure is defined to be $\max_{h \in [1,n]} d(p_h, p'_h)$. Since different distance measures have different definitions on estimated positions, in the following, we focus on how to define estimated positions for some representative distance measures.

(1) *Closest Euclidean Distance:* With this distance measure, p'_h is defined to be the location on the segment $\overline{p_{s_k} p_{s_{k+1}}}$ of T' with $s_k \leq h \leq s_{k+1}$, which has the smallest Euclidean distance from p_h. We define a mapping function $M_{\mathcal{C}}$ which maps p_h and T' to p'_h for this distance measure. That is, $p'_h = M_{\mathcal{C}}(p_h, T')$.

(2) *Synchronous Euclidean Distance:* Under this distance measure, p'_h can be calculated with the following steps. The first step is to find the segment $\overline{p_{s_k} p_{s_{k+1}}}$ of T' with

$s_k \leq h \leq s_{k+1}$. The second step is to find a point along the line passing through two points, namely $(x_{s_k}, y_{s_k}, t_{s_k})$ and $(x_{s_{k+1}}, y_{s_{k+1}}, t_{s_{k+1}})$, in a three-dimensional space such that the third dimensional value (representing the time dimension) of this point is t_h. Then, p'_h is set to be the first two-dimensional values of this point. Similarly, we define a mapping function $M_{\mathcal{S}}$ which maps both p_h and T' to p'_h. That is, $p'_h = M_{\mathcal{S}}(p_h, T')$.

However, all of them adopt *position-based* distances instead of the *direction-based* distance studied in this thesis. In the next section, we show that they do not give any error guarantee on the direction information.

Theoretical Properties

The *length* between two positions p_i and p_j wrt T ($i < j$), denoted by $len(p_i, p_j|T)$, is defined to be the length of the trace from p_i to p_j in T. That is,

$$len(p_i, p_j|T) = \sum_{k=i}^{j-1} d(p_k, p_{k+1})$$

The (average) *speed* between two positions p_i and p_j wrt T ($i < j$), denoted by $speed(p_i, p_j|T)$, is equal to $len(p_i, p_j|T)/(t_j - t_i)$, where t_i (t_j) is the time stamp corresponding to p_i (p_j).

Interestingly, DPTS gives error guarantees on the length and speed information. Consider that T is a trajectory and T' is an ϵ_t-simplification of T with $\epsilon_t < \pi/2$. For any two adjacent positions p_i and p_{i+1} in T where $i \in [1, n)$, both the length and the speed between the two corresponding *estimated* positions wrt T' are theoretically bounded. The estimated positions are determined by a mapping function. In the remaining of this chapter, if no distance measure is specified explicitly, we assume that the Closest Euclidean Distance (i.e., $M_{\mathcal{C}}(\cdot, \cdot)$) is used by default. The results based on the other mapping function (i.e., $M_{\mathcal{S}}(\cdot, \cdot)$) can be found in Section A.2 in the Appendix.

Lemma 4.3.1 (Bounded Length/Speed) *Let T be a trajectory and T' be an ϵ_t-simplification of T with $\epsilon_t < \pi/2$. For any two adjacent positions p_i and p_{i+1} in T where $i \in [1, n)$,*

$$\cos(\epsilon_t) \leq \frac{len(p'_i, p'_{i+1}|T')}{len(p_i, p_{i+1}|T)} \leq 1 \; and \; \cos(\epsilon_t) \leq \frac{speed(p'_i, p'_{i+1}|T')}{speed(p_i, p_{i+1}|T)} \leq 1$$

where $p'_i = M_{\mathcal{C}}(p_i, T')$ and $p'_{i+1} = M_{\mathcal{C}}(p_{i+1}, T')$. □

Proof. Let $\overline{p_{s_k} p_{s_{k+1}}}$ be the segment of T' such that p_{s_k} is the last position with $s_k \leq i$ and $p_{s_{k+1}}$ is the first position with $s_{k+1} \geq i + 1$. Consider Figure 4.4(a) for illustration. Since $\epsilon_t < \pi/2$, we can verify that p'_i and p'_{i+1} are located along $\overline{p_{s_k} p_{s_{k+1}}}$.

Let ψ_i be the angle formed by the two lines that pass through $\overline{p_i p_{i+1}}$ and $\overline{p_{s_k} p_{s_{k+1}}}$. Thus,

$$len(p'_i, p'_{i+1}|T') = d(p'_i, p'_{i+1}) = \cos(\psi_i) \cdot d(p_i, p_{i+1})$$

$$= \cos(\psi_i) \cdot len(p_i, p_{i+1}|T) \geq \cos(\epsilon_t) \cdot len(p_i, p_{i+1}|T)$$

which implies that $\cos(\epsilon_t) \leq \frac{len(p'_i, p'_{i+1}|T')}{len(p_i, p_{i+1}|T)} \leq 1$. Besides, since $speed(p'_i, p'_{i+1}|T') = len(p'_i, p'_{i+1}|T')/(t_{i+1} - t_i)$ and $speed(p_i, p_{i+1}|T) = len(p_i, p_{i+1}|T)/(t_{i+1} - t_i)$, we know $\cos(\epsilon_t) \leq \frac{speed(p'_i, p'_{i+1}|T')}{speed(p_i, p_{i+1}|T)} \leq 1$. ■

Interestingly, DPTS gives an error bound on the position information (in addition to the length/speed information).

Lemma 4.3.2 (Bounded Position Error) *Let T be a trajectory and T' be an ϵ_t-simplification of T with $\epsilon_t < \pi/2$. For each position p_i in T where $i \in [1, n]$, we have*

$$d(p_i, p'_i) \leq 0.5 \cdot \tan(\epsilon_t) \cdot L_{max}$$

where $p'_i = M_{\mathcal{C}}(p_i, T')$ and $L_{max} = \max_{k \in [1,m)} len(p_{s_k}, p_{s_{k+1}}|T')$. □

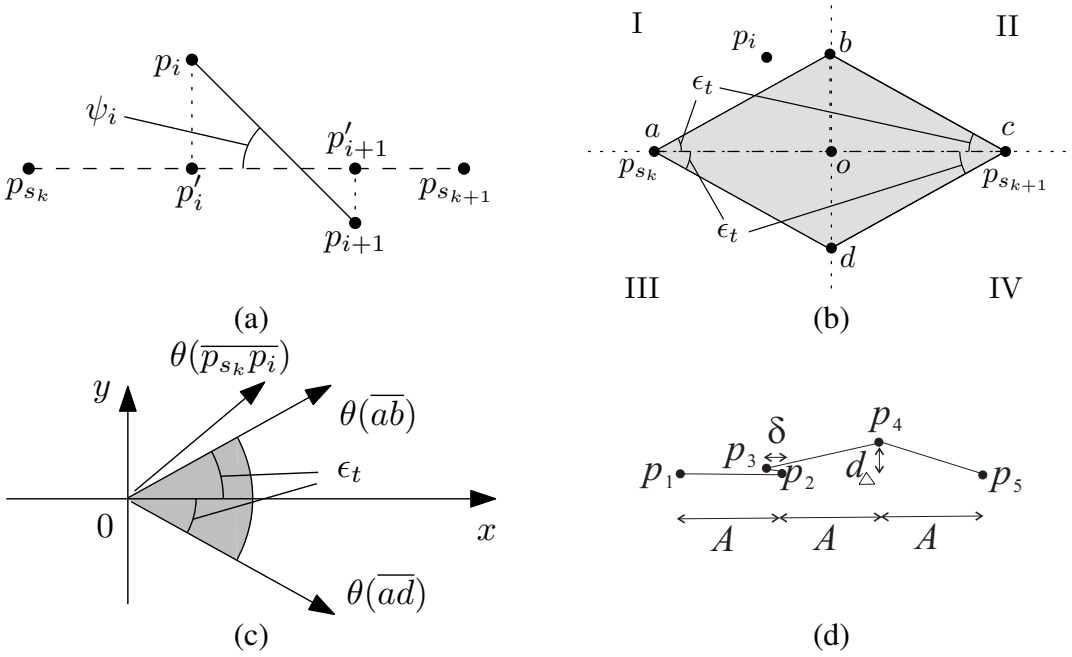


Figure 4.4: Proofs of Lemma 4.3.1, Lemma 4.3.2 and Lemma 4.3.3

Proof. Let $\overline{p_{s_k}p_{s_{k+1}}}$ be the segment of T' such that p_{s_k} is the last position with $p_{s_k} \leq i$ and $p_{s_{k+1}}$ is the first position with $p_{s_{k+1}} \geq i$. We construct a rhombus $\diamond_{abcd}$ with four corners, namely a, b, c and d, such that a is at p_{s_k}, c is at $p_{s_{k+1}}$ and the angle between $\overline{ab}$ ($\overline{cb}$) and $\overline{ad}$ ($\overline{cd}$) is equal to $2 \cdot \epsilon_t$. Consider Figure 4.4(b) for illustration where $\diamond_{abcd}$ is indicated by the shaded area. We claim that p_i is inside $\diamond_{abcd}$ which we prove by contradiction.

Assume that p_i is outside $\diamond_{abcd}$. We partition the plane into 4 parts with the two lines that pass through $\overline{ac}$ and $\overline{bd}$ as indicated by I, II, III and IV in Figure 4.4(b), where o is the intersection of the two lines. We consider 4 cases of which partition p_i is in. Without loss of generality, suppose p_i falls in part I.

Since p_i is outside $\diamond_{abcd}$, we know $\theta(\overline{p_{s_k}, p_i})$ falls outside range $[\theta(\overline{ad}), \theta(\overline{ab})]$. For illustration, consider Figure 4.4(c).

Consider the segments between p_{s_k} and p_i in T. For each such segment $\overline{p_h p_{h+1}}$ ($s_k \leq h < i$), we denote by $\overrightarrow{p_h p_{h+1}}$ the vector from p_h to p_{h+1}. We know that the direction of each such vector falls in range $[\theta(\overline{ad}), \theta(\overline{ab})]$ since otherwise $\epsilon(T') > \epsilon_t$. As a result, we know $\theta(\overline{p_{s_k}p_i})$ falls in range $[\theta(\overline{ad}), \theta(\overline{ab})]$ since $\overrightarrow{p_{s_k}p_i} = \sum_{s_k \leq h < i} \overrightarrow{p_h p_{h+1}}$.

This, however, contradicts the fact that $\theta(\overline{p_{s_k}p_i})$ falls outside range $[\theta(\overline{ad}), \theta(\overline{ab})]$.

Thus, we know that p_i falls in $\diamond_{abcd}$. Therefore, we have

$$d(p_i, p_i') \leq d(b, o) = \tan(\epsilon_t) \cdot d(a, o) = 0.5 \cdot \tan(\epsilon_t) \cdot d(a, c)$$
$$= 0.5 \cdot \tan(\epsilon_t) \cdot len(p_{s_k}, p_{s_{k+1}}|T') \leq 0.5 \cdot \tan(\epsilon_t) \cdot L_{max}$$

which finishes the proof. ■

Next, we show that existing position-preserving simplified trajectories do not have bounds on the direction information.

Lemma 4.3.3 (Unbounded Direction Error) *Let T be a trajectory and T' be a (direction-based) ϵ_t-simplification of T with $\epsilon_t < \pi/2$. Let $T_{\mathcal{C}}$ be a (position-based) simplified trajectory of T such that $|T_{\mathcal{C}}| = |T'|$ and the error of $T_{\mathcal{C}}$ under the Closest Euclidean Distance is minimized. There exists a dataset such that $\epsilon(T_{\mathcal{C}}) \approx \pi$ and $\epsilon(T') \approx 0$.* □

Proof. We prove by constructing a problem instance as shown in Figure 4.4(d). $T = (p_1, p_2, p_3, p_4, p_5)$ is a trajectory, where p_1, p_2, p_3 and p_5 are located at a horizontal line and p_4 has its perpendicular distance from this line equal to a small real number $d_{\triangle}$. Besides, $d(p_2, p_3) = \delta$ where $\delta << d_{\triangle}$.

Suppose that we can only keep 4 positions in the simplified trajectory. In other words, we have to remove 1 position from the 5 positions. If we consider preserving the direction information, p_4 will be removed and thus $T' = (p_1, p_2, p_3, p_5)$. Thus, $\epsilon(T') = \epsilon(\overline{p_3p_5}) \approx 0$. If we consider preserving the position information, p_2 will be removed and thus $T_{\mathcal{C}} = (p_1, p_3, p_4, p_5)$. Hence, $\epsilon(T_{\mathcal{C}}) = \epsilon(\overline{p_1p_3}) = \triangle(\theta(\overline{p_1p_3}), \theta(\overline{p_2p_3})) \approx \pi$. ■

4.3.2 Existing Trajectory Simplification

Many trajectory simplification techniques have been proposed. We categorize them by the main idea employed in the algorithm as follows. They are *Split* [72, 31], *Merge* [79, 74], *Greedy* [56, 72] and *Dead-Reckoning* [60]. *Split* is an approach which finds a position in a given trajectory, according to the *heuristic* value of the position, to *split* the whole trajectory into two sub-trajectories and continues the process iteratively on each of the split sub-trajectories which cannot be approximated by a line segment connecting its start position and its end position. *Merge* is an approach which finds two adjacent segments in a given trajectory, according to the *heuristic* value computed from these two adjacent segments, discards the position p bridging these two segments, and create a segment connecting the non-bridging end position of one segment and the non-bridging end position of the other segment. It continues the process iteratively until discarding any position p violates the error tolerance. *Greedy* is an approach which finds a sequence of the greatest number of consecutive segments to be discarded and create a segment connecting the two end positions of this sequence iteratively until discarding any sequence of 2 consecutive segments violates the tolerance constraint. *Dead-Reckoning* is an online algorithm which reads each position sequentially and determines whether this position is discarded or not according to a *heuristic* criterion.

The aforementioned ideas of *Split*, *Merge* and *Greedy* can be adapted to the Min-Size problem. The only change is to change the error measurement to our simplification error (Equation (4.2)). However, they have their drawbacks. First, they cannot return *optimal* solutions. Second, as shown in our experiments, they are not efficient compared with our proposed *Intersect* algorithm. Details of the adaptation can be found in Section B.1 of the Appendix.

Other related studies include [12] which studies the error bounds of several queries on the simplified trajectories with bounded simplification errors mainly measured by

the position information, [23] which studies the trajectory simplification problem with the consideration of the shape and also the semantic meanings of the trajectory, [22] which introduces a multi-resolution *polygonal curve approximation* (also called *line simplification*) algorithm for trajectory simplification, and [55] which studies the trajectory simplification problem where the trajectories are constrained to a road network. None of these studies pay attention to the direction information for trajectory simplification.

4.4 Exact Algorithm

A naive solution for the Min-Size problem is to traverse each possible simplification of T with its simplification error at most ϵ_t and then to pick the one with the smallest size. Since the number of all possible simplifications of a trajectory T is $2^{|T|-2}$, this solution is not feasible in practice. Alternatively, one may adopt a common dynamic programming (DP) technique for the Min-Size problem. Unfortunately, the time complexity of this technique is cubic. For the sake of space, we include this DP algorithm in Section B.2 in the Appendix. Instead, we propose a method called *SP* which is much faster and scalable.

Algorithm *SP* involves the following three steps.

- **Step 1 (Graph Construction):** It first constructs a graph based on the given trajectory.
- **Step 2 (Shortest Path Finding):** It computes a shortest path in this graph.
- **Step 3 (Solution Generation):** It finally returns the solution for Min-Size according to the shortest path found.

In Step 1, it constructs a graph wrt ϵ_t, denoted by $G_{\epsilon_t}(V, E)$, as follows. For each

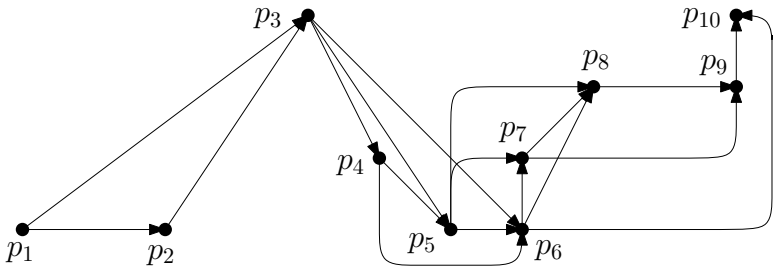


Figure 4.5: The graph G_{ϵ_t} constructed based on the running example when ϵ_t is set to be $\pi/4 = 0.785$

position p_i of T where $1 \leq i \leq n$, it creates a vertex for p_i in V. For each pair of two positions (p_i, p_j) where $i < j$, it creates an edge (p_i, p_j) in E if $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$.

In Step 2, it finds the shortest path from p_1 to p_n in G_{ϵ_t} by a shortest path algorithm (e.g., a BFS search). Here, the *length* of a path is defined to be the number of edges involved along the path.

In Step 3, it generates the solution for Min-Size according to the shortest path found. Note that all vertices involved in this shortest path correspond to all positions in the ϵ_t-simplification of T with the smallest size. Thus, if the ordering of the positions (or vertices) involved in the shortest path is "p_{s_1}-p_{s_2}-...-p_{s_m}", it returns the solution T' as $(p_{s_1}, p_{s_2}, ..., p_{s_m})$.

Example 6 (Algorithm SP) Consider our running example in Figure 4.2. Suppose that $\epsilon_t = 0.785$. In Step 1 of the *SP* algorithm, we can construct graph G_{ϵ_t} accordingly as shown in Figure 4.5. In this figure, we construct a vertex for each position in T. Besides, for each pair of positions p_i and p_j where $i < j$, if $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$, we create an edge (p_i, p_j). Note that $\epsilon(\overline{p_i p_{i+1}}) = 0$ for each $i \in [1, n-1]$.

In Step 2, we can find the shortest path in this graph. It is easy to verify that p_1-p_3-p_6-p_{10} is the shortest path. Finally, in Step 3, we construct the solution of Min-Size as (p_1, p_3, p_6, p_{10}). □

Let us analyze the time complexity of a *straightforward* implementation of algorithm *SP*. For Step 1, a straightforward solution for constructing G_{ϵ_t} is to try all

Algorithm 7 The *SP* algorithm with the practical enhancement

Require: A trajectory $T = (p_1, p_2, ..., p_n)$ and the error tolerance ϵ_t

```
1: H_0 ← {p_1}; U ← {p_2, p_3, ..., p_n}; l ← 1
2: while true do
3:    H_l ← ∅
4:    //process the positions in H_{l-1} and U in a reversed order
5:    for each p_i in H_{l-1} and each p_j in U where i < j do
6:       if ε(p_i p_j) ≤ ε_t then
7:          if p_j = p_n then
8:             return the trajectory corresponding to the shortest path from p_1 to p_n
9:          U ← U\{p_j}; H_l ← H_l ∪ {p_j}
10:   l ← l + 1
```

possible pairs of (p_i, p_j) where $1 \leq i < j \leq n$ and to check whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$. Since there are $O(n^2)$ possible such pairs and the checking cost for each pair is $O(n)$, the time complexity of Step 1 is $O(n^3)$. For Step 2, a simple BFS could be adopted to find the shortest path from p_1 to p_n in G_{ϵ_t}, which takes $O(|V| + |E|)$ time. Since $|V| = O(n)$ and $|E| = O(n^2)$, we know that the cost of BFS is $O(n^2)$. Step 3 which returns the solution takes $O(n)$ time. As we can see, the time complexity of Step 1 (i.e., the graph construction) dominates those of Step 2 and Step 3. Thus, the overall time complexity of a straightforward implementation of algorithm *SP* is $O(n^3)$. Besides, the space complexity of *SP* is simply $O(|V| + |E|)$ which corresponds to the space cost of maintaining G_{ϵ_t}.

In the following, we propose two kinds of enhancement techniques in order to improve the efficiency of our *SP* algorithm. The first one is called the *practical enhancement* (Section 4.4.1) which is to improve the performance of the algorithm in a practical way. The second one is called the *complexity improvement* (Section 4.4.2) which is to improve the theoretical time complexity of the algorithm from cubic to quadratic with some properties.

4.4.1 Practical Enhancement

The practical enhancement is to construct G_{ϵ_t} (in Step 1) *on the demand* of the BFS procedure used in Step 2. Specifically, the straightforward implementation has

to materialize *all* possible edges in G_{ϵ_t} in Step 1 and then perform the BFS procedure in Step 2. Here, the enhancement constructs only *some* of the edges in G_{ϵ_t} which are needed in the BFS procedure. Since some other edges need not be constructed, the space consumption can be reduced and some computations can be also saved.

Given a position p in T and a non-negative integer l, p is said to be an *l-length position* if the length of the shortest path from p_1 to p in G_{ϵ_t} is equal to l. Given a non-negative integer l, we define the *l-length unique set*, denoted by H_l, to be the set of all l-length positions in T. For example, the 0-length unique set H_0 is $\{p_1\}$. Consider the BFS procedure starting from p_1 on G_{ϵ_t}. It first retrieves the set of positions which are the out-neighbors of p_1. This set corresponds to H_1. Then, starting from each position p in H_1, it retrieves the set of positions which are the out-neighbors of p and *have not been retrieved before*. This set corresponds to H_2. The above process continues from H_2 in the same manner until p_n is retrieved.

In view of the above discussion, we design our *SP* algorithm with this enhancement as follows. We maintain the l-length unique sets H_l ($l = 0, 1, 2, ...$) which store the positions retrieved by the BFS procedure and U for storing the remaining positions that have not been retrieved by the BFS procedure. We initialize H_0 to be $\{p_1\}$ and U to be $\{p_2, p_3, ..., p_n\}$. We then compute H_l based on H_{l-1} for $l = 1, 2, ...$ iteratively as follows. We start from each position p_i in H_{l-1}. For each position p_j in U, we compute $\epsilon(\overline{p_i p_j})$. If $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$, we further check whether p_j is p_n. If so, we stop the process since the shortest path from p_1 to p_n has been found; otherwise, we exclude p_j from U and include it in H_l. Besides, when processing the positions in H_{l-1} and U, we impose a *reversed* order, which corresponds to $p_n, p_{n-1}, ..., p_1$. The intuition is that we expect that p_n could be retrieved earlier in this way. We present our enhanced *SP* algorithm in Algorithm 7.

Complexity Analysis. The worst-case time complexity of the *SP* algorithm with the

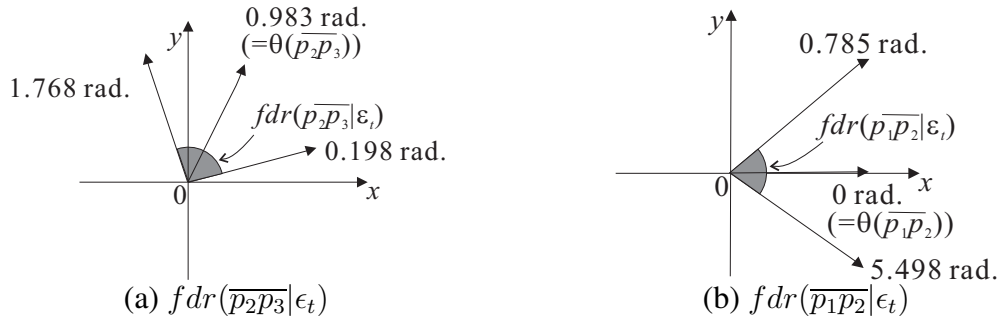


Figure 4.6: Illustration of $fdr(\overline{p_2p_3}|\epsilon_t)$ and $fdr(\overline{p_1p_2}|\epsilon_t)$

practical enhancement keeps the same as that of the straightforward implementation, i.e., it is still $O(n^3)$. However, in practice, with the practical enhancement, the *SP* algorithm is more efficient since some computations of $\epsilon(\overline{p_ip_j})$ are avoided, and it is also more scalable since there is no need to materialize G_{ϵ_t}. The space complexity of *SP* with the practical enhancement is simply $O(n)$ since it maintains each position once and does not materialize G_{ϵ_t} explicitly.

4.4.2 Complexity Improvement

The complexity improvement is to improve the time complexity of our *SP* algorithm from cubic to quadratic by using some properties for our algorithm. In this section, we focus on the complexity improvement based on the straightforward implementation for illustration. In Section 4.4.3, we describe how this complexity improvement can be incorporated with the practical enhancement.

As can be noticed, the cost of the straightforward implementation is dominated by the construction of graph G_{ϵ_t}. In this section, we propose a technique to reduce the cost of constructing the graph from $O(n^3)$ to $O(C \cdot n^2)$ time, where C is shown to be a small constant in most cases. The major idea of such an improvement is to reduce the time complexity of checking whether $\epsilon(\overline{p_ip_j}) \leq \epsilon_t$ (in the graph construction step) from $O(n)$ to $O(C)$ by utilizing a new concept called "*feasible direction range*". Before we present the main idea, we first introduce some related concepts.

Given two angles θ_1 and θ_2 in [0, 2π), an *angular range*, represented in the form of $[\theta_1, \theta_2]$, is defined to be a set of all possible angles of a vector originated from the origin when it is rotated anti-clockwise from θ_1 to θ_2. For example, the shaded part in Figure 4.6(a) shows the angular range of $[0.198, 1.768]$, and the shaded part in Figure 4.6(b) shows the angular range of $[5.498, 0.785]$. Since the direction of $\overline{p_2p_3}$ is 0.983 radian, we say that its direction is in $[0.198, 1.768]$ but not in $[5.498, 0.785]$. Similarly, since the direction of $\overline{p_1p_2}$ is 0 radian, we say that its direction is in $[5.498, 0.785]$ but not in $[0.198, 1.768]$.

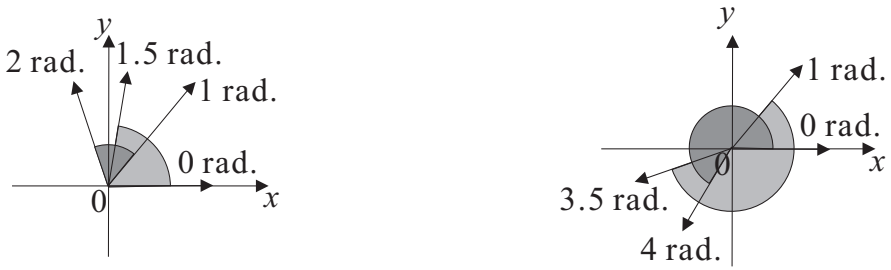
Definition 4.4.1 (feasible direction range) *Given a segment $\overline{p_hp_{h+1}}$ ($1 \leq h < n$) in T, the **feasible direction range** of $\overline{p_hp_{h+1}}$ wrt ϵ_t, denoted by $fdr(\overline{p_hp_{h+1}}|\epsilon_t)$, is defined to be the angular range in the form of $[\theta_1, \theta_2]$ with $\theta_1 = [(\theta(\overline{p_hp_{h+1}}) - \epsilon_t) \mod 2\pi]$ and $\theta_2 = [(\theta(\overline{p_hp_{h+1}}) + \epsilon_t) \mod 2\pi]$.* □

The feasible direction range of $\overline{p_hp_{h+1}}$ wrt ϵ_t corresponds to a set of all possible directions each of which has its angular difference from $\overline{p_hp_{h+1}}$ at most ϵ_t. We can write fdr as follows.

$$fdr(\overline{p_hp_{h+1}}|\epsilon_t) = [\theta(\overline{p_hp_{h+1}}) - \epsilon_t, \theta(\overline{p_hp_{h+1}}) + \epsilon_t] \mod 2\pi \tag{4.3}$$

Consider our running example. Suppose that ϵ_t is set to 0.785. Since the direction of $\overline{p_2p_3}$ is 0.983 radian, $fdr(\overline{p_2p_3}|\epsilon_t) = ([0.983 - 0.785, 0.983 + 0.785] \mod 2\pi) = [0.198, 1.768]$ (See Figure 4.6(a)). Similarly, since the direction of $\overline{p_1p_2}$ is 0 radian, $fdr(\overline{p_1p_2}|\epsilon_t) = ([0 - 0.785, 0 + 0.785] \mod 2\pi) = [5.498, 0.785]$ (See Figure 4.6(b)).

We denote by $T[i, j]$ the sub-trajectory of T that is between position p_i and position p_j ($1 \leq i < j \leq n$), i.e., $T[i, j] = (p_i, p_{i+1}, ..., p_j)$. We define the *feasible direction range* of a sub-trajectory $T[i, j]$ wrt ϵ_t, denoted by $fdr(T[i, j]|\epsilon_t)$, to be the *intersection*



(a) $[0, 1.5] \cap [1.0, 2.0]$ (b) $[0, 4.0] \cap [3.5, 1.0]$

Figure 4.7: Illustration of intersection operations between two angular ranges

of the fdr's of the segments in $T[i, j]$. That is,

$$fdr(T[i,j]|\epsilon_t) = \cap_{i \leq h < j} fdr(\overline{p_h p_{h+1}}|\epsilon_t) \tag{4.4}$$

In our running example, $fdr(T[1,3]|\epsilon_r)$ is equal to $\cap_{1 \leq h < 3} fdr(\overline{p_h p_{h+1}}|\epsilon_t) = fdr(\overline{p_1 p_2}|\epsilon_t) \cap fdr(\overline{p_2 p_3}|\epsilon_t) = [5.498, 0.785] \cap [0.198, 1.768] = [0.198, 0.785]$. Figures 4.6(a) and (b) illustrate this scenario.

In the following, we simply write $fdr(\overline{p_h p_{h+1}})$ ($fdr(T[i,j])$) for $fdr(\overline{p_h p_{h+1}}|\epsilon_t)$ ($fdr(T[i,j]|\epsilon_t)$) if the context of ϵ_t is clear.

In some cases, an intersection between an angular range and another angular range results in a *single* angular range. In some other cases, this intersection operation results in *multiple* disjoint angular ranges. To illustrate, an intersection between $[0, 1.5]$ and $[1.0, 2.0]$ results in a single angular range $[1, 1.5]$ (as shown in Figure 4.7(a)). An intersection between $[0, 4.0]$ and $[3.5, 1.0]$ results in two angular ranges $[3.5, 4.0]$ and $[0, 1.0]$ (as shown in Figure 4.7(b)).

Thus, from Equation (4.4), since $fdr(T[i,j])$ involves multiple intersection operations of angular ranges, it may consist of multiple disjoint angular ranges. We denote by $||fdr(T[i,j])||$ the number of disjoint angular ranges in $fdr(T[i,j])$.

With the concept of "feasible direction range", we are now ready to describe how we can check whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ efficiently.

Lemma 4.4.1 *Let $T = (p_1, p_2, ..., p_n)$ be a trajectory. $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ iff $\theta(\overline{p_i p_j})$ is in $fdr(T[i, j])$.* □

Proof. "if": Assume that $\theta(\overline{p_i p_j})$ is in $fdr(T[i, j])$. It follows that $\theta(\overline{p_i p_j})$ is in $fdr(\overline{p_h p_{h+1}})$ and thus $\triangle(\theta(\overline{p_i p_j}), \theta(\overline{p_h p_{h+1}})) \leq \epsilon_t$ for $i \leq h < j$. Therefore, $\epsilon(\overline{p_i p_j}) = \max\{\triangle(\theta(\overline{p_h p_{h+1}}), \theta(\overline{p_i p_j}))|i \leq h < j\} \leq \epsilon_t$.

"only-if": this direction could be verified similarly. ■

Lemma 4.4.1 suggests that checking whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ is equivalent to checking whether $\theta(\overline{p_i p_j})$ is in $fdr(T[i, j])$. Suppose that $fdr(T[i, j])$ has been computed. Then, checking whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ takes $O(||fdr(T[i, j])||)$ only (compared with $O(n)$ in the straightforward implementation).

Note that in some cases, $\epsilon(\overline{p_i p_j}) > \epsilon_t$ but $\epsilon(\overline{p_i p_{j+k}}) \leq \epsilon_t$ where $j > i > 0$ and $k > 0$. By Lemma 4.4.1, we know that $\theta(\overline{p_i p_j})$ is not in $fdr(T[i, j])$ but $\theta(\overline{p_i p_{j+k}})$ is in $fdr(T[i, j + k])$. To illustrate, in our running example (Figure 4.2), if we set $\epsilon_t = \pi/4$, then $\epsilon(\overline{p_6 p_9}) > \epsilon_t$ but $\epsilon(\overline{p_6 p_{10}}) \leq \epsilon_t$. By Lemma 4.4.1, $\theta(\overline{p_6 p_9})$ is not in $fdr(T[6, 9])$ but $\theta(\overline{p_6 p_{10}})$ is in $fdr(T[6, 10])$.

Now, we know that the checking step can be done in $O(||fdr(T[i, j])||)$. There are two remaining issues related to this checking step. The first issue is related to the size of $fdr(T[i, j])$. If this size is very large, the checking step is still expensive. Fortunately, we find that this size is usually a small constant. When $\epsilon_t \leq \pi/2$, it is equal to 1. The second issue is how to compute $fdr(T[i, j])$ efficiently for each i and j where $i < j$.

Issue 1: Size of $fdr(T[i, j])$

Lemma 4.4.2 *Let $T = (p_1, p_2, ..., p_n)$ be a trajectory and ϵ_t be the error tolerance. Then, given two integers i and j ($1 \leq i < j \leq n$), $||fdr(T[i, j])||$ is bounded by*

$\min\{1 + \lfloor \frac{\epsilon_t}{(\pi - \epsilon_t)} \rfloor, j - i\}$. □

Proof. We first give some concepts based on an angular range and provide a lemma which is used to prove Lemma 4.4.2.

We denote the *universe* angular range $[0, 2\pi)$ by $\mathcal{U}$, Given an angular range $[a, b]$, we denote its *complement* wrt $\mathcal{U}$, which is the angular range (b, a), by $[a, b]^c$. We define the *span* of an angular range $[a, b]$, denoted by $[a, b].span$, to be equal to $(b - a)(\mod 2\pi)$. We define the *span* of $fdr(T[i, j])$ $(1 \leq i < j \leq n)$, denoted by $fdr(T[i, j]).span$, to be equal to the sum of the spans of the disjoint angular ranges that are involved in $fdr(T[i, j])$.

Lemma 4.4.3 *Let $[a, b]$ and $[a', b']$ be two angular ranges. $[a, b] \cap [a', b']$ involves two disjoint angular ranges iff $[a', b']^c$ (i.e., (b', a')) falls in $[a, b]$ completely.* □

Proof. This could be verified easily by the fact that $[a, b] \cap [a', b']$ involves two disjoint angular ranges, namely $[a, b']$ and $[a', b]$, iff (b', a') falls in $[a, b]$ completely. ■

Suppose $fdr(\overline{p_k p_{k+1}})$ is $[a_k, b_k]$ for $i \leq k < j$. Then, $fdr(T[i, j])$ is equal to $\cap_{k=i}^{j-1}[a_k, b_k]$. Besides, we know $[a_k, b_k].span = 2\epsilon_t$ for $i \leq k < j$ since $a_k = \theta(\overline{p_k p_{k+1}}) - \epsilon_t \mod 2\pi$ and $b_k = \theta(\overline{p_k p_{k+1}}) + \epsilon_t \mod 2\pi$ (Definition 4.4.1).

First, we prove $||fdr(T[i, j])|| \leq j - i$ by induction of $k = j - i$. Base step: $k = 1$. The correctness is obvious since $fdr(T[i, i + 1]) = [a_i, b_i]$ which involves one angular range only (i.e., $[a_i, b_i]$) and thus $||fdr(T[i, i + 1])|| = 1$. Induction step: $||fdr(T[i, i + k)|| \leq k$ implies $||fdr(T[i, i + k + 1)|| \leq k + 1$. Assume $||fdr(T[i, i + k)|| = r$ $(r \leq k)$. Specifically, let $[a'_1, b'_1], [a'_2, b'_2], ..., [a'_r, b'_r]$ be the r disjoint angular ranges involved in $fdr(T[i, i + k])$. Then, $fdr(T[i, i + k + 1])$, which is equal to $fdr(T[i, i + k]) \cap [a_{i+k}, b_{i+k}]$, corresponds to r intersections, $[a'_h, b'_h] \cap [a_{i+k}, b_{i+k}]$ for $1 \leq h \leq r$, each two of which are disjoint. Among these r intersections, we show that

at most one involves two disjoint angular ranges by contradiction. Assume that there exists h_1 and h_2 ($i \leq h_1 \neq h_2 \leq r$) such that both $[a'_{h_1}, b'_{h_1}] \cap [a_{i+k}, b_{i+k}]$ and $[a'_{h_2}, b'_{h_2}] \cap [a_{i+k}, b_{i+k}]$ involve two disjoint angular ranges. According to Lemma 4.4.3, we know that (b_{i+k}, a_{i+k}) is in both $[a'_{h_1}, b'_{h_1}]$ and $[a'_{h_2}, b'_{h_2}]$, which leads to a contradiction since $[a'_{h_1}, b'_{h_1}]$ and $[a'_{h_2}, b'_{h_2}]$ are disjoint. As a result, $fdr(T[i, i+k_1])$ involves at most $r+1$ disjoint angular ranges. That is, $||fdr(T[i, i+k+1])|| \leq r+1 \leq k+1$.

Second, we prove $||fdr(T[i,j])|| \leq 1 + \lfloor \frac{\epsilon_t}{\pi - \epsilon_t} \rfloor$. We compute $fdr(T[i, i+1]), fdr(T[i, i+2]), ..., fdr(T[i,j])$ sequentially based on the following equation.

$$\begin{aligned} fdr(T[i, i+k+1]) &= fdr(T[i, i+k]) \cap [a_{i+k}, b_{i+k}] \\ &= fdr(T[i, i+k]) \backslash (b_{i+k}, a_{i+k}) \end{aligned} \tag{4.5}$$

We have two cases regarding Equation 4.5. Case 1: $||fdr(T[i, i+k+1])|| = ||fdr(T[i, i+k])|| + 1$. In this case, according to Lemma 4.4.3, (b_{i+k}, a_{i+k}) is in one of the disjoint angular ranges that are involved in $fdr(T[i, i+k+1])$. Then, we deduce that $fdr(T[i, i+k+1]).span = fdr(T[i, i+k]).span - (2\pi - 2\epsilon_t)$ since $(a_{i+k} - b_{i+k}) \mod 2\pi = 2\pi - 2\epsilon_t$. Case 2: $||fdr(T[i, i+k+1)|| = ||fdr(T[i, i+k)||$. In this case, we have $fdr(T[i, i+k+1]).span \leq fdr(T[i, i+k]).span$ since the span is non-increasing after an intersection operation. In view of the above two cases, we conclude that the *increase* of the number of disjoint angular ranges by 1 (in Case 1 only) is due to the *decrease* of the span by $(2\pi - 2\epsilon_t)$. Since at the beginning, $||fdr(T[i, i+1])|| = 1$ and $fdr(T[i, i+1]).span = 2\epsilon_t$, $||fdr(T[i, i+k])||$ has its greatest value equal to $(1 + \lfloor \frac{2\epsilon_t}{2\pi - 2\epsilon_t} \rfloor)$. □ ■

According to Lemma 4.4.2, $||fdr(T[i,j])||$ is usually bounded by a small constant. Let $C = \min\{1 + \lfloor \frac{\epsilon_t}{(\pi - \epsilon_t)} \rfloor, j - i\}$. In particular, when $\epsilon_t \leq \pi/2$, $\lfloor \frac{\epsilon_t}{(\pi - \epsilon_t)} \rfloor$ is equal to 0. In this case, $||fdr(T[i,j])||$ is exactly equal to $\min\{1, j-i\} = 1$ (since $j > i$).

Issue 2: How to Compute $fdr(T[i,j])$ Efficiently

A straightforward method to compute $fdr(T[i,j])$ is to compute $fdr(\overline{p_h p_{h+1}})$ ($i \leq h < j$) *independently* and then to intersect these fdr's. This straightforward method, nevertheless, incurs the cost of $\Omega(n)$ on average. Since we have $\Theta(n^2)$ instances of $fdr(T[i,j])$, computing all instances of $fdr(T[i,j])$ with this method incurs the total cost of $\Omega(n^3)$. Fortunately, this method could be improved significantly since it involves a lot of redundant work. A better method only takes $O(C)$ instead of $\Omega(n)$ to compute $fdr(T[i,j])$ based on the following *incremental property*.

Given two integers i and j where $1 \leq i < j < n$,

$$fdr(T[i,j+1]) = fdr(T[i,j]) \cap fdr(\overline{p_j p_{j+1}}) \tag{4.6}$$

Suppose that the content of $fdr(T[i,j])$ is known, since the total number of angular ranges in $fdr(\overline{p_j p_{j+1}})$ is 1, we can compute $fdr(T[i,j+1])$ in $O(C)$ time. Note that C is the greatest number of angular ranges in $fdr(T[i,j])$ and the intersection operation between two intervals could be finished in $O(1)$ time.

Thus, we propose to compute $fdr(T[i,j])$ ($1 \leq i < j \leq n$) using the incremental property. Specifically, it involves n rounds.

- At round 1, it computes $fdr(T[h,h+1])$ (i.e., $fdr(\overline{p_h p_{h+1}})$) for $1 \leq h < n$.
- At round r ($r > 1$), it computes $fdr(T[h,h+r])$ for $1 \leq h \leq n-r$. Specifically, it computes $fdr(T[h,h+r])$ by intersecting $fdr(T[h,h+r-1])$ (which has been maintained at round $r-1$) and $fdr(\overline{p_{h+r-1} p_{h+r}})$. Note that this operation takes $O(C)$ time.

Complexity Analysis. The time complexity of the above method for computing $fdr(T[i,j])$'s for $1 \leq i < j \leq n$ is $O(C \cdot n^2)$ since it involves n rounds and each

round incurs the cost of $O(C \cdot n)$. Since there are $O(n^2)$ times of checking whether $\epsilon(\overline{p_i p_{i+1}}) \leq \epsilon_t$ and each checking can be done in $O(C)$ time with the $fdr(T[i,j])$ information, the time complexity of the *SP* algorithm with the complexity improvement is $O(C \cdot n^2)$. Besides, the above method of computing $fdr(T[i,j])$'s has its space complexity of $O(C \cdot n)$ since at each round, it is sufficient to maintain $O(n)$ $fdf(T[i,j])$'s each of which involves $O(C)$ intervals. Since the *SP* algorithm with the complexity improvement materializes G_{ϵ_t} explicitly, we know that its space complexity is $O(C \cdot n + |V| + |E|)$, where V (E) is the vertex (edge) set of G_{ϵ_t}.

4.4.3 Combining The Two Enhancements

The practical enhancement involves the construction of only *some* edges, which means that we just need to perform the checking of $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ for *some* pairs of positions (p_i, p_j) only where $1 \leq i < j \leq n$. However, the cost of checking whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ is $O(n)$ which is expensive. In contrast, the complexity improvement reduces the cost of checking whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ from $O(n)$ to $O(C)$, but the undesired part is that it always computes $fdr(T[i,j])$ for *all* pairs of positions (p_i, p_j).

In this part, we propose to unify the good aspects of both the practical enhancement and the complexity improvement and at the same time, to avoid their undesired aspects. Our strategy is to maintain $fdr(T[i,j])$ $(1 \leq i < j \leq n)$ (i.e., the idea of complexity improvement) on the demand of the BFS process (i.e., the idea of the practical enhancement). Specifically, when checking whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ $(1 \leq i < j \leq n)$, we have two cases. Case 1: $fdr(T[i,j])$ has been computed. In this case, the checking could be finished in $O(C)$ time. Case 2: $fdr(T[i,j])$ has not been computed. In this case, we recursively resort to $fdr(T[i,j-1])$ for computing $fdr(T[i,j])$. This recursive process stops either in Case 1 or $fdr(T[i,i+1])$ is acquired. Note that $fdr(T[i,i+1])$ could be computed in $O(1)$ time by Equation (4.3).

This version of *SP* algorithm enjoys the benefit of the practical enhancement since it checks whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ $(1 \leq i < j \leq n)$ on demand of the BFS process and does not materialize G_{ϵ_t} explicitly, and it also enjoys the benefit of the complexity improvement since it adopts the "feasible direction range" concept for checking whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$, which is fast.

It could be verified that the worse-case time complexity of the *SP* algorithm with both the practical enhancement and the complexity improvement is $O(C \cdot n^2)$ since it computes only a sub-set of all possible $fdr(T[i,j])$'s.

Besides, the space complexity of this version of *SP* is $O(C \cdot n)$ (it is sufficient to maintain for each i the computed $fdr[i,j]$ with the *largest* j among all *computed* $fdr[i,j]$'s throughout the execution of the algorithm since each $fdr[i,j]$ is enquired at most once and note that it does not materialize G_{ϵ_t}).

4.5 Approximate Algorithm

According to the discussion in Section 4.4, the time complexity of an exact algorithm for Min-Size is at least quadratic. This, however, is not scalable enough when the datasets involves millions of positions. In this section, we develop an approximate algorithm called *Intersect* for the Min-Size problem, which runs in linear time and gives a certain degree of quality guarantee.

Before we describe *Intersect*, we first give a concept of "feasibility" used in the algorithm. Given an error tolerance ϵ_t and two integers i and j where $1 \leq i < j \leq n$, $\overline{p_i p_j}$ is said to be ϵ_t-*feasible* iff $fdr(T[i,j]|\epsilon_t)$ is non-empty. With this concept, we have the following property.

Lemma 4.5.1 (Feasibility) *Given an error tolerance ϵ_t and two integers i and j where $1 \leq i < j \leq n$, if $\overline{p_i p_j}$ is $\frac{\epsilon_t}{2}$-feasible, then $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$.*

Proof. Since $\overline{p_i p_j}$ is$\frac{\epsilon_t}{2}$-feasible, $fdr(T[i,j]|\epsilon_t/2)$ is non-empty.

We prove with two steps.

First, we deduce that for any two segments between i and j in T, $\overline{v_h v_{h+1}}$ and $\overline{v_{h'} v_{h'+1}}$ ($i \leq h < h' < j$), we have $\triangle(\theta(\overline{p_h p_{h+1}}), \theta(\overline{p_{h'} p_{h'+1}})) \leq \epsilon_t$. Let $\theta_1 = \theta(\overline{p_h p_{h+1}})$ and $\theta_2 = \theta(\overline{p_{h'} p_{h'+1}})$. Since $fdr(T[i,j]|\epsilon_t/2)$ is non-empty, we know $fdr(\overline{p_h p_{h+1}}|\epsilon_t/2)$ and $fdr(\overline{p_{h'} p_{h'+1}}|\epsilon_t/2)$ intersects. Let θ_3 be a direction that is in both $fdr(\overline{p_h p_{h+1}}|\epsilon_t/2)$ and $fdr(\overline{p_{h'} p_{h'+1}}|\epsilon_t/2)$. Thus, $\triangle(\theta_1, \theta_3) \leq \epsilon_t/2$ and $\triangle(\theta_2, \theta_3) \leq \epsilon_t/2$. As a result, according to Equation (4.1), we have

$$\triangle(\theta_1, \theta_3) = \min\{|\theta_1 - \theta_3|, 2\pi - |\theta_1 - \theta_3|\} \leq \epsilon_t/2 \tag{4.7}$$

$$\triangle(\theta_2, \theta_3) = \min\{|\theta_2 - \theta_3|, 2\pi - |\theta_2 - \theta_3|\} \leq \epsilon_t/2 \tag{4.8}$$

Next, we show that $\triangle(\theta_1, \theta_2) \leq \epsilon_t$. Consider four cases.

Case 1: $|\theta_1 - \theta_3| \leq 2\pi - |\theta_1 - \theta_3|$ and $|\theta_2 - \theta_3| \leq 2\pi - |\theta_2 - \theta_3|$. In this case, we have $|\theta_1 - \theta_3| \leq \epsilon_t/2$ and $|\theta_2 - \theta_3| \leq \epsilon_t/2$. Thus,

$$|\theta_1 - \theta_2| = |\theta_1 - \theta_3 + \theta_3 - \theta_2| \leq |\theta_1 - \theta_3| + |\theta_2 - \theta_3| \leq \epsilon_t$$

Therefore, $\triangle(\theta_1, \theta_2) = \min\{|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|\} \leq \epsilon_t$.

Case 2: $|\theta_1 - \theta_3| > 2\pi - |\theta_1 - \theta_3|$ and $|\theta_2 - \theta_3| \leq 2\pi - |\theta_2 - \theta_3|$. In this case, we have $|\theta_1 - \theta_3| > 2\pi - \epsilon_t/2$ and $|\theta_2 - \theta_3| \leq \epsilon_t/2$.

$$|\theta_1 - \theta_2| = |\theta_1 - \theta_3 - (\theta_2 - \theta_3)| \geq |\theta_1 - \theta_3| - |\theta_2 - \theta_3| >$$

$$2\pi - \epsilon_t/2 - \epsilon_t/2 = 2\pi - \epsilon_t$$

Therefore, $\triangle(\theta_1, \theta_2) = \min\{|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|\} \leq \epsilon_t$.

Case 3: $|\theta_1 - \theta_3| \leq 2\pi - |\theta_1 - \theta_3|$ and $|\theta_2 - \theta_3| > 2\pi - |\theta_2 - \theta_3|$. This case is symmetric with Case 2 and the proof is similar.

Case 4: $|\theta_1 - \theta_3| > 2\pi - |\theta_1 - \theta_3|$ and $|\theta_2 - \theta_3| > 2\pi - |\theta_2 - \theta_3|$. In this case, we have $2\pi - |\theta_1 - \theta_3| \leq \epsilon_t/2$ and $2\pi - |\theta_2 - \theta_3| \leq \epsilon_t/2$. As a result, we have $|\theta_1 - \theta_3| > \pi$ which implies that θ_1 and θ_3 are at different sides of the x-axis, and $|\theta_2 - \theta_3| > \pi$ which implies that θ_2 and θ_3 are at different sides of the x-axis as well. Without loss of generality, assume that θ_3 is at the upper side of the x-axis. Then, both θ_1 and θ_2 are at the lower side. Furthermore, we assume assume $\theta_1 \geq \theta_2$ without loss of generality. Then, $|\theta_1 - \theta_2| \leq 2\pi - |\theta_2 - \theta_3| \leq \epsilon_t/2 < \epsilon_t$. Consider Figure 4.8(a) for illustration. Therefore, $\triangle(\theta_1, \theta_2) \leq |\theta_1 - \theta_2| \leq \epsilon_t$.

By combining the above four cases, we have $\triangle(\theta_1, \theta_2) \leq \epsilon_t$.

Second, we show that $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$. According to the results in the first step, for any two segments between p_i and p_j in T, the angular difference between their directions is at most ϵ_t. It is easy to verify that there exists an angular range $[a, b]$ such that the directions of all the segments between p_i and p_j in T fall in $[a, b]$ and the span of $[a, b]$ is bounded by ϵ_t (where the definition of "span" can be found in the proof of Lemma 4.4.2). Given two positions p and p', we denote the *vector* from p to p' by $\overrightarrow{pp'}$. Besides, it could be verified that $\theta(\overline{p_i p_j})$ falls in the angular range $[a, b]$ by utilizing the fact that $\overrightarrow{p_i p_j} = \sum_{h=i}^{j-1} \overrightarrow{p_h p_{h+1}}$. For illustration, consider Figure 4.8(b), where $[a, b]$ corresponds to $[\theta(\overline{p_{i+1} p_{i+2}}), \theta(\overline{p_{i+2} p_{i+3}})]$. Therefore, we know $\triangle(\theta(\overline{p_i p_j}), \theta(\overline{p_h p_{h+1}})) \leq \epsilon_t$ for $i \leq h < j$ since the angular difference between any two directions falling in range $[a, b]$ is smaller than that the angular difference between a and b, which is bounded by ϵ_t. Thus, we know $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$. ■

Specifically, *Intersect* has the following steps. Let T' be a variable storing the simplified trajectory to be returned. Let e be a variable storing the position *index* of the last position in T'. Let h be a variable storing the position index of the position in T being processed. Initially, *Intersect* initializes T' to be (p_1), and then sets e to be 1 (since p_1 is currently the last position in T'). Then, it reads each of the remaining

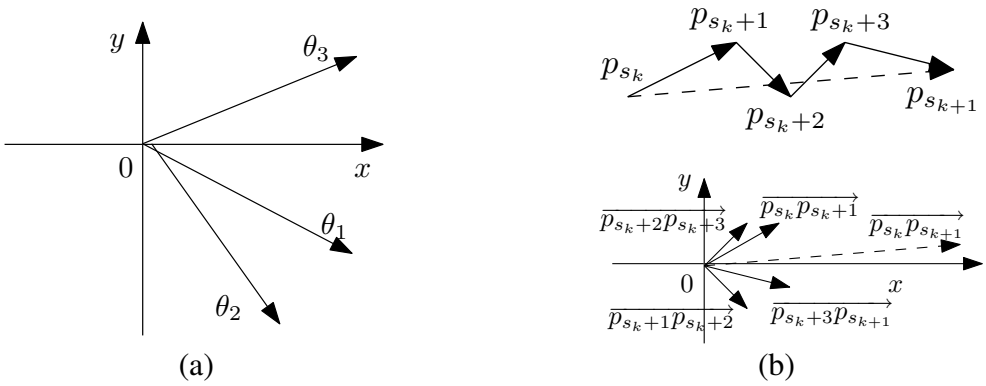


Figure 4.8: Proof of Lemma 4.5.2

Algorithm 8 The *Intersect* Algorithm

Require: A trajectory $T = (p_1, p_2, ..., p_n)$; an error tolerance ϵ_t
Ensure: An ϵ_t-simplification of T'
1: $T' \leftarrow (p_1)$; $e \leftarrow 1$; $h \leftarrow 2$
2: **while** $h \leq n$ **do**
3: **while** $h \leq n$ and $\overline{p_e p_h}$ is $\frac{\epsilon_t}{2}$-feasible **do**
4: increment h by 1
5: append p_{h-1} to T'; $e \leftarrow h - 1$
6: **return** T'

positions sequentially. It sets h to 2 (since p_2 is the position in T to process next). It proceeds with an iterative step as follows. Whenever $h \leq n$ and $\overline{p_e p_h}$ is $\frac{\epsilon_t}{2}$-feasible, it increments h by 1. It terminates this iterative step when either (1) $h > n$ or (2) h has *just* been incremented to a value such that $\overline{p_e p_h}$ is not $\frac{\epsilon_t}{2}$-feasible. For both stopping conditions, we know that $\overline{p_e p_{h-1}}$ is $\frac{\epsilon_t}{2}$-feasible and thus by Lemma 4.5.1, we have $\epsilon(\overline{p_e p_{h-1}}) \leq \epsilon_t$. Thus, p_{h-1} is appended to T'. Then, e is set to $h - 1$. It repeats the above iterative step whenever $h \leq n$. At the end, it returns T'. The pseudo-code of *Intersect* is shown in Algorithm 8.

With Lemma 4.5.1, it is easy to verify that the trajectory returned by *Intersect* is an ϵ_t-simplification of T.

Lemma 4.5.2 *Let T' be the output of the* Intersect *algorithm in Algorithm 8. Then, T' is an ϵ_t-simplification of T.* □

Intersect not only scans the data once only and returns an ϵ_t-simplification of T at

the end, but also provides a certain degree of guarantee on the size of the simplified trajectory.

Lemma 4.5.3 (Size Bound) *Let T' be the output of the* Intersect *algorithm in Algorithm 8. We have $|T'| \leq |T''|$, where T'' is the $\epsilon_t/2$-simplification of T with the minimum size.* □

Proof. Let $T = (p_1, ..., p_n)$, $T' = (p_{s_1}, ..., p_{s_m})$ and $T'' = (p_{r_1}, ..., p_{r_l})$. By definition, we have $s_1 = r_1 = 1$ and $s_m = r_l = n$. Note that $n = |T|, m = |T'|$ and $l = |T''|$.

First, we prove that $fdr(T[r_k, r_{k+1}]|\epsilon_t/2)$ is non-empty for $1 \leq k < l$. Consider a specific $k \in [1, l-1]$. Since T'' is an $\epsilon_t/2$-simplification, i.e., $\epsilon(T'') \leq \epsilon_t/2$, we know $\epsilon(\overline{p_{r_k}p_{r_{k+1}}}) \leq \epsilon_t/2$. As a result, we know $\triangle(\theta(\overline{p_h p_{h+1}}), \theta(\overline{p_{r_k}p_{r_{k+1}}})) \leq \epsilon_t/2$, which further implies that $\theta(\overline{p_{r_k}p_{r_{k+1}}})$ falls in $fdr(\overline{p_h p_{h+1}}|\epsilon_t/2)$ for $r_k \leq h < r_{k+1}$. Thus, $fdr(T[r_k, r_{k+1}]|\epsilon_t/2)$ is non-empty.

Next, we prove $m \leq l$ by contradiction. Assume that $m > l$. In the following, we want to show that $s_k \geq r_k$ for $k = 1, 2, ..., l$. When $k = 1$, it is true since $s_1 = r_1 = 1$. Consider $k = 2$. According to Algorithm *Intersect*, we know that $fdr(T[s_1, s_2]|\epsilon_t/2)$ is non-empty but $fdr(T[s_1, s_2 + 1]|\epsilon_t/2)$ is empty. We can say that s_2 is the *greatest* possible position such that $fdr(T[s_1, s_2]|\epsilon_t/2)$ is non-empty. Besides, since $fdr(T[r_1, r_2]|\epsilon_t/2)$ is non-empty and $r_1 = s_1$, we derive that $s_2 \geq r_2$. Consider $k = 3$. Since $fdr(T[r_2, r_3]|\epsilon_t/2)$ is non-empty and $s_2 \geq r_2$, we consider two cases. *Case 1:* $s_2 < r_3$. In this case, we know that $fdr(T[s_2, r_3]|\epsilon_t/2)$ is non-empty and thus $s_3 \geq r_3$. *Case 2:* $s_2 \geq r_3$. In this case, we also deduce that $s_3 \geq r_3$ (since $s_3 > s_2$). By combining these two cases, we conclude that $s_3 \geq r_3$. Continuing the above procedure, we can deduce that $s_k \geq r_k$ for

$k = 1, 2, ..., l$. Since $m > l$, we have $s_l < s_m$ and thus $s_l + 1 \leq s_m = r_l$. Note that $fdr(T[r_{l-1}, r_l]|\epsilon_t/2) = fdr(T[r_{l-1}, s_{l-1}]|\epsilon_t/2) \cap fdr(T[s_{l-1}, s_l+1]|\epsilon_t/2) \cap fdr(T[s_l+1, s_m]|\epsilon_t/2)$ (When $s_l + 1 = s_m$, we define $fdr(T[s_l + 1, s_m]|\epsilon_t/2)$ to be the universe angular range). Besides, since $fdr(T[r_{l-1}, r_l]|\epsilon_t/2)$ is non-empty, we deduce that $fdr(T[s_{l-1}, s_l + 1]|\epsilon_t/2)$ is non-empty, which, however leads to a contradiction because according to algorithm *Intersect*, $fdr(T[s_{l-1}, s_l + 1]|\epsilon_t/2)$ is empty. Thus, we conclude that $m \leq l$, which implies that $|T'| \leq |T''|$. ■

Complexity Analysis. We know that variable h is incremented whenever line 4 of Algorithm 8 is executed. Both the stopping condition of the outer while-loop (line 2) and one of the stopping conditions of the inner while-loop (line 3) are "$h \leq n$". We conclude that there are $O(n)$ times to execute the steps in line 3, line 4 and line 5. The step in line 4 and the step in line 5 take $O(1)$ time. In the following, we show that the step in line 3 also takes O(1) time. Thus, the time complexity of *Intersect* is $O(n)$.

The remaining issue is to derive the time complexity of the step in line 3. In line 3, checking whether $h \leq n$ can be done in $O(1)$ time. However, a straightforward implementation of checking whether $\overline{p_e p_h}$ is $\frac{\epsilon_t}{2}$-feasible (line 3) (or equivalently checking whether $fdr(T[e, h]|\epsilon_t/2)$ is non-empty) is expensive. This is because as described in Section 4.4.2, computing $fdr(T[e, h]|\epsilon_t/2)$ from scratch is expensive. By using the same technique in Section 4.4.2, we can perform this checking operation in $O(1)$ time. Specifically, we introduce a variable called fdr to store the content of $fdr(T[e, h]|\epsilon_t/2)$. We have the following two changes in the algorithm due to this variable. Firstly, between line 2 and line 3, we insert a statement that "fdr $\leftarrow fdr(\overline{p_e p_h}|\epsilon_t/2)$". Note that $h = e + 1$ at this moment, and the time complexity of this statement is $O(C')$ where $C' = \min\{1 + \lfloor \frac{\epsilon_t/2}{(\pi - \epsilon_t/2)} \rfloor, h - e\}$ (by using Lemma 4.4.2). Secondly, in the inner while-loop, just after line 4, we insert a statement that "fdr $\leftarrow$ fdr $\cap fdr(\overline{p_{h-1} p_h}|\epsilon_t/2)$". Note that the time complexity of this statement is $O(C')$. With these two changes, checking

whether $\overline{p_e p_h}$ is $\frac{\epsilon_t}{2}$-feasible (line 3) is equivalent to checking whether the current content of fdr is non-empty because the current content of fdr is equal to $fdr(T[e,h]|\epsilon_t/2)$ (by Equation (4.6)). Furthermore, since ϵ_t is at most π and thus $\epsilon_t/2$ is at most $\pi/2$, we deduce that $C' = O(1)$. We conclude that the time complexity of the step in line 3 is $O(1)$.

The space complexity of *Intersect* is $O(n)$ (which corresponds to the memory usage for storing the simplified trajectory T').

4.6 Empirical Studies

We introduce the datasets and also the algorithms for our empirical study in Section 4.6.1, and then present the empirical results of DPTS, the exact algorithms for the Min-Size problem, and the approximate algorithms for the Min-Size problem, in Section 4.6.2, in Section 4.6.3, and in Section 4.6.4, respectively.

4.6.1 Datasets and Algorithms

We used 5 real datasets in our experiments, namely Deer, Elk, Hurricane, Geolife and T-Drive. Deer and Elk[1] are two animal movement datasets which contain the radio-telemetry locations of deers in 1995 and elks in 1993, respectively. Hurricane[2] contains the trajectories of the Atlantic hurricanes from year 1950 to year 2004. These three datases (i.e., Deer, Elk and Hurricane) are benchmark datasets for trajectory clustering [64]. Geolife[3] records the outdoor movements of 182 users in a period of 5 years and T-Drive[4] is a set of taxi trajectories in Beijing. These two datasets are widely-used

[1] http://www.fs.fed.us/pnw/starkey/data/tables/

[2] http://weather.unisys.com/hurricane/atlantic/

[3] http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13/

[4] http://research.microsoft.com/apps/pubs/?id=152883

	# of trajectories	total # of positons	average # of positions per trajectory	directional difference between two adjacent segments (Mean, S.D.)
Deer	32	20,065	627	(1.669, 0.948)
Elk	33	47,204	1,430	(1.647, 0.984)
Hurricane	570	17,736	31	(0.213, 0.300)
Geolife	17,621	24,876,978	1,412	(0.364, 0.615)
T-Drive	10,359	17,740,902	1,713	(0.657, 0.803)

Table 4.1: Real datasets (DPTS and Min-Size)

for a broad range of applications on trajectory data [107, 101]. The statistics of these datasets are summarized in Table 4.1.

We study 5 exact algorithms with the following notions. DP is the dynamic programming algorithm and SP is the straightforward implementation of the *SP* algorithm. SP-prac (SP-theo) is the *SP* algorithm with the practical enhancement (complexity improvement) only and SP-both is the one with all enhancements. Besides, we study 4 approximate algorithms, Split, Merge, Greedy and Intersect. The first three are the adaptations of the existing trajectory simplification methods and the forth is proposed in this thesis.

All algorithms were implemented in C/C++ and run on a Linux platform with a 2.66GHz machine and 4GB RAM.

4.6.2 Relevance to Existing Studies

In this section, we conducted experiments to show how DPTS is relevant to existing studies.

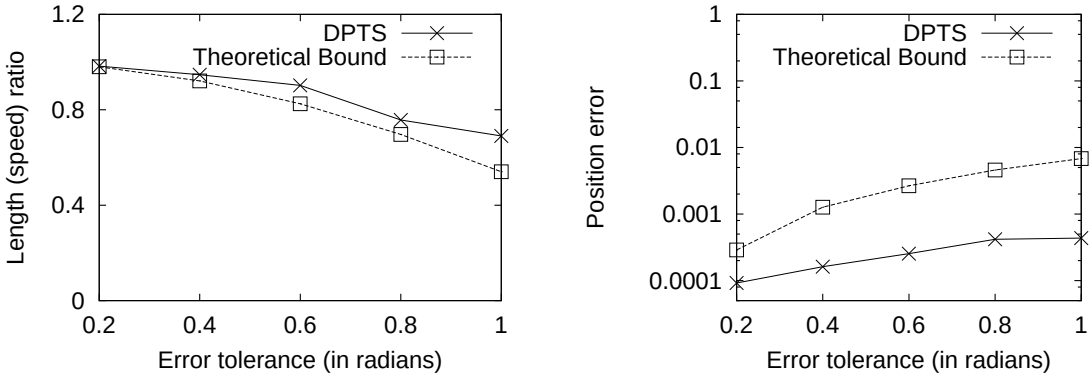
Bounds of DPTS wrt Existing Measurements

In this part, we verify the theoretical bounds of the length (speed) error and the position error of DPTS as introduced in Section 4.3.1.

We vary the tolerance ϵ_t on $\{0.2, 0.4, 0.6, 0.8, 1\}$. The results about length (speed)

errors are shown in Figure 4.9(a), where the "length (speed) ratio" is defined to be $\min_{i\in[1,n)}\{len(p'_i, p'_{i+1}|T')/len(p_i, p_{i+1}|T)\}$ where p'_i is the estimated position of p_i on T' for $i \in [1, n]$. Thus, the larger this ratio is, the more accurate the length (speed) information of the simplified trajectory is. Note that the length ratio and the speed ratio (with respect to a segment in the original trajectory) are exactly the same since the speed is equal the length divided by the time difference between the time stamps of the two end-points of the segment, and the time difference in the original trajectory is kept to be the same as the time difference in the simplified trajectory. We observe that the theoretical bound of the length (speed) ratio is usually good (e.g., it is about 0.92 when $\epsilon_t = 0.4$).

The results about the position error are shown in Figure 4.9(b). We observe that the empirical position error is usually significantly smaller than the theoretical bound (by near to one order of magnitude). Besides, when ϵ_t increases, the increase in the position error of DPTS becomes smaller. When ϵ_t becomes large, the position error of DPTS keeps quite stable.



(a) Length and speed error (b) position error

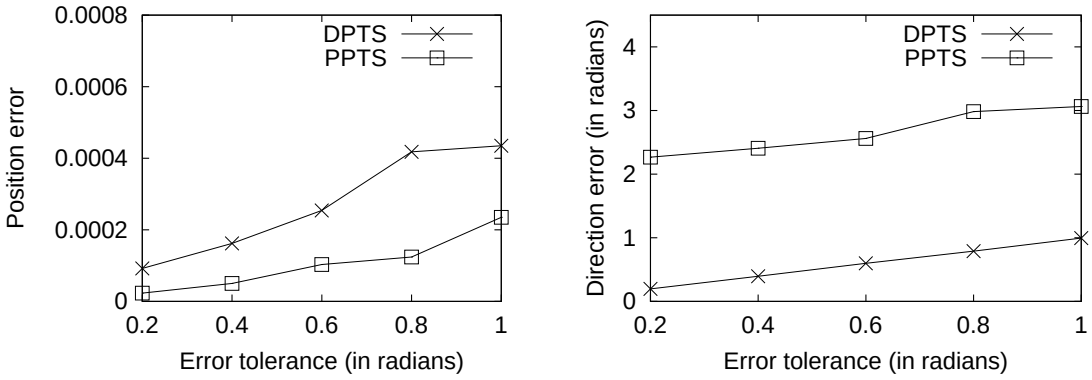
Figure 4.9: Verification of theoretical error bounds (Geolife, DPTS)

DPTS vs. PPTS

In this part, we want to compare DPTS with Position-Preserving Trajectory Simplification (PPTS) in terms of two measurements, namely the position error and

the direction error. In Section 4.3, we show that the position error of DPTS is *bounded* (Lemma 4.3.2) while the direction error of PPTS is *un-bounded* (Lemma 4.3.3). We study *how worst the position error of DPTS compared with PPTS is* and *how worst the direction error of PPTS compared with DPTS is*.

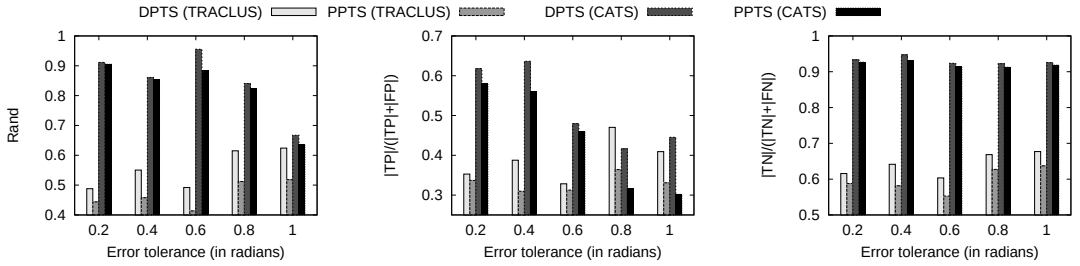
We adopt the *Douglas-Peucker* algorithm [31] for PPTS which is the most popular existing algorithm for PPTS [72, 12, 43], and we use our *SP* algorithm for DPTS. We vary ϵ_t for DPTS. For a fair comparison, we enforce that the simplified trajectories from DPTS and PPTS have the same size. The results are shown in Figure 4.10(a) for position errors and in Figure 4.10(b) for direction errors. Consider Figure 4.10(a). It could be noticed that though the position errors of DPTS are usually larger than those of PPTS, the difference is small. For example, the ratio of the position errors is between 1.85 to 3. Consider Figure 4.10(b). We observe that the direction errors of PPTS are significantly larger than that of DPTS. For example, when $\epsilon_t = 0.2$, the ratio is more than 10. Besides, the direction errors of PPTS are greater than 2, a value greater than $\pi/2$, even with a small value of ϵ_t and is nearly to π, the greatest possible direction error, with a medium value of ϵ_t, which implies that PPTS can hardly preserves the direction information. In conclusion, our DPTS preserves the direction information by its nature and also the position information to a certain degree, but PPTS preserves the position information only but not the direction information.



(a) Position error (b) Direction error

Figure 4.10: Comparison with existing PPTS (Geolife, DPTS)

An Application Study (Trajectory Clustering)



(a) Rand (b) $|TP|/(|TP| + |FP|)$ (c) $|TN|/(|TN| + |FN|)$

Figure 4.11: Trajectory Clustering Study (Deer, DPTS)

In Section 4.6.2, we compared DPTS and PPTS with their *favor-metrics* (i.e., the position error, favoring PPTS, and the direction error, favoring DPTS). In this section, we compare DPTS with PPTS with a *neutral* metric, the clustering quality, for a real-life application, trajectory clustering.

The main idea is as follows. Let $\mathcal{D}$ be a set of raw trajectories. We perform DPTS (PPTS) on each trajectory in $\mathcal{D}$ and obtain a set of simplified trajectories, denoted by $\mathcal{D}^d$ ($\mathcal{D}^p$). Then, we perform a clustering procedure on each of these two sets of trajectories and obtain the corresponding clustering results. We regard the clustering results based on $\mathcal{D}$ as ground truth and measure the qualities of the clustering results on $\mathcal{D}^d$ and $\mathcal{D}^p$. We verify DPTS by showing that the clustering results on $\mathcal{D}^d$ are consistently better (closer to the ground truth) than those on $\mathcal{D}^p$.

Consider the clustering procedure on $\mathcal{D}$ first. The clustering results based on $\mathcal{D}$ (i.e., the cluster membership of each trajectory) could be represented by a binary matrix $M_{n_t \times n_c}$, where n_t is the number of trajectories in the set and n_c is the number of resulting clusters. Note that under some clustering mechanisms such as the one in [64], a trajectory could belong to multiple clusters. For each trajectory $T \in \mathcal{D}$, its cluster membership could be represented by an n_c-dimensional binary vector. For each pair of trajectories T_1 and T_2 in $\mathcal{D}$, we measure the similarity between T_1 and T_2 by the Euclidean distance between T_1's cluster membership (which is a vector) and T_2's clus-

ter membership (which is a vector). If the distance is below a pre-set threshold σ, we regard T_1 and T_2 to be *similar*; otherwise, we regard T_1 and T_2 to be *dissimilar*. Thus, based on the clustering results on $\mathcal{D}$, we can always obtain a *similarity matrix* which indicates for each pair of two trajectories in $\mathcal{D}$ whether they are similar or not. Let S be such a similarity matrix corresponding to $\mathcal{D}$. The distance threshold σ for deciding whether two trajectories are similar is set to 0.5 by default (all distances are normalized to [0, 1]).

For a specific trajectory $T \in \mathcal{D}$, we denote its simplification in $\mathcal{D}^d$ and $\mathcal{D}^p$ by T^d and T^p, respectively.

Similarly, we perform the same clustering procedure on $\mathcal{D}^d$ and $\mathcal{D}^p$ as we did on $\mathcal{D}$ and obtain their corresponding similarity matrices, denoted by S^d and S^p, respectively.

We measure the quality of the clustering on $\mathcal{D}^d$ (and the clustering on $\mathcal{D}^p$) as follows. For a pair of two trajectories T_1 and T_2 in $\mathcal{D}$, we have 4 cases. Case 1: T_1 and T_2 are similar (wrt S) and T_1^d and T_2^d (the simplified trajectories of T_1 and T_2 under DPTS) are similar (wrt S^d). In this case, we have an occurrence of true positive (TP). Case 2: T_1 and T_2 are dissimilar (wrt S) and T_1^d and T_2^d are similar (wrt S^d). In this case, we have an occurrence of false positive (FP). Similarly, Case 3 and Case 4 correspond to the occurrences of false negative (FN) and true negative (TN), respectively. We adopt three measures in our experiments for measuring the clustering results. The first is called *Rand*, which is defined to be $(|TP| + |TN|)/(|TP| + |FP| + |FN| + |TN|)$ where $|\cdot|$ denotes the number of occurrences. The second and the third are defined to be $|TP|/(|TP| + |FP|)$ and $|TN|/(|TN| + |FN|)$, respectively. The larger the measure is, the better the clustering is.

For the trajectory clustering procedure, we adopt the *TRACLUS* algorithm [64] and the *CATS* algorithm [50]. According to [50], existing trajectory clustering algorithms fall in two categories. The first category includes those algorithms which take each

trajectory *as a whole* for clustering while the second category includes the algorithms which use *sub-trajectories* for clustering. *CATS* is the state-of-the-art in the first category and *TRACLUS* is the state-of-the-art in the second category [50].

For the PPTS procedure, again, we adopt the popular *Douglas-Peucker* algorithm.

We vary the error tolerance ϵ_t with the values of 0.2, 0.4, 0.6, 0.8 and 1 for DPTS. Figure 4.11(a), (b) and (c) show the results about the Rand measure, the measure of $|TP|/(|TP| + |FP|)$ and the measure of $|TN|/(|TN| + |FN|)$ on the Deer dataset, respectively.

We observe that the clustering based on the simplified trajectories returned by DPTS is consistently better than that based on the simplified trajectories returned by PPTS. This might be explained by the fact that the direction information is heavily used in the trajectory clustering algorithms and the direction information loss due to DPTS is bounded while that due to PPTS is un-bounded.

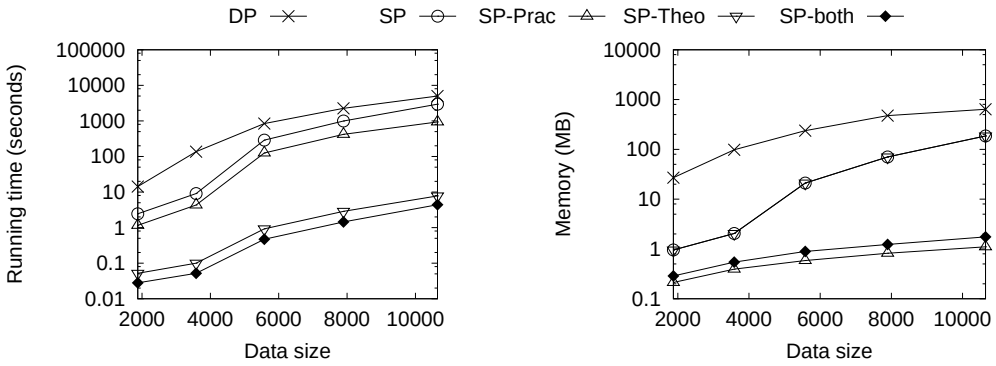
4.6.3 Performance Study of the Exact Algorithms

In the part, we study the effects of 2 factors, namely the data size (i.e., $|T|$) and the error tolerance (i.e., ϵ_t), on the performance of the exact algorithms. We use 2 measures, namely the running time and the memory.

Effect of $|T|$. The values used for $|T|$ are around $2k$ $4k$, $6k$, $8k$ and $10k$ (ϵ_t is fixed to be 1). For each setting of $|T|$, we select a set of 10 trajectories each of which has its size near to this value and run DPTS on each of these trajectories. Then, we average the experimental results on these trajectories (this policy is used throughout our experiments without specification). Figure 4.12 show the results on Geolife. According to these results, SP-both is the fastest while DP is the slowest due to its high time complexity. Besides, the complexity improvement helps to reduce the running time dramatically (e.g., SP-theo is faster than SP by 2-3 orders of magnitude). This

could be easily explained by the fact that with the complexity improvement, the cost of checking whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$ is reduced from $O(n)$ to $O(C)$ (C is a small constant). Though the practical enhancement improves the time efficiency a little, it helps to reduce the memory significantly (e.g., the memory occupied by SP-theo is 1-3 orders of magnitude larger than that occupied by SP-both and the difference increases when $|T|$ increases on Geolife).

The experimental results on T-Drive are similar and thus they are omitted.

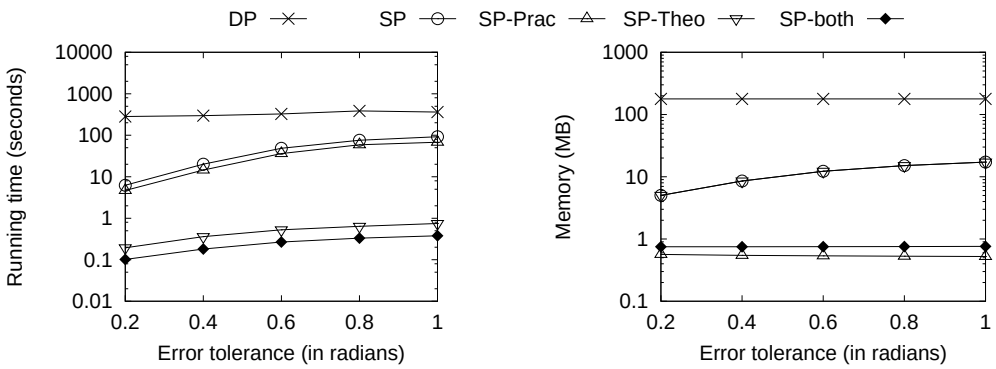


(a) Running time (b) Memory

Figure 4.12: Effect of data size $|T|$ (Geolife, Min-Size)

Effect of ϵ_t. The values used for ϵ_t are 0.2, 0.4, 0.6, 0.8 and 1 in radians ($|T|$ is fixed to be $5k$). Figure 4.13 shows the results on Geolife. According to these results, ϵ_t affects the SP algorithms only. Specifically, the running times of all SP algorithms increase slightly when ϵ_t becomes larger. This is because a larger ϵ_t usually results in G_{ϵ_t} with more edges and thus the BFS process on G_{ϵ_t} needs more time.

Compression Rate. We also study the effect of ϵ_t on the *size ratio* which is defined to be equal to $\sum_{T' \in \mathcal{D}'} |T'| / \sum_{T \in \mathcal{D}} |T|$, where $\mathcal{D}$ is the set of raw trajectories and $\mathcal{D}'$ is the set of the corresponding simplified trajectories. Note that a smaller size ratio means a higher compression rate. The results are shown in Figure 4.14. We have the following observations. First, the size ratio decreases significantly when we increase the tolerance from 0 slightly. This is good since it implies that under DPTS, the trajectory data could be simplified significantly with a small error. Second, we observe that the size



(a) Running time (b) Memory

Figure 4.13: Effect of error tolerance ϵ_t (Geolife, Min-Size)

ratio is strictly smaller than 1 (e.g., it is about 0.9 for the Geolife datasets) even if the error tolerance is set to be 0. This implies that the real-life trajectories usually involve a certain degree of redundancy and could be simplified without incurring any error.

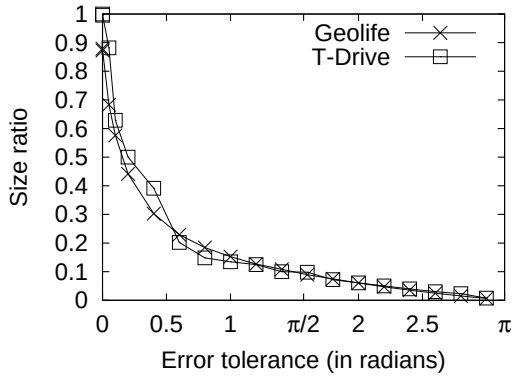
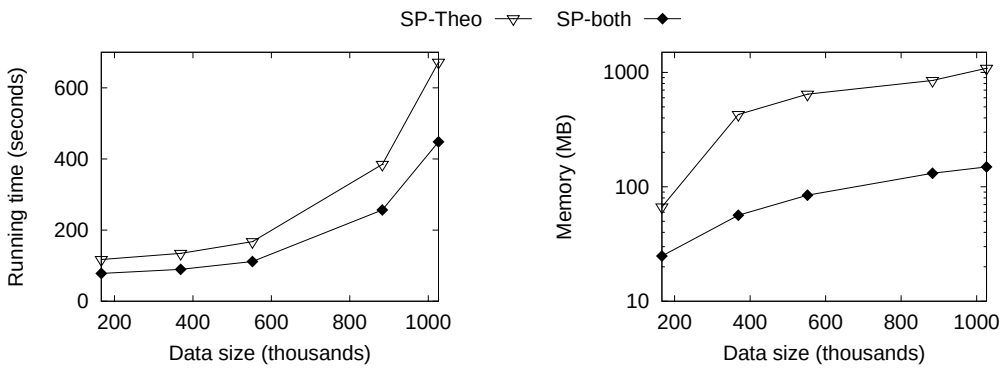


Figure 4.14: Compression rate (Min-Size)

Scalability Test. Figure 4.15 shows the results of the scalability test on the exact algorithms. We only show the results of SP-theo and SP-both since the other exact algorithms are not scalable on large datasets due to their expensive time complexities. According to these results, both SP-theo and SP-both are scalable to large trajectory datasets with millions of positions, and SP-both runs slightly faster than SP-theo. It is noted that SP-both occupies significantly less memory than SP-theo and thus SP-both is more scalable than SP-theo. This is because SP-both does not materialize G_{ϵ_t} explicitly while SP-theo does.



(a) Running time (b) Memory

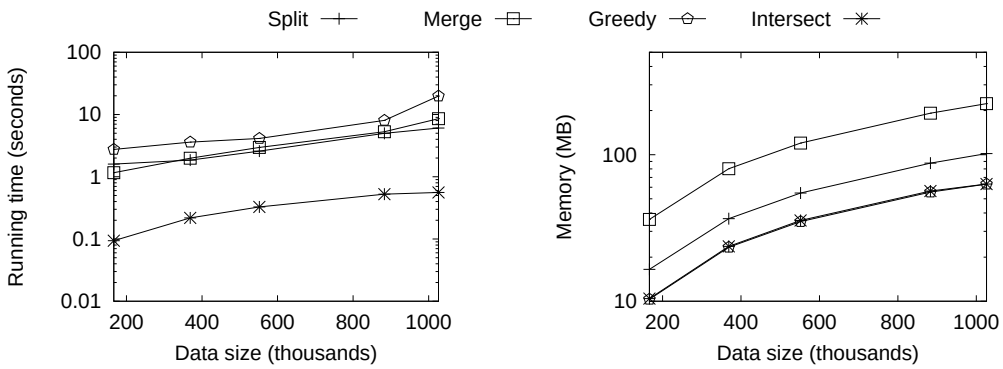
Figure 4.15: Scalability test for exact algorithms (Geolife, Min-Size)

4.6.4 Performance Study of the Approximate Algorithms

In this part, we study the effects of $|T|$ and ϵ_t on the approximate algorithms. We use 3 measures, namely the running time, the memory and the approximation error. The approximation error of an approximate algorithm is defined to be $|T'|/|T^*|$, where T' is the simplified trajectory returned by the approximate algorithm on a given raw trajectory and T^* is the simplified trajectory returned by an exact algorithm on the same raw trajectory.

Effect of $|T|$. The values used for $|T|$ are around $200k$, $400k$, $600k$, $800k$ and $1000k$ (ϵ_t is fixed to be 1). Figure 4.16 shows the results. According to these results, Intersect is the fastest, which is at least 1 order of magnitude faster than other approximate algorithms. This is because Intersect runs in linear time while the other algorithms run in quadratic time in the worst case.Besides, Intersect occupies the least memory and Greedy occupies slightly more memory than Intersect (though the difference in Figure 4.16(b) is not obvious).

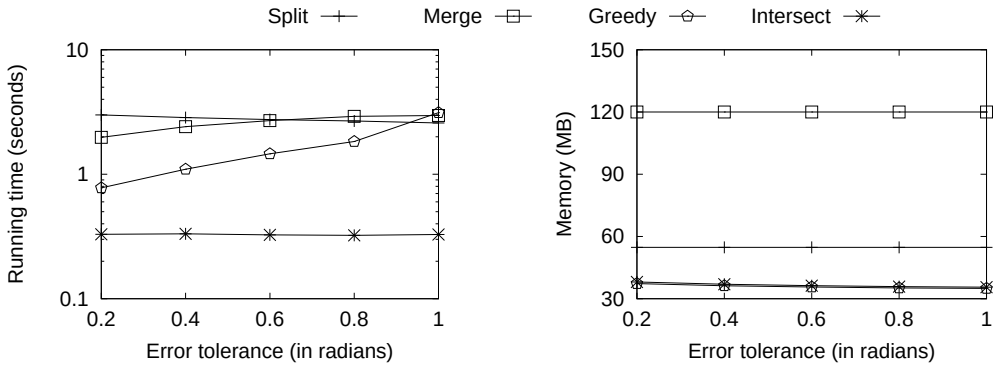
Effects of ϵ_t. Figure 4.17 shows the effects of ϵ_t on the approximate algorithms, where we vary ϵ_t with 0.2, 0.4, 0.6, 0.8 and 1 ($|T|$ is fixed to be $500k$). According to these results, Greedy runs faster than Split and Merge with small ϵ_t's. This is because with a smaller ϵ_t, it is less likely that a long sequence of consecutive segments could be



(a) Running time (b) Memory

Figure 4.16: Effect of data size $|T|$ (Geolife, Min-Size)

approximated with one segment and thus the cost of checking the error of the segment linking the start position and the end position is small.

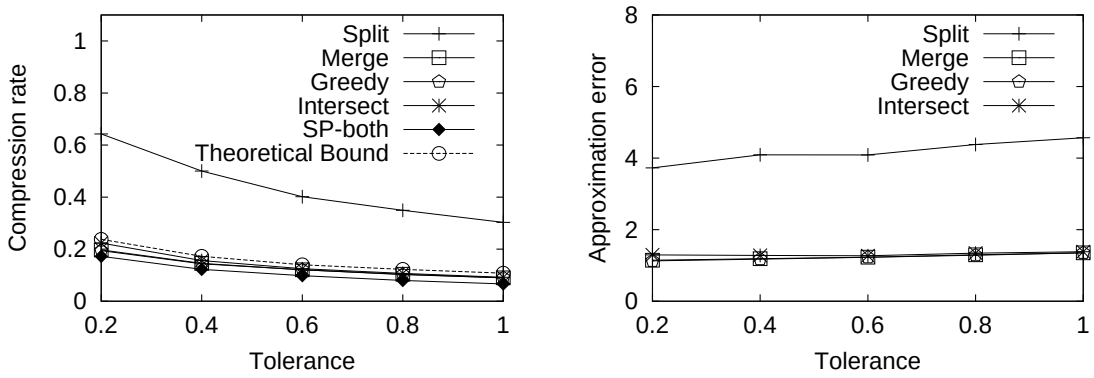


(a) Running time (b) Memory

Figure 4.17: Effect of the error tolerance ϵ_t (Geolife, Min-Size)

Compression Rate & Approximation Error. Figure 4.18(a) shows the results on the compression rates of the approximate algorithms. We also show the theoretical bound of the size the trajectory returned by Intersect (Lemma 4.5.3). Figure 4.18(b) shows the results on the approximate errors of the approximate algorithms. Both these results verify our Intersect algorithm.

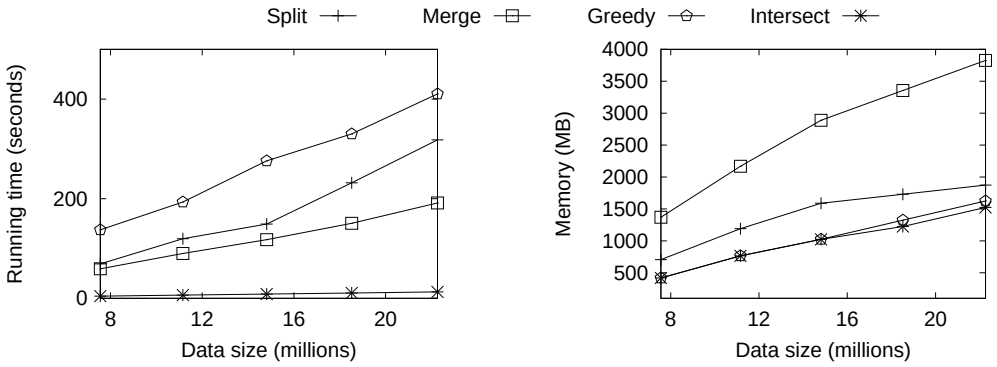
Scalability Test. The largest trajectory in our real datasets contains around 2M positions only. In order to generate larger trajectories, we concatenate multiple trajectories in ascending order of their time stamps into one. Figure 4.19 shows the results of the scalability test on the approximate algorithms. According to these results, Intersect is



(a) Compression rate (b) Approximation error

Figure 4.18: Compression rates and approximate errors of the approximate algoritms (Geolife, Min-Size)

very fast on large datasets with more than 20M positions. For example, Intersect runs in 13.3s on the trajectory with 24,876,978 positions. In contrast, the running times of other approximate algorithms increase much faster when $|T|$ increases.



(a) Running time (b) Memory

Figure 4.19: Scalability test for approximate algorithms (Geolife, Min-Size)

4.7 Conclusion

In this chapter, we propose direction-preserving trajectory simplification, which has not been studied in the literature, as a novel alternative to the traditional position-preserving trajectory simplification. We propose an exact algorithm called *SP* and an approximate algorithm called *Intersect*. We conducted experiments to show the efficiency and the scalability of our proposed methods.

CHAPTER 5

DIRECTION-PRESERVING TRAJECTORY SIMPLIFICATION: MINIMIZING THE ERROR

5.1 Introduction

In Chapter 4, we study the Min-Size problem, which is to simplify a given trajectory such that the error of the simplified trajectory is at most a given *error threshold* and its *size* is minimized. Here, the size of a simplified trajectory is defined to be the total number of positions kept in the trajectory. The Min-Size problem is suitable only when users have clear knowledge about the error tolerance.

In some cases, users might not know how to specify the error tolerance clearly. This could be because the simplified trajectories will be used *in the future* and thus the details are not available at the moment or the simplified trajectories will be used in different applications which might have different accuracy requirements and thus it is not suitable to specify *one* error tolerance for simplifying trajectories. In these cases, a better way is to retain the accuracy as much as possible while achieving a certain degree of compression rate for simplifying trajectories. Specifically, we are given a *storage budget* denoting the greatest size of a simplified trajectory to be stored (note that the storage budget implies a compression rate requirement), and the goal is to minimize the error of the simplified trajectory. We call this problem the *Min-Error* problem which corresponds to the dual problem of the Min-Size problem.

In this thesis, we develop multiple algorithms for the Min-Error problem, both exact and faster approximate algorithms. Specifically, our major contributions are sum-

marized as follows. First, we define a new problem called *Min-Error* which minimizes the simplification error under a storage budget. Second, to solve the Min-Error problem exactly, we explore the idea of *dynamic programming* and *binary search*, resulting in two different algorithms, with the time complexities of $O(Wn^3)$ and $O(n^2C \log n)$, respectively (W is the storage budget, n is the size of the trajectory and C is a small constant). Third, motivated by the relatively high complexities of the exact algorithms, we further develop an approximate algorithm which runs in $O(n \log^2 n)$ time and gives a 2-factor approximation. Fourth, we conducted extensive experiments on real datasets which verified our proposed algorithms.

The remainder of this chapter is organized as follows. Section 5.2 defines the Min-Error problem. Section 5.4 and Section 5.5 introduce our exact and approximate algorithms, respectively. Section 5.6 gives the empirical study. Section 5.3 studies the related work and Section 5.7 concludes the chapter.

5.2 Problem Definition

The concepts and notations are the same as those defined in Section 4.2. We use Figure 5.1 as a running example which would be used throughout this chapter. In Figure 5.1, there is a trajectory $T = (p_1, p_2, ..., p_8)$. T has 8 positions, i.e., $p_1, p_2..., p_8$, and thus the size of T is equal to 8. T has 7 segments, i.e., $\overline{p_1p_2}, \overline{p_2p_3}, ..., \overline{p_7p_8}$, which correspond to the solid line segments in the figure.

The *Min-Error* problem is to simplify a given trajectory such that the error of the simplified trajectory is the smallest and the size of the simplified trajectory is at most a given positive integer W called the *storage budget*. The formal definition is as follows.

Problem 3 (Min-Error) *Given a trajectory T and a positive integer W, **the Min-Error problem** is to find a simplification T' of T such that $|T'| \leq W$ and $\epsilon(T')$ is*

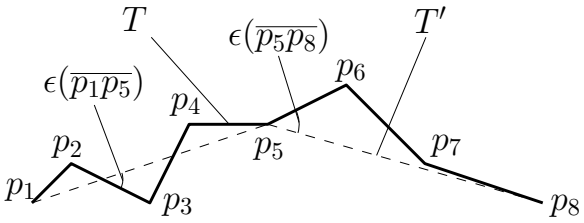


Figure 5.1: A running example (Min-Error)

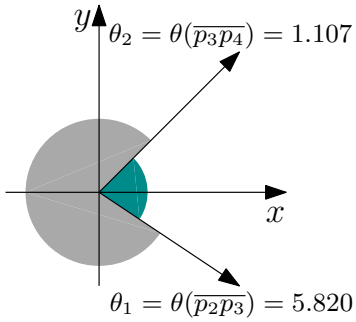


Figure 5.2: An angular range

minimized. □

To illustrate, consider the Min-Error problem with its input trajectory as T in Figure 4.2 and its input storage budget as 3. Then, $T' = (p_1, p_5, p_8)$ is the optimal solution since we cannot find any other simplification of T with its size at most 3 and its error smaller than $\epsilon(T')$ $(= 0.785)$.

5.3 Related Work

As we have discussed in Chapter 4, most existing studies on trajectory simplification aim to preserve the *position information* of the trajectory, which we call *position-preserving trajectory simplification* (PPTS), by adopting a *position-based error measurement* for measuring the error of the simplified trajectory [31, 72, 79, 60, 74, 56]. A position-based error of a simplified trajectory is usually defined to be the maximum Euclidean distance between a position on the original trajectory and its "mapped" position on the simplified trajectory. Two major methods have been proposed to define for a position p on the original trajectory its "mapped" position on the simplified trajectory, namely the *closest distance function* [31], which defines the "mapped" position to be the closest position from p on the simplified trajectory, and the *synchronous distance function* [72, 79, 60, 74, 56], which defines the "mapped" position to be the position

with the same time stamp on the simplified trajectory as p. The algorithms used by these studies are mainly heuristic-based.

Some other existing studies on trajectory simplification include [67] which aims to minimize the area enclosed by the original trajectory and the simplified trajectory, [82, 23] which consider the semantic information of a trajectory for trajectory simplification, [54, 38, 55] which study the trajectory simplification problem on trajectories constrained on road networks, [94, 89, 79, 47, 58, 60, 48, 74, 59, 56] which study the online trajectory simplification problem, [21] which combines the trajectory simplification process and the encoding process for better compression rate, [12, 35] which study the effects of trajectory simplification on some spatio-temporal queries, [73] which provides a preliminary empirical study on several trajectory simplification algorithms, [22] which proposes a multi-resolution trajectory simplification method, and [108] which provides a preliminary literature study on trajectory simplification.

Another closely related topic is *polygonal curve approximation* [3] (a good survey could be found in [44]). However, none of these studies consider the direction-based error as adopted in this thesis.

5.4 Exact Algorithm

Given a simplification T' of T, we say that T' is *affordable* iff $|T'| \leq W$. Let T'_o be the optimal solution of the Min-Error problem and ϵ_o be the error of T'_o, i.e., $\epsilon_o = \epsilon(T'_o)$. Then, T'_o corresponds to one affordable simplification with the smallest error.

Let $\mathcal{F}$ be the set containing all affordable simplifications of T. A naive method for the Min-Error is to perform an exhaustive search over $\mathcal{F}$ and find the one with the smallest error, which, however, is not feasible since the size of $\mathcal{F}$ is exponential in terms of W (specifically, $|\mathcal{F}| = \binom{n-2}{W-2}$). A better way is to design a *dynamic pro-*

gramming algorithm since we have the following *sub-problem optimality property*: If $T' = (p_{s_1}, p_{s_2}, ..., p_{s_m})$ is an optimal solution for the Min-Error problem instance with its input trajectory of T and its input storage budget of W, then $T'' = (p_{s_2}, ..., p_{s_m})$ is also an optimal solution for another Min-Error problem instance with its input trajectory of $T[s_2 : n]$ and its input storage budget of $W - 1$. We call this dynamic programming algorithm *DP*, and since there is not much surprise in the development of *DP*, we omit the details of *DP* here and put them in Section B.3 of the Appendix. Unfortunately, *DP* has a time complexity of $O(Wn^3)$, which is prohibitively expensive on large datasets. Thus, in the following, we design a *binary search* algorithm called *Error-Search* for the Min-Error problem. *Error-Search* has a time complexity of $O(n^2 C \log n)$ (C is usually a small constant) which is significantly smaller than that of *DP*.

Let $\mathcal{E}$ be the set containing all $\epsilon(\overline{p_i p_j})$'s for $1 \leq i < j \leq n$, i.e., $\mathcal{E} = \{\epsilon(\overline{p_i p_j}) | 1 \leq i < j \leq n\}$. Note that $|\mathcal{E}| = O(n^2)$. We observe that the minimized error ϵ_o is contained in $\mathcal{E}$, i.e., $\epsilon_o \in \mathcal{E}$. This could be easily verified by the fact that any simplification has its error equal to the greatest simplification error of its segment, which is covered by $\mathcal{E}$ by definition.

Given a non-negative real value ϵ, we say that ϵ is an *affordable error* if there exists an affordable simplification T' in $\mathcal{F}$ such that $\epsilon(T') \leq \epsilon$. Thus, ϵ_o corresponds to the *smallest* affordable error.

In view of the above discussion, we design an algorithm called *Error-Search* as follows. Firstly, we construct the search space $\mathcal{E}$. Secondly, for each $\epsilon \in \mathcal{E}$, we check whether there exists an affordable simplification T' with $\epsilon(T') \leq \epsilon$ (i.e., we check whether ϵ is an affordable error) which we call the *error affordability check on* ϵ. We note here that we can adopt a *binary search* strategy (instead of a linear scan strategy) for searching on $\mathcal{E}$ since we have the following *monotonicity property*: if ϵ is

an affordable error, then any $\epsilon' > \epsilon$ is also an affordable error. Thirdly, we return as T'_o the affordable simplification corresponding to the smallest affordable error found (which is exactly ϵ_o).

The correctness of *Error-Search* is obvious. In the following, we discuss (1) how to construct the search space $\mathcal{E}$, (2) how to perform the error affordability check on a given ϵ, and (3) the time and space complexities of *Error-Search*.

(1) Construction of $\mathcal{E}$. Recall that $\mathcal{E} = \{\epsilon(\overline{p_i p_j}) | 1 \leq i < j \leq n\}$. Thus, we have $O(n^2)$ instances of $\epsilon(\overline{p_i p_j})$'s in $\mathcal{E}$. A straightforward method for computing $\epsilon(\overline{p_i p_j})$ $(1 \leq i < j \leq n)$ is to compare $\theta(\overline{p_i p_j})$ with $\theta(\overline{p_h p_{h+1}})$ for each $h \in [i, j)$. This method, though simple, incurs the worst-case cost of $O(n)$. Thus, the overall cost of constructing $\mathcal{E}$ based on this method is $O(n^3)$, which is too costly. In the following, we develop a more efficient method for computing $\epsilon(\overline{p_i p_j})$ $(1 \leq i < j \leq n)$ which runs in $O(\log n)$ time only instead of $O(n)$ time, resulting in the overall cost of constructing $\mathcal{E}$ being $O(n^2 \log n)$.

Our method is based on the concept of "opposite direction" which will be described in detail next. Recall that $\epsilon(\overline{p_i p_j})$ corresponds to the *greatest* angular difference between $\theta(\overline{p_i p_j})$ and a direction in $\theta[i : j]$. Thus, computing $\epsilon(\overline{p_i p_j})$ could be finished by finding the direction in $\theta[i : j]$ which has the *greatest* angular difference from $\theta(\overline{p_i p_j})$. Let θ^* denote this direction. With θ^*, we can easily compute $\epsilon(\overline{p_i p_j})$ by computing the angular difference between $\theta(\overline{p_i p_j})$ and θ^* with Equation (4.1). In the following, we focus on how to find θ^*.

Let $\theta(\overline{p_i p_j})^-$ be the *opposite direction* of $\theta(\overline{p_i p_j})$, i.e., $\theta(\overline{p_i p_j})^- = [(\theta(\overline{p_i p_j}) + \pi) \mod 2\pi]$. We observe that θ^* is exactly the direction in $\theta[i : j]$ which has the *smallest* angular difference from $\theta(\overline{p_i p_j})^-$. This is simply because any direction θ in $\theta[i : j]$ has its angular difference from $\theta(\overline{p_i p_j})$ equal to π minus its angular difference from $\theta(\overline{p_i p_j})^-$, i.e., $\triangle(\theta, \theta(\overline{p_i p_j})) = \pi - \triangle(\theta, \theta(\overline{p_i p_j})^-)$.

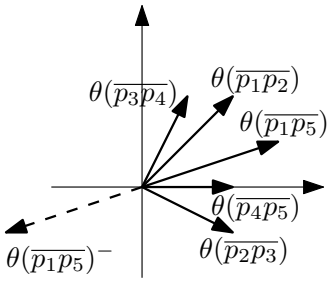


Figure 5.3: Opposite direction

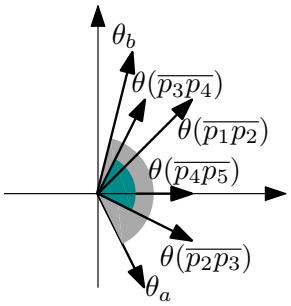


Figure 5.4: $mcar(\cdot)$

To illustrate, consider Figure 5.3. $\theta(\overline{p_1p_2})$, $\theta(\overline{p_2p_3})$, $\theta(\overline{p_3p_4})$, and $\theta(\overline{p_4p_5})$ correspond to $\theta[1:5]$. $\theta(\overline{p_1p_5})$ and $\theta(\overline{p_1p_5})^-$ are also shown. As could be verified, $\theta(\overline{p_2p_3})$ is the direction in $\theta[1:5]$ which has the *greatest* angular difference from $\theta(\overline{p_1p_5})$ and also the *smallest* angular difference from $\theta(\overline{p_1p_5})^-$.

Thus, we propose to search θ^* with two steps. First, we sort the directions in $\theta[i:j]$ in ascending order and let $\theta_1, \theta_2, ..., \theta_{j-i}$ be the resulting sorted list (note that sorting from scratch incurs a cost of $O(n \log n)$ here, and what we do is to *incrementally* maintain the sorted list of $\theta[i:j]$ based on the one of $\theta[i:j-1]$ which has already been maintained for computing $\epsilon(\overline{p_ip_{j-1}})$ if we compute $\epsilon(\overline{p_ip_{j-1}})$ first, and thus this step could be finished in $O(\log n)$ time). Second, we find the direction in the sorted list which has the smallest angular difference from $\theta(\overline{p_ip_j})^-$ (i.e., θ^*) and this step could also be done in $O(\log n)$ time with a binary search process based on the sorted list. In combination of the first step and the second step, our method finds θ^* in $O(\log n)$ time.

In view of the above discussion, we know that $\mathcal{E}$ could be constructed in $O(n^2 \log n)$ time since we have $O(n^2)$ instances of $\epsilon(\overline{p_ip_j})$ each with a computation cost of $O(\log n)$.

(2) Error Affordability Check on ϵ. Given a value ϵ, the task is to check whether there exists an affordable simplification T' in $\mathcal{F}$ with $\epsilon(T') \leq \epsilon$. A linear scan method over $\mathcal{F}$ is not feasible since the size of $\mathcal{F}$ is exponential. In the following, we propose a method which runs in $O(n^2 C)$ time where C is usually a small constant.

Lemma 5.4.1 *Let ϵ be a non-negative value and T' be a simplification of T with its error at most ϵ and its size minimized. Then, ϵ is an affordable error iff T' is affordable.*

□

Proof. "⇒": Suppose that ϵ is an affordable error, i.e., there exists an affordable simplification T'' with $\epsilon(T'') \leq \epsilon$. We have $|T'| \leq |T''| \leq W$, and thus we know that T' is affordable.

"⇐": Clearly, ϵ is an affordable error since T' is affordable and has its error at most ϵ by definition. ■

Lemma 5.4.1 suggests that the error affordability check on a given value ϵ can be implemented with the following two steps. First, we compute the simplification T' of T with its error at most ϵ and its size minimized. This essentially corresponds to solving a *Min-Size* problem instance with its input trajectory as T and its input error tolerance as ϵ. Thus, this step can be done by executing an exact algorithm of the Min-Size problem. Second, we check whether T' is affordable (i.e., the size of T' is at most W). If yes, then ϵ is an affordable span. Otherwise, it is not. According to the results in Chapter 4, the time complexity of the exact algorithm for the Min-Size problem is $O(n^2C)$ (where C is usually a small constant, e.g., $C = 1$ if $\epsilon \leq \pi/2$), and hence we know that the time complexity of the above method of performing an error affordability check is also $O(n^2C)$.

(3) Time & Space Complexity of *Error-Search*. In conclusion, the time complexity of *Error-Search* is $O(n^2C \log n)$ since the cost of constructing $\mathcal{E}$ is $O(n^2 \log n)$, the cost of sorting $\mathcal{E}$ (for binary search) is $O(n^2 \log n^2)$ $(= O(n^2 \log n))$, and the cost of performing the error affordability check $O(\log n^2)$ times (in the binary search) is $O(n^2C \cdot \log n^2)$ $(= O(n^2C \log n))$. The space complexity of *Error-Search* is $O(n^2)$ which corresponds to the space cost of storing the search space $\mathcal{E}$.

5.5 Approximate Algorithm

In this section, we present our approximate algorithm called *Span-Search* for the Min-Error problem which runs in $O(n \log^2 n)$ time and gives a 2-factor approximation. Specifically, in Section 5.5.1, we introduce an *estimator* of the error of a simplification of T called *span*. In Section 5.5.2, based on this estimator, we define a new problem called *Min-Span* whose optimal solution corresponds to a 2-factor approximation of the Min-Error problem. In Section 5.5.3, we give an overview of *Span-Search* which returns the optimal solution of the Min-Span problem in $O(n \log^2 n)$ time. In Section 5.5.4, we give the details of *Span-Search* and analyze its time and space complexities.

5.5.1 An Estimator of Error

We define the *span* of an angular range $[\theta_1, \theta_2]$, denoted by $\xi([\theta_1, \theta_2])$, to be equal to the angle of an *anti-clockwise* rotation from a vector with its direction equal to θ_1 to another vector with its direction equal to θ_2. Specifically, we have

$$\xi([\theta_1, \theta_2]) = \begin{cases} \theta_2 - \theta_1 & \text{if } \theta_2 \geq \theta_1 \\ 2\pi - (\theta_1 - \theta_2) & \text{if } \theta_2 < \theta_1 \end{cases} \tag{5.1}$$

Note that $\xi([\theta_1, \theta_2])$ is non-negative, and for any θ_1 and θ_2 in $[0, 2\pi)$, we have $\xi([\theta_1, \theta_2]) + \xi([\theta_2, \theta_1]) = 2\pi$.

To illustrate, consider Figure 5.2 where we have $\theta_1 = 5.820$ and $\theta_2 = 1.107$. Thus, we know $\xi([\theta_1, \theta_2]) = \xi([5.820, 1.107]) = 2\pi - (5.820 - 1.107) = 1.570$ and $\xi([\theta_2, \theta_1]) = \xi([1.107, 5.820]) = 5.820 - 1.107 = 4.713$.

Let $\mathcal{D}$ be the set of the directions of all possible segments in T, i.e., $\mathcal{D} = \theta[1 : n]$. Note that $|\mathcal{D}| = n-1$. Given a set $\mathcal{D}' \subseteq \mathcal{D}$, any angular range that covers *all* directions in $\mathcal{D}'$ is said to be a *covering angular range* of $\mathcal{D}'$. Among all covering angular ranges

of $\mathcal{D}'$, the one with the *smallest* span is called the *minimum covering angular range* of $\mathcal{D}'$ which we denote by $mcar(\mathcal{D}')$.

To illustrate, consider Figure 5.4 where we show $\mathcal{D}' = \theta[1 : 5] = \{\theta(\overline{p_1p_2}), \theta(\overline{p_2p_3}), \theta(\overline{p_3p_4}), \theta(\overline{p_4p_5})\}$ and two other directions θ_a and θ_b. Then, $[\theta_a, \theta_b]$ (see the sector area in lighter color) is a covering angular range of $\mathcal{D}'$ since all directions in $\mathcal{D}'$ fall in $[\theta_a, \theta_b]$. Besides, the minimum covering angular range of $\mathcal{D}'$, i.e., $mcar(\mathcal{D}')$, is $[\theta(\overline{p_2p_3}), \theta(\overline{p_3p_4})]$ (see the sector area in darker color) since $[\theta(\overline{p_2p_3}), \theta(\overline{p_3p_4})]$ covers all directions in $\mathcal{D}'$ (i.e., $[\theta(\overline{p_3p_4}), \theta(\overline{p_5, p_6})]$ is a covering angular range of $\mathcal{D}'$) and there exists no other covering angular range of $\mathcal{D}'$ with its span smaller than that of $[\theta(\overline{p_2p_3}), \theta(\overline{p_3p_4})]$ $(= 1.570)$ (See Figure 5.2).

Note that the two boundaries of $mcar(\mathcal{D}')$ always come from $\mathcal{D}'$ since otherwise the range could be shrunk further and it does not have the minimum span.

Let $T = (p_1, p_2, ..., p_n)$ be a trajectory and $T' = (p_{s_1}, p_{s_2}, ..., p_{s_m})$ be a simplification of T. The *span* of T', denoted by $\xi(T')$, is defined to be the greatest span of the minimum covering angular ranges of $\theta[s_k : s_{k+1}]$ where $k \in [1, m)$, i.e.,

$$\xi(T') = \max_{1 \leq k < m}\{\xi(mcar(\theta[s_k : s_{k+1}]))\} \tag{5.2}$$

To illustrate, consider our running example in Figure 4.2. $T' = (p_1, p_5, p_8)$ is a simplification of T. As mentioned before, $mcar(\theta[1 : 5]) = [\theta(\overline{p_2p_3}), \theta(\overline{p_3p_4})] = [5.821, 1.107]$ and thus $\xi(mcar(\theta[1 : 5])) = \xi([5.821, 1.107]) = 1.570$. Besides, $mcar(\theta[5 : 8]) = [\theta(\overline{p_6p_7}), \theta(\overline{p_5p_6})] = [5.498, 0.464]$ and thus $\xi(mcar(\theta[5 : 8])) = \xi([5.498, 0.464]) = 1.249$. Therefore, $\xi(T') = \max\{\xi(mcar(\theta[1 : 5])), \xi(mcar(\theta[5 : 8]))\} = \max\{1.570, 1.249\} = 1.570$.

5.5.2 The Min-Span Problem

In this part, we define a problem called *Min-Span* which is quite similar to *Min-Error*, but with a different objective.

Problem 4 (Min-Span) *Given a trajectory T and a positive integer W, the **Min-Span** problem is to find a simplification T' of T such that $|T'| \leq W$ and $\xi(T')$ is minimized.*

□

To illustrate, consider a Min-Span problem instance with its input trajectory as T in Figure 4.2 and its input W as 3. It could be verified that $T' = (p_1, p_5, p_8)$ is the optimal solution of this problem instance since we cannot find any other simplification of T which has its size at most 3 and its span smaller than $\xi(T')$ $(= 1.570)$.

Interestingly, the *optimal* solution of the Min-Span problem is a *2-factor approximation* of the Min-Error problem.

Lemma 5.5.1 *Let T'_o be the optimal solution of the Min-Error problem with its input trajectory as T and its input storage budget as W. Let T'_ξ be the optimal solution of the Min-Span problem with its input trajectory and its input storage budget both the same as the Min-Error problem. Then, $\epsilon(T'_\xi) \leq 2 \cdot \epsilon(T'_o)$.* □

Proof. This proof is divided into two parts. In the first part, we show that any simplification $T' = (p_{s_1}, p_{s_2}, ..., p_{s_m})$ of T satisfies $\frac{\xi(T')}{\epsilon(T')} \in [1, 2]$ which we prove with two steps.

First, we show that for any $k \in [1, m)$, we have

$$\frac{\epsilon(\overline{p_{s_k} p_{s_{k+1}}})}{\xi(mcar(\theta[s_k : s_{k+1}]))} \in [1/2, 1] \tag{5.3}$$

Suppose that $mcar(\theta[s_k : s_{k+1}])$ is $[\theta_a, \theta_b]$. Note that θ_a and θ_b are two directions in $\theta[s_k : s_{k+1}]$. We have two cases.

Case 1: $\xi([\theta_a, \theta_b]) \leq \pi$. For illustration, consider Figure 5.5(a). In this case, $\theta(\overline{p_{s_k} p_{s_{k+1}}})$ is covered by $[\theta_a, \theta_b]$. Therefore, we have

$$\begin{aligned}
&\epsilon(\overline{p_{s_k} p_{s_{k+1}}}) = \max\{\triangle(\theta(\overline{p_{s_k} p_{s_{k+1}}}), \theta_a), \triangle(\theta(\overline{p_{s_k} p_{s_{k+1}}}), \theta_b)\} \\
&\in [1/2, 1] \cdot (\triangle(\theta(\overline{p_{s_k} p_{s_{k+1}}}), \theta_a) + \triangle(\theta(\overline{p_{s_k} p_{s_{k+1}}}), \theta_b)) \\
&= [1/2, 1] \cdot \xi([\theta_a, \theta_b])
\end{aligned}$$

Case 2: $\xi([\theta_a, \theta_b]) > \pi$. In this case, $\theta(\overline{p_{s_k} p_{s_{k+1}}})$ could be or not be covered by $[\theta_a, \theta_b]$. We further consider two sub-cases.

Case 2(a): $\theta(\overline{p_{s_k} p_{s_{k+1}}})$ is covered by $[\theta_a, \theta_b]$. For illustration, consider Figure 5.5(b). The proof of this case is similar to the one of Case 1 and thus it is omitted here.

Case 2(b): $\theta(\overline{p_{s_k} p_{s_{k+1}}})$ is not covered by $[\theta_a, \theta_b]$. Then, $\theta(\overline{p_{s_k} p_{s_{k+1}}})$ is covered by $[\theta_b, \theta_a]$. For illustration, consider Figure 5.5(c). Let θ_c and θ_d be two directions in $\theta[s_k : s_{k+1}]$ such that θ_c and θ_d are in $[\theta_a, \theta_b]$ and no directions in $\theta[s_k : s_{k+1}]$ other than θ_c and θ_d are in $[\theta_c, \theta_d]$ (Note that θ_c and θ_d always exist). Then, we know that $[\theta_d, \theta_c]$ corresponds to a covering angular range of $\theta[s_k : s_{k+1}]$ and $\theta(\overline{p_{s_k} p_{s_{k+1}}})$ falls in $[\theta_d, \theta_c]$. Besides, we know $\xi([\theta_d, \theta_c]) \geq \xi([\theta_a, \theta_b])$ since $[\theta_a, \theta_b]$ is the minimum covering angular range of $\theta[s_k : s_{k+1}]$. Similar to Case 1, we have $\frac{\epsilon(\overline{p_{s_k} p_{s_{k+1}}})}{\xi([\theta_d, \theta_c])} \in [1/2, 1]$ which implies that $\epsilon(\overline{p_{s_k} p_{s_{k+1}}}) \geq \frac{1}{2} \cdot \xi([\theta_d, \theta_c]) \geq \frac{1}{2} \cdot \xi([\theta_a, \theta_b])$. Furthermore, we have $\epsilon(\overline{p_{s_k} p_{s_{k+1}}}) \leq \pi < \xi([\theta_a, \theta_b])$. In combination, we have $\frac{\epsilon(\overline{p_{s_k} p_{s_{k+1}}})}{\xi([\theta_a, \theta_b])} \in [\frac{1}{2}, 1]$.

Second, Let $k' = \arg\max_{k \in [1,m)}\{\xi(mcar(\theta[s_k : s_{k+1}]))\}$ and $k'' = \arg\max_{k \in [1,m)}\{\epsilon(\overline{p_{s_k} p_{s_{k+1}}})\}$. By using Equation (5.3), we have

$$\begin{aligned}
\xi(T') &= \xi(mcar(\theta[s_{k'} : s_{k'+1}])) \leq 2 \cdot \epsilon(\overline{p_{s_{k'}} p_{s_{k'+1}}}) \\
&\leq 2 \cdot \epsilon(\overline{p_{s_{k''}} p_{s_{k''+1}}}) = 2 \cdot \epsilon(T') \qquad (5.4)
\end{aligned}$$

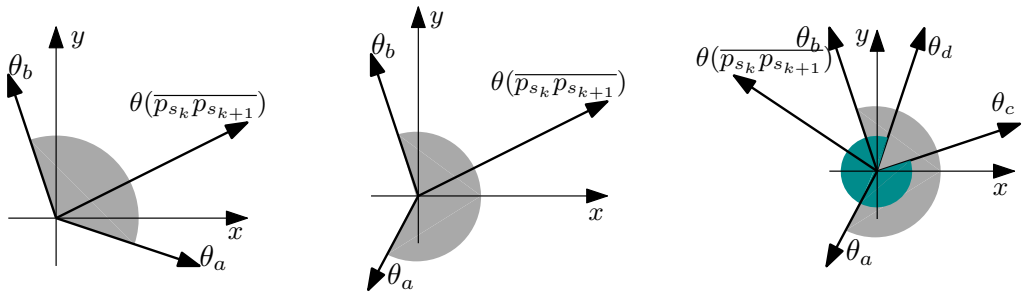


Figure 5.5: Proof of Lemma 5.5.1

and

$$\xi(T') = \xi(mcar(\theta[s_{k'} : s_{k'+1}])) \geq \xi(mcar(\theta[s_{k''} : s_{k''+1}]))$$
$$\geq \epsilon(\overline{p_{s_{k''}}p_{s_{k''+1}}}) = \epsilon(T') \tag{5.5}$$

By using Equations (5.4) and (5.5), we obtain $\frac{\xi(T')}{\epsilon(T')} \in [1, 2]$.

In the second part, we show that $\epsilon(T'_\xi) \leq 2 \cdot \epsilon(T'_o)$ as follows.

$$\epsilon(T'_\xi) \leq \xi(T'_\xi) \leq \xi(T'_o) \leq 2 \cdot \epsilon(T'_o)$$

■

5.5.3 The Span-Search Algorithm: Overview

In this part, we develop an algorithm called *Span-Search* which returns the optimal solution of the Min-Span problem in $O(n \log^2 n)$ time and thus gives a 2-factor approximation for the Min-Error problem (Lemma 5.5.1).

Let T'_ξ be the optimal solution of the Min-Span problem and ξ_o be the span of T'_ξ. Essentially, T'_ξ corresponds to the affordable simplification with the smallest span. *Span-Search* first maintains a search space $\mathcal{S}$ containing ξ_o and then searches ξ_o over $\mathcal{S}$ (T'_ξ can also be retrieved when ξ_o is found).

Concepts & Search Space $\mathcal{S}$

We introduce some concepts used for defining the search space $\mathcal{S}$ and then give a precise definition of $\mathcal{S}$. Suppose that T'_ξ is $(p_{s_1}, p_{s_2}, ..., p_{s_m})$. From Equation (5.2), we know $\xi(T'_\xi) = \max_{1 \le k < m}\{\xi(mcar(\theta[s_k : s_{k+1}]))\}$. Let $k^* = \arg\max_{1 \le k < m}\{\xi(mcar(\theta[s_k : s_{k+1}]))\}$. Then, we have $\xi_o = \xi(T'_\xi) = \xi(mcar(\theta[s_{k^*} : s_{k^*+1}]))$. Consider $\xi(mcar(\theta[s_{k^*} : s_{k^*+1}]))$. Let $[\theta, \theta']$ be $mcar(\theta[s_{k^*} : s_{k^*+1}])$. Note that θ and θ' are two directions in $\theta[s_{k^*} : s_{k^*+1}]$. Then, we derive that $\xi_o = \xi([\theta, \theta']))$. By Equation (5.1), we have

$$\xi_o = \begin{cases} \theta' - \theta & \text{if } \theta' \ge \theta \\ 2\pi - (\theta - \theta') & \text{if } \theta' < \theta \end{cases} \tag{5.6}$$

Essentially, θ and θ' could be the directions of any two segments of T. Thus, we have the following observation.

Observation 2 (Pairwise Direction Difference) *Let ξ_o be the optimal span of the Min-Span problem. There exist two segments of T such that ξ_o is equal to either $\theta' - \theta$ or $2\pi - (\theta - \theta')$ where θ and θ' are the directions of the two segments.* □

Based on the above observation, we construct an $(n-1) \times (n-1)$ matrix Θ containing both $\theta' - \theta$ and $2\pi - (\theta - \theta')$ for each possible pair $(\theta, \theta') \in \mathcal{D} \times \mathcal{D}$, where $\mathcal{D}$ is the set of the directions of all possible segments of T, and define the search space $\mathcal{S}$ to be the multi-set of all values in the matrix Θ. Note that the size of $\mathcal{S}$ is $(n-1)^2 = O(n^2)$. Specifically, Θ is defined as follows. Let $L = (\theta_1, \theta_2, ..., \theta_{n-1})$ be the sorted list of the values in $\mathcal{D}$ in *ascending* order. For each $i \in [1, n-1]$ and each $j \in [1, n-1]$, we define

$$\Theta[i][j] = \begin{cases} \theta_j - \theta_i & \text{if } j \ge i \\ 2\pi - (\theta_i - \theta_j) & \text{if } j < i \end{cases} \tag{5.7}$$

	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6	θ_7
L:	0	0.464	0.785	1.107	5.498	5.820	5.961

Table 5.1: Sorted list of the directions in $\theta[1:8]$

0	0.464	0.785	1.107	5.498	5.820	5.961
5.819	0	0.321	0.643	5.034	5.356	5.497
5.498	5.962	0	0.322	4.713	5.035	5.176
5.176	5.640	5.961	0	4.391	4.713	4.854
0.785	1.249	1.570	1.892	0	0.322	0.463
0.463	0.927	1.248	1.570	5.961	0	0.141
0.322	0.786	1.107	1.429	5.820	6.142	0

Table 5.2: Matrix Θ defined by Equation (5.7)

$\Theta[1:1][1]$	$\Theta[1:2][2]$	$\Theta[1:3][3]$	$\Theta[1:4][4]$	$\Theta[1:5][5]$	$\Theta[1:6][6]$	$\Theta[1:7][7]$
$\Theta[2:7][1]$	$\Theta[3:7][2]$	$\Theta[4:7][3]$	$\Theta[5:7][4]$	$\Theta[6:7][5]$	$\Theta[7:7][6]$	

Table 5.3: The array set representing matrix Θ

(1,1,1), (1,2,2), (1,3,3), (1,4,4), (1,5,5), (1,6,6), (1,7,7)
(2,7,1), (3,7,2), (4,7,3), (5,7,4), (6,7,5), (7,7,6)

Table 5.4: The index triplet set (for the original search space $\mathcal{S}$)

~~(1,**0**,1)~~, (1,**1**,2), (1,**1**,3), (1,**2**,4), (1,5,5), (1,6,6), (1,7,7)
(2,**4**,1), (3,**4**,2), (4,**5**,3), (5,7,4), (6,7,5), (7,7,6)

Table 5.5: The index triplet set (for the updated search space resulted from the pruning based on pivot $\xi = 1.249$)

To illustrate, consider Table 5.1 which shows the sorted list of $\mathcal{D}$ of the trajectory T in Figure 4.2 and Table 5.2 which shows the corresponding matrix Θ.

With Observation 2, it is easy to verify that ξ_o is in $\mathcal{S}$. We present this result in the following lemma.

Lemma 5.5.2 *The span of the optimal solution of the Min-Span problem (i.e., ξ_o) is in $\mathcal{S}$.* □

For example, as mentioned before, for the Min-Span problem with its input trajectory as T presented in Figure 4.2 and its input W as 3, ξ_o is equal to 1.570 which corresponds to $\Theta[6][4]$.

Given a value ξ, we say that ξ is an *affordable span* iff there exists an affordable simplification of T with its span at most ξ. It immediately follows that ξ_o corresponds to the *smallest* affordable span. With Lemma 5.5.2, we know that ξ_o is the smallest affordable span in $\mathcal{S}$.

Strategy of Searching over $\mathcal{S}$

After we introduced the concepts and defined the search space $\mathcal{S}$ in the previous section, in this section, we present a strategy called *Span-Search* for finding the optimal span ξ_o on $\mathcal{S}$. Given a value ξ, we call the procedure of checking whether ξ is an affordable span the *span affordability check on* ξ. This procedure, when called with an input of ξ, also returns an affordable simplification T' with $\xi(T') \leq \xi$ if ξ is an affordable span.

As we described before, we know that ξ_o is the smallest affordable span in $\mathcal{S}$. Thus, we propose to find ξ_o with three steps.

- **Step 1 (Searching Step):** Step 1 is to find a value ξ from $\mathcal{S}$ and perform a span affordability check on ξ. If ξ is an affordable span, it also obtains an affordable simplification T'. Let ξ_{best} be a variable denoting the best-known affordable span in $\mathcal{S}$ (i.e., the smallest affordable span in $\mathcal{S}$ seen so far), initialized to ∞. Let T'_{best} be a variable denoting the simplified trajectory with its span at most ξ_{best}. If ξ is an affordable span and $\xi < \xi_{best}$, it updates ξ_{best} and T'_{best} with ξ and T', respectively.

- **Step 2 (Iterative Step):** Step 2 is to perform Step 1 iteratively with one of the remaining values in $\mathcal{S}$ to be found until there is no remaining value in $\mathcal{S}$.

- **Step 3 (Output Step):** Step 3 is to return ξ_{best} and T'_{best}.

A simple strategy of implementing Step 1 called *Random-Search* is to select a *ran-*

dom value from $\mathcal{S}$ as ξ. The algorithm with this strategy is too costly since $\mathcal{S}$ involves $O(n^2)$ values and thus the algorithm needs to perform $O(n^2)$ span affordability checks.

Another strategy of implementing Step 1 called *Binary-Search* is to always select the *median* of the values in the current search space as ξ since the result of the span affordability check on the median could be used to prune at least half of the current search space due to the following *monotonicity* property.

Property 5 (Monotonicity) *Let ξ and ξ' be two real numbers where $\xi < \xi'$. If ξ is an affordable span, then ξ' is also an affordable span.* □

Specifically, if ξ is an affordable span, we can prune all values at least ξ in the current search space; otherwise, we can prune all values at most ξ in the current search space. Although the algorithm with the *Binary-Search* strategy performs $2 \log n = O(\log n)$ span affordability checks only, it is still not scalable (since it needs to materialize a search space $\mathcal{S}$ which occupies $O(n^2)$ space) and too costly (since it introduces extra cost for finding the medians which takes $O(n^2 \log n)$ time[1]).

In this thesis, we propose a new strategy called *Span-Search*. which differs from *Random-Search* and *Binary-Search* as follows.

- *Span-Search* does not materialize the search space $\mathcal{S}$ *explicitly* as *Linear-Search* and *Binary-Search* do, instead, it materializes a concise representation of $\mathcal{S}$ called *index triplet set* (the details will be introduced in Part I of Section 5.5.4) which occupies $O(n)$ space only.

- *Span-Search* performs the span affordability check always on a *pivot* wrt the current search space (the details will be introduced in Part II of Section 5.5.4)

[1]One can either sort the values in $\mathcal{S}$ at the right beginning with $O(n^2 \log n^2) = O(n^2 \log n)$ time and then pick the medians each with $O(1)$ time afterwards or run a *median selection* algorithm [7] which returns the median of N values with $O(N)$ time whenever a median is required. Both take $O(n^2 \log n)$ time.

Algorithm 9 *Span-Search*

1: Initialize the *index triplet set* $\mathcal{T}$ of $\mathcal{S}$ (Part I of Section 5.5.4)
2: //Steps 1 & 2
3: **while** there exist values in the current search space represented by $\mathcal{T}$ **do**
4: Find a *pivot* ξ wrt the current search space (Part II of Section 5.5.4)
5: Perform a span affordability check on ξ (Part III of Section 5.5.4)
6: Update ξ_{best} and T'_{best} if necessary
7: Prune the search space with ξ by updating $\mathcal{T}$ (Part IV of Section 5.5.4)
8: //Step 3
9: **return** ξ_{best} and T'_{best}

at Step 1, which is different from *Random-Search* (on an *random* value from the current search space) or *Binary-Search* (on the *median* of the current search space). The details of how to perform a span affordability check on a given value will be introduced in Part III of Section 5.5.4.

- *Span-Search* prunes at least $\frac{1}{4}$ of the current search space after each span affordability check (whose details will be introduced in Part IV of Section 5.5.4). This implies that *Span-Search* needs to perform $O(\log n)$ span affordability checks only.

- *Span-Search* has the time complexity of $O(n \log^2 n)$ and the space complexity of $O(n)$ both superior over those of *Linear-Search* and *Binary-Search* (the details will be discussed in Part V of Section 5.5.4).

The pseudo-code of *Span-Search* is given in Algorithm 9.

5.5.4 The Span-Search Algorithm: Details

In this section, we give the details of *Span-Search*.

Part I: Concise Representation of $\mathcal{S}$

In this part, we introduce our *index triplet set* which can concisely represent the search space $\mathcal{S}$ with $O(n)$ space (note that a full materialization of $\mathcal{S}$ occupies $O(n^2)$ space).

We introduce some related concepts first. Given an l-sized array X and two integers $i, i' \in [1, l]$, if $i < i'$, then $X[i]$ is said to be *before* the position of $X[i']$ in the array X, and $X[i']$ is said to be *after* the position of $X[i]$ in the array X. If $i = i'$, $X[i]$ is said to be *at* the position of $X[i']$ in the array X.

Since $\mathcal{S}$ is the multi-set containing all values in matrix Θ, we focus on describing how to represent Θ concisely.

For each s, e and $j \in [1, n-1]$ where $s \leq e$, we denote by $\Theta[s:e][j]$ the array containing the values between the s^{th} position and the e^{th} position in the j^{th} column of Θ, i.e., $\Theta[s:e][j] = \{\Theta[s][j], \Theta[s+1][j], ..., \Theta[e][j]\}$.

For each column of Θ, say, $\Theta[1:n-1][j]$ where $j \in [1, n-1]$, which itself is an array, we maintain it with two arrays, namely $\Theta[s_1:e_1][j]$ and $\Theta[s_2:e_2][j]$, where $s_1 = 1$, $e_1 = j$, $s_2 = j+1$, and $e_2 = n-1$ (note that the $(n-1)^{th}$ column corresponds to one array (i.e., $\Theta[1:n-1][n-1]$) only). As a result, the values in Θ are organized with $2(n-1)-1$ $(= O(n))$ arrays each in the form of $\Theta[s:e][j]$. Let $\mathcal{A}$ be the set containing all these arrays. Thus, the size of $\mathcal{A}$ is $O(n)$. Note that the multi-set of values of the arrays in $\mathcal{A}$ is exactly equal to $\mathcal{S}$. In the following, for clarity, when we write $\mathcal{A}$, we mean the array set corresponding to the matrix Θ of the search space $\mathcal{S}$.

To illustrate, consider Table 5.2 where each column is divided into two arrays: one with white background and the other one with gray background. Table 5.3 shows the corresponding array set $\mathcal{A}$.

A nice feature about $\mathcal{A}$ is that all arrays in $\mathcal{A}$ are *non-increasing* [2] which could be verified easily by using Equation (5.7) and the fact that for each $i, i' \in [1, n-1]$ where $i \leq i'$, we have $\theta_i \leq \theta_{i'}$.

Property 6 (Non-Increasing Arrays) *Each array in $\mathcal{A}$ is non-increasing.* □

[2]Given an l-sized array X where l is a positive integer, X is said to be *non-increasing* if for each $i, i' \in [1, l]$ where $i < i'$, $X[i] \geq X[i']$.

To illustrate, consider the arrays in Table 5.3 and their corresponding content shown in Table 5.2. It could be easily noticed that all the arrays are non-increasing.

We have introduced the concepts used to define the index triplet set. Let $\mathcal{S}$ be the search space and $\mathcal{A}$ be the corresponding array set. The *index triplet set* of $\mathcal{S}$, denoted by $\mathcal{T}$, is the set containing triplets of the indices of all arrays in $\mathcal{A}$. That is,

$$\mathcal{T} = \{(s, e, j) | \Theta[s : e][j] \in \mathcal{A}\} \tag{5.8}$$

Note that each triplet in $\mathcal{T}$ identifies an array in $\mathcal{A}$ concisely. The space complexity of $\mathcal{T}$ is $O(n)$ only since we have $O(n)$ arrays in $\mathcal{A}$ each with its space cost of $O(1)$ in $\mathcal{T}$. For example, Table 5.4 shows the index triplet set corresponding the array set shown in Table 5.3.

Interestingly, $\mathcal{T}$ alone concisely represents the multi-set of values of the arrays in $\mathcal{A}$ (or the search space $\mathcal{S}$). This is because for each triplet $(s, e, j) \in \mathcal{T}$, we know that *conceptually*, we have $\Theta[i][j]$ where $i = s, s + 1, ..., e$. We do not materialize the content of $\Theta[i][j]$ explicitly since given the indices (i.e., i and j), the content can be retrieved in $O(1)$ time by using Equation (5.7). Instead, we materialize $\mathcal{T}$ only. In the following, for the sake of convenience, we refer $\mathcal{A}$ instead of $\mathcal{T}$ to represent the entire search space $\mathcal{S}$ (though $\mathcal{T}$ is the materialized version for $\mathcal{A}$).

Part II: Definition, Search Space & Retrieval of a Pivot

In this part, we answer three questions: (1) *what is a pivot*, (2) *where can we find a pivot* and (3) *how to find a pivot*.

(1) What is a pivot? Before we define what is a *pivot*, we introduce a concept called *bisector* and its related concepts.

For each array $\Theta[s : e][j]$ in $\mathcal{A}$, we define its *bisector*, denoted by $b(s, e, j)$, to be $\Theta[\lceil \frac{s+e}{2} \rceil][j]$. Since $\Theta[s : e][j]$ is non-increasing, we know that at least half of the

values in $\Theta[s : e][j]$ are at most its bisector $b(s, e, j)$ and at least half of the values in $\Theta[s : e][j]$ are at least its bisector $b(s, e, j)$. For example, the bisector of array $\Theta[3 : 7][2]$ is $\Theta[5][2]$ which is equal to 1.249 (See Table 5.2).

For a given value $\xi \in \mathcal{S}$, the arrays in $\mathcal{A}$ could be categorized into three disjoint groups, namely the group containing those arrays with the bisectors strictly *smaller than* ξ which we denote by $\mathcal{A}(\xi, -)$, the group containing those arrays with the bisectors exactly *equal to* ξ which we denote by $\mathcal{A}(\xi, =)$, and the group containing those arrays with the bisectors strictly *larger than* ξ which we denote by $\mathcal{A}(\xi, +)$. Let $N(\mathcal{A}(\xi, -))$, $N(\mathcal{A}(\xi, =))$, and $N(\mathcal{A}(\xi, +))$ be the size of the multi-set of the values of the arrays in $\mathcal{A}(\xi, -)$, $\mathcal{A}(\xi, =)$, and $\mathcal{A}(\xi, +)$, respectively. Note that $N(\mathcal{A}(\xi, -)) + N(\mathcal{A}(\xi, =)) + N(\mathcal{A}(\xi, +)) = |\mathcal{S}|$.

To illustrate, consider the array set $\mathcal{A}$ shown in Table 5.3. Suppose $\xi = 1.249$. Then, we know $\mathcal{A}(\xi, -) = \{\Theta[1 : 1][1], \Theta[2 : 7][1], \Theta[1 : 2][2], \Theta[1 : 3][3], \Theta[4 : 7][3], \Theta[1 : 4][4]\}$, $\mathcal{A}(\xi, =) = \{\Theta[3 : 7][2]\}$, and $\mathcal{A}(\xi, +) = \{\Theta[5 : 7][4], \Theta[1 : 5][5], \Theta[6 : 7][5], \Theta[1 : 6][6], \Theta[7 : 7][6], \Theta[1 : 7][7]\}$. As a result, we have $N(\mathcal{A}(\xi, -)) = 20$, $N(\mathcal{A}(\xi, =)) = 5$, and $N(\mathcal{A}(\xi, +)) = 24$. Note that $N(\mathcal{A}(\xi, -)) + N(\mathcal{A}(\xi, =)) + N(\mathcal{A}(\xi, +)) = 49 = |\mathcal{S}|$.

Now, we are ready to define what is a *pivot*.

Definition 5.5.1 (Pivot) *Given a value $\xi \in \mathcal{S}$, ξ is defined to be a* pivot *wrt $\mathcal{S}$ if* $\min\{N(\mathcal{A}(\xi, -)) + N(\mathcal{A}(\xi, =)), N(\mathcal{A}(\xi, +)) + N(\mathcal{A}(\xi, =))\} \geq \frac{|\mathcal{S}|}{2}$. □

For example, $\xi = 1.249$ corresponds to a pivot wrt $\mathcal{S}$ since $\min\{N(\mathcal{A}(\xi, -)) + N(\mathcal{A}(\xi, =)), N(\mathcal{A}(\xi, +)) + N(\mathcal{A}(\xi, =))\} = \min\{20+5, 24+5\} = 25 \geq \frac{|\mathcal{S}|}{2}$ $(= 49/2)$.

In the following, we simply write "a pivot wrt $\mathcal{S}$" as "a pivot" if the context of $\mathcal{S}$ is clear.

(2) Where can we find a pivot? Before we give the details, we introduce a property first.

Property 7 *Given $\xi, \xi' \in \mathcal{B}$ with $\xi < \xi'$, we have*

$$N(\mathcal{A}(\xi, -)) + N(\mathcal{A}(\xi, =)) \leq N(\mathcal{A}(\xi', -)) + N(\mathcal{A}(\xi', =))$$

$$N(\mathcal{A}(\xi, +)) + N(\mathcal{A}(\xi, =)) \geq N(\mathcal{A}(\xi', +)) + N(\mathcal{A}(\xi', =))$$

This essentially says that $N(\mathcal{A}(\xi, -)) + N(\mathcal{A}(\xi, =))$ is non-decreasing *while $N(\mathcal{A}(\xi, +)) + N(\mathcal{A}(\xi, =))$ is* non-increasing *when ξ increases.* □

Proof. This is simply because $\mathcal{A}(\xi, -) \cup \mathcal{A}(\xi, =) \subseteq \mathcal{A}(\xi', -)$ and $\mathcal{A}(\xi', +) \cup \mathcal{A}(\xi', =) \subseteq \mathcal{A}(\xi, +)$. ■

Let $\mathcal{B}$ be the multi-set containing the bisectors of all arrays in $\mathcal{A}$, i.e., $\mathcal{B} = \{\Theta[\lceil \frac{s+e}{2} \rceil][j] | \Theta[s : e][j] \in \mathcal{A}\}$. Note that the size of $\mathcal{B}$ is $O(n)$, and $\mathcal{B} \subseteq \mathcal{S}$. We claim that there exists a pivot in $\mathcal{B}$.

Lemma 5.5.3 *At least one of the values in $\mathcal{B}$ is a pivot wrt $\mathcal{S}$.* □

Proof. Let $\xi_1, \xi_2, ..., \xi_{|\mathcal{B}|}$ be the sorted list of $\mathcal{B}$ in ascending order. We prove Lemma 5.5.3 by contradiction. Assume that none of the values in $\mathcal{B}$ is a pivot.

Consider ξ_1. Clearly, $N(\mathcal{A}(\xi_1, -)) = 0$ and thus $N(\mathcal{A}(\xi_1, =)) + N(\mathcal{A}(\xi_1, +)) = |\mathcal{S}|$. Therefore, we know $N(\mathcal{A}(\xi_1, -)) + N(\mathcal{A}(\xi_1, =)) < |\mathcal{S}|/2$ since otherwise ξ_1 is a pivot which leads to a contradiction.

Consider $\xi_{|\mathcal{B}|}$. Similarly, we know $N(\mathcal{A}(\xi_{|\mathcal{B}|}, +)) = 0$ and thus $N(\mathcal{A}(\xi_{|\mathcal{B}|}, -)) + N(\mathcal{A}(\xi_{|\mathcal{B}|}, =)) = |\mathcal{S}|$. Therefore, we know $N(\mathcal{A}(\xi_{|\mathcal{B}|}, +)) + N(\mathcal{A}(\xi_{|\mathcal{B}|}, =)) < |\mathcal{S}|/2$ since otherwise $\xi_{|\mathcal{B}|}$ is a pivot wrt $\mathcal{S}$ which leads to a contradiction.

By using Property 7 and the above results, we know there exists $h_1 \in [1, |\mathcal{B}|)$ such that $N(\mathcal{A}(\xi_{h_1}, -)) + N(\mathcal{A}(\xi_{h_1}, =)) < |\mathcal{S}|/2$ and $N(\mathcal{A}(\xi_{h_1+1}, -)) + N(\mathcal{A}(\xi_{h_1+1}, =)) \geq$

$|\mathcal{S}|/2$. Similarly, there exists $h_2 \in (1, |\mathcal{B}|]$ such that $N(\mathcal{A}(\xi_{h_2}, +)) + N(\mathcal{A}(\xi_{h_2}, =)) < |\mathcal{S}|/2$ and $N(\mathcal{A}(\xi_{h_2-1}, +)) + N(\mathcal{A}(\xi_{h_2-1}, =)) \geq |\mathcal{S}|/2$.

We consider 3 cases. Case 1: $h_2 < h_1+1$. We have $N(\mathcal{A}(\xi_{h_2}, -))+2 \cdot N(\mathcal{A}(\xi_{h_2}, =)) + N(\mathcal{A}(\xi_{h_2}, +)) < |\mathcal{S}|/2 + |\mathcal{S}|/2 = |\mathcal{S}|$ which leads to a contradiction. Case 2: $h_2 = h_1+1$. This contradicts the fact that $N(\mathcal{A}(\xi_{h_1}, -))+N(\mathcal{A}(\xi_{h_1}, =))+N(\mathcal{A}(\xi_{h_2}, =)) + N(\mathcal{A}(\xi_{h_2}, +)) = |\mathcal{S}|$. Case 3: $h_2 > h_1 + 1$. We deduce that ξ_{h_1+1} is a pivot which leads to a contradiction. That is, we deduce a contradiction in all cases which finishes our proof. ■

To illustrate, consider the search space $\mathcal{S}$ corresponding to the array set shown in Table 5.3. We can compute $\mathcal{B} = \{0, 0.785, 0, 1.249, 0.321, 1.248, 0.322, 1.570, 4.713, 5.820, 4.713, 6.142, 4.854\}$. As mentioned before, 1.249 is a pivot wrt $\mathcal{S}$ which is contained in $\mathcal{B}$.

Lemma 5.5.3 is very usefully since it not only implies that there always exists a pivot, but also implies that we can focus on $\mathcal{B}$ which has its size of $O(n)$ for finding a pivot.

(3) How to find a pivot? According to Lemma 5.5.3, we can focus on $\mathcal{B}$ for finding a pivot. A straightforward method is to traverse the values in $\mathcal{B}$ one by one, check whether it is a pivot, and stop when a pivot is found. Note that given a value ξ, the cost of checking whether ξ is a pivot or not is $O(n)$ since we have $O(n)$ arrays in $\mathcal{A}$ and the number of values in an array $\Theta[s:e][j]$ is simply $e - s + 1$ which could be computed in $O(1)$ time.

Fortunately, we can find a pivot in a smarter way with a binary search over $\mathcal{B}$ based on the monotonicity properties shown in Property 7. Specifically, we first sort the values in $\mathcal{B}$ in ascending order and obtain a sorted list. Let b_m be the value at the middle of the list. Then, we compute $N(\mathcal{A}(b_m, -))$, $N(\mathcal{A}(b_m, =))$, and $N(\mathcal{A}(b_m, +))$. If $\min\{N(\mathcal{A}(b_m, -))+N(\mathcal{A}(b_m, =)), N(\mathcal{A}(b_m, +))+N(\mathcal{A}(b_m, =))\} \geq \frac{|\mathcal{S}|}{2}$, we return

b_m as a pivot; otherwise, we have two cases.

- Case 1: $N(\mathcal{A}(b_m, -)) + N(\mathcal{A}(b_m, =)) < \frac{|\mathcal{S}|}{2}$. In this case, we can safely prune all values that are *at most* b_m in $\mathcal{B}$ (Here, pruning a value means that we ignore this value for finding a pivot, which is considered in this section, but this value is still in the current search space $\mathcal{S}$ (or $\mathcal{A}$)).
- Case 2: $N(\mathcal{A}(b_m, +)) + N(\mathcal{A}(b_m, =)) < \frac{|\mathcal{S}|}{2}$. In this case, we can safely prune all values that are *at least* b_m in $\mathcal{B}$.

In conclusion, if b_m is a pivot, we are done, and otherwise we can prune at least half of the search space $\mathcal{B}$ and repeat the process based on the remaining search space until we find a pivot.

The time complexity of the above method is simply $O(n \log n)$ since the sorting procedure has the cost of $O(n \log n)$ and the binary search procedure has $O(\log n)$ iterations each has the cost of $O(n)$ for checking whether a given value is a pivot.

Part III: Span Affordability Check on ξ

In this part, we introduce our method for performing the span affordability check on a given ξ.

Let ξ be a non-negative value. Given a simplification T' of T, we say that T' is a ξ-simplification (of T) iff $\xi(T') \leq \xi$.

Similar to the error affordability check described in Section 5.4, we perform the span affordability check on a given ξ as follows. First, we compute the ξ-simplification with the smallest size, say, T'. Then, we compare $|T'|$ with W. If $|T'| \leq W$, we conclude that ξ is an affordable span; otherwise, we conclude that ξ is not an affordable span. The correctness of this method is obvious and the remaining issue is how to find the ξ-simplification with the smallest size for a given ξ.

Algorithm 10 Finding ξ-simplification with the smallest size

1: $T' \leftarrow (p_1)$
2: $i \leftarrow 1$; $j \leftarrow i + 1$
3: **while** $j \leq n$ **do**
4: **while** $j \leq n$ and $\xi(mcar(\theta[i : j])) \leq \xi$ **do**
5: $j \leftarrow j + 1$
6: Append p_{j-1} to T'
7: $i \leftarrow j - 1$
8: **return** T'

We design our algorithm as follows. Let i be the position index of T where the algorithm starts at. Initially, i is set to 1 and p_i is appended to T'. It tries to approximate as many consecutive segments starting from p_i in T as possible while adhering to the constraint that the span of the minimum covering angular range of the set containing the directions of these segments is at most ξ. To do it, it checks the position index j starting from $i + 1$ one by one. If $\xi(mcar(\theta[i : j])) \leq \xi$, it continues to check the next position index by updating j to $j + 1$ until either $j > n$ (i.e., $j = n + 1$) or $\xi(mcar(\theta[i : j])) > \xi$. Then, it appends p_{j-1} to T' since in either the case of $j > n$ (i.e., $j = n + 1$), or the case of $\xi(mcar(\theta[i : j])) > \xi$, the segments between p_i and p_{j-1} form the longest possible sequence starting from p_i that could be approximated by one segment in T'. After that, it continues the process from p_{j-1} by updating i with $j - 1$. It stops if $j > n$ which implies $j = n + 1$. The pseudo-code of the algorithm is shown in Algorithm 10.

We illustrate Algorithm 10 with the input trajectory as T in Figure 4.2 and ξ as 1.249. Note that $\xi = 1.249$ is a pivot. In this case, $n = 8$. T' is first initialized as (p_1) and $i = 1$. It starts from $j = i + 1 = 2$. It computes $\xi(mcar(\theta[1 : 2])) = 0$ since $\theta[1 : 2] = \{\theta(\overline{p_1p_2})\} = \{0.785\}$ and thus $mcar(\theta[1 : 2]) = [0.785, 0.785]$. Since $j \leq n$ and $\xi(mcar(\theta[1 : 2])) \leq \xi = 1.249$, it updates j to be $j + 1 = 3$. Again, it computes $\xi(mcar(\theta[1 : 3])) = 1.248$. Since $j \leq n$ and $\xi(mcar(\theta[1 : 3])) \leq \xi = 1.249$, it updates j to be $j + 1 = 4$. Then, it computes $\xi(mcar(\theta[1 : 4])) = 1.570$. This time, since $\xi(mcar(\theta[1 : 4])) > \xi = 1.249$, it stops updating j, but appends p_{j-1} (i.e., p_3) to

T' (thus T' becomes (p_1, p_3)) and updates i to be $j - 1 = 3$. It implies that (p_1, p_2, p_3) is the longest possible sequence starting from p_1 which has the span at most $\xi = 1.249$. It repeats the same process with the new starting position p_3 and keeps increasing j by 1 until $j = 7$ since $\xi(\theta[3:7]) = 1.892 > \xi = 1.249$. Then, it appends p_{j-1} (i.e., p_6) to T' (thus T' becomes (p_1, p_3, p_6)) and updates i to be $j - 1 = 6$. It continues the same process with the new starting position p_6 and keeps increasing j by 1 until $j = 9$ since $j > n$. Then, it appends p_{j-1} (i.e., p_8) to T' (thus T' becomes (p_1, p_3, p_6, p_8)) and stops the process. At the end, it returns T' which is (p_1, p_3, p_6, p_8).

Lemma 5.5.4 *Algorithm 10 finds the ξ-simplification with the smallest size for a given ξ.* □

Proof. Let $T' = (p_{s_1}, p_{s_2}, ..., p_{s_m})$ be the simplification returned by Algorithm 10. Let $T'' = (p_{t_1}, p_{t_2}, ..., p_{t_l})$ be the ξ-simplification with the smallest size. By definition, we have $s_1 = r_1 = 1$ and $s_m = t_l = n$. Note that $|T'| = m$ and $|T''| = l$.

Assume that $m > l$. We prove that for each $k \in [1, l]$, we have $s_k \geq r_k$ by deduction.

Base step: $k = 1$. We have $s_k = r_k = 1$.

Deduction step: $k > 1$. Assume that we have $s_{k-1} \geq r_{k-1}$. According to Algorithm 10, we have $\xi(mcar(\theta[s_{k-1} : j])) \leq \xi$ for $j \in [s_{k-1} + 1, s_k]$ while $\xi(mcar(\theta[s_{k-1} : s_k + 1])) > \xi$. Since $s_{k-1} \geq r_{k-1}$, we know $r_k \leq s_k$ since otherwise $\xi(mcar(\theta[r_{k-1} : r_k])) \geq \xi(mcar(\theta[r_{k-1} : s_k + 1)) \geq \xi(mcar(\theta[s_{k-1} : s_k + 1])) > \xi$, which leads to a contradiction. The above inequalities are based on the fact that $\xi(mcar(\mathcal{D}))$ is non-decreasing when the set $\mathcal{D}$ includes more directions.

Therefore, we have $s_l \geq r_l = n$, which leads to a contradiction that $s_l < s_m = n$. This finishes our proof. ■

Algorithm 10 has the time complexity of $O(n \log n)$, whose implementation details and time complexity analysis could be found in Section B.4 of the Appendix.

Recall that the span affordability check on a given ξ is performed by first finding the ξ-simplification T' with the smallest size via Algorithm 10 and then comparing $|T'|$ with W. Thus, the cost of performing the span affordability check is dominated by the cost of Algorithm 10, which is $O(n \log n)$.

To illustrate, consider the span affordability check on $\xi = 1.249$ with the input trajectory as T in Figure 4.2 and the input W as 3. As discussed before, the ξ-simplification T' with the smallest size is (p_1, p_3, p_6, p_8). Since T' has its size equal to 4 which is larger than $W = 3$, we know that $\xi = 1.249$ is not an affordable span.

Part IV: How to Prune Search Space with a Pivot

In this part, we describe how we can prune at least $\frac{1}{4}$ of the current search space based on a pivot. Suppose that ξ is a pivot. We can prune the current search space based on two different cases.

- *Case 1:* ξ is not an affordable span. In this case, we know that $\xi_o > \xi$ and we can prune values at most ξ. To do this, for each array $\Theta[s : e][j]$ in $\mathcal{A}(\xi, -) \cup \mathcal{A}(\xi, =)$, we prune its values that are at or after the position of its bisector (because they are at most its bisector and its bisector is at most ξ) by shrinking it to $\Theta[s : \lceil \frac{s+e}{2} \rceil - 1][j]$ ($\Theta[s : \lceil \frac{s+e}{2} \rceil - 1][j]$ is dropped if $\lceil \frac{s+e}{2} \rceil - 1 < s$). Note that the number of values pruned is at least $\frac{N(\mathcal{A}(\xi,-)) + N(\mathcal{A}(\xi,=))}{2}$. Since ξ is a pivot, we know $N(\mathcal{A}(\xi, -)) + N(\mathcal{A}(\xi, =)) \geq \frac{|\mathcal{S}|}{2}$ which implies that we have pruned at least $\frac{|\mathcal{S}|}{4}$ values. Here, by shrinking $\Theta[s : e][j]$ to $\Theta[s : \lceil \frac{s+e}{2} \rceil - 1][j]$ in $\mathcal{A}$, we mean updating the triple (s, e, j) with $(s, \lceil \frac{s+e}{2} \rceil - 1, j)$ in the index triplet set $\mathcal{T}$ corresponding to $\mathcal{A}$.

- *Case 2:* ξ is an affordable span. We can perform the pruning operation in a

symmetric way as Case 1 by shrinking each array $\Theta[s:e][j]$ in $\mathcal{A}(\xi,+)\cup\mathcal{A}(\xi,=)$ to $\Theta[\lceil\frac{s+e}{2}\rceil+1:e][j]$ ($\Theta[\lceil\frac{s+e}{2}\rceil+1:e][j]$ is dropped if $\lceil\frac{s+e}{2}\rceil+1>e$). Similar to Case 1, we derive that $\frac{1}{4}$ of the current search space is pruned, and the corresponding shrinking operation is executed on $\mathcal{T}$.

In conclusion, using a pivot can prune $\frac{1}{4}$ of the current search space.

To illustrate, consider our running example where Table 5.3 shows the array set corresponding to the current search space $\mathcal{R}$ containing 49 values. Suppose that we have found a pivot $\xi = 1.249$. Now, we illustrate the pruning process based on ξ. Since ξ is not an affordable span (we know it from the examples discussed before), we prune the search space $\mathcal{R}$ be updating each array $\Theta[s:e][j]$ in $\mathcal{A}(\xi,-)\cup\mathcal{A}(\xi,=)$ to be $\Theta[s:\lceil\frac{s+e}{2}\rceil-1][j]$. As discussed before, $\mathcal{A}(\xi,-)=\{\Theta[1:1][1],\Theta[2:7][1],\Theta[1:2][2],\Theta[1:3][3],\Theta[4:7][3],\Theta[1:4][4]\}$ and $\mathcal{A}(\xi,=)=\Theta[3:7][2]\}$. The arrays in these two sets will be updated and the index triplet set of the updated array set is shown in Table 5.5. As could be verified, the number of values in the search space represented by this updated index triplet set is equal to 35, i.e., $(49-35)=14$ values have been pruned (note that $14>\frac{49}{4}$).

Note that our index triplet set for representing the search space makes the process of executing the pruning operations extremely convenient, i.e., all we need is to update the indices (i.e., s and e) of each array $\Theta[s:e][j]$, and thus the pruning operations could be executed in $O(n)$ time since we have $O(n)$ arrays only.

Remark. The pruning operations only shrink the arrays and thus Property 6 still holds for the updated array set which further implies that we can repeat our process to find a pivot ξ wrt the updated search space, perform a span affordability on ξ and prune the updated search space at the next iteration until the search space becomes empty. Note that the process involves $O(\log n)$ iterations only since at least $\frac{1}{4}$ of the search space is pruned at each iteration.

	# of trajectories	total # of positons	average # of positions per trajectory
Geolife	17,621	24,876,978	1,412
T-Drive	10,359	17,740,902	1,713

Table 5.6: Real datasets (DPTS and Min-Error)

Part V: Time & Space Complexity of *Span-Search*

In this part, we analyze the time and space complexities of *Span-Search*. *Span-Search* proceeds with iterations. At each iteration, it first finds a pivot ξ wrt the current search space (which can be done in $O(n \log n)$ as shown in Part II of Section 5.5.4), checks the span affordability on ξ (which can be done in $O(n \log n)$ as shown in Part III of Section 5.5.4), and prunes at least $\frac{1}{4}$ of the current search space (which can be done in $O(n)$ as shown in Part IV of Section 5.5.4). It could be verified easily that the process involves $2 \log n / \log(4/3) = O(\log n)$ iterations. Therefore, the time complexity of *Span-Search* is $O(\log n \cdot (n \log n + n \log n + n)) = O(n \log^2 n)$. Besides, the space complexity of *Span-Search* is simply $O(n)$ which corresponds to the space cost for maintaining the index triplet set.

5.6 Experiments

We used two real datasets in our experiments, namely Geolife and T-Drive. Geolife[3] records the outdoor movements of 182 users in a period of 5 years and T-Drive[4] is a set of taxi trajectories in Beijing. These two datasets are widely used for a broad range of applications on trajectory data [107, 101]. The statistics of these datasets are summarized in Table 5.6.

Since the experimental results in Chapter 4 already show the advantage of using DPTS over PPTS, we focus on the performance of our proposed algorithms in this thesis. All algorithms were implemented in C/C++ and ran on a Linux platform with a

[3]http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13/

[4]http://research.microsoft.com/apps/pubs/?id=152883

2.66GHz machine and 40GB RAM.

5.6.1 Comparison with Wavelet Transformation

First, following [12], we use the *Haar Wavelet Transformation* as a baseline of trajectory simplification and compare it with our Min-Error mechanism in terms of how good they are for preserving the direction information. The major idea of wavelet transformation is to transform the raw data which corresponds to a set of n numbers into a set of n coefficients (this step does not introduce any information loss and the raw data could be completely restored with these n coefficients) and store k coefficients only where $k < n$, e.g., top-k coefficients (note that this step saves some storage space with the compression rate of k/n, but introduces some information loss since $n - k$ coefficients are dropped). To get an approximation of the raw data (which contains n values), a set of n values is constructed based on the k stored coefficients. We adopt wavelet transformation for trajectory simplification with the purpose of preserving the direction information as follows. We maintain the set of the directions of the segments of a given trajectory (this corresponds to the direction information of the trajectory), perform wavelet transformation on the set of directions and store a certain number of coefficients according to the storage budget. The goodness of wavelet transformation for preserving the direction information is measured by the *maximum* and also *average* angular difference between an original direction and its corresponding approximated direction constructed based on the stored coefficients. For both measures, the smaller, the better.

We conducted experiments on Min-Error and wavelet transformation by varying the storage budget W, and the results are shown in Figure 5.6 where "Wavelet trans. (max.)" and "Wavelet trans. (avg.)" denote the maximum and the average angular difference of wavelet transformation, respectively, and "Min-Error (max.)" denotes

the maximum angular difference between the direction of a segment $\overline{pp'}$ in the original trajectory and the direction of the segment that approximates $\overline{pp'}$ in the simplified trajectory generated by the exact algorithm, *Error-Search*, for *Min-Error* (note that this corresponds to the direction-based error). Note that for Min-Error, we do not show the average angular difference since it is extremely small. According to these results, we have the following observations. First, wavelet transformation performs poorly when being used for preserving the direction information, e.g., in most cases, wavelet transformation results in high maximum and average angular difference, and this holds even when the storage budget W is near to $|T|$. This essentially tells that wavelet transformation is not suitable for preserving the direction information when being used for trajectory simplification. Second, Min-Error performs significantly better than wavelet transformation in terms of preserving the direction information. Thus, in the following, we focus on Min-Error only in our experiments.

We also show the effects of the storage budget W on the direction-based error in more detail in Figure 5.7 and we observe that when W is relatively small (e.g., $W \leq 0.2$), a small increase on W yields a significant reduction on the optimal error, while when W is relatively large (e.g., $W \geq 0.5$), even a large increase on W helps a little to reduce the optimal error.

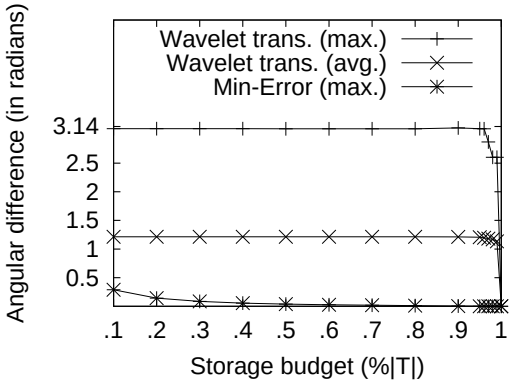


Figure 5.6: MinError vs. Wavelet

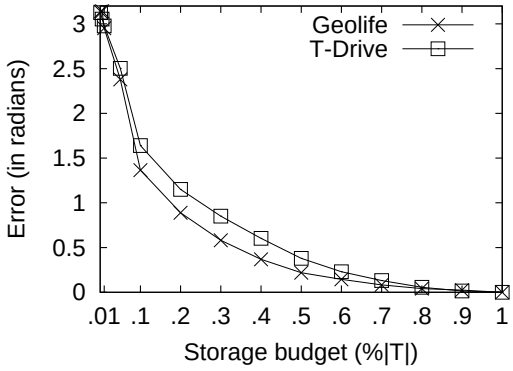


Figure 5.7: Error trends

We do not adopt the principle of *minimum description length* (MDL) [42] for our

Min-Error problem since MDL is for balancing between the size and the error of the simplied trajectory and thus it does not allow users to specify a size constraint or optimize the simplification error.

Next, we study the performance of our exact and approximate algorithm in Section 5.6.2 and in Section 5.6.3, respectively.

5.6.2 Performance Study of the Exact Algorithms

In this part, we study the effects of 2 factors, namely the data size (i.e., $|T|$) and the storage budget (i.e., W) on the performance of our exact algorithms, namely *DP* and *Error-Search*. We use 2 measures, namely the running time and the memory.

Effect of $|T|$. The values used for $|T|$ are around 2,000, 4,000, 6,000, 8,000 and 10,000 (W is fixed to be 0.2, i.e., $W = 0.2 * |T|$). For each setting of $|T|$, we select a set of 10 trajectories each of which has its size near to this value and run our exact algorithms on each of these trajectories. Then, we average the experimental results on these trajectories (this policy is used throughout our experiments without specification). Figure 5.8 show the results on Geolife. According to these results, *Error-Search* is always faster than *DP*, and the efficiency gap between them becomes larger when the data size increases. This could be easily explained by the fact that *Error-Search* has smaller time/space complexities than *DP*.

The experimental results on T-Drive are similar and thus they are omitted due to page limit.

Effect of W. The values used for W are 0.1, 0.2, 0.3, 0.4 and 0.5 ($|T|$ is fixed to about 6,000). The results are presented in Figure 5.9. We observe that *DP* has both its running time and its memory increase with W, which could be explained by the fact that *DP* has its problem space proportional to W. In contrast, W has no significant effects on *Error-Search* since *Error-Search* has its time/space complexities independent of W.

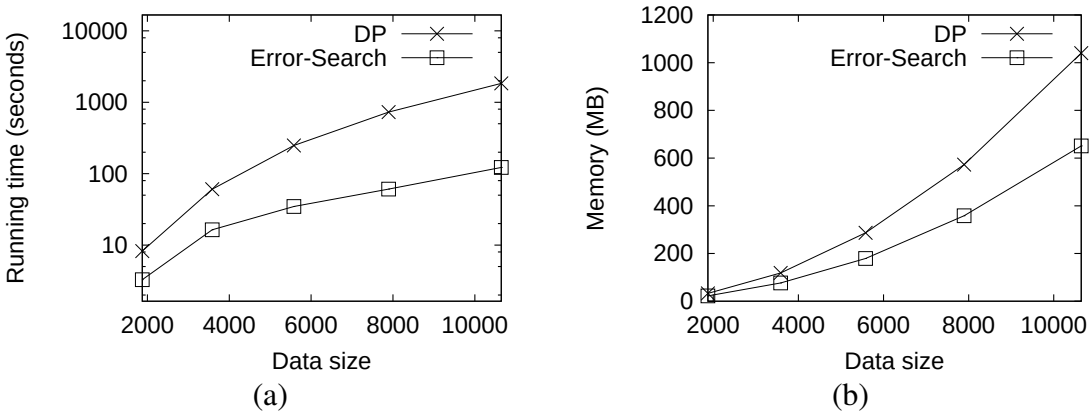


Figure 5.8: Effects of data size $|T|$ (Geolife, Min-Error)

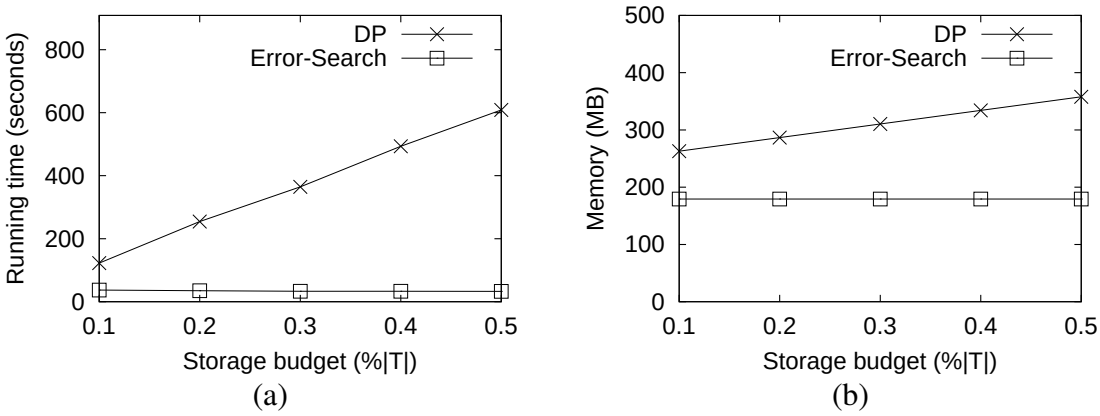


Figure 5.9: Effects of storage budget W (Geolife, Min-Error)

Scalability test. Figure 5.10 shows the scalability test results on the exact algorithms. We observe that *DP* is limited to medium-sized datasets only while *Error-Search* can go much further. For example, on a trajectory with about 50,000 positions, *DP* runs for several days and occupies nearly 30GB memory, while *Error-Search* runs for about 1hr and occupies about 10GB memory.

5.6.3 Performance Study of the Approximate Algorithms

In this part, we study the effects of $|T|$ and W on two approximate algorithms, namely *Span-Search* and *Douglas-Peucker*. *Douglas-Peucker* is an adaptation of the traditional Douglas-Peucker algorithm [31], whose major idea is to recursively cut the trajectory at one of the end of the segment that has the *greatest* angular difference from the segment

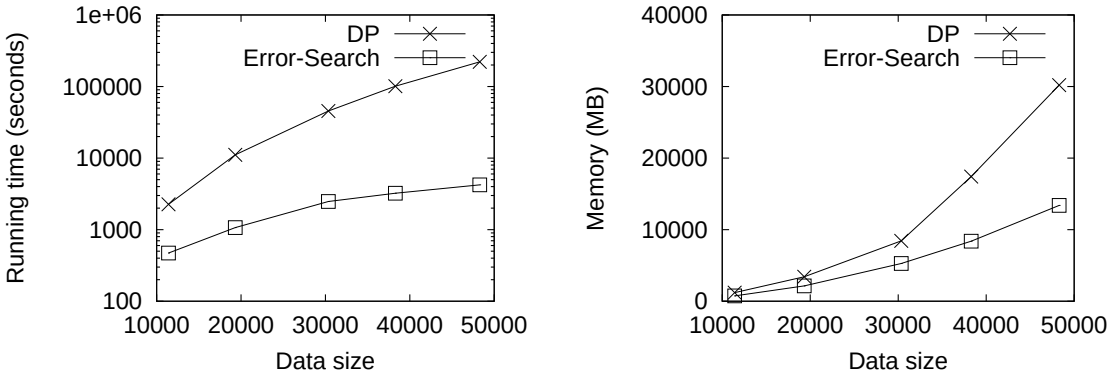


Figure 5.10: Scalability test (Geolife, Min-Error)

linking the start position and the end position of this trajectory until we have $W - 1$ sub-trajectories and then use one segment to approximate each sub-trajectory. We note here that *Douglas-Peucker* is the most popular algorithm for trajectory simplification in the literature [31, 72, 43]. We use 3 measures, namely the running time, the memory and the *approximation factor*. The approximation factor of an approximate algorithm is defined to be $\epsilon(T')/\epsilon(T'_o)$, where T' is the simplified trajectory returned by this approximate algorithm on a given raw trajectory and T'_o is the simplified trajectory returned by an exact algorithm on the same raw trajectory. Clearly, the smaller the approximation factor is, the better approximation quality the algorithm has.

Approximation factor. We present the results with two figures, Figure 5.11(a) and Figure 5.11(b). Figure 5.11(a) shows for each approximate algorithm, the (absolute) error of the simplified trajectory returned and also the optimal error (i.e., the error of the simplified trajectory returned by an exact algorithm such as *Error-Search*), and Figure 5.11(b) shows the approximation factors of the approximate algorithms. According to these results, *Span-Search* is consistently better than *Douglas-Peucker* in terms of approximation quality. We emphasize here that Douglas-Peucker has its approximation factor usually around 3. In contrast, Span-Search usually achieves an approximation factor around 1.5, though its theoretical worst-case bound is 2. In other words, Douglas-Peucker has an error that is 200% greater than optimum while Span-

Search achieves an error only 50% greater than optimum.

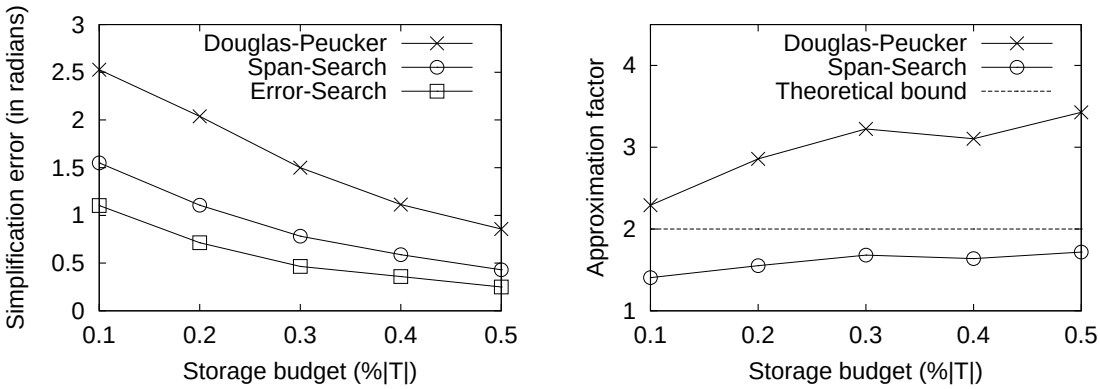


Figure 5.11: Approximation quality (Geolife, Min-Error)

Effect of $|T|$. The values used for $|T|$ are around 20,000, 40,000, 60,000, 80,000 and 100,000 (W is fixed to 0.2). Figure 5.12 shows the results. According to these results, *Span-Search*, though slower than *Douglas-Peucker*, runs reasonably fast (e.g., on a dataset with about 100,000 positions, *Span-Search* runs less than 1000s). Besides, both *Span-Search* and *Douglas-Peucker* are space efficient (e.g., they occupy less than 30MB) which could be explained by the fact that *Span-Search* has a linear space complexity and so does *Douglas-Peucker*.

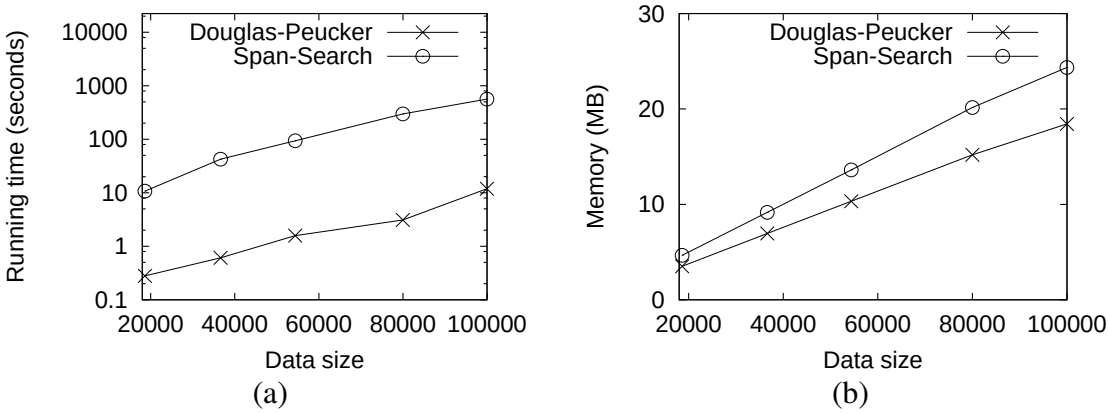


Figure 5.12: Effects of data size $|T|$ (Geolife, Min-Error)

Effect of W. The values used for W are 0.1, 0.2, 0.3, 0.4 and 0.5 ($|T|$ is fixed to about 60,000). The results are shown in Figure 5.13. We notice that both *Span-Search* and *Douglas-Peucker* are only slightly affected by W. Specifically, when W increases,

both the algorithms run a little bit slower. For *Span-Search*, with a larger W, the span affordability check procedure would probably maintain a larger binary search tree and also a larger priority queue which incurs more cost. For *Douglas-Peucker*, with a larger W, it would do more "cut" operations and thus it incurs more cost.

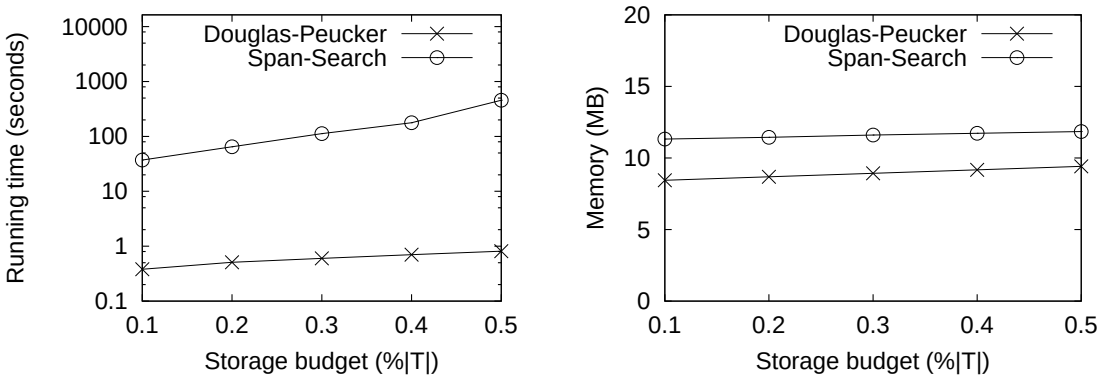


Figure 5.13: Effects of storage budget W (Geolife, Min-Error)

Scalability test. Figure 5.14 shows the scalability test results on the approximate algorithms. According to results, we know that *Span-Search* is scalable to large datasets. For example, on a dataset with about 500,000 positions, *Span-Search* runs for a couple of hours and occupies less than 150MB memory.

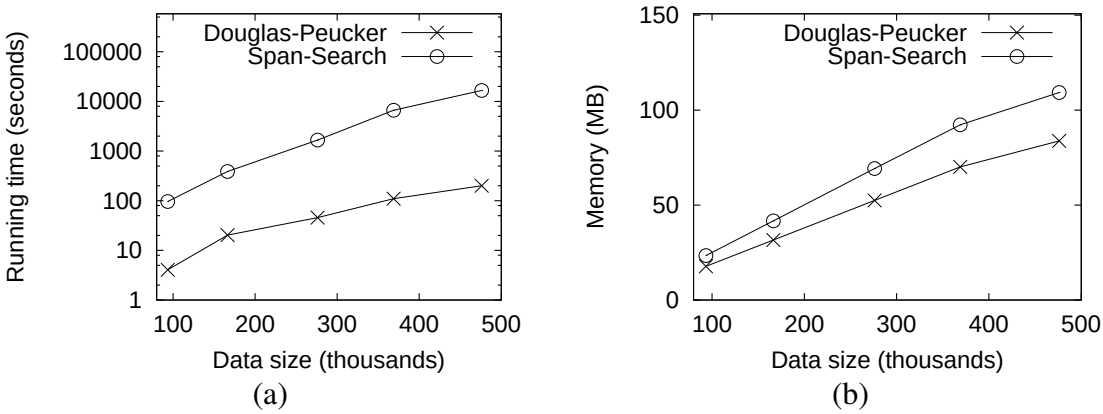


Figure 5.14: Scalability test (Geolife, Min-Error)

Additional experiments. We also conducted experiments on a variant of *Error-Search* which adopts the *Douglas-Peucker* algorithm for performing each error affordability check *approximately* and thus it corresponds to an approximate algorithm for the Min-

Error problem. We observed that this variant of *Error-Search* was dominated by our *Span-Search* algorithm in terms of both the running time and the minimized error. Due to the page limit, the details are omitted.

Empirical conclusion. About the exact algorithms, *Error-Search* has its superiority over *DP* in terms of both time and space efficiency. About the approximate algorithms, *Span-Search* has its approximation quality consistently better than *Douglas-Peucker* and is scalable to large datasets.

5.7 Conclusion

In this chapter, we identified a new application scenario for DPTS and defined a corresponding problem, i.e., the Min-Error problem. Then, we designed two exact algorithms, *DP* and *Error-Search*, based on dynamic programming and binary search, respectively. Since the time complexities of the exact algorithms are relatively high, we further developed an approximate algorithm *Span-Search* which runs in $O(n \log^2 n)$ time and gives a 2-factor approximation. We conducted extensive experiments on real datasets which verified our proposed algorithms.

CHAPTER 6

CONCLUSION AND FUTURE WORK PLAN

In this thesis, we introduce three techniques for boosting the spatial computations which are central to some location-based services, namely the collective spatial keyword query, worst-case optimized spatial matching, and direction-preserving trajectory simplification.

We have the future work plans as follows as regarding to techniques introduced in this thesis. As for the collective spatial keyword query (CoSKQ), we plan to study the problem of finding a feasible set with the smallest cost where the cost is defined based on the *shortest route* that traverses all objects in the set, the problem of finding a feasible set with the smallest *cost per object*, and the CoSKQ problem in a dynamic setting where the query point is moving. As for the worst-case optimized spatial matching, we plan to study the problem of finding the worst-case optimized spatial matching in a dynamic setting where the customers and/or the service-providers are moving and the problem of finding a location for setting up a new service-provider such that the maximum matching distance is reduced as much as possible. As for the direction-preserving trajectory simplification (DPTS), we plan to study the problem of performing DPTS in an *online* setting, the problem of doing DPTS where each line segment of the trajectory is associated with a weight (i.e., different segments have different importance; for example, it is fairly intuitive that a longer road segment is regarded to be more important than a shorter one), and the problem of trajectory simplification where the error is defined based on some other information than the directional information, e.g., the speed information.

REFERENCES

[1] Fire services department, hong kong: Performance pledge 2012. http://www.hkfsd.gov.hk/eng/performance.html.

[2] *MaxFlow Revisited: An Empirical Comparison of Maxflow Algorithms for Dense Vision Problems*, 2012.

[3] P. K. Agarwal and K. R. Varadarajan. Efficient algorithms for approximating polygonal chains. *Discrete Computational Geometry*, 23(2):273–291, 2000.

[4] A. Aggarwal, H. Imai, N. Katoh, and S. Suri. Finding k points with minimum diameter and related problems. *Journal of Algorithms*, 12(1):38–56, 1991.

[5] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[6] E. M. Arkiny and R. Hassinz. Minimum diameter covering problems. *Networks*, 36(3), 2000.

[7] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *JCSS*, 7(4):448–461, 1973.

[8] E. Bortnikov, S. Khuller, J. Li, Y. Mansour, and J. S. Naor. The load-distance balancing problem. *Networks*, 2010.

[9] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *VLDB*, pages 853–864, 2015.

[10] S. Brakatsoulas, D. Pfoser, and N. Tryfona. Modeling, storing and mining moving object databases. In *IDEAS*, pages 68–77. IEEE, 2014.

[11] R. E. Burkard, M. Dell'Amico, and S. Martello. *Assignment problems*. Society for Industrial Mathematics, 2009.

[12] H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *The VLDB Journal*, 2006.

[13] X. Cao, L. Chen, G. Cong, and X. Xiao. Keyword-aware optimal route search. *Proceedings of the VLDB Endowment*, 5(11):1136–1147, 2012.

[14] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *VLDB*, 3(1-2):373–384, 2010.

[15] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD*, pages 373–384. ACM, 2011.

[16] A. Cary, O. Wolfson, and N. Rishe. Efficient and scalable method for processing top-k spatial boolean queries. In *Scientific and Statistical Database Management*, pages 87–95. Springer, 2010.

[17] T. M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. In *SODA*, 2006.

[18] M. S. Chang, C. Y. Tang, and R. C. T. Lee. Solving the euclidean bottleneck matching problem by k-relative neighborhood graphs. *Algorithmica*, 8(1):177–194, 1992.

[19] B. Chazelle. New upper bounds for neighbor searching. In *Information and Control*, 1986.

[20] H. Chen, W.-S. Ku, M.-T. Sun, and R. Zimmermann. The multi-rule partial sequenced route query. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, page 10. ACM, 2008.

[21] M. Chen, M. Xu, and P. Franti. Compression of gps trajectories. In *Data Compression Conference (DCC), 2012*, pages 62–71. IEEE, 2012.

[22] M. Chen, M. Xu, and P. Franti. A fast o(n) multiresolution polygonal approximation algorithm for gps trajectory simplification. *IEEE Transactions on Image Processing*, 21(5), 2012.

[23] Y. Chen, K. Jiang, Y. Zheng, C. Li, and N. Yu. Trajectory simplification method for location-based social networking services. In *IWLBSN*, pages 33–40. ACM, 2009.

[24] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel. Text vs. space: efficient geo-search query processing. In *CIKM*, pages 423–432. ACM, 2011.

[25] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *VLDB*, 2(1):337–348, 2009.

[26] G. Cong, H. Lu, B. C. Ooi, D. Zhang, and M. Zhang. Efficient spatial keyword search in trajectory databases. *Arxiv preprint arXiv:1205.2880*, 2012.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2009.

[28] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. Computational geometry: Algorithms and applications. In *Springer*, 2000.

[29] 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 International Conference on Advances in Geographic Information Systems*, pages 324–333. ACM, 2013.

[30] T. Dokka, A. Kouvela, and F. C. R. Spieksma. Approximating the multi-level bottleneck assignment problem. In *Operations Research Letter*, 2012.

[31] D. Douglas and T. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 11(2):112–122, 1973.

[32] A. Efrat, A. Itai, and M. J. Katz. Geometry helps in bottleneck matching and related problems. *Algorithmica*, 31(1):1–28, 2001.

[33] A. Efrat and M. J. Katz. Computing euclidean bottleneck matchings in higher dimensions. *Information processing letters*, 2000.

[34] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665. IEEE, 2008.

[35] E. Frentzos and Y. Theodoridis. On the effect of trajectory compression in spatiotemporal querying. In *Advances in Databases and Information Systems*, pages 217–233. Springer, 2007.

[36] H. N. Gabow and R. E. Tarjan. Algorithms for two bottleneck optimization problems. *Journal of Algorithms*, 9(3):411–417, 1988.

[37] D. Gale and L. Shapley. College admissions and the stability of marriage. *Amer. Math. Monthly*, 69:9–15, 1962.

[38] O. Ghica, G. Trajcevski, O. Wolfson, U. Buy, P. Scheuermann, F. Zhou, and D. Vaccaro. Trajectory data reduction in wireless sensor networks. *International Journal of Next-Generation Computing*, 1(1), 2010.

[39] F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi. Trajectory pattern mining. In *SIGMOD*, pages 330–339. ACM, 2007.

[40] A. V. Goldberg, S. Hed, H. Kaplan, R. E. Tarjan, and R. F. F. Werneck. Maximum flows by incremental breadth-first search. In *ESA*, 2011.

[41] O. Gross. The bottleneck assignment problem. *The Rand Corporation*, 1959.

[42] P. D. Grünwald, I. J. Myung, and M. A. Pitt. *Advances in minimum description length: Theory and applications*. MIT press, 2005.

[43] J. Gudmundsson, J. Katajainen, D. Merrick, C. Ong, and T. Wolle. Compressing spatio-temporal trajectories. *Algorithms and Computation*, pages 763–775, 2007.

[44] P. S. Heckbert. Survey of polygonal surface simplification algorithms. *Carnegie Mellon University technical report*, 1997.

[45] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM TODS*, 24(2):265–318, 1999.

[46] D. S. Hochbaum and D. B. Shmoys. A best possible heuristic for the k-center problem. *Mathematics of operations research*, 1985.

[47] N. Hönle, M. Grossmann, D. Nicklas, and B. Mitschang. Preprocessing position data of mobile objects. In *MDM*, pages 41–48. IEEE, 2008.

[48] N. Hönle, M. Grossmann, S. Reimann, and B. Mitschang. Usability analysis of compression algorithms for position data streams. In *SIGSPATIAL*, pages 240–249. ACM, 2010.

[49] J. E. Hopcroft and R. M. Karp. A n5/2 algorithm for maximum matchings in bipartite graphs. In *Switching and Automata Theory*, 1971.

[50] C.-C. Hung, W.-C. Peng, and W.-C. Lee. Clustering and aggregating clues of trajectories for mining trajectory patterns and routes. *The VLDB Journal*, pages 1–24, 2011.

[51] J. M. Kang, M. F. Mokbel, S. Shekhar, T. Xia, and D. Zhang. Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors. In *ICDE*, 2007.

[52] L. Kazemi and C. Shahabi. Geocrowd: enabling query answering with spatial crowdsourcing. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, pages 189–198. ACM, 2012.

[53] L. Kazemi, C. Shahabi, and L. Chen. Geotrucrowd: trustworthy query answering with spatial crowdsourcing. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 314–323. ACM, 2013.

[54] G. Kellaris, N. Pelekis, and Y. Theodoridis. *Trajectory compression under network constraints*, pages 392–398. Advances in Spatial and Temporal Databases. Springer, 2009.

[55] G. Kellaris, N. Pelekis, and Y. Theodoridis. Map-matched trajectory compression. *Journal of Systems and Software*, 2013.

[56] A. Kolesnikov. Efficient online algorithms for the polygonal approximation of trajectory data. In *MDM'11*, volume 1, pages 49–57. IEEE, 2011.

[57] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *SIGMOD*, 2000.

[58] R. Lange, F. Dürr, and K. Rothermel. Online trajectory data reduction using connection-preserving dead reckoning. In *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, page 52. ICST, 2008.

[59] R. Lange, F. Dürr, and K. Rothermel. Efficient real-time trajectory tracking. *The VLDB Journal*, 20(5):671–694, 2011.

[60] R. Lange, T. Farrell, F. Dürr, and K. Rothermel. Remote real-time trajectory simplification. In *PerComm'09*, pages 1–10. IEEE, 2009.

[61] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *SIGKDD*, pages 467–476. ACM, 2009.

[62] J. G. Lee, J. Han, and X. Li. Trajectory outlier detection: A partition-and-detect framework. In *ICDE*, pages 140–149. IEEE, 2008.

[63] J. G. Lee, J. Han, X. Li, and H. Gonzalez. Traclass: trajectory classification using hierarchical region-based and trajectory-based clustering. *PVLDB'08*, 1(1):1081–1094, 2008.

[64] J. G. Lee, J. Han, and K. Y. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD*, pages 593–604. ACM, 2007.

[65] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. *On trip planning queries in spatial databases*, pages 273–290. Advances in Spatial and Temporal Databases. Springer, 2005.

[66] Z. Li, K. Lee, B. Zheng, W. Lee, D. Lee, and X. Wang. Ir-tree: An efficient index for geographic document search. *TKDE*, pages 585–599, 2011.

[67] G. Liu, M. Iwai, and K. Sezaki. A method for online trajectory simplification by enclosed area metric. *ICMU'12*, 2012.

[68] W. Liu, W. Sun, C. Chen, Y. Huang, Y. Jing, and K. Chen. Circle of friend query in geo-social networks. In *Database Systems for Advanced Applications*, pages 126–137. Springer, 2012.

[69] W. Liu, Y. Zheng, S. Chawla, J. Yuan, and X. Xing. Discovering spatio-temporal causal interactions in traffic data streams. In *SIGKDD*, 2011.

[70] J. Lu, Y. Lu, and G. Cong. Reverse spatial and textual k nearest neighbor search. In *SIGMOD*, pages 349–360. ACM, 2011.

[71] W. Luo, H. Tan, L. Chen, and L. M. Ni. Finding time period-based most frequent path in big trajectory data. In *SIGMOD*, 2013.

[72] N. Meratnia and R. de By. Spatiotemporal compression techniques for moving point objects. *EDBT*, pages 561–562, 2004.

[73] J. Muckell, J. H. Hwang, C. T. Lawson, and S. Ravi. Algorithms for compressing gps trajectory data: an empirical evaluation. In *SIGSPATIAL*, pages 402–405. ACM, 2010.

[74] J. Muckell, J. H. Hwang, V. Patil, C. T. Lawson, F. Ping, and S. Ravi. Squish: an online approach for gps trajectory compression. In *Proceedings of the 2nd International Conference on Computing for Geospatial Research and Applications*, pages 1–8. ACM, 2011.

[75] K. Mulmuley. Computational geometry: An introduction through randomized algorithms. In *Prentice Hall*, 1993.

[76] D. Patel, C. Sheng, W. Hsu, and M. L. Lee. Incorporating duration information for trajectory classification. In *ICDE*, pages 1132–1143. IEEE, 2012.

[77] N. Pelekis, I. Kopanakis, G. Marketos, I. Ntoutsi, G. Andrienko, and Y. Theodoridis. Similarity search in trajectory databases. In *TIME*, pages 129–140. IEEE, 2007.

[78] H. Pham, C. Shahabi, and Y. Liu. Ebm-an entropy-based model to infer social strength from spatiotemporal data. *SIGMOD*, 2013.

[79] M. Potamias, K. Patroumpas, and T. Sellis. Sampling trajectory streams with spatiotemporal criteria. In *SSDBM*, pages 275–284. IEEE, 2006.

[80] J. Rocha, O. Gkorgkas, S. Jonassen, and K. Nørvåg. Efficient processing of top-k spatial keyword queries. *Advances in Spatial and Temporal Databases*, pages 205–222, 2011.

[81] J. B. Rocha-Junior and K. Nørvåg. Top-k spatial keyword queries on road networks. In *EDBT*, pages 168–179. ACM, 2012.

[82] F. Schmid, K.-F. Richter, and P. Laube. *Semantic trajectory compression*, pages 411–416. Advances in Spatial and Temporal Databases. Springer, 2009.

[83] S. Shang, R. Ding, B. Yuan, K. Xie, K. Zheng, and P. Kalnis. User oriented trajectory search for trip recommendation. In *EDBT*, 2012.

[84] M. Sharifzadeh, M. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. *The VLDB journal*, 17(4):765–787, 2008.

[85] M. Singh, Q. Zhu, and H. Jagadish. Swst: A disk based index for sliding window spatio-temporal data. In *ICDE*, pages 342–353. IEEE, 2012.

[86] Statistics Canada. Postal code conversion file (pccf). In *http://www5.statcan.gc.ca/bsolc/olc-cel/olc-cel?lang=eng&catno= 92F0153X*, 2012.

[87] L. A. Tang, Y. Zheng, J. Yuan, J. Han, A. Leung, C. C. Hung, and W. C. Peng. On discovery of traveling companions from streaming trajectories. In *ICDE*, pages 186–197. IEEE, 2012.

[88] H. To, G. Ghinita, and C. Shahabi. A framework for protecting worker location privacy in spatial crowdsourcing. *Proc.VLDB Endow*, 7(10):919–930, 2014.

[89] G. Trajcevski, H. Cao, P. Scheuermanny, O. Wolfsonz, and D. Vaccaro. Online data reduction and the quality of history in moving objects databases. In *Proceedings of the 5th ACM international workshop on Data engineering for wireless and mobile access*, pages 19–26. ACM, 2006.

[90] L. H. U, K. Mouratidis, and N. Mamoulis. Continuous spatial assignment of moving users. *VLDB Journal*, 2010.

[91] L. H. U, M. L. Yiu, K. Mouratidis, and N. Mamoulis. Capacity constrained assignment in spatial databases. In *ACM SIGMOD*, 2008.

[92] S. Vaid, C. Jones, H. Joho, and M. Sanderson. Spatio-textual indexing for geographical search on the web. *Advances in Spatial and Temporal Databases*, pages 923–923, 2005.

[93] L.-Y. Wei, Y. Zheng, and W.-C. Peng. Constructing popular routes from uncertain trajectories. In *SIGKDD*, pages 195–203. ACM, 2012.

[94] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and parallel databases*, 7(3):257–387, 1999.

[95] R. C.-W. Wong, M. T. Özsu, P. S. Yu, A. W.-C. Fu, and L. Liu. Efficient method for maximizing bichromatic reverse nearest neighbor. *Proc. VLDB Endow.*, 2(1), Aug. 2009.

[96] R. C.-W. Wong, Y. Tao, A. W.-C. Fu, and X. Xiao. On efficient spatial matching. *VLDB*, September 2007.

[97] D. Wu, M. Yiu, G. Cong, and C. Jensen. Joint top-k spatial keyword query processing. *TKDE*, 2011.

[98] D. Wu, M. L. Yiu, C. S. Jensen, and G. Cong. Efficient continuously moving top-k spatial keyword query processing. In *ICDE*, pages 541–552. IEEE, 2011.

[99] X. Xiao, Y. Zheng, Q. Luo, and X. Xie. Inferring social ties between users with human location history. *Journal of Ambient Intelligence and Humanized Computing*, pages 1–17, 2012.

[100] H. Q. Ye and D. D. Yao. Utility-maximizing resource control: Diffusion limit and asymptotic optimality for a two-bottleneck model. In *Operations Research*, 2010.

[101] J. Yuan, Y. Zheng, X. Xie, and G. Sun. Driving with knowledge from the physical world. In *KDD*, pages 316–324. ACM, 2011.

[102] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: driving directions based on taxi trajectories. In *Proceedings of the 18th SIGSPATIAL International conference on advances in geographic information systems*, pages 99–108. ACM, 2010.

[103] D. Zhang, Y. M. Chee, A. Mondal, A. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, pages 688–699. IEEE, 2009.

[104] D. Zhang, B. C. Ooi, and A. K. H. Tung. Locating mapped resources in web 2.0. In *ICDE*, pages 521–532. IEEE, 2010.

[105] J. Zhang, X. Meng, X. Zhou, and D. Liu. Co-spatial searcher: Efficient tag-based collaborative spatial search on geo-social network. In *DASFAA*, DASFAA'12, pages 560–575, 2012.

[106] K. Zheng, Y. Zheng, N. J. Yuan, and S. Shang. On discovery of gathering patterns from trajectories. In *ICDE*, 2013.

[107] Y. Zheng, X. Xie, and W. Y. Ma. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data Engineering Bulletin*, 33(2):32–40, 2010.

[108] Y. Zheng and X. Zhou. *Computing with Spatial Trajectories*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[109] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W. Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, pages 155–162. ACM, 2005.

APPENDIX A

THEORETICAL RESULTS

A.1 The Approximation Factor of MaxSum-Appro in General Case

In this part, we show that the approximation factor of MaxSum-Appro with a general setting of α is equal to $(2 - \sqrt{2}/2 \cdot \alpha)$.

We use the same notations as defined in the proof of Theorem 2.4.2.

Consider $cost(S_o)$. Same as the proof of Theorem 2.4.2, we have $\max_{o' \in S_o} d(o', q) = d(o, q) = r_2$ and $\max_{o_1, o_2 \in S_o} d(o_1, o_2) \geq d(o, o_f) = r_1$. As a result, we have

$$cost(S_o) \geq \alpha \cdot r_2 + (1 - \alpha) \cdot r_1 \tag{A.1}$$

Consider $cost(S')$. Same as the proof of Theorem 2.4.2, we consider two cases.

Case 1: $r_1 \leq \sqrt{2} r_2$ (See Figure 2.2(a) for illustration). In this case, we have $\max_{o_1, o_2 \in S'} d(o_1, o_2) \leq d(a, b) = 2\sqrt{r_1^2 - r_1^4/4r_2^2}$. Recall that $\max_{o' \in S'} d(o', q) = d(o, q) = r_2$. As a result, we have

$$cost(S') \leq \alpha \cdot r_2 + (1 - \alpha) \cdot 2\sqrt{r_1^2 - r_1^4/4r_2^2} \tag{A.2}$$

Therefore,

$$\begin{aligned}
\frac{cost(S')}{cost(S_o)} &\leq \frac{\alpha \cdot r_2 + (1-\alpha) \cdot 2\sqrt{r_1^2 - r_1^4/4r_2^2}}{\alpha \cdot r_2 + (1-\alpha) \cdot r_1} \\
&= 1 + \frac{(1-\alpha) \cdot (2\sqrt{r_1^2 - r_1^4/4r_2^2} - r_1)}{\alpha \cdot r_2 + (1-\alpha) \cdot r_1} \\
&= 1 + \frac{(1-\alpha) \cdot (2\sqrt{1 - r_1^2/4r_2^2} - 1)}{\alpha \cdot r_2/r_1 + 1 - \alpha} \\
&< 1 + \frac{(1-\alpha) \cdot (2\sqrt{1-0} - 1)}{\alpha \cdot \sqrt{2}/2 + 1 - \alpha} \\
&= 1 + \frac{1-\alpha}{\alpha \cdot \sqrt{2}/2 + 1 - \alpha} \\
&= 2 - \frac{\sqrt{2}/2 \cdot \alpha}{1 - (1 - \sqrt{2}/2) \cdot \alpha} \\
&< 2 - \frac{\sqrt{2}/2 \cdot \alpha}{1-0} = 2 - \sqrt{2}/2 \cdot \alpha
\end{aligned}$$

Case 2: $r_1 > \sqrt{2}r_2$ (See Figure 2.2(b) for illustration). Similar to Case 1, it could be verified that $\max_{o_1,o_2 \in S'} d(o_1, o_2) \leq d(a, b) = 2r_2$. As a result, we have

$$cost(S') \leq \alpha \cdot r_2 + (1-\alpha) \cdot 2r_2 \tag{A.3}$$

Therefore,

$$\begin{aligned}
\frac{cost(S')}{cost(S_o)} &\leq \frac{\alpha \cdot r_2 + (1-\alpha) \cdot 2 \cdot r_2}{\alpha \cdot r_2 + (1-\alpha) \cdot r_1} = \frac{\alpha + (1-\alpha) \cdot 2}{\alpha + (1-\alpha) \cdot r_1/r_2} \\
&\leq \frac{2-\alpha}{\alpha + (1-\alpha) \cdot \sqrt{2}} < 2 - \alpha < 2 - \sqrt{2}/2 \cdot \alpha
\end{aligned}$$

Thus, by combining Case 1 and Case 2, we have $cost(S') \leq (2 - \sqrt{2}/2 \cdot \alpha) \cdot cost(S_o)$, which finishes the discussion.

A.2 Theoretical Error Bounds wrt the Synchronous Euclidean Distance

In this part, we show that when the mapping function $M_{\mathcal{S}}(\cdot,\cdot)$ (i.e., Synchronous Euclidean Distance) is used, the position error of DPTS is bounded (Lemma A.2.1) and the direction error of PPTS is un-bounded (Lemma A.2.2).

Lemma A.2.1 (Bounded Position Error) *Let T be a trajectory and T' be an ϵ_t-simplification of T with $\epsilon_t < \pi/2$. For each position p_i in T where $i \in [1,n]$, we have*

$$d(p_i, p_i') \leq 0.5 \cdot (1 + 1/\cos(\epsilon_t)) \cdot L_{max}$$

where $p_i' = M_{\mathcal{C}}(p_i, T')$ and $L_{max} = \max_{i\in[1,n)} len(p_{s_i}, p_{s_{i+1}}|T')$. □

Proof. Let $\overline{p_{s_k}p_{s_{k+1}}}$ be the segment of T' such that p_{s_k} is the last position with $s_k \leq i$ and $p_{s_{k+1}}$ is the first position with $s_{k+1} \geq i+1$. Consider Figure A.1(a) for illustration.

By using *triangle inequality*, we have

$$d(p_i, p_i') \leq len(p_{s_k}, p_i|T) + d(p_{s_k}, p_i')$$

and

$$d(p_i, p_i') \leq len(p_i, p_{s_{k+1}}|T) + d(p_i', p_{s_{k+1}})$$

By using the above two equations, we have

$$
\begin{aligned}
d(p_i, p_i') &\leq len(p_{s_k}, p_i|T) + d(p_{s_k}, p_i') \\
&\quad + len(p_i, p_{s_{k+1}}|T) + d(p_i', p_{s_{k+1}}) \\
&= len(p_{s_k}, p_{s_{k+1}}|T) + len(p_{s_k}, p_{s_{k+1}}|T') \\
&\leq 1/\cos(\epsilon_t) \cdot len(p_{s_k}, p_{s_{k+1}}|T') + len(p_{s_k}, p_{s_{k+1}}|T') \\
&= (1 + 1/\cos(\epsilon_t)) \cdot len(p_{s_k}, p_{s_{k+1}}|T') \\
&\leq (1 + 1/\cos(\epsilon_t)) \cdot L_{max}
\end{aligned}
$$

which completes the proof. ■

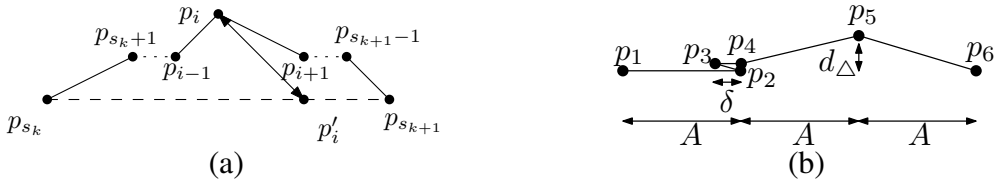


Figure A.1: Proofs of Lemma A.2.1 and Lemma A.2.2

Lemma A.2.2 (Unbounded Direction Error) *Let T be a trajectory and T' be a (direction-based) ϵ_t-simplification of T with $\epsilon_t < \pi/2$. Let $T_{\mathcal{C}}$ be a (position-based) simplified trajectory of T such that $|T_{\mathcal{C}}| = |T'|$ and the error of $T_{\mathcal{C}}$ under the Synchronous Euclidean Distance is minimized. There exists a dataset such that $\epsilon(T_{\mathcal{C}}) \approx \pi$ and $\epsilon(T') \approx 0$.* □

Proof. We prove by constructing a problem instance as shown in Figure A.1(b). $T = (p_1, p_2, p_3, p_4, p_5, p_6)$ is a trajectory, where all positions except p_5 are along a horizontal line such that $d(p_2, p_1) = d(p_4, p_1) = A$, $d(p_3, p_1) = A - \delta$ and $d(p_6, p_1) = 3A$. Here, A is a positive real number. Besides, p_5 is located a little bit above the horizontal line with its perpendicular distance to the horizontal line equal to $d_\triangle$. We have $\delta << d_\triangle$.

Suppose that we can only keep 5 positions in the simplified trajectory. In other words, we have to remove 1 position from the 6 positions. If we consider preserving

the direction information, p_4 will be removed and thus $T' = (p_1, p_2, p_3, p_5, p_6)$. As a result, $\epsilon(T') \approx 0$. If we consider preserving the position information, p_3 will be removed (since the position error of p_3 under the Synchronous Euclidean Distance is the smallest) and thus $T_{\mathcal{C}} = (p_1, p_2, p_4, p_5, p_6)$. As a result, $\epsilon(T_{\mathcal{C}}) \approx \pi$. ■

APPENDIX B

ALGORITHMS AND ADAPTATIONS OF EXISTING ALGORITHMS

B.1 Adaptations of Existing Trajectory Simplification Methods

The existing trajectory simplification methods described in Section 4.3 can be adapted to the Min-Size problem. However, these adaptations have limitations. We discuss both the adaptations and their limitations next.

Let $T = (p_1, p_2, ..., p_n)$ be a trajectory and ϵ_t be the error tolerance.

Split. It splits T at a *splitting position* into two sub-trajectories if the error of approximating T with one segment exceeds ϵ_t. Let i be $\arg\max_{1<h<n}\{\triangle(\theta(\overline{p_h p_{h+1}}), \theta(\overline{p_1 p_n}))\}$. We select p_i as the splitting position. It then performs the same procedure recursively on each of the sub-trajectories. At the end, it approximates each resulting trajectory with one segment.

The time complexity of Split is $O(n^2)$ because in the worst case, it always chooses the second position of the remaining trajectory as the splitting point.

Merge. It regards each segment in T as a sub-trajectory. It then iteratively merge two adjacent sub-trajectories T_1 and T_2 into one if the error of approximating the merged trajectory with one segment is bounded by ϵ_t until no such merge operations are possible. When multiple candidates of two adjacent sub-trajectories for merging are available, we choose the one with the smallest error when the corresponding merged trajectory is approximated with one segment. At the end, it approximates each resulting

sub-trajectory with one segment.

The time complexity of Merge is $O(n^2)$ since, in the worst case, it always merges the first two remaining sub-trajectories and the final simplified returned is $\overline{p_1 p_n}$.

Greedy. Let T' be the simplified trajectory to be computed by Greedy, which is initialized as (p_1) at the beginning. Greedy scans the positions of T starting from p_1 sequentially. It iteratively finds the position p_i such that $\epsilon(\overline{p_e p_i}) \leq \epsilon_t$ and $\epsilon(\overline{p_e p_{i+1}}) > \epsilon_t$, where p_e is the last position in T' currently maintained in the algorithm, and then appends p_i to T'. At the end, it appends p_n to T'.

The time complexity of Greedy is $O(n^2)$. This is because, in the worst case, where the output T' contains only two positions, p_1 and p_n, it has to compute $\epsilon(\overline{p_1 p_3})$, $\epsilon(\overline{p_1 p_4})$, ..., $\epsilon(\overline{p_1 p_n})$. Since each $\epsilon(\overline{p_1 p_3})$ takes $O(n)$ time, the time complexity is $O(n^2)$.

B.2 The Dynamic Programming Algorithm for the Min-Size Problem

In the following, we design a dynamic programming (DP) algorithm for the Min-Size problem. Let $T[i, j]$ be the sub-trajectory of T, which starts at p_i and ends at p_j where $1 \leq i < j \leq n$, i.e., $T[i, j] = (p_i, p_{i+1}, ..., p_j)$. We denote by $S[i, j]$ the optimal solution of the Min-Size problem where the trajectory input is $T[i, j]$ and the error tolerance is still ϵ_t. Let $N[i, j]$ denote the number of segments maintained in $S[i, j]$ (which is equal to the size of the optimal solution (i.e., $S[i, j]$) minus one). Note that minimizing the number of segments in the simplified trajectory is exactly equal to minimizing the size of the simplified trajectory.

Consider $S[i, j]$. There are two cases.

- *Case 1:* $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$. That is, $T[i, j]$ could be approximated by one segment (i.e., $\overline{p_i p_j}$) with the error bounded by the tolerance. In this case, we have $S[i, j] =$

(p_i, p_j) and $N[i, j] = 1$.

- *Case 2:* $\epsilon(\overline{p_i p_j}) > \epsilon_t$. In this case, $T[i, j]$ cannot be approximated with one segment only such that the error tolerance constraint is satisfied. Thus, there exists at least one position p_{k^*} in $T[i, j]$ excluding its start position and its end position where $i < k^* < j$. Besides, we have $N[i, j] = N[i, k^*] + N[k^*, j]$ and $k^* = \arg\min_{i<k<j}\{N[i, k] + N[k, j]\}$.

In conclusion, we can compute $N[i, j]$ in the way as shown in Equation (B.1).

$$N[i, j] = \begin{cases} 1 & \text{if } \epsilon(\overline{p_i p_j}) \leq \epsilon_t \\ \min_{i<k<j}\{N[i, k] + N[k, j]\} & Otherwise \end{cases} \tag{B.1}$$

Based on Equation (B.1), we can design a DP algorithm for the Min-Size problem. Note that $N[1, n]$ corresponds to the number of segments maintained in the optimal solution of the Min-Size problem with the input as T.

Complexity Analysis. For each problem instance $N[i, j]$, we first check whether $\epsilon(\overline{p_i p_j}) \leq \epsilon_t$. The cost is simply $O(n)$ since it computes $\triangle(\overline{p_i p_j}, \overline{p_h p_{h+1}})$ for $i \leq h < j$ in order to obtain $\epsilon(\overline{p_i p_j})$. Besides, it takes $O(1)$ and $O(n)$ time to obtain $N[i, j]$ in Case 1 and Case 2, respectively. Thus, the overall cost for a specific problem instance of $N[i, j]$ is still $O(n)$. Since we have in total $O(n^2)$ possible problem instances of $N[i, j]$, the time complexity of the DP algorithm is $O(n^2 \cdot n) = O(n^3)$.

B.3 The Dynamic Programming Algorithm for the Min-Error Problem

Let $T = (p_1, p_2, ..., p_n)$ $(n \geq 2)$ be a trajectory and $W \leq n$ be a positive integer. We denote by Min-Error(T, W) the Min-Error problem with the input trajectory as T and the input storage budget as W. Then, we denote by $S[i, k]$ ($i \in [1, n]$ and $k \in [2, W]$)

the solution of Min-Error($T[i:n]$, k) which is a simplification of $T[i:n]$ and denote by $E[i,k]$ the error of $S[i,k]$ (note that the case of $k = 1$ is trivial and thus it is ignored). Then, we have the following Equation (B.2) for computing $E[i,k]$. The idea is that in the case of $k \geq n-i+1$, the optimal solution is T itself and thus it has the error equal to 0; in the case of $W = 2$, the optimal solution is (p_i, p_n) and thus it has the error equal to $\epsilon(\overline{p_i p_n})$; and in other cases, the optimal solution is best one among $(p_i, p_h) \diamond S[h, k-1]$ for $h = i+1, i+2, ..., n$, where $\diamond$ is the "concatenation" operator, and thus it has the error equal to $\min_{h\in[i+1,n]} \max\{\epsilon(\overline{p_i p_h}), E[h, k-1]\}$.

$$E[i,k] = \begin{cases} 0 & k \geq n-i+1 \\ \epsilon(\overline{p_i p_n}) & k = 2 \\ \min_{h\in[i+1,n]} \max\{\epsilon(\overline{p_i p_h}), E[h,k-1]\} & \text{Otherwise} \end{cases} \quad \text{(B.2)}$$

Correspondingly, we have the following Equation (B.3) for computing $S[i,k]$ ($i \in [1,n]$ and $k \in [2,W]$).

$$S[i,k] = \begin{cases} T[i:n] & k \geq n-i+1 \\ (p_i, p_n) & k = 2 \\ (p_i, p_{h^*}) \diamond S[h^*, k-1] \text{ where } h^* = & \\ \arg\min_{h\in[i+1,n]} \max\{\epsilon(\overline{p_i p_h}), E[h,k-1]\} & \text{Otherwise} \end{cases} \quad \text{(B.3)}$$

DP is the dynamic programming algorithm based on Equation (B.2) and Equation (B.3).

We analyze the time complexity of the *DP* algorithm as follows. The process of computing $S[i,k]$ (via Equation (B.3)) relies on the results of $E[i,k]$ (via Equation (B.2)) ($i \in [1,n]$ and $k \in [2,W]$).

Consider the part of computing all instances of $E[i,k]$. We have $O(n \cdot W)$ instances of $E[i,k]$. For a specific $E[i,k]$, the cost of case $k \geq n-i+1$ is simply $O(1)$, the cost of case $k = 2$ is simply $O(n)$, and the cost of one of the other cases is $O(n^2)$. Note that a straightforward method for computing an instance of $\epsilon(\overline{p_i p_j})$ ($1 \leq i \leq$

$j \leq n$) runs in $O(n)$ time. Thus, the time cost of computing all $E[i,k]$'s is $O(Wn^3)$ (=$O(Wn \cdot (1+n+n^2))$).

Consider the part of computing all instances of $S[i,k]$. Again, we have $O(n \cdot W)$ instances of $S[i,k]$. For a specific $S[i,k]$, based on the result of $E[i,k]$, the cost of case $k \geq n-i+1$ is $O(1)$, the cost of case $k=2$ is $O(1)$, and the cost of one of the other cases is $O(n)$ (provided that $\epsilon(\overline{p_i p_h})$ has been maintained when computing $E[i,k]$'s). As a result, the time cost of computing all $S[i,k]$'s (based on the results of $E[i,k]$'s) is $O(Wn^2)$ (=$(Wn \cdot (1+1+n))$).

In conclusion, the time complexity of *DP* is $O(Wn^3)$. Besides, the space complexity of *DP* is simply $O(n^2)$.

B.4 Implementation & Time Complexity (Algorithm 10)

Before we analyze the time complexity of Algorithm 10, we discuss how we compute $mcar(\theta[i:j])$ for $1 \leq i < j \leq n$ first.

We compute $mcar(\theta[i:j])$ as follows. Suppose $[\theta_a, \theta_b]$ is $mcar(\theta[i:j])$. As discussed before, θ_a and θ_b are two directions in $\theta[i:j]$. We construct a search space $\mathcal{R}$ for $[\theta_a, \theta_b]$ as $\mathcal{R} = \{[\theta_{h+1}, \theta_h] | h \in [1, j-i]\}$ where $\theta_1, \theta_2, ..., \theta_{j-i}$ is the list of the directions in $\theta[i:j]$ sorted in ascending order and θ_{h+1} corresponds to θ_1 if $h = j-i$. We claim that $[\theta_a, \theta_b]$ is in $\mathcal{R}$.

Lemma B.4.1 *Let* $[\theta_a, \theta_b]$ *be* $mcar(\theta[i:j])$ *where* $i, j \in [1,n]$ *and* $i < j$. *Then,* $[\theta_a, \theta_b] \in \mathcal{R}$. □

Proof. First, we know that $[\theta_a, \theta_b]$ belongs to $\{[\theta_c, \theta_d] | \theta_c, \theta_d \in \theta[i:j]\}$ which we denote by $\mathcal{R}'$.

Second, we prove the lemma by showing that any angular range in $\mathcal{R}' - \mathcal{R}$ is not a covering angular range of $\theta[i:j]$. For any angular range $[\theta_h, \theta_k]$ ($h, k \in [1, j-i]$) in $\mathcal{R}' - \mathcal{R}$, we have $h \neq k+1$ ($k+1$ corresponds to 1 if $k = j-i$). We then deduce that θ_{k+1} is not covered by $[\theta_h, \theta_k]$ and thus $[\theta_h, \theta_k]$ is not a covering angular range of $\theta[i:j]$. ■

$\mathcal{R}$ has the following nice property. Each of the ranges in $\mathcal{R}$ is a covering angular range of $\theta[i:j]$ which could be verified by the fact that for each range $[\theta_{h+1}, \theta_h]$ where $h \in [1, j-i]$, all directions in $\theta[i:j]$ (except for θ_h and θ_{h+1}) do not fall in $[\theta_h, \theta_{h+1}]$ (since θ_h and θ_{h+1} are adjacent in the sorted list) and thus they fall in $[\theta_{h+1}, \theta_h]$.

Property 8 *Any range in $\mathcal{R}$ is a covering angular range of $\theta[i:j]$.* □

By using Lemma B.4.1 and Property 8, we compute $mcar(\theta[i:j])$ by first constructing $\mathcal{R}$ which has the cost of $O((j-i)\log(j-i))$ and then finding the range with the smallest span in $\mathcal{R}$ which has the cost of $O(j-i)$. Thus, $mcar(\theta[i:j])$ (and thus $\xi(\theta[i:j])$) could be computed in $O((j-i)\log(j-i))$ time.

Now, we are ready to analyze the time complexity of Algorithm 10. Suppose that $T' = (p_{s_1}, p_{s_2}, ..., p_{s_m})$ is a simplification returned by Algorithm 10. Then, the time cost of Algorithm 10 is dominated by the costs of computing $\xi(mcar(\theta[s_1:j]))$'s for $j = s_1+1, s_1+2, ..., s_2+1$, $\xi(mcar(\theta[s_2:j]))$'s for $j = s_2+1, s_2+2, ..., s_3+1$, ..., and $\xi(mcar(\theta[s_{m-1}:j]))$ for $j = s_{m-1}+1, s_{m-1}+2, ..., s_m$.

We compute $\xi(mcar(\theta[s_k:j]))$ for $j = s_k+1, s_k+2, ..., s_{k+1}+1$ *incrementally* as follows. When computing $\xi(mcar(\theta[s_k:j]))$ where $j \in [s_k+1, s_{k+1}+1]$, we maintain two data structures, namely a *binary search tree* and a *priority heap*, *dynamically* maintaining the sorted list of $\theta[s_k:j]$ and the corresponding $\mathcal{R}$ (for each angular range in $\mathcal{R}$, its span is used as its key in the priority queue), respectively. With the information of the priority heap built on $\mathcal{R}$, $\xi(mcar(\theta[s_k:j]))$ could be computed in

$O(1)$ time.

We consider the problem of how to maintain the binary search tree and the priority heap for computing $\xi(mcar(\theta[s_k : j+1))$ based on those for computing $\xi(mcar(\theta[s_k : j]))$. Note that the only difference between $\theta[s_k : j]$ and $\theta[s_k : j+1]$ is that $\theta[s_k : j+1]$ includes one more direction, i.e., $\theta(\overline{p_j p_{j+1}})$. Thus, the cost of maintaining the binary search tree for $\theta[s_k : j+1]$ is simply $O(\log(s_{k+1} - s_k))$ (note that $j \in [s_k + 1, s_{k+1} + 1]$). Besides, the only difference between the search space $\mathcal{R}$ for computing $\xi(mcar(\theta[s_k : j]))$ and the one for computing $\xi(mcar(\theta[s_k : j+1))$ is that one range in the former, say, $[\theta_c, \theta_d]$, is split into two, i.e., $[\theta_c, \theta(\overline{p_j p_{j+1}})]$ and $[\theta(\overline{p_j p_{j+1}}), \theta_d]$, in the latter, where θ_c and θ_d are two directions adjacent to $\theta(\overline{p_j p_{j+1}})$ in the sorted list of $\theta[s_k : j+1]$ which could be retrieved in $O(\log(s_{k+1} - s_k))$ time by using the binary search tree for $\theta[s_k : j+1]$. Therefore, we can also maintain the priority heap for computing $\xi(mcar(\theta[s_k : j+1]))$ in $O(\log(s_{k+1} - s_k))$ time (we first retrieve θ_c and θ_d and then update three elements in the priority heap, namely deleting $[\theta_c, \theta_d]$ and inserting $[\theta_c, \theta(\overline{p_j p_{j+1}})]$ and $[\theta(\overline{p_j p_{j+1}}), \theta_d]$). In conclusion, the cost for maintaining the binary search tree and the priority heap for computing $\xi(mcar(\theta[s_k : j+1]))$ based on those for computing $\xi(mcar(\theta[s_k : j]))$ is $O(\log(s_{k+1} - s_k))$. As a result, the cost of computing $\xi(mcar(\theta[s_k : j]))$ for $j = s_k + 1, s_k + 2, ..., s_{k+1} + 1$ using the above incremental implementation is $O((s_{k+1} - s_k) \cdot \log(s_{k+1} - s_k))$, which further implies that the time complexity of Algorithm 10 using this implementation is

$$\sum\nolimits_{k \in [1,m)} O((s_{k+1} - s_k) \log(s_{k+1} - s_k)) = O(n \log n)$$