FRAMEWORKS, ALGORITHMS, AND SYSTEMS FOR EFFICIENT DISCOVERY OF DATA-BACKED FACTS

by

GENSHENG ZHANG

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
DECEMBER 2017

To my wife Changling and my son Maxwell.

ACKNOWLEDGEMENTS

I would like to express my appreciation and thanks to my advisor Professor Dr. Chengkai Li. I would like thank Dr. Li for constantly being an inspiring/encouraging mentor during the course of my doctoral study. I would also like to thank Dr. Sharma Chakravarthy, Dr. Gautam Das, and Dr. Leonidas Fegaras for serving as my committee members. I would especially like to thank my labmates of the Innovative Database and Information Systems Research (IDIR) Lab and friends from the Computer Science and Engineering department – Ning Yan, Nandish Jayaram, Naeemul Hassan, Afroza Sultana, Fatma Arslan, Damian Jimenez, Josue Caraballo, Abolfazl Asudeh, Sona Hasani, and many others. All of you have been there to support me and make research fun.

A special thanks to my family for all of the sacrifices that youve made on my behalf. Words cannot express how grateful I am to my wife, my parents, my brother and my sisters. You are the people who set the example and who made me who I am.

SEPTEMBER 22, 2017

ABSTRACT

FRAMEWORKS, ALGORITHMS, AND SYSTEMS
FOR EFFICIENT DISCOVERY OF DATA-BACKED FACTS

GENSHENG ZHANG, Ph.D.
The University of Texas at Arlington, 2017

Supervising Professor: Dr. Chengkai Li

This thesis studies the problem of finding facts from semi-structured and structured data. The amount of data in our world is exploding, and the proliferation of data is making them increasingly inaccessible. It is now more challenging than ever how to efficiently identify useful information where a vast amount of data is available.

This thesis first studies the problem of finding facts in semi-structured data, specifically, in knowledge graphs. We built Maverick, a general, extensible framework that discovers exceptional facts about entities in knowledge graphs. We model an exceptional fact about an entity of interest as a context-subspace pair, in which a subspace is a set of attributes and a context is defined by a graph query pattern of which the entity is a match. The entity is exceptional among the entities in the context, with regard to the subspace. The search spaces of both patterns and subspaces are exponentially large. Maverick conducts beam search on the patterns which uses a match-based pattern construction method to evade the evaluation of invalid patterns. It applies two heuristics to select promising patterns to form the beam in each iteration. Maverick traverses and prunes the subspaces organized as

a set enumeration tree by exploiting the upper bound properties of exceptionality scoring functions. Maverick demonstrated substantial performance improvement of the proposed framework over the baselines as well as its effectiveness in discovering exceptional facts.

This thesis further investigates the problem of finding facts in structured data. In particular, our objective is to find top-k prominent streaks in multi-dimensional multi-sequence data. Given a sequence of values, a prominent streak is a long consecutive subsequence consisting of only large (small) values, e.g., consecutive games of outstanding performance in sports. To efficiently discover prominent streaks, we exploited the properties of local prominent streaks (LPS), which is a superset of prominent streaks. We showed that LPS-based algorithms exhibited orders of magnitude performance improvement against the baseline method.

This thesis also presents a fact-finding system FactWatcher, which helps journalists identify data-backed, attention-seizing facts which serve as leads to news stories. FactWatcher discovers three types of facts, including situational facts, one-of-the-few facts, and prominent streaks, through a unified suite of data model, algorithm framework, and fact ranking measure. Furthermore, FactWatcher provides multiple features in striving for an end-to-end system, including fact ranking, fact-to-statement translation and keyword-based fact search.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

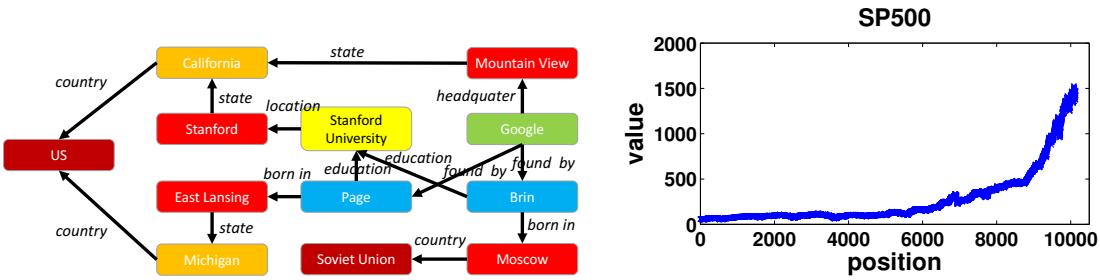
Introduction

The amount of data in our world is exploding, and the proliferation of data is making them increasingly inaccessible. Data produced by an application can be one of the following three types: unstructured (e.g., text documents), semi-structured (e.g., XML documents, graphs), and structured (e.g., relational databases). It is unstructured data the most prevalent type of data.[1] Only a very small percentage of data produced is truly structured. Unstructured data is typically text-heavy, and difficult to be understood by automated systems as compared to (semi-)structured data, due to irregularities and ambiguities in texts. Recently, works such as Linked Open Data (LOD)[2] and Open Information Extraction (Open IE)[1] have gained a lot of traction, which focus on adding/extracting (semi-)structured information in/from unstructured data. The promising progress of these efforts greatly improves the availability of public information as structured and semi-structured data.

This thesis mainly investigates finding facts from semi-structured data and structured data. Both structured and semi-structured data are produced and accumulated in a rich variety of applications, including search, recommendation, and business intelligence. In particular, we are interested in knowledge graphs and sequence data. Knowledge graphs typical have loosely defined schema mainly for semantic interpretation, which makes them semi-structured data. Knowledge graphs such as DBpedia [2], Freebase [3], Wikidata [4], and YAGO [5] record properties of

[1] *https://www.forbes.com/2007/04/04/teradata-solution-software-biz-logistics-cx_rm_0405data.html*

[2] *http://linkeddata.org/*



a An excerpt of a knowledge graph.

b An excerpt of stock price sequence (S&P500).

Figure 1.1: Examples of knowledge graph and sequence data.

and relationships between real-world entities. For example, as shown in Fig. 1.1a, both Page and Brin are founders of Google and alumni of Stanford University, while Page was born in US, Brin was born in Soviet Union. Sequence data is a type of structured data, in which tuples are ordered temporally. Sequence data record series of values or events. Examples include stock quotes, sports statistics, temperature measurement, Web usage logs, network traffic logs, Web clickstream, customer transaction sequence, social media statistics. Fig. 1.1b Shows the stock price of S&P500 from June 01, 1960 (position 0) to June 01, 2000.

We call information that is stated in or can be inferred from data data-backed facts, or simply facts. Based on availability, facts can be roughly categorized into three types: (1) Primitives. The primitives are the facts that are immediately available. For example, given a knowledge graph such as Fig. 1.1a, the fact "Page is a founder of Google" is a primitive, since it is stated in the graph. We assume that the data of interest contains primitives only. (2) Aggregates. The aggregates are the facts that can be derived by performing some simple analysis. For example, one can derive a fact such as "S&P 500 recorded historical high on June 3, 1987." by applying the *max* function on all S&P 500 prices prior to the date. (3) Obscures. Obscures are the

facts that may only be unearthed by non-trivial analysis. Consider several factual statements in published news articles:

1. "Denzel Washington followed Sidney Poitier as only the second black to win the Best Actor award." (*abcnews.go.com*)
2. "This was Brazil's first own goal in World Cup history ..." (*yahoo.com*)
3. "Hillary Clinton becomes first female presidential nominee." (*chicagotribune.com*)
4. "This month the Chinese capital has experienced 10 days with a maximum temperature in around 35 degrees Celsius – the most for the month of July in a decade." (chinadaily.com.cn)
5. "The Nikkei 225 closed below 10000 for the 12th consecutive week, the longest such streak since June 2009." (bloomberg.com)
6. "He (LeBron James) scored 35 or more points in nine consecutive games and joined Michael Jordan and Kobe Bryant as the only players since 1970 to accomplish the feat." (nba.com)
7. "Only player in NBA history to average at least 20 points, 10 rebounds and 5 assists per game for 6 consecutive seasons. (Kevin Garnett)" (en.wikipedia.org)

All above statements usually cannot be easily obtained without conducting complex analysis.

This thesis mainly focuses on finding obscure facts in knowledge graphs and sequence data. Specifically, we are interested in the facts similar to above statements 1—7. We call statements 1—3 *exceptional facts* (chapter 2), which reveal extraordinary characteristics about entities in knowledge graphs; and call statements 4—7 *prominent streaks* (chapter 3), which exhibit sensational properties about streaks in sequence data. Discovery of exceptional facts and prominent streaks is useful to important applications such as computational journalism [6, 7], recommendation systems, and data cleaning. a) In *fact-finding* [8, 9, 10, 11, 12], journalists are interested

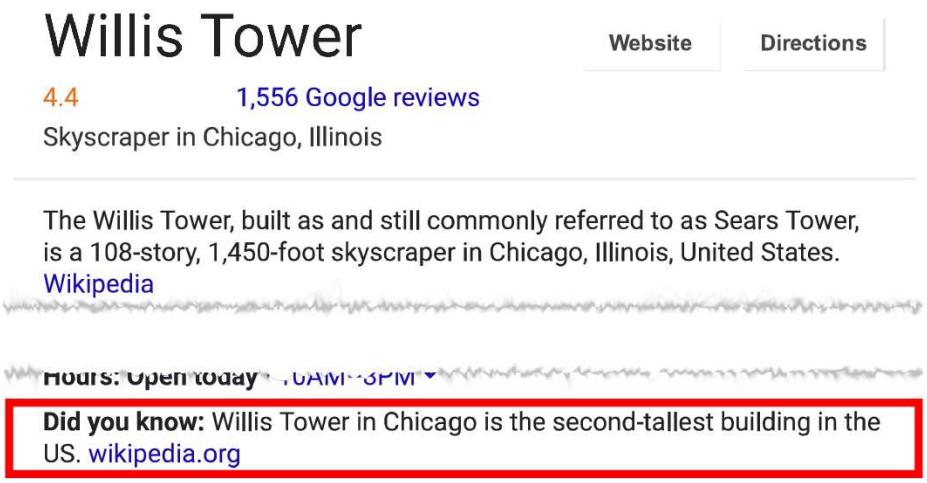


Figure 1.2: Google's "Did you know" feature displays exceptional facts about entities in search results. (*Google.com*)

in monitoring data and discovering attention-seizing factual statements. Examples include statements 1, 2, and 4–7. These facts help make news stories substantiated and interesting, and they may even become leads to news stories. b) In *fact-checking* [13, 14, 15], for vetting the statements made by humans, fact-checkers at news organizations such as The Washington Post, CNN, and PolitiFact can compare the statements with automatically-discovered facts. For example, an algorithm may find that Hillary Clinton is the second female presidential nominee, which contradicts with the statement 3 above. [3] c) Exceptional facts can help promote friends, news, products, and search results in various recommendation systems. For example, Google's "Did you know" feature displays exceptional facts about entities in search results, as shown in Fig. 1.2. d) When the discovered facts are inconsistent with known truth or apparent common sense, it reveals incomplete data or data errors. Such insights aid knowledge base cleaning and completion. For example, the above statement 3

[3]The first female presidential nominee was Victoria Woodhull, according to *http://www.snopes.com/victoria-woodhull-hillary-clinton/*.

may be generated using an incomplete source that misses the nomination of Victoria Woodhull.

One of the major challenges faced by exceptional fact discovery and prominent streak discovery is the prohibitively large search space. To tackle the challenges, we introduce efficient frameworks and algorithms for discovery of exceptional facts and prominent streaks, such as beam-search based framework Maverick (Chapter 2) and local prominent streak based algorithms (Chapter 3). We also present system FactWatcher to demonstrate the effectiveness of the frameworks and algorithms in Chapter 4.

CHAPTER 2

Discovering Exceptional Facts from Knowledge Graphs

2.1 Introduction

This chapter introduces Maverick [16], a framework that, given an entity in a knowledge graph, discovers *exceptional facts* about the entity. Informally, such exceptional facts separate the entity from many other entities. An exceptional fact consists of three components: an *entity of interest*, a *context*, and a set of *qualifying attributes*. In each exceptional fact, among all entities in the context, the entity of interest is one of the few or even the only one that bears a particular value combination on the qualifying attributes.

Given an entity in a knowledge graph, an integer k, and an exceptionality scoring function, the objective of *exceptional fact discovery* is to find the top-k highest scored pairs of (context, attribute set). The entity is exceptional with regard to the attributes, while at the same time belonging to the context together with other entities. This description hinges upon two concepts—context and attribute—which we explain below.

- The *attributes* of an entity are the entity's incoming/outgoing edge labels, and the attribute values are the entity's direct neighbors. For example, Fig. 2.1 is an

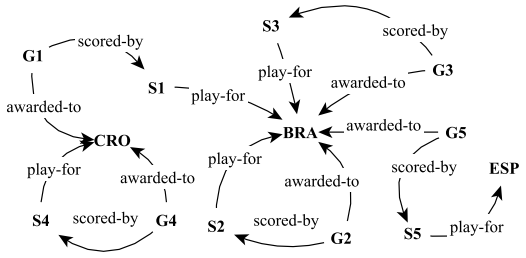


Figure 2.1: An excerpt of a knowledge graph.

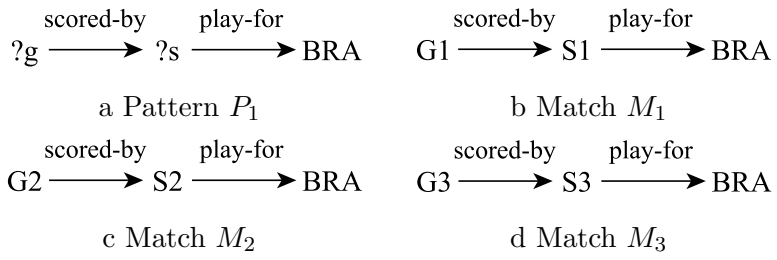


Figure 2.2: Pattern P_1 and variable ?g define a context consisting of all the goals scored by BRA players; M_1, M_2, M_3 are matches to P_1 in Fig. 2.1.

excerpt of a knowledge graph about FIFA World Cup, in which the edge labeled *awarded-to* from node G1 to CRO captures the fact that the goal is awarded to the team Croatia. Entity G1 has two attributes *scored-by* and *awarded-to*, with values S1 and CRO, respectively.

- A *context* is a set of entities sharing some common characteristics defined in a pattern query. In Fig. 2.2a, pattern P_1 and the variable ?g in it define a context C_1 of all the goals scored by players of team Brazil. Figs. 2.2b-2.2d show P_1's matches in Fig. 2.1. For instance, match M_1 (Fig. 2.2b) is a subgraph of Fig. 2.1, in which ?g of P_1 is mapped to G1. Hence, G1 belongs to context C_1. Similarly, G2 and G3 in Fig. 2.1 also belong to C_1 based on M_2 and M_3, while G4 and G5 are not part of C_1.
- With respect to a *subspace* (i.e., a set of attributes), an entity is exceptional in a context if its attribute values deviate from the values of other entities in the same context. For example, the value of attribute *awarded-to* for G1 is CRO, while the value is BRA for both G2 and G3. The degree of exceptionality of an entity varies by different contexts and subspaces. For instance, one interpretation of statement 1 is that the context is the Academy Award Best Actor winners and the qualifying attribute is ethnicity; an alternative interpretation is that the context is all African Americans and the qualifying attribute is the award. Under some definitions of exceptionality, the second interpretation may render Denzel Washington

more exceptional, since there are a lot more African Americans than winners of the award.

A holistic solution to exceptional fact discovery may be expected to synthesize whatever types of available data (structured databases, graphs, text, and so on), which is beyond the scope of this work. Instead, our focus is on knowledge graphs which are becoming increasingly important to analytics and intelligence applications. To the best of our knowledge, there is no previous study on discovering exceptional facts about entities in knowledge graphs. The two most related areas are *outlier detection* in graphs [17, 18, 19, 20] and *outlying aspect mining* [21, 22, 10, 12, 23, 24, 25]. Duan et al. [24] and Vinh et al. [25] discussed the differences between these two areas. They achieve different goals. Outlier detection searches for all outlying objects among a set of objects. Outlying aspect mining, however, focuses on only one given object and returns the subspaces of attributes in which the object is relatively outlying, regardless of its true degree of outlyingness. In terms of objectives and problem modeling, the exceptional fact discovery problem formulated in this chapter is closer to outlying aspect mining than outlier detection. However, it focuses on graph data. In contrast, existing outlying aspect mining methods [21, 10, 12, 23] assume a single relational table. These methods take a tuple as input and returns two disjoint attribute sets. The first set of attributes define the context, i.e., the tuples having values identical to that of the input tuple on the attributes. On the second set of attributes, the input tuple has peculiar values compared to other tuples belonging to the context.

However, these methods for outlying aspect mining cannot be effectively applied to knowledge graphs, since they are specifically devised for single tables only. A seemingly plausible idea can be to represent a knowledge graph as a single table and then to apply the existing methods on the table. Consider the single-table model of RDF proposed in [26]. When adapting it for a knowledge graph, each tuple (row)

is for an entity v and each attribute (column) corresponds to an edge label in the knowledge graph. The attribute is also associated with an edge direction—either incoming into or outgoing from v. The value at the junction of the row and the column is an entity or a set of entities adjacent to v via edges with the label and direction given by the column. Given this single-table representation of the knowledge graph, at least a few major problems render the existing outlying aspect mining methods inapplicable. *First*, in these methods a context, defined by a set of attributes, consists of the tuples having values identical to that of the input tuple. In other words, the context is the result of a conjunctive query over the attributes. For knowledge graphs, however, a context is defined by a graph pattern query, which cannot be captured by conjunctive queries on attributes in the aforementioned single-table representation. More specifically, an edge in the pattern may not be adjacent to the input entity and thus does not correspond to any of the entity's attributes. Hence, evaluating a pattern may involve self-joins of the single-table. Existing outlying aspect mining methods are not designed to accommodate joins. *Second*, the aforementioned set values in the single-table representation are not considered in the existing methods. An adaptation of the methods will thus require at least joins which, as mentioned above, are not supported by the methods. *Third*, due to the heterogeneity and scale of a large knowledge graph, such a single-table is extremely wide and sparse, which is well beyond the capacity of the existing methods because of the intrinsic exponential complexity of the problem's search space.

To discover the exceptional facts about an entity, we must explore two extremely large search spaces, one of patterns and the other of attribute subspaces. Section 2.5.1 shows that the number of patterns is at least exponential in the size of the graph. It is also clear that the number of subspaces is exponential in the number of attributes since a subspace is a combination of attributes. It is not computationally

feasible to exhaustively enumerate all possible patterns and subspaces. Furthermore, it is challenging to prune patterns and subspaces, due to the non-existence of *downward closure property* (i.e., *anti-monotone property*) on typical exceptionality scoring functions.

To tackle these challenges, this chapter introduces Maverick, a beam-search based framework. Given an input entity, Maverick discovers the top-k context-subspace pairs that give the entity the highest expectionality scores. Maverick allows an application to plug in any exceptionality scoring function based on the application needs. Conceptually, Maverick organizes the search space of patterns as a partial order defined by the subsumption relation on patterns and the search space of attribute subspaces as a set enumeration tree [27]. Intuitively, the search for top-k context-subspace pairs is performed in a nested-loop fashion in which the outer loop enumerates patterns and the inner loop enumerates subspaces. Maverick conducts breath-first beam search [28] on the space of patterns, starting from a pattern with a single variable node. On each visited pattern, Maverick applies a set of heuristics to prune its children so that Maverick visits at most w patterns at each level, where w is the beam width. Each visited pattern is evaluated over the knowledge graph to obtain the contexts it defines. For each context, Maverick calculates the input entity's exceptionality scores in different subspaces. It exploits an upper bound for exceptionality score to guide the traversal of the subspaces. The supersets of a subspace are pruned if their upper-bound scores are below the current top-k scores.

This chapter reports the results of experiments on two real-world knowledge graphs, which verify Maverick's effectiveness in finding exceptional facts. The experiments compared the performance of a breath-first search method and the beam search method coupled with different candidate-selection heuristics. The experiment results establish that, even though the breath-first search method may evaluate more

patterns in a fixed time frame than the beam search methods, it is not as effective as the beam search method using the proposed heuristics. We have also included some exceptional facts discovered by Maverick to demonstrate its practicality.

2.2 Problem Formulation

In this section we formally define the data model of knowledge graphs, the concepts of context, attribute, and subspace, and the problem of exceptional fact discovery.

Knowledge Graphs

A knowledge graph $G(V_G, E_G)$ is a set of RDF [29] triples with node set $V_G \subseteq \mathcal{I}$ and edge set $E_G \subseteq V_G \times \mathcal{I} \times V_G$, where $\mathcal{I}$ is the universe of IRIs.[1] In Fig. 2.1, there are three kinds of entities: goals (e.g., G1), players (e.g., S1), and teams (e.g., BRA). (Without loss of generality, we use an entity's name as its identifier (IRI) in the ensuing examples, assuming entity names are unique.) Three different types of edge labels represent different relationships: each player plays for a team (*play-for*), and each goal is scored by a player (*scored-by*) and is awarded to a team (*awarded-to*). For example, there is an own goal, as G1 is scored by S1, a player of BRA, but awarded to CRO.

Patterns and Contexts

Definition 1 (Pattern P)**.** A pattern is a weakly connected graph[2] $P(V_P, E_P)$, where $V_P \subseteq \mathcal{I} \cup \mathcal{V}$, $E_P \subseteq V_P \times \mathcal{I} \times V_P$, and $\mathcal{V}$ is the universe of variables. We also denote by $X_P \subseteq V_P$ the variables occurring in P. △

[1] For the sake of simplicity, we do not consider blank nodes and literals.

[2] A weakly connected graph is a directed graph of which the corresponding undirected graph is connected.

Definition 2 (Match M)**.** A match $M(V_M, E_M)$ to a pattern $P(V_P, E_P)$ is a subgraph of G ($V_M \subseteq V_G$ and $E_M \subseteq E_G$) such that there exists a bijection $f : V_P \to V_M$ satisfying the following conditions:

- $|V_M| = |V_P|$, $|E_M| = |E_P|$;
- $\forall\ (v_i, l, v_j) \in E_P \Rightarrow (f(v_i), l, f(v_j)) \in E_M$;
- $\forall\ (u_i, l, u_j) \in E_M \Rightarrow (f^{-1}(u_i), l, f^{-1}(u_j)) \in E_P$;
- $\forall\ v \in \mathcal{I} \Rightarrow f(v) = v$.

In short, a subgraph M of G is a match to pattern P if M is edge-isomorphic to P and, for each non-variable node v in P, $f(v)$ has the same identifier. △

Note that the semantics of patterns in our definition is similar to that of *basic graph patterns* in [30, 31]. However, there are two main differences. One is that patterns in this work are weakly connected. The other is that a match to a pattern is required to be edge-isomorphic to the pattern. Neither of them is enforced in [30, 31].

Definition 3 (Range of Variable R_x^P)**.** Let $\mathcal{M}_P$ be all the matches to pattern P in a knowledge graph G. (I.e., $\mathcal{M}_P$ is $[[P]]_G$, the evaluation of P against G, using the terminology in [30].) For a variable $x \in X_P$, the range of x, denoted R_x^P, is a set of entities defined as

$$R_x^P = \{f(x) \mid M \in \mathcal{M}_P, f : V_P \to V_M\}.$$ △

For example, P_1 in Fig. 2.2a has two variable nodes, `?g` and `?s`. (To distinguish variables from entities, the names of variable nodes always start with the symbol `?`.) Figs. 2.2b-2.2d show P_1's matches in Fig. 2.1. $R_{?g}^{P_1} = \{\mathsf{G1}, \mathsf{G2}, \mathsf{G3}\}$ and $R_{?s}^{P_1} = \{\mathsf{S1}, \mathsf{S2}, \mathsf{S3}\}$.

Definition 4 (Context $C_v^{P,x}$)**.** Given an entity v, a pattern P, a variable $x \in X_P$ such that $v \in R_x^P$, the context of v defined by P and x is denoted $C_v^{P,x}$ and $C_v^{P,x} = R_x^P$. △

For example, the context of G1 in the running example—goals scored by BRA players—is defined by pattern P_1 in Fig. 2.2a and variable $?g$ in the pattern: $C^{P_1,?\mathsf{g}}_{\mathsf{G1}} = R^{P_1}_{?\mathsf{g}} = \{\mathsf{G1}, \mathsf{G2}, \mathsf{G3}\}$. On the other hand, since $\mathsf{G1} \notin R^{P_1}_{?\mathsf{s}}$, ?s in P_1 does not define a context of G1. Note that a pattern may define multiple contexts of v, since v may be mapped to different variables in the pattern. For example, consider pattern $P = \{(?\mathsf{g}, \textit{awarded-to}, ?\mathsf{a}), (?\mathsf{g}, \textit{scored-by}, ?\mathsf{s}), (?\mathsf{s}, \textit{play-for}, ?\mathsf{t})\}$. It defines two different contexts of BRA: $C^{P,?\mathsf{a}}_{\mathsf{BRA}} = \{\mathsf{CRO}, \mathsf{BRA}\}$, $C^{P,?\mathsf{t}}_{\mathsf{BRA}} = \{\mathsf{ESP}, \mathsf{BRA}\}$.

Entity Attributes and Subspaces

Given an entity of interest v, an attribute corresponds to the label of an edge incoming into or outgoing from v, and its value is the entity at the other end of the edge. Note that we need to distinguish between incoming attributes and outgoing attributes since an entity can be both sources and destinations of edges of the same label. For instance, a person can have a manager and meanwhile be the manager of someone else.

Definition 5 (Entity Attributes A_v)**.** Given an entity v, its attributes A_v is the union of its incoming and outgoing attributes: $A_v = A^{\mathsf{i}}_v \cup A^{\mathsf{o}}_v$. The incoming attributes are a set of edge labels $A^{\mathsf{i}}_v = \{(l, \leftarrow) \mid \exists(x,\ l,\ v) \in E_G\}$. Given an incoming attribute $a = (l, \leftarrow) \in A^{\mathsf{i}}_v$, v's value on attribute a is the set $v.a = \{x \mid (x,\ l,\ v) \in E_G\}$. Similarly, the outgoing attributes are $A^{\mathsf{o}}_v = \{(l, \rightarrow) \mid \exists(v,\ l,\ x) \in E_G\}$. Given an outgoing attribute $a = (l, \rightarrow) \in A^{\mathsf{o}}_v$, v's value is $v.a = \{x \mid (v,\ l,\ x) \in E_G\}$. △

Definition 6 (Subspace A)**.** A subspace A is a subset of v's attributes, i.e., $A \subseteq A_v$. The projection of v's attribute values onto subspace A is denoted $v.A$, and $v.\varnothing =$ `null`. △

For example, in Fig. 2.1, $A^{\mathsf{i}}_{\mathsf{CRO}} = \{(\textit{play-for}, \leftarrow), (\textit{awarded-to}, \leftarrow)\}$; $\mathsf{CRO}.(\textit{awarded-to}, \leftarrow) = \langle\{\mathsf{G1},\ \mathsf{G4}\}\rangle$ and $A^{\mathsf{o}}_{\mathsf{G1}} = \{(\textit{scored-by}, \rightarrow),\ (\textit{awarded-to}, \rightarrow)\}$; $\mathsf{G1}.(\textit{awarded-to}, \rightarrow) =$

$\langle\{\mathsf{CRO}\}\rangle$. Let subspace $A = \{(\textit{scored-by}, \leftarrow), (\textit{play-for}, \rightarrow)\}$. $A_{\mathsf{S1}} = A$ and $\mathsf{S1}.A = \langle\{\mathsf{G1}\}, \{\mathsf{BRA}\}\rangle$.

Exceptionality Score

Definition 7 (Exceptionality Scoring Function χ)**.** An exceptionality scoring function $\chi(v, A, C) \in \mathbb{R}$ measures entity v's degree of exceptionality with regard to subspace A in comparison with other entities in context C. Without loss of generality, we assume the range of χ is $[0, 1]$, with larger χ implying greater exceptionality. We also set $\chi(v, A, C)$=0 if $A \not\subseteq A_v$ or $v \notin C$, to make χ a total function. $\triangle$

The Maverick framework is indifferent to the choice of the exceptionality scoring function. It can accommodate many different interestingness/outlyingness functions (see surveys such as [32, 33]). Hence, the focus of this chapter is not on the design, evaluation and comparison of such exceptionality scoring functions. Rather, the goal is to develop a general framework for efficiently finding exceptional facts under various scoring functions. Nevertheless, to make the discussion concrete, we consider several representative functions, of which one is introduced below and two more are discussed in Section 2.4.2. To ensure consistency, the discussion uses our own notations and terminologies in presenting the adaptation of existing functions.

One-of-the-Few χ_f The one-of-the-few concept is adapted from [9]. The crux of the idea is that a factual claim about an entity is interesting when equally or more significant claims can be made about only few other entities. For example, in Fig. 2.1, it is interesting to claim "G1 is the only own goal among the goals scored by BRA players", since such a unique claim cannot be made about any other goal scored by a BRA player. On the contrary, "G1 is the only goal scored by S1" is not impressive, because the same kind of claim "Gx is the only goal scored by Sy" can be made for all 5 goals in Fig. 2.1.

The one-of-the-few measure [9] is based on multi-criteria dominance relationship which is irrelevant to this work. Our adaptation of [9] quantifies the rareness of attribute values based on frequency. Consider a context C, a subspace A, and any entity u in the context. We denote by p_S^A, or simply p_S when A is clear, the probability (or "frequency" as in [21]) of u taking values S in subspace A, i.e.,

$$p_S^A = p(u.A = S \mid u \in C) = \mid \{u \mid u \in C,\ u.A = S\} \mid / \mid C \mid . \tag{2.1}$$

Ranking facts directly by frequency is not robust, regarding which detailed analysis can be found in [9]. To intuitively understand the insight, consider an extreme example. Suppose in an organization everyone has a unique name. Given an particular individual x, a fact "x is the only person with that name" has high exceptionality measured by frequency itself. However, it is not truly exceptional since the same kind of fact can be stated for everyone.

Based on the definition of p_S^A, the one-of-the-few χ_f quantifies the exceptionality of an entity of interest v by the pessimistic rank of the frequency of $v.A$. Specifically, the exceptionality of v is:

$$\chi_f(v, A, C) = \mid \{u \mid u \in C,\ p_{u.A} > p_{v.A}\} \mid / \mid C \mid . \tag{2.2}$$

For example, consider entity of interest $v_0 = \mathsf{G1}$ in Fig. 2.1 and context C defined by pattern P_1 and variable ?g in Fig. 2.2a, i.e., $C = C_{\mathsf{G1}}^{P_1,?g} = \{\mathsf{G1}, \mathsf{G2}, \mathsf{G3}\}$. Table 2.1 shows the frequencies of attribute values in all subspaces. According to Table 2.1, $p_{\mathsf{G2}.A} = p_{\mathsf{G3}.A} = p_{\langle\{\mathsf{BRA}\}\rangle} = \frac{2}{3} > p_{\mathsf{G1}.A} = p_{\langle\{\mathsf{CRO}\}\rangle} = \frac{1}{3}$. Hence, $\chi_f(\mathsf{G1}, A, C) = \frac{\mid \{\mathsf{G2}.\mathsf{G3}\} \mid}{\mid C \mid} = \frac{2}{3}$. For $A = \{(\textit{awarded-to}, \rightarrow), (\textit{scored-by}, \rightarrow)\}$, $\chi_f(\mathsf{G1},\ A,\ C) = 0$, since there exists no $u \in C$ such that $p_{u.A} > p_{\mathsf{G1}.A}$.

Definition 8 (Top-k Exceptional Facts F_v)**.** With regard to an entity v, the *rank* of a context-subspace pair (C, A) is the number of context-subspace pairs with greater

Table 2.1: The frequencies of attribute values in all subspaces for entity of interest G1 with regard to context $C = \{\mathsf{G1}, \mathsf{G2}, \mathsf{G3}\}$.

A	$v.A : p^C_{v.A}$	G1.A
$\{(\textit{awarded-to}, \rightarrow)\}$	⟨{CRO}⟩:1/3, ⟨{BRA}⟩:2/3	⟨{CRO}⟩
$\{(\textit{scored-by}, \rightarrow)\}$	⟨{S1}⟩:1/3, ⟨{S2}⟩:1/3, ⟨{S3}⟩:1/3	⟨{S1}⟩
$\{(\textit{awarded-to}, \rightarrow), (\textit{scored-by}, \rightarrow)\}$	⟨{CRO}, {S1}⟩:1/3, ⟨{BRA}, {S2}⟩:1/3, ⟨{BRA}, {S3}⟩:1/3	⟨{CRO}, {S1}⟩

exceptionality scores, i.e., $rank(C, A) = |\ \{(C', A') \in \mathcal{C}_v \times A_v \mid \chi(v, A', C') > \chi(v, A, C)\}\ |$. $\mathcal{C}_v$ is the universe of v's contexts: $\mathcal{C}_v = \{C_v^{P,x} \mid P \in \mathcal{P}, x \in P, v \in R_x^P\}$, in which $\mathcal{P}$ is the universe of patterns over G, i.e., $\mathcal{P} = \{P(V_P, E_P) \mid V_P \subseteq X \cup V_G, E_P \subseteq (X \cup V_G) \times L \times (X \cup V_G), P(V_P, E_P)$ is weakly connected$\}$ where X is the universe of variables. (C, A) is a *top-k exceptional fact* if its rank is lower than k. Hence, the set of top-k exceptional facts about v, F_v, is defined as $F_v = \{(C, A) \in \mathcal{C}_v \times A_v \mid rank(C, A) < k\}$.[3] △

Problem Statement Given a knowledge graph G, an entity of interest v_0, an integer k, and an exceptionality scoring function χ, the problem of *exceptional fact discovery* is to find F_{v_0}—the top-k exceptional facts about v_0.

Continue the running example. With regard to G1, the context-subspace pair $(C_{\mathsf{G1}}^{P_1,?\mathsf{g}}, \{(\textit{awarded-to}, \rightarrow)\})$ may be exceptional. The context $C_{\mathsf{G1}}^{P_1,?\mathsf{g}}$ is $\{\mathsf{G1}, \mathsf{G2}, \mathsf{G3}\}$, i.e., the goals scored by BRA players. An interpretation of G1's exceptionality with regard to the pair is: among all the goals scored by BRA players, G1 is the only own goal.

Alternative Problem Modeling

There could be other ways of defining context and subspace. Definition 4 allows contexts based on arbitrary patterns. It is possible to adopt a more simplified and restricted definition that only allows such patterns to be in certain "shapes" such as paths, star graphs, and trees. Definition 6 dictates that a subspace must be a set of entity attributes. In other words, when comparing an entity with other entities in a

[3]The size F_v may be greater than k due to ties in exceptionality scores and thus ranks.

context, the entity stands out with respect to a subspace if it satisfies the conjunctive condition formed on the attributes in the subspace (i.e., a star query) while most other entities do not. It is plausible to adopt a more complex and expressive definition that allows the framework to assess exceptionality of entities using more complex, general graph queries instead of only star queries.

The current choices of Definitions 4 and 6 are formed based on several considerations related to usability and practicality. Particularly, the exceptionality scoring functions in this section and Section 2.4.2, adapted from functions in the literature that define outlyingness of tuples in relational tables, are defined on the aforementioned star queries. It is thus unclear how to define a scoring function using more complex graph queries. While such is an interesting question to ponder, it falls outside this chapter's scope. As mentioned earlier, the Maverick framework is indifferent to the choice of the exceptionality scoring function. The current simple definition of subspace also eases the task of ensuring the discovered facts can be intuitively expressed by the system and interpreted by users. On a related note, while conducting the experiments (Section 2.6) we limited the sizes of the context-defining patterns and subspaces to be very small, only involving at most a handful of nodes and edges.

There could also be other ways of defining attributes. For example, one can define an entity's attributes as a vector of values either independent to or derived from the graph. For instance, [18, 19] consider, for each node, an associated mini-table containing information from external sources. A prevalent model of entities in knowledge graphs is embedding-based [34, 35, 36, 37], in which each entity is represented by a vector capturing its neighborhood information. Such vectors can also be used as entities' attributes. However, the vectors are indecipherable to human beings. Furthermore, in general knowledge graphs, an entity may have sub-properties, functional properties, and transitive properties [38, 39]. This work does not consider

such models and thus is lack of reasoning capacity based on such properties. Some of such properties may be leveraged by pre-processing. For example, one may materialize the transitive properties. This can be an interesting future direction to explore.

2.3 Overview of Framework

We propose Maverick, an iterative framework for exceptional fact discovery. Intuitively, the process of discovering context-subspace pairs can be viewed as nested loops. The outer loop enumerates contexts, while the inner loop enumerates subspaces for each context. Given the entity of interest v_0, while subspace enumeration in the inner loop enumerates the subsets of A_{v_0}, the outer loop enumerates contexts by patterns, since each context of an entity is defined by a pattern and one of the pattern's variables (c.f. Definition 4). Conceptually, Maverick organizes all the possible contexts as a partial order on patterns, i.e., a Hasse diagram, in which each node is a pattern and each edge represents the subsumption (subgraph-supergraph) relationship between the two patterns. The essence of the outer loop is thus a traversal of the search space of patterns.

Given that the search space of patterns can be extremely large (Section 2.5), it is impractical to adopt breath-first, depth-first, or heuristic search approaches due to memory and time constraints [40]. To address this challenge, we propose to traverse the search space by *beam search* [41]. Since beam search maintains a "beam" of heuristically w best nodes and prunes all other nodes, it is not guaranteed to be complete or optimal. However, good solutions can be found quickly if the heuristic is sound enough.

Fig. 2.3 and Alg. 1 illustrate the framework of Maverick, which has three main components: Context Evaluator (CE), Exceptionality Evaluator (EE), and Pattern Generator (PG). The beam search at the outer loop starts with a pattern P_0 with a

Algorithm 1: Discovering exceptional context-subspace pairs.

FACT-DISCOVER *(G, v_0, χ, k, w)*

Input: G : the knowledge graph; $v_0 \in V_G$: the entity of interest; χ : the exceptionality scoring function; k : the size of output; w : the beam width

Output: H : k most exceptional context-subspace pairs

$P_0 \leftarrow (V_{P_0} = \{x_0\}, E_{P_0} = \varnothing)$; `// Initial state.` x_0 `is a variable.`

$B \leftarrow \{P_0\}$; `// Beam.`

$i \leftarrow 1$; `// Iteration number.`

while $B \neq \varnothing$ *and* $i \leq$ MAX_ITERATION **do**

  $i \leftarrow i + 1$; $B_{tmp} \leftarrow \varnothing$;

  foreach $P \in B$ **do**

    `// Obtain contexts of` v_0 `and matches to` P`.`
    `(Section 2.3.1)`

    $\mathcal{C}^P_{v_0}, \mathcal{M}_P \leftarrow$ `CONTEXT-EVALUATOR`(P, v_0, G);

    foreach $C \in \mathcal{C}^P_{v_0}$ **do**

      `// Exceptionality Evaluation. (Section 2.4)`

      $\mathcal{A} \leftarrow$ `EXCEPTIONALITY-EVALUATOR`(v_0, C, k, χ);

      foreach $A \in \mathcal{A}$ **do** $H \leftarrow H \cup \{(C, A)\}$;

    `// Find` $\mathcal{Y}$ `--- the children of` P`. (Section 2.5)`

    $\mathcal{Y} \leftarrow$ `PATTERN-GENERATOR`$(v_0, P, \mathcal{M}_P, w, G)$;

    $B_{tmp} \leftarrow B_{tmp} \cup \mathcal{Y}$;

  $B \leftarrow$ top-w of B_{tmp} based on heuristics h ; `// Section 2.5.4`

return top-k pairs in H based on exceptionality scores;

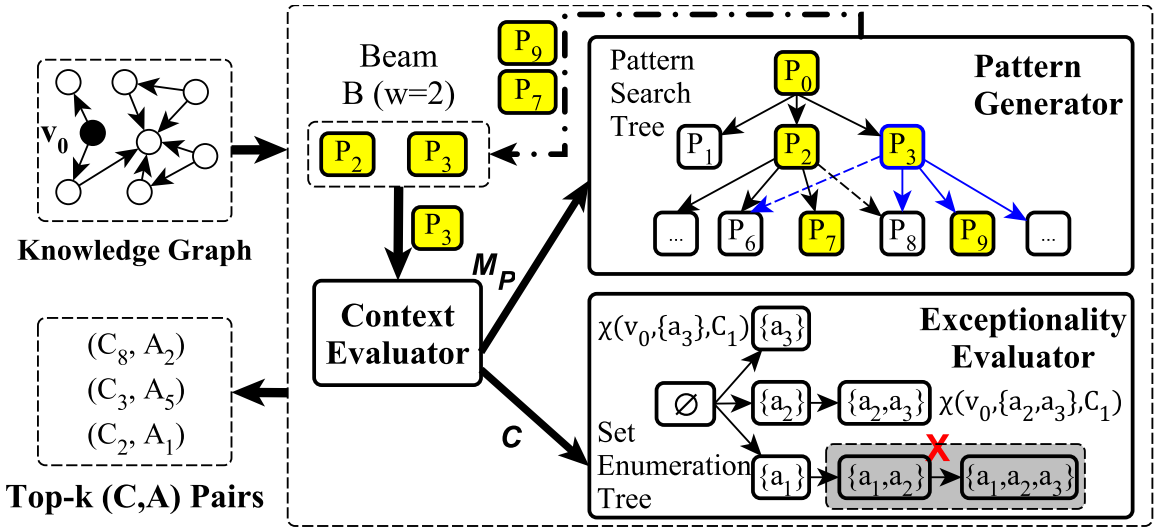


Figure 2.3: The framework of Maverick.

single variable node x_0 (Lines 2–3 in Alg. 1). The search results in a *pattern search tree*, of which the root is P_0. At each iteration, Maverick maintains a beam B of a fixed size w (Lines 6, 13, 14). The beam consists of heuristically the best w patterns (e.g., P_2, P_3 in Fig. 2.3 where $w = 2$) at the visited level of the pattern search tree. For each pattern P in B, component CE obtains the matches $\mathcal{M}_P$ to the pattern and the corresponding contexts $\mathcal{C}^P_{v_0}$ of v_0 (Line 8). For each context C in $\mathcal{C}^P_{v_0}$ (e.g. C_1 in Fig. 2.3), component EE finds the top-k scored subspaces according to a given exceptionality scoring function χ (Line 10, and Section 2.4). Component PG finds the children of the visited pattern based on its matches (Line 12, and Section 2.5). Since there are usually much more children than what the beam size w allows, PG applies a set of heuristics (Section 2.5.4) to prune the child patterns. Each child pattern is given a score that measures how promising it is according to the heuristics. The best w patterns among all the children of patterns in B will become the new beam B (Line 14), which is the input to the next iteration, e.g., $\{P_7, P_9\}$ in Fig. 2.3. The process ends when the limit on the number of iterations has reached. The limit is set to avoid overly-complex patterns which correspond to facts that are only convolutedly

interesting. It also practically bounds the resource spent for running the algorithm. When the algorithm terminates, Maverick returns the k context-subspace pairs with the highest exceptionality scores (Line 15). Below, we discuss component CE in Section 2.3.1, EE in Section 2.4, and PG in Section 2.5.

2.3.1 Context Evaluator

The context evaluator (CE, Line 8 in Alg. 1) is responsible for obtaining the matches to a given pattern as well as the corresponding contexts. Its working is depicted in Alg. 2. We expect a graph query system to take a pattern as the input and return all the matches to the pattern (Line 3). The Maverick framework is agnostic to the choice of the specific query processing system. According to Definition 4, for each variable in the pattern ($x \in X_P$), CE returns its range R_x^P as a context if the entity of interest v_0 is in the range (Line 5).

Algorithm 2: Context evaluator.

```
1 CONTEXT-EVALUATOR (P, v_0, G)
2     C^P_{v_0} ← ∅ ;                          // The set of contexts defined by P.
3     M_P ← match(G, P) ;                                  // Matches to P.
4     foreach x ∈ X_P do
          // Refer to Definitions 3 and 4 for R^P_x and C^{P,x}_{v_0}.
5         if v_0 ∈ R^P_x then C^P_{v_0} ← C^P_{v_0} ∪{C^{P,x}_{v_0}} ;
6     return (C^P_{v_0}, M_P);
```

For example, consider graph G in Fig. 2.1, the entity of interest $v_0 = \mathsf{G1}$, and the pattern P_1 in Fig. 2.2a. $\mathcal{M}_{P_1} = \{M_1, M_2, M_3\}$, where M_1, M_2, and M_3 are in Figs. 2.2b–2.2d. P_1 has two variables, $?g$ and $?s$. Since $\mathsf{G1} \in R^{P_1}_{?g} = \{\mathsf{G1}, \mathsf{G2}, \mathsf{G3}\}$

and $\mathsf{G1} \notin R^{P_1}_{?s}$, P_1 defines one and only one context of $\mathsf{G1}$, which is $C^{P_1,?g}_{\mathsf{G1}} = R^{P_1}_{?g}$. Therefore, $\mathcal{C}^P_{\mathsf{G1}} = \{C^{P_1,?g}_{\mathsf{G1}}\}$.

Context defined by initial pattern P_0 Maverick starts with a trivial pattern P_0 in which the only node x_0 is a variable. According to Definition 2, every node in the knowledge graph G is a match to P_0. Therefore, the context defined by P_0 includes the nodes of all different types in G. Although Maverick allows such contexts consisting of heterogeneous entities, it brings two practical challenges. First, since different types of entities have different attributes, the exceptionality of an entity could be unrealistically bloated due to the sparsity of attributes that are common to many entities in the context. Second, the computation of exceptionality scores, of which the complexity is at least linear to the size of the context, is highly expensive when the context includes all the nodes in the data graph.

Due to these two reasons related to semantics and efficiency, it is more desirable that a context is homogeneous, i.e., it only includes entities of the same type. That imposes a requirement for a type system on the knowledge graph, which can be either predefined (e.g., DBpedia Ontology, Freebase Schema) or derived from the data graph. Maverick accommodates both kinds of type systems in knowledge graphs, which ensures its general applicability. Particularly, when an explicit type system does not exist, there can be different ways of deriving entity types. Although Maverick is oblivious to the specific approach for deriving the type system, the particular approach implemented in our system is as follows. In this approach, two entities u and v both

belong to an implicit type if they have at least one common attribute, i.e., $A_u \cap A_v \neq \varnothing$ (see Definition 5 for A_v).[4] We then define the context $C_{v_0}^{P_0,x_0}$ for P_0 as:

$$C_{v_0}^{P_0,x_0} = \{u \in V_G | A_u \cap A_{v_0} \neq \varnothing\}.$$

For example, $C_{\mathsf{G1}}^{P_0,x_0} = \{\mathsf{G1}, \mathsf{G2}, \mathsf{G3}, \mathsf{G4}, \mathsf{G5}\}$, which excludes team nodes (such as CRO), player nodes (such as S1), and so on. Note that this way of deriving entity types is compatible with Freebase's type system, in which the types of the source/destination nodes of an edge are determined by the label (i.e., type) of the edge.

2.4 Exceptionality Evaluator

The Exceptionality Evaluator (EE) operates in the inner loop of the Maverick framework (function `EXCEPTIONALITY-EVALUATOR` $(v_0, C,\ k, \chi)$ at Line 10 of Alg. 1). For each context C of the entity of interest v, it finds the k subspaces A with the highest $\chi(v, A, C)$ scores. Note that it is sufficient to find these k subspaces, since the eventual output of Maverick is the top-k context-subspace pairs across all contexts of v. A naive solution of EE can exhaustively enumerate all possible subspaces of A_v and calculate the exceptionality score of v in each subspace. The apparent $O(2^{|A_v|})$ complexity of this approach renders it prohibitively expensive since many entities may have a lot of attributes. For instance, Denzel Washington has more than 40 attributes in the August 9, 2015 Freebase graph. It is thus crucial for Maverick to have an efficient subspace enumeration method in order to discover more exceptional context-subspace pairs. Section 2.4.1 discusses how Maverick uses a set enumeration tree to avoid exhaustively enumerating subspaces. Specifically, Maverick exploits the upper bound properties of exceptionality scoring functions to guide the traversal of the set

[4]By this definition, a type is similar to a cluster of entities, and the clustering is non-exclusive, i.e., an entity can belong to multiple types.

enumeration tree. Section 2.4.2 introduces three representative exceptionality scoring functions along with their upper bound functions.

2.4.1 Finding Top-k Subspaces

EE applies a *set enumeration tree* (SE-tree) [27] to avoid exhaustively enumerating subspaces. Each node in the tree is a subspace—a subset of v's attributes A_v. The children of a node correspond to various supersets of the node's associated attributes. Formally, let r be an (arbitrary) total order on A_v. The root of an SE-tree for A_v is the empty set. The children of a node $A \subset A_v$ in the tree form the set $\{A \cup \{a\} \mid a \in A_v \setminus A, \forall a' \in A, a' <_r a\}$. An SE-tree for $A_v = \{a_1, a_2, a_3\}$ is shown in Fig. 2.3. The gist is to explore the set enumeration tree using heuristic search methods such as best-first search and to prune branches that are guaranteed to not contain highly-scored subspaces.

What is particularly challenging is that an exceptionality scoring function χ usually does not have the *downward closure property* with respect to subspace inclusion, i.e., $\chi(v, A, C)$ can be greater than, less than, or equal to $\chi(v, A', C)$ for any $A' \supseteq A$. As a matter of fact, none of the three representative functions that will be introduced in Section 2.4.2 satisfies the property (proof omitted). The lack of downward closure property makes it infeasible to prune the set enumeration tree based on exact exceptionality scores.

EE uses upper bounds on the exceptionality scoring function χ to allow for pruning of the set enumeration tree. Alg. 3 presents its pseudo code. The set enumeration tree nodes (i.e., subspaces) are visited in the descending order of their upper bounds (Line 6). If the upper bound score of a node is not greater than the score of the current k-th ranked subspace, the node and all its children are pruned (Line 12). Otherwise, the exact exceptionality score of the node is calculated (Line 13). The

Algorithm 3: Exceptionality evaluator.

```
1  EXCEPTIONALITY-EVALUATOR (v, C, k, χ)
       // CS: current subspace; UA: attributes to visit; Tk: top-k
          subspaces.
2      CS ← ∅; UA ← A_v; Tk ← ∅;
3      return EXPLORE-SUBSPACE(v, C, k, χ, CS, UA, Tk);
4  EXPLORE-SUBSPACE (v, C, k, χ, CS, UA, Tk)
5      while UA ≠ ∅ do
           // Calculate upper bounds.
6          a_max ← arg max_{a∈UA} upper(v, CS ∪ {a}, C);
7          A_max ← CS ∪ {a_max}; upper_max ← upper(v, A_max, C);
8          UA ← UA \ {a_max};
9          if |Tk| < k then
               // −1 indicates the top-k list Tk is not full.
10             score_min ← −1; A_min ← ∅;
11         else (A_min, score_min) ← arg min_{(A,score)∈Tk} score ;
12         if upper_max > score_min then
13             score ← χ(v, A_max, C);
14             if score > score_min then
15                 if score_min ≥ 0 then
16                     Tk ← Tk \ {(A_min, score_min)};
17                 Tk ← Tk ∪ {(A_max, score)};
               // Explore children subspaces.
18             Tk ← EXPLORE-SUBSPACE(v, C, k, χ, A_max, UA, Tk);
19     return Tk;
```

subspace is used to purge the current k-th subspace if its exact score is still greater (Lines 14–17). Regardless of whether the node makes into the top-k list, its children are enumerated recursively (Line 18).

The general upper bound function *upper* in Alg. 3 is defined as follows. By the definition, it is sound to prune a node and all its children if the condition in Line 12 is not satisfied.

Definition 9 (Upper bound of an exceptionality scoring function *upper*)**.** Given an exceptionality scoring function χ, an upper bound of χ is a function that, for any entity v, context C, and subspace $A \subseteq A_v$, bounds the exceptionality score of v with respect to C and any superset of A, i.e.,

$$upper(v, A, C) \geq \max_{A \subseteq A' \subseteq A_v} \chi(v, A', C). \qquad \triangle$$

The general upper bound function *upper* must be instantiated for specific exceptionality scoring functions χ. The Maverick framework expects an application developer to supply *upper* while specifying χ. Various outlying aspect mining methods [21, 22, 24] also devise upper bound functions for pruning set enumeration tree. They operate on the single-table data model and are thus inapplicable for graphs, as explained in Section 2.1. EE must use different scoring functions and upper bound functions designed for knowledge graphs. The ensuing discussion in this section entails that.

2.4.2 Exceptionality Scoring Functions

As mentioned in Section 2.2, the general Maverick framework accommodates different exceptionality scoring functions beyond the one-of-the-few function χ_f. We discuss two more representative functions in this section.

Outlyingness χ_o This measure, adopted from [21], is based on the distribution of attribute values. An entity receives a high score if it has rare attribute values while a lot of other entities share common attribute values. It quantifies the rareness of attribute values by $p_S^A = p(u.A = S \mid u \in C)$ (the same as for χ_f). Let $\mathcal{S}_A$ be all possible attribute values on subspace A and in context C, i.e., $\mathcal{S}_A = \{u.A \mid u \in C\}$. The outlyingness score of an entity v is given by:

$$\chi_o(v, A, C) = \sum\nolimits_{S \in \mathcal{S}_A} p_S \times (p_S - p_{v.A}) \times \mathbb{1}(p_S > p_{v.A}) \tag{2.3}$$

where $\mathbb{1}(\cdot)$ is the indicator function that returns 1 for a true condition and 0 otherwise. Essentially, the outlyingness score is the area *above* the accumulated frequency histogram of the context C with respect to the subspace A, starting from the frequency of $v.A$. The score is designed to quantify the "degree of unbalance" between the frequencies of entities in the context [21].

For instance, consider the same example used in explaining χ_f: $v_0 = \mathsf{G1}$, $C = C_{\mathsf{G1}}^{P_1,?g} = \{\mathsf{G1}, \mathsf{G2}, \mathsf{G3}\}$, and $A = \{(\textit{awarded-to}, \rightarrow)\}$. According to Table 2.1, $\chi_o(\mathsf{G1},\ \{(\textit{awarded-to}, \rightarrow)\}, C) = p_{\langle\{\mathsf{CRO}\}\rangle} \times (p_{\langle\{\mathsf{CRO}\}\rangle} - p_{\langle\{\mathsf{CRO}\}\rangle}) \times 0 + p_{\langle\{\mathsf{BRA}\}\rangle} \times (p_{\langle\{\mathsf{BRA}\}\rangle} - p_{\langle\{\mathsf{CRO}\}\rangle}) \times 1 = \frac{2}{3}(\frac{2}{3} - \frac{1}{3}) = \frac{2}{9}$. Another example is, for $A = \{(\textit{awarded-to}, \rightarrow), (\textit{scored-by}, \rightarrow)\}$, $\chi_o(\mathsf{G1}, A, C) = 0$ since there exists no $u \in C$ such that $p_{u.A} > p_{\mathsf{G1}.A}$.

Isolation Score χ_i The isolation score χ_i is inspired by iForest [42] and iPath [25]. Both iForest and iPath are applicable on real value attributes. By randomly choosing a pivot in the range of an attribute, both methods iteratively split a set of entities into two disjoint subsets, until the entities in each set have an identical value. The iForest score and iPath score are defined using the number of splits applied. iPath only iteratively splits the subsets containing the entity of interest. The entity's score is the number of splits until the entity has the same value as other entities in its subsuming set. iForest splits all subsets. Essentially, both iForest and iPath follow the *minimum*

description length principle, and both scores are functions of the estimated description length of the entity's attribute value, which is the number of splits. Inspired by iForest score, we define isolation score χ_i as follows:

$$\chi_i(v, A, C) = 1 - 2^{-\frac{-\log_2 p_{v.A}}{-\sum_{S \in \mathcal{S}_A}(p_S \times \log_2 p_S)}} \tag{2.4}$$

where the numerator in the exponent is the description length of v's attribute value, while the denominator is the average description length of attribute values in subspace A. Intuitively, if $v.A$ is peculiar, then the description length of $v.A$ is longer than average and $\chi_i(v, A, C)$ is closer to 1.

For example, let the conditions be the same as the ones used in explaining χ_o and χ_f: $v_0 = \mathsf{G1}$, $C = C_{\mathsf{G1}}^{P_1,?g} = \{\mathsf{G1}, \mathsf{G2}, \mathsf{G3}\}$, and $A = \{(\textit{awarded-to}, \rightarrow)\}$. According to Table 2.1, $-\sum_{S \in \{\langle\{\mathsf{BRA}\}\rangle, \langle\{\mathsf{CRO}\}\rangle\}}(p_S \times \log_2 p_S) = -(\frac{1}{3} \times \log_2 \frac{1}{3} + \frac{2}{3} \times \log_2 \frac{2}{3}) = \log_2 3 - \frac{2}{3} = 0.91$, $-\log_2 p_{\mathsf{G1}.A} = 1.58$, then $\chi_f(\mathsf{G1},\ A,\ C) = 0.7$. Similarly, $\chi_f(\mathsf{G1},\ \{(\textit{awarded-to}, \rightarrow),\ (\textit{scored-by}, \rightarrow)\},\ C) = 0$.

2.4.3 Upper Bound Functions

In this section we devise upper bound functions for the three representative exceptionality functions introduced in Section 2.2 (χ_f) and Section 2.4.2 (χ_o and χ_i). We prove that these designs satisfy Definition 9 and thus ensure the soundness of Alg. 3, with regard to any given entity v, context C, and subspaces $A \subseteq A' \subseteq A_v$. Recall that we denote by p_S^A, or simply p_S, the frequency of entity's attribute value S in subspace A (Eq. (2.1)).

Theorem 1 (Upper bound of χ_f)**.** $upper_f(v, A, C) \geq \chi_f(v,\ A', C)$, given the following definition in which $\overline{C_v}$=$C \setminus \{v\}$:

$$upper_f(v, A, C) = \mid \{u \mid u \in \overline{C_v},\ p_{u.A} > 1/\mid C \mid\} \mid \ / \mid C \mid \tag{2.5}$$

The theorem holds because $\frac{1}{\mid C \mid} \leq p_{u.A'} \leq p_{u.A}$ for any $A' \supseteq A$. We omit the detailed proof here.

Theorem 2 (Upper bound of χ_o)**.** $upper_o(v, A, C) \geq \chi_o(v, A', C)$, given the following definition where $\mathcal{S}_A = \{u.A \mid u \in C\}$:

$$upper_o(v, A, C) = \sum\nolimits_{S \in \mathcal{S}_A} (p_S)^2 - \frac{(2\ p_{v.A} + 1) \times \mid C \mid - 2}{\mid C \mid^2}. \tag{2.6}$$

Proof. Let $\{p_{v.A}, p_{S_1}, \cdots, p_{S_N}\}$ be the probability distribution of attribute values in subspace A. According to [21], for any $A' \supseteq A$, $\chi_o(v, A', C)$ is maximized when the additional attributes in $A' \setminus A$ preserve the current attribute value distribution, except that the additional attributes make v different from all other entities, i.e., the optimal distribution of attribute values in subspace A' is $\{p_{v.A'},\ p_{v.A} - p_{v.A'}, p_{S_1}, \cdots, p_{S_N}\}$, where $p_{v.A'} = \frac{1}{\mid C \mid}$. (Note that $p_S \geq \frac{1}{\mid C \mid}$ for any S.) In other words, the entities having value $v.A$ on subspace A are partitioned into v itself (having value $v.A'$ on subspace A') and the rest (having identical value on A'). Based on Eq. (2.3), after a few polynomial manipulations, which we omit here, we have $\chi_o(v, A', C) \leq \sum_{S \in \mathcal{S}_A} {p_S}^2 - \frac{(2\ p_{v.A}+1) \times \mid C \mid - 2}{\mid C \mid^2}$. ∎

Theorem 3 (Upper bound of χ_i)**.** $upper_i(v, A, C) \geq \chi_i(v, A', C)$, given the following definition:

$$upper_i(v, A, C) = 1 - 2^{\frac{-\log_2 \frac{1}{\mid C \mid}}{-q_{v.A} - \sum_{S \in \mathcal{S}_A \setminus \{v.A\}} (p_S \times \log_2 p_S)}} \tag{2.7}$$

where $q_{v.A} = \frac{1}{\mid C \mid} \times \log_2 \frac{1}{\mid C \mid} + (p_{v.A} - \frac{1}{\mid C \mid}) \times \log_2 (p_{v.A} - \frac{1}{\mid C \mid})$.

Proof. By Eq. (2.4), $\chi_i(v, A', C)$ is maximized when the denominator in the exponent is minimized and the numerator is maximized. Let $\{p_{v.A}, p_{S_1}, \cdots, p_{S_N}\}$ be the probability distribution of attribute values in subspace A. Similar to the proof of Theorem 2, we prove that $\chi_i(v, A', C)$ is maximized when the distribution in subspace A' is $\{p_{v.A'}, p_{v.A} - p_{v.A'}, p_{S_1}, \cdots, p_{S_N}\}$, where $p_{v.A'} = \frac{1}{\mid C \mid}$. Partition the entities having

value S in A into two disjoint subsets that have values $S1$ and $S2$ in A', respectively, i.e., $P_S^A = P_{S'}^{A'} + P_{S''}^{A'}$. Without loss of generality, assume $P_{S'}^{A'} \leq P_{S''}^{A'}$. We have

1. $-p_{S'}^{A'} \log_2 p_{S'}^{A'} - p_{S''}^{A'} \log_2 p_{S''}^{A'} \geq -p_S^A \log_2 p_S^A$,
2. $-p_{S'}^{A'} \log_2 p_{S'}^{A'} - p_{S''}^{A'} \log_2 p_{S''}^{A'} \geq$
   $-p_{v.A'}^{A'} \log_2 p_{v.A'}^{A'} - (p_S^A - p_{v.A'}^{A'}) \log_2 (p_S^A - p_{v.A'}^{A'})$.

They can be derived by

1. $-p_{S'}^{A'} \log_2 p_{S'}^{A'} - p_{S''}^{A'} \log_2 p_{S''}^{A'} \geq -p_{S'}^{A'} \log_2 p_{S''}^{A'} - p_{S''}^{A'} \log_2 p_{S''}^{A'}$
   $= -(p_{S'}^{A'} + p_{S''}^{A'}) \log_2 p_{S''}^{A'} = -p_S^A \log_2 p_{S''}^{A'} \geq -p_S^A \log_2 p_S^A$,
2. Let $p_S^A = n\ p_{v.A'}^{A'} = \frac{n}{\mid C \mid}$, $p_{S'}^{A'} = m\ p_{v.A'}^{A'} = \frac{m}{\mid C \mid}$, then $1 \leq m < n$, $-p_{S'}^{A'} \log_2 p_{S'}^{A'} -$
   $p_{S''}^{A'} \log_2 p_{S''}^{A'} - (-p_{v.A'}^{A'} \log_2 p_{v.A'}^{A'} - (p_S^A - p_{v.A'}^{A'}) \log_2 (p_S^A - p_{v.A'}^{A'})) = -m\ p_{v.A'}^{A'} \log_2 m\ p_{v.A'}^{A'} -$
   $(n\ p_{v.A'}^{A'} - m\ p_{v.A'}^{A'}) \log_2 (n\ p_{v.A'}^{A'} - m\ p_{v.A'}^{A'}) + p_{v.A'}^{A'} \log_2 p_{v.A'}^{A'} + (n\ p_{v.A'}^{A'} - p_{v.A'}^{A'}) \log_2 (n\ p_{v.A'}^{A'} -$
   $p_{v.A'}^{A'}) = p_{v.A'}^{A'}((n-1) \log_2 (n-1) - m log_2 m - (n-m) log_2 (n-m)) = p_{v.A'}^{A'} \log_2 \frac{(n-1)^{(n-1)}}{m^m (n-m)^{(n-m)}}$
   $\geq p_{v.A'}^{A'} \log_2 \min(\frac{(n-1)^{(n-1)}}{1^1 (n-1)^{(n-1)}}, \frac{(n-1)^{(n-1)}}{(n-1)^{(n-1)} (n-(n-1))^{(n-(n-1))}}) = 0$.

As a result, since $-\log_2 p_{v.A'} \geq \log_2 p_i$ for any $p_i \geq p_{v.A'}$, by dividing $p_{v.A}$ to $p_{v.A'}$ and $p_{v.A} - p_{v.A'}$, $\chi_i(v, A', C)$ is maximized. In other words, the optimal distribution in subspace A' is $\{p_{v.A'}, p_{v.A} - p_{v.A'}, p_{S_1}, \cdots, p_{S_N}\}$. ■

Subspace pruning across contexts Since Maverick only needs to return top-k context-subspace pairs, the Exceptionality Evaluator (Alg. 3) employs a simple optimization that uses the current top-k context-subspace pairs H to prune the subspaces in a new context. Specifically, a subspace in a context is pruned if its upper bound exceptionality score is lower than the score of the k-th best context-subspace pair in H. This optimization is simply implemented by modifying the function `EXCEPTIONALITY-EVALUATOR` in Alg. 3 to accept an additional argument H and changing Line 2 from `Tk` $\leftarrow \varnothing$ to `Tk` $\leftarrow H$. The experiment results reported in Section 2.6 already reflect this optimization.

Our final note is that an upper bound function may have limited pruning power when it gives loose bounds on exceptionality scores, resulting in exponential complexity in subspace enumeration. Our empirical results in Section 2.6.2, though, verified that the several upper bound functions proposed in this chapter (Eqs. (2.6)–(2.7)) substantially reduced the overhead of subspace enumeration.

2.5 Pattern Generator

The Pattern Generator (PG) is used in Line 12 of Alg. 1 in the Maverick framework. Its pseudo code is in Alg. 4. At each iteration of the beam search on patterns, it finds the children of each visited pattern P (Line 3, see Alg. 5) in the current beam. A child pattern, if not pruned (see Section 2.5.3), is given a score that measures how promising it is according to a few heuristics (Line 5, see Section 2.5.4). Among all the children of the patterns in the current beam, the w children with the highest scores are returned to form the new beam (Line 14 in Alg. 1), where w is the predefined beam width. The new beam becomes the input to the next iteration. This section first describes the search space of patterns (Section 2.5.1) and then discusses how to efficiently explore the space by applying pruning rules (Section 2.5.3) and selection heuristics (Section 2.5.4).

2.5.1 Search Space of Patterns

The search space of patterns is a Hasse diagram of valid patterns, where a pattern is *valid* if it contains at least one variable node and it has a match (Definition 2) in the knowledge graph G. We exclude invalid patterns since they cannot lead to relevant facts. For example, pattern $\{(?\mathsf{g}, \textit{scored-by}, ?\mathsf{s1}), (?\mathsf{g}, \textit{scored-by}, ?\mathsf{s2})\}$ does not have a match and is thus invalid because no goal is scored by more than one player. Formally, the search space of patterns is a Hasse diagram $\mathbb{P}(V_{\mathbb{P}}, E_{\mathbb{P}})$, where $V_{\mathbb{P}}$ is the

Algorithm 4: Pattern generator.

```
1 PATTERN-GENERATOR (v_0, P, M_P, w, G)
2     Y ← ∅ ;                                   // Promising children of P.
      // Find P's children, see Alg. 5.
3     children ← FIND-CHILDREN(v_0, P, M_P, G);
4     foreach child ∈ children do
5         score ← h(v_0, child);                // See Section 2.5.4 for h.
6         Y ← Y ∪ {(child, score)};
7     return top-w of Y based on score;
```

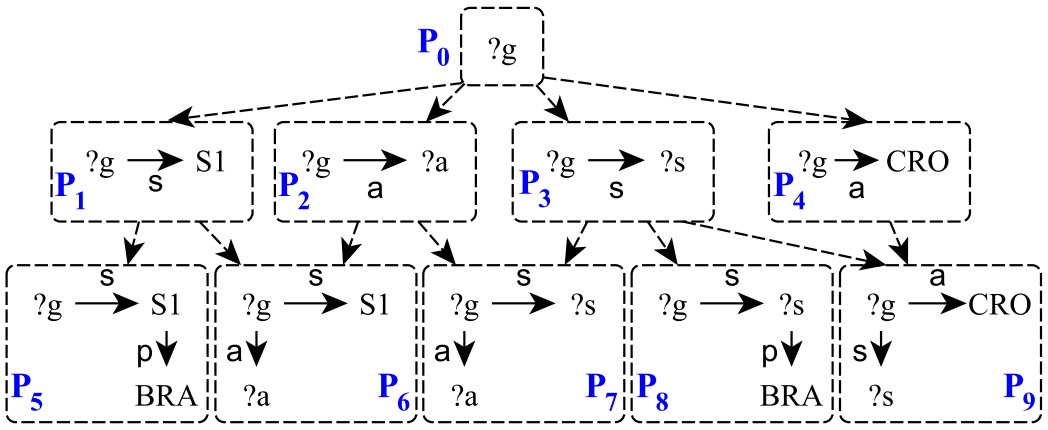


Figure 2.4: An excerpt of the search space of patterns over Fig. 2.1. Edge labels: *a*: *awarded-to*, *p*: *play-for*, *s*: *scored-by*.

set of valid patterns and $E_{\mathbb{P}} \subseteq V_{\mathbb{P}} \times V_{\mathbb{P}}$ is the set of edges. There exists an edge from *parent* pattern P_i to *child* pattern P_j if P_i is an immediate subgraph of P_j, i.e., P_i has exactly one edge less than P_j. A pattern can have multiple children and multiple parents. Fig. 2.4 shows an excerpt of the search space of patterns over the data graph in Fig. 2.1. In the figure, P_6 and P_7 are the children of P_2, and both P_2 and P_3 are the parents of P_7.

One may realize already that $\mathbb{P}$ can be extremely large. We prove in Theorem 4 that the *order* of $\mathbb{P}$ (i.e., the cardinality of $V_{\mathbb{P}}$) is exponential to the orders of G's

weakly connected components (WCCs).[5] Given that knowledge graphs are all well connected, it is impossible to exhaustively enumerate the patterns. For example, according to Theorem 4, the data graph in Fig. 2.1 has at least $2^{13+1} - 2 - 13 + 13 = 16,382$ patterns. (The graph itself is the only WCC, with 13 nodes.) Note that Theorem 4 only provides a loose bound. In practice, the number is even much larger, exacerbating the challenge. Section 2.6.2 shows that the tiny graph has more than $69,000$ patterns with merely no more than 5 edges.

Theorem 4. Let $\mathcal{W}$ be the set of WCCs in a knowledge graph G, a lower bound on $\mathbb{P}$'s order is:

$$| V_{\mathbb{P}} | \geq \sum\nolimits_{W \in \mathcal{W}} (2^{| V_W |+1} - 2) - | V_G | + \max_{W \in \mathcal{W}} | V_W |.$$

Proof. Given a WCC $W \in \mathcal{W}$, it has at least one subgraph of order i, for every $i \in [\ 1, |V_W|\ ]$. For each subgraph of size i, there are 2^i corresponding patterns that can be constructed by replacing some nodes with variables. Hence, for each W, there are at least $\sum_{i=1}^{|V_W|} 2^i = 2^{|V_W|+1} - 2$ patterns. Since every such pattern is isomorphic to a subgraph of W, it is guaranteed to be valid. Note that two patterns of the same order constructed from two subgraphs in two different WCCs can be equivalent if all their nodes are variables. Therefore, each $W \in \mathcal{W}$ has at most $| V_W |$ patterns that are equivalent to others. There are at least $\max_W |V_W|$ unique patterns in which all nodes are variables. Thus, after excluding double-counted patterns,

$$\begin{aligned} | V_{\mathbb{P}} | &\geq \sum\nolimits_W (2^{|V_W|+1} - 2) - \sum\nolimits_W |V_W| + \max_W |V_W| \\ &= \sum\nolimits_W (2^{|V_W|+1} - 2) - |V_G| + \max_W |V_W|. \end{aligned}$$

■

[5] A weakly connected component is a maximal subgraph of a directed graph, in which every pair of vertices are connected, ignoring edge direction.

2.5.2 Match-based Construction of Patterns

Given the current beam of patterns, Maverick finds top context-subspace pairs using its context evaluator (Section 2.3.1) and exceptionality evaluator (Section 2.4). Among the child patterns of the evaluated patterns, the promising ones are chosen to form the new beam for the next iteration. While Section 2.5.4 discusses how to select the promising patterns, this section proposes an efficient way of generating the child patterns. Note that the aforementioned Hasse diagram of patterns is not pre-materialized. Rather, the patterns need to be constructed before we can evaluate them.

To construct the child patterns of an evaluated pattern P, a simple approach is to enumerate all possible ways of expanding P by adding one more edge. A major drawback of this approach is it may construct many invalid patterns that do not have any match. Some invalid patterns can be easily recognized by referring to the schema graph of the data. However, chances are most of the schema-abiding patterns are still invalid because they do not have matching instances in the data graph, given the sheer diversity of a knowledge graph. The system will evaluate such patterns in vain to get empty results in order to realize they are invalid.

To avoid evaluating invalid patterns, we propose a *match-based pattern construction method*. Instead of constructing the child patterns by directly expanding P, this method expands the matches of P and constructs the child patterns from the expanded matches. It guarantees to construct only valid patterns and evade the evaluation of invalid patterns. The method is based on the following theorem.

Theorem 5. Suppose P' is a child of $P \in \mathbb{P}$, i.e., $(P, P') \in E_{\mathbb{P}}$ and thus P' is a valid pattern with matches. Given any match M' to P', there exists a match M to P that is a subgraph of M', i.e., $\forall M' \in \mathcal{M}_{P'}$, $\exists M \in \mathcal{M}_P$ s.t. $V_M \subseteq V_{M'}$ and $E_M \subseteq E_{M'}$.

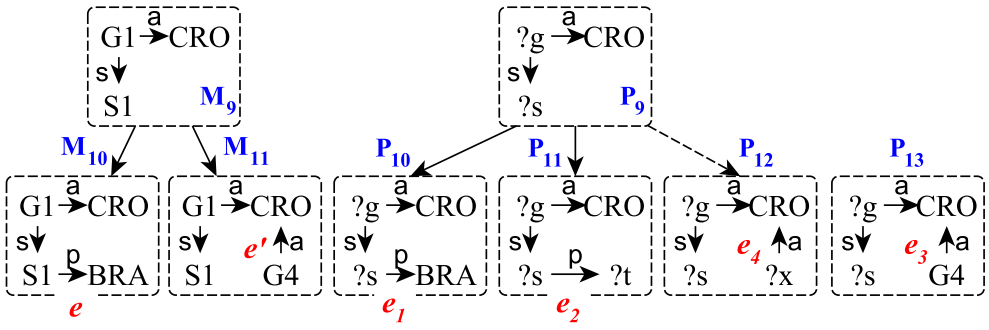


Figure 2.5: Illustration of how the child patterns of a pattern are constructed. P_{10} and P_{11} are obtained based on M_{10} and edge e. P_{12} and P_{13} can be obtained based on M_{11} and edge e', but they are pruned based on rules in Section 2.5.3.

Proof. Since P' is a child of P, P' has one edge more than P. Suppose $(u, l, w) = E_{P'} \setminus E_P$, and f' is the bijection $f' : V_{P'} \to V_{M'}$. We prove the theorem by constructing M. More specifically, let $E_M = E_{M'} \setminus \{(f'(u), l, f'(w))\}$, and $V_M = \cup_{(v_i,l',v_j)\in E_M}\{v_i, v_j\}$. We can construct a bijection $f : V_P \to V_M$ such that $f(u) = f'(u)$ for any $u \in V_P$. Since f' satisfies the edge isomorphism, f also satisfies it, i.e., $\forall(v_i, l', v_j) \in E_P$, $(f(v_i), l', f(v_j)) \in E_M$, and vice versa. By Definition 2, M is a match to P. ■

Based on Theorem 5, the method that constructs the child patterns of P is illustrated in Alg. 5. For a match M of P, it finds each of its weakly connected supergraphs by adding an edge that exists in the data graph G and is adjacent to a node in M (Line 6). Given each such resulting supergraph M', let $(u, l, w) = E_{M'} \setminus E_M$ and, without loss of generality, assume $u \in V_M$. If $w \in V_M$, then the only child of P obtained from M' is $P + (f^{-1}(u), l, f^{-1}(w))$ (Line 13). [6] If $w \notin V_M$, then two child patterns are obtained: $P + (f^{-1}(u), l, w)$ and $P + (f^{-1}(u), l, z)$, where z is a variable and $z \notin X_P$ (Line 17; Line 21 for the symmetric case). Fig. 2.5 shows an example of obtaining a pattern's children. For instance, P_{10} can be obtained by adding e_1, which is obtained by replacing `S1` of edge `e` with variable `?s`.

[6] For brevity, we denote by $P + e$ the supergraph of P by adding edge e.

Algorithm 5: Find all the children of a given pattern.

FIND-CHILDREN $(v_0, P, \mathcal{M}_P, G)$

$D \leftarrow \varnothing$; `// The set of` P`'s children.`

$\mathbb{M} \leftarrow \{M \in \mathcal{M}_P \mid f{:}V_P \to V_M$ and $\exists x \in X_P$ s.t. $f(x) = v_0\}$; `// Rule 1`

foreach $M \in \mathbb{M}$ **do**

Let f be the bijection $f : V_P \to V_M$;

$\overline{E_M} = \{(u, l, w) \in E_G \setminus E_M \mid u \in V_M$ or $w \in V_M\}$;

foreach $(u, l, w) \in \overline{E_M}$ **do**

$z \leftarrow$ a new variable and $z \notin X_P$;

if $\nexists\ x \in X_P$ *s.t.* $f(x) = u$ *or* $f(x) = w$ **then**

continue; `// Rule 2`

else if $u \in V_M$ *and* $w \in V_M$ **then**

$x \leftarrow f^{-1}(u), y \leftarrow f^{-1}(w)$;

$P_1 \leftarrow P + (x, l, y)$;

$D \leftarrow D \cup \{P_1\}$;

else if $w \notin V_M$ **then** `//` $\exists\ x \in X_P$ `s.t.` $f(x) = u$

$x \leftarrow f^{-1}(u)$;

$P_1 \leftarrow P + (x, l, w)$; $P_2 \leftarrow P + (x, l, z)$;

$D \leftarrow D \cup \{P_1, P_2\}$;

else `//` $\exists\ y \in X_P$ `s.t.` $f(y) = w$

$y \leftarrow f^{-1}(w)$;

$P_1 \leftarrow P + (u, l, y)$; $P_2 \leftarrow P + (z, l, y)$;

$D \leftarrow D \cup \{P_1, P_2\}$;

return D;

(1) (x, l, y)	(2) (x, l, z)	(3) (x, l, w)	(4) (u, l, y)	(5) (u, l, z)
(6) (u, l, v)	(7) (u, l, w)			

Figure 2.6: Consider a pattern P and its child pattern P'. The 7 types of the extra edge $e = E_{P'} \setminus E_P$. x, y, z are variables, $x, y \in X_P$, $z \notin X_P$. u, v, w are non-variables, $u, v \in V_P \cap \mathcal{I}$, and $w \in V_G \setminus V_P$.

2.5.3 Pattern Pruning Strategies

The search space $\mathbb{P}$ of patterns as defined in Section 2.5.1 and constructed using the match-based pattern construction method in Section 2.5.2 has an enormous size. To ensure efficiency, the Pattern Generator (PG) employs two pruning rules to exclude irrelevant patterns from $\mathbb{P}$ and to avoid repeated constructions of patterns from certain type of parent patterns.

Rule 1 (*RelevantOnly*)**.** Exclude a pattern if it does not define any context for the entity of interest v_0.

The rational behind Rule 1 is, for discovering exceptional facts about v_0, a pattern is relevant only if it defines a context for v_0. By this rule, the match-based pattern construction method only expands a match in which v_0 is an image of a variable in P. It is guaranteed that the patterns obtained define v_0's contexts.

Rule 2 (*VarOnly*)**.** Expand a pattern only if the new edge has at least one variable.

Let P' be a child pattern of P. The extra edge in P', i.e., $e{=}E_{P'} \setminus E_P$, belongs to one of the 7 types in Fig. 2.6. Rule 2 avoids constructing P' from P if e belongs to types 6-7. This rule is based on Theorem 6. Simply put, enforcing Rule 2 will not miss any contexts of v_0.

Theorem 6. Let P' be a child of $P \in \mathbb{P}$, $e = E_{P'} \setminus E_P$, $C^P_{v_0}$ be all the contexts of v_0 defined by P: $C^P_{v_0} = \{R^P_{x'} \mid x' \in X_P, v_0 \in R^P_{x'}\}$, then $C^{P'}_{v_0} = C^P_{v_0}$, if e belongs to types 6-7.

Proof. Since both ends of e are entities, we have $X_P = X_{P'}$. By Theorem 5, $\forall M' \in \mathcal{M}_{P'}$, there exists $M \in \mathcal{M}_P$ which is a subgraph of M'. Let $e' = E_{M'} \setminus E_M$, then $e' = e$

by Definition 2. Therefore, $\forall x \in X_{P'}$, $R^{P'}_x \subseteq R^P_x$. Similarly, $\forall M \in \mathcal{M}_P$, the graph $M+e$ is a match to P'. As a result, $\forall x \in X_P$, $R^P_x \subseteq R^{P'}_x$. In sum, $\forall x \in X_P$, $R^P_x = R^{P'}_x$, and $C^P_{v_0} = \{R^P_{x'} \mid x' \in X_P, v_0 \in R^P_{x'}\} = \{R^{P'}_{x'} \mid x' \in X_{P'}, v_0 \in R^{P'}_{x'}\} = C^{P'}_{v_0}$. ■

2.5.4 Pattern Selection Heuristics (h)

Even with the rules proposed in Section 2.5.3, there are still too many patterns. In this section, we propose two scoring heuristics for selecting promising patterns to visit, to substantiate the function h in Line 5 of Alg. 4. A heuristic gives each pattern a score, based on which the w patterns with the highest scores form the beam for the next iteration of beam search.

Heuristic 1 (Optimistic)**.** Given a pattern P, the entity of interest v_0, let $\mathcal{C}^P_{v_0}$ be the set of contexts defined by P, i.e., $\mathcal{C}^P_{v_0} = \{C^{P,x}_{v_0} | x \in X_P, v_0 \in R^P_x\}$, then

$$h_{opt}(v_0, P) = \max_{C \in \mathcal{C}^P_{v_0}} upper(v_0, \varnothing, C)$$

where $upper(v_0, \varnothing, C)$ is a upper bound of χ with regard to C for any subspace (see Defintion 9).

h_{opt} simply uses the exceptionality score upper bound of P. It optimistically assumes the ideal case for each pattern, where the entity of interest is most exceptional among the entities in a context defined by the pattern. In Section 2.4, we discussed the upper bound functions for various exceptionality functions. Note that we have $p_{v_0.\varnothing} = 1$ (Eq. (2.1)) since $v.\varnothing = \texttt{null}$ (Definition 6) and we consider all null values equal, $upper_o(v_0, \varnothing, C) = 1 - \frac{3\times|\ C\ |-2}{|\ C\ |^2}$, $upper_f(v_0, \varnothing, C) = 1 - \frac{1}{|\ C\ |}$, and $upper_i(v_0, \varnothing, C) = 1 - 2^{-|C|\log_2|C|\ /\ (|C|\log_2|C|-|C-1|-1)\log_2(|C-1|))}$. In sum, all the three upper bounds increase when the context size increases. In other words, h_{opt} selects the patterns that define large contexts. However, a large context may contain many entities of different characteristics, which may actually make the entity of interest less

exceptional. Note that, since h_{opt} depends on context size $|C|$, all the child patterns of P need to be evaluated in order to get $|C|$. It is also required for heuristic h_{conv} below for the same reason.

Heuristic 2 (Convergent)**.** Consider a pattern P and the entity of interest v_0. Given P', a parent of P in the pattern search tree, we define $r_x = |C_{v_0}^{P,x}| \;/\; |C_{v_0}^{P',x}|$. The score of P is

$$h_{conv}(v_0, P) = \max_{(P',P)\in E_{\mathbb{P}} \text{ and } P'\in B, C_{v_0}^{P',x}\in\mathcal{C}_{v_0}^{P'}} \Big[r_x \times \max_{A\subseteq A_{v_0}} \chi(v_0, A, C_{v_0}^{P',x}) + (1 - r_x) \times \mathit{upper}(v_0, \varnothing, C_{v_0}^{P,x}) \Big]$$

The h_{conv} score of P is a weighted sum of the upper bound of P (for any subspace) and the best score of the parent pattern P'. Note that Maverick performs a beam search and the patterns visited form a pattern search tree. P could be constructed from different parent patterns in the current beam B. The above equation thus uses the best score across all such parents. For this reason, the edge adjacent to P in the pattern search tree comes from the parent P' that gives it the best score. If P' posses some highly-scored context-subspace pairs, h_{conv} gives favorable score to P if P and P' define similar contexts; otherwise, h_{conv} favors a P that defines smaller contexts. Compared with h_{opt}, h_{conv} is potentially both more efficient and more effective. It can be more efficient since it may favor child patterns that define smaller contexts. Such child patterns usually can be evaluated more efficiently since they have less matches. It can be more effective since it discards child patterns that define contexts where the entity of interest may not be exceptional, based on the highest score of the context-subspace pairs for the parent pattern. When h_{conv} is used for choosing patterns to form the beams, the sizes of the contexts defined by the

patterns in a path of the tree may gradually become smaller and eventually converge. We thus call h_{conv} Convergent.

Complexity Analysis of the Beam Search Method We present a brief analysis of the complexity of our beam search method. Let w be the beam size, if Maverick stops at level l of the Hasse diagram $\mathbb{P}$, which is the search sapce of patterns, then it evaluates exceptionality in at least $(l-1)w+1$ contexts (patterns). For each context, the complexity of computing exceptionality is $O(2^{|A_{v_0}|})$, as we discussed in Section 2.4. Assume the average degree of an entity is d, the average number of variables in a pattern of size k is $\frac{k}{2}$, then a pattern of size k has $(d+1)^{k/2}$ children. For each child, Maverick needs to compute h scores for selecting promising candidates (Section 2.5.4). The main computational cost of h scores is the calculation of context sizes, which requires pattern evaluation. Given a pattern of size k, its evaluation can be done in $O(|E(G)|^k)$, since it may require k self-join operations on E_G. In sum, the estimated complexity is $O((w(l-1)+1)(2^{|A_{v_0}|}+\sum_{k=0}^{l}(d+1)^{k/2}\times|E(G)|^k)) = O(2^{|A_{v_0}|}+d^l|E(G)|^l)$.

2.6 Experiments

2.6.1 Experiment Setup

The framework and algorithms of Maverick are implemented in Python. The experiments were conducted on a 16-core, 32GB-RAM node in Stampede—a cluster of the Extreme Science and Engineering Discovery Environment (XSEDE: *https://www.xsede.org*).

Datasets[7] The experiments used the following two real-world graphs:

[7]All graphs in experiments are available in Neo4j format at *https://goo.gl/vavMcK*.

- WCGoals. It was constructed by crawling data from the FIFA World Cup website (*http://www.fifa.com/worldcup/index.html*). It consists of 49,078 nodes, 158,114 edges, 13 different edge labels, and 11 entity types: WorldCup, RoundCategory, Round, Stadium, Team, Game, Group, Player, Bibnum, Participant, and Goal.
- OscarWinners. This is a subgraph of Freebase. It has 42,148 nodes, 63,187 edges, 24 distinct edge labels, and 13 entity types including Person, FilmCrew, AwardWon, FilmCharacter, AwardCategory, Performance, Genre, Award, Film, Country, FilmCrewRole, Ceremony, and SpecialPerformanceType. Each film in the graph has won at least one Academy Award (Oscar).

The two graphs were stored using Neo4j (*https://neo4j.com*) graph database. The patterns are expressed in Neo4j's query language Cypher. The experiment results using the two graphs and different expectionality scoring functions are similar. Therefore, we only report our findings on WCGoals and exceptionality function χ_o, except that Section 2.6.6 reports the discovered exceptional facts using both WCGoals and OscarWinners.

Methods Compared The experiments compared the performance of a breadth-first search method and several beam search methods (Section 2.3) coupled with different heuristics (Section 2.5.4):

- Beam-Rdm: Beam search that randomly selects child patterns.
- Beam-Opt: Beam search using h_{opt} in selecting child patterns.
- Beam-Conv: Beam search using h_{conv} in selecting child patterns.
- Breadth-First: The breadth-first search method that enumerates all possible patterns.

The family of beam search methods and Breadth-First differ in two ways. *Firstly*, beam search only visits a fixed number of patterns at each level of the pattern search

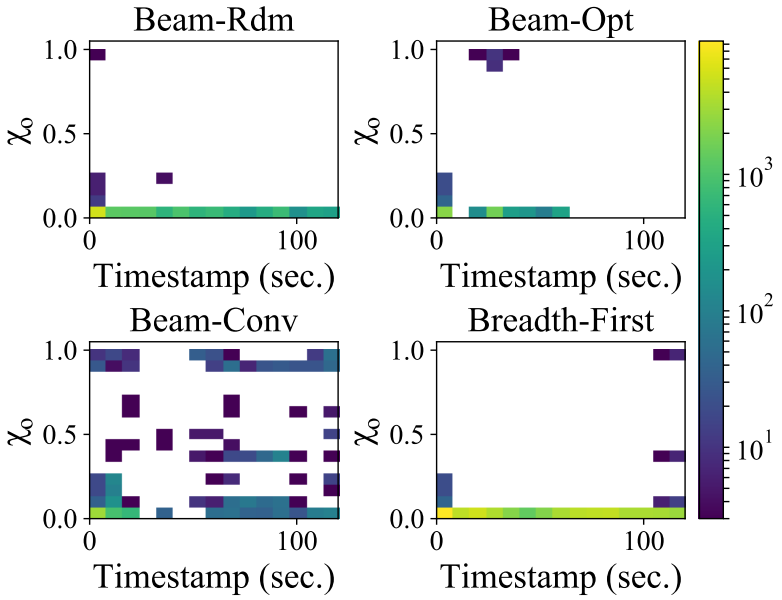
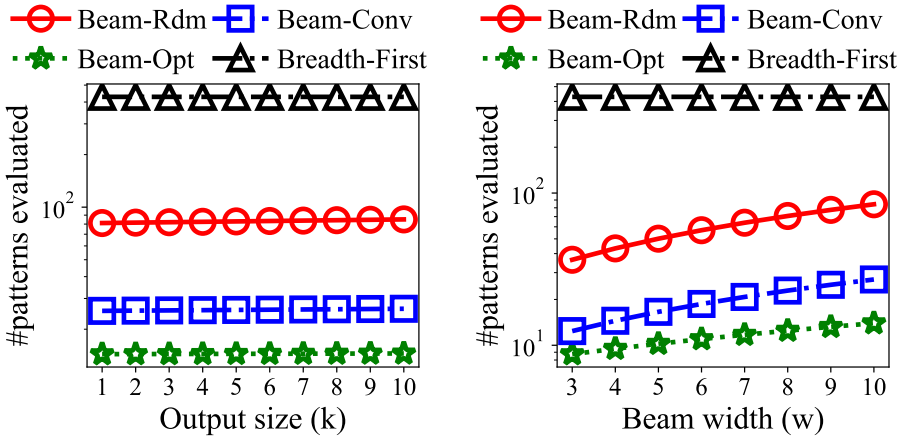


Figure 2.7: The heat map of exceptionality scores (χ_o) and timestamps of all the discovered context-subpsace pairs during 2-minute runs for 10 entities of interest (v_0) in WCGoals ($k = 10$, $w = 10$).

tree, whereas Breadth-First visits all. *Secondly*, beam search visits the patterns by the decreasing order of their scores, whereas Breadth-First does not assume any order. The experiment results establish that, even though Breadth-First may evaluate more patterns than the beam search methods in a fixed time frame, it is not as effective as Beam-Conv which discovers more highly-scored context-subspace pairs using less time.

2.6.2 Efficiency

We measured how fast Maverick discovers highly-scored context-subspace pairs and how fast it explores the search space of patterns. We executed Maverick for multiple 2-minute runs and recorded a) the scores of discovered context-subspace pairs; b) the time when each context-subspace pair was discovered; and c) the number of visited patterns in the outer loop of the framework.



a Varying k, fixing $w = 10$.

b Varying w, fixing $k = 10$.

Figure 2.8: Effect of k and w on the number of evaluated patterns.

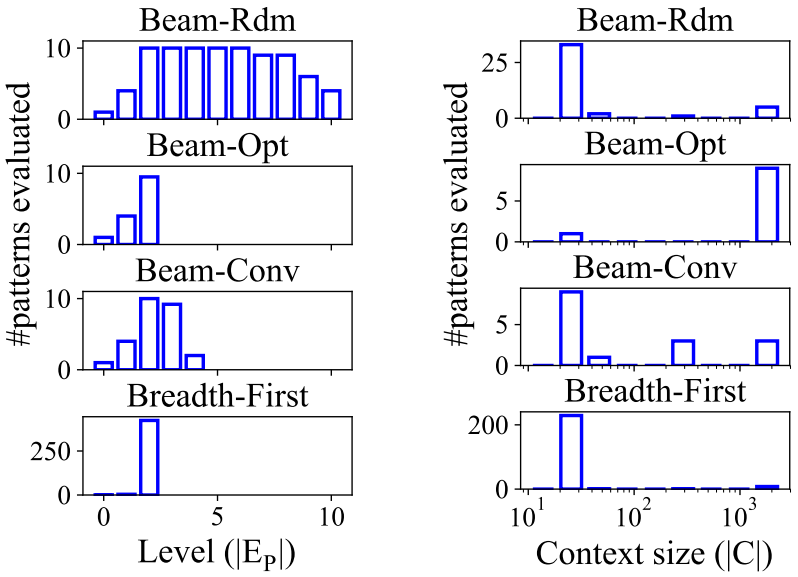
Fig. 2.7 shows the heat map of the context-subspace pairs' exceptionality scores by their timestamps. It includes all the discovered context-subspace pairs during the 2-minute runs for 10 entities of interest in WCGoals. We run 10 times per entity for all the methods, since Beam-Rdm selects child patterns randomly. Both the output size k and the beam width w were set to 10. The 10 entities were randomly chosen from those that have highly-scored context-subspace pairs. Each bucket in the figure corresponds to a particular range of scores and a 8-second time frame in the 2-min run. The color of the bucket reflects how many context-subspace pairs (from all 100 runs for the 10 entities) discovered during the time frame fall into the corresponding score range. Intuitively, if the upper left portion of a heat map is more populated, the corresponding method performs better, since it means the method discovers highly-scored pairs faster. If the upper portion of a heat map is more populated, it means the method discovers more highly-scored pairs. The figure shows that Beam-Conv is both efficient and effective in discovering highly-scored context-subspace pairs. In contrast, Beam-Opt performed poorly. The results confirm the analysis in Section 2.5.4: preferring patterns that produce large contexts (h_{opt})

degrades not only the efficiency but also the effectiveness of Maverick. It is because such patterns are usually more expensive to evaluate and the produced contexts may include more varieties of entities, which makes the entity of interest less exceptional. With regard to Breadth-First, since it enumerates candidate patterns exhaustively, it may discover some highly-scored pairs that reside in the low levels of the pattern search tree. For example, some highly-scored pairs for entity Goal(46683) were found using the 2-edge pattern in Fig. 2.2a. Given that the number of patterns with no more than 2 edges is small (more details in the discussion of results regarding pruning strategies), Beam-Rdm is likely to hit such small patterns that define contexts in which the entity of interest is exceptional.

Fig. 2.8 shows the impact of output size k and beam width w on how many patterns Maverick can manage to evaluate, in other words, how many nodes in $\mathbb{P}$ it can manage to visit. The y-axis is the average number of evaluated patterns across the aforementioned 10 runs. Since we observed similar results on the 10 entities from WCGoals, the figure only depicts the results on Goal(46683). Fig. 2.8a shows that varying k from 1 to 10 (fixing w at 10) barely had any impact on the number of evaluated patterns. Since k controls the number of context-subspace pairs that Maverick returns, it mainly affects EE, which is responsible for finding top-k subspaces with regard to each context. Thus k needs to be very large in order to have a significant impact on the number of evaluated patterns, since EE is the least time-consuming component, as explained as follows. Table 2.2 provides the breakdown of execution time of different search methods into the three components in the workflow—Context Evaluator (CE), Exceptionality Evaluator (EE), and Pattern Generator (PG). The results are the average of the runs which are the same as in Fig. 2.7. (The summation in each column is slightly less than 100%, since we do not include operations such as framework initialization in the breakdown.) Another observation from Table 2.2

Table 2.2: Breakdown of execution time by components.

	Beam-Rdm	Beam-Opt	Beam-Conv	Breadth-First
CE	25.52%	1.56%	1.90%	28.36%
EE	0.41%	0.65%	0.32%	2.79%
PG	61.49%	97.69%	95.92%	53.89%



a By level. b By context size.

Figure 2.9: Number of evaluated patterns by level and context size.

is that the execution time of PG dominates more substantially in Beam-Opt and Beam-Conv than in Beam-Rdm and Breadth-First. The reason is PG in both Beam-Opt and Beam-Conv needs to compute h for each child pattern based on the pattern selection heuristics, which entails evaluating the child patterns to obtain the context sizes. In fact, on average, PG in Beam-Opt and Beam-Conv spent more than 99% and 96% of its time on applying the heuristics.

Fig. 2.8b depicts the results when w varied from 3 to 10 and k was fixed at 10. It shows the number of evaluated patterns increased by w in the three beam search methods. When w increases, the methods evaluate more patterns from lower levels in the pattern search tree, which have less edges and can be evaluated more efficiently

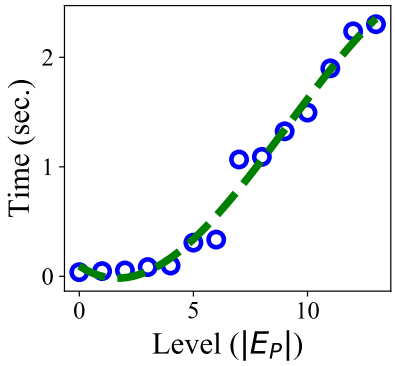


Figure 2.10: Time spent by Context Evaluator (Alg. 2) on evaluating patterns at different levels.

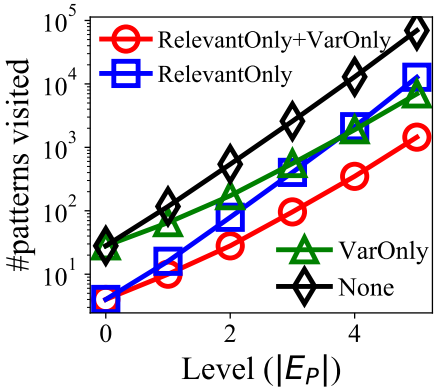


Figure 2.11: The pruning power of different pruning strategies.

than those from higher levels. In a fixed time frame, the methods can then evaluate more patterns in total, as shown in the figure. Fig. 2.10 shows the average time that Context Evaluator (Alg. 2) spends on pattern evaluation increases when the level of pattern (i.e., the number of edges) increases. Since Breadth-First does not need to calculate scores for patterns and does not have a limit on the number of patterns to visit at each level, it tends to evaluate more patterns but may only evaluate patterns at low levels. Fig. 2.9a compares the numbers of patterns evaluated at different levels by the four methods, when both k and w stayed at 10. Breadth-First evaluated patterns up to level 3 and spent most of its time on level 3. On the contrary, the beam search methods evaluated at most 10 patterns at each level and covered more levels.

Fig. 2.8b also suggests that Beam-Rdm evaluated more patterns than Beam-Conv and Beam-Opt. It is because Beam-Rdm (like Breadth-First) does not compute scores for child patterns, which is expensive. Since Beam-Opt favors patterns that define larger contexts, it evaluated the fewest patterns since it spent more time to calculate the sizes of the contexts. On the other hand, Beam-Conv prefers patterns defining smaller contexts, which allowed it to evaluate more patterns. This is verified

in Fig. 2.9b, which shows the numbers of evaluated patterns with different context sizes, when k and w were both 10.

Effect of pruning strategies We examined the effectiveness of the two pruning rules from Section 2.5.3 by comparing the following pruning strategies. In order to comprehensively compare these strategies, we used Breath-First as the search method since it exhaustively enumerates all possible candidate patterns at all levels.

- None: No child pattern pruning rule is applied;
- RelevantOnly (Rule 1);
- VarOnly (Rule 2);
- RelevantOnly+VarOnly: Apply both *RelevantOnly* and *VarOnly*.

Fig. 2.11 shows the number of patterns visited by Breath-First on the data graph in Fig. 2.1. The figure reveals that both rules can significantly reduce the number of candidate patterns. For instance, there are $69,582$ candidate patterns at level 5 when no pruning rule is applied (*None*). The number is reduced to $12,740$ and $6,963$ by following *RelevantOnly* and *VarOnly*, respectively. It is further reduced to $1,448$ with both rules applied (*RelevantOnly*+*VarOnly*). The figure also shows that the number of patterns still grows exponentially to the level of the pattern search tree even with both pruning rules applied, which suggests an enormous search space of patterns. Since *VarOnly* is stricter than *RelevantOnly*, as *VarOnly* only allows expanding on variable nodes, the growth rate of *VarOnly* can be smaller than *RelevantOnly*. Fig. 2.11 also confirms that.

Effect of upper bound of exceptionality functions Fig. 2.12 depicts the effect of using upper bound functions in pruning subspaces (Section 2.4.3). It shows the time and the number of subspaces visited for Game(903)—one of the 10 entities used in Fig. 2.7—with/without applying upper bound functions under varying k. (Results

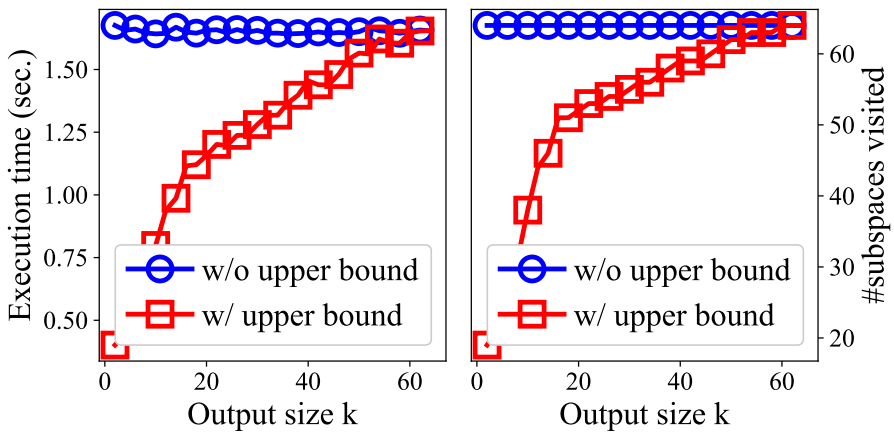


Figure 2.12: Effect of subspace pruning (upper bound functions).

for the other 9 entities are similar.) The measures are averages of 10 runs. The figure verifies that the upper bound functions significantly improve the performance of exceptionality score calculations. Under relatively small k (e.g. 10), the execution time was reduced by more than half when the upper bound was applied. As k increased, the upper bound function's pruning power gradually diminished. Eventually, it was no longer able to prune any subspaces after $k = 60$.

2.6.3 Effectiveness

Section 2.6.6 reports a few examples of discovered exceptional facts using both WCGoals and OscarWinners. We also conducted experiments to verify if Maverick can effectively discover highly-scored context-subspace pairs. Fig. 2.13 shows the score distributions of the top-10 context-subspace pairs for the same 10 entities in Section 2.6.2. There were 10 2-minute runs per entity. Both k and w were set to 10. The results in Fig. 2.13 are averaged over all entities and all runs. The last row are the results when Maverick uses patterns mined by a frequent pattern (FP) mining

algorithm [43] as candidate patterns, instead of the ones discovered in $\mathbb{P}$. We set the minimum support to be $1,000$, as lower value leads to excessive execution time.[8]

The results show that the output of Beam-Rdm mainly consists of pairs scored low. It is expected because the chance of hitting a promising pattern by a random method is very low due to the large search space of patterns. It is not surprising either to observe Beam-Opt performed badly as explained in Section 2.6.2. In contrast, Beam-Conv significantly outperformed other beam search methods, as it found much more highly-scored context-subspace pairs. It also found substantially more highly-scored pairs than Breadth-First in score range $[0.8 - 1.0]$. This observation confirms that a wide pattern search tree hinders Breadth-First's performance. Using FPs was not effective in discovery of exceptional facts. There are mainly two reasons. 1) Due to practical considerations such as efficiency and resources, FP mining techniques usually consider only node types (e.g., Team) but not node IDs (e.g., BRA). 2) An FP mining algorithm is not designed for individual entities. This leads to two consequences. One is that there may be no FP that defines some context for a given entity. The other is that the exceptionality of the entity may not be revealed in the contexts defined by the FPs. In fact, the experiments on WCGoals yielded only 12 FPs and all of them are about only two node types: Participant and Goal.

We also use a variation of *coverage error* [44] to measure the effectiveness of the four methods. For each method, we evaluated the result of its 2-minute run, using the result of its 10-hour run as the ground truth. The ground truth is the list of discovered context-subspace pairs during the 10-hour run, ranked by their exceptionality scores. Given the set of discovered context-subspace pairs in a 2-minute run, H, the coverage error is the average rank position of the pairs in the ground truth,

[8]The algorithm did not finish after more than 10 hours on graph WCGoals when the minimum support was set to 500.

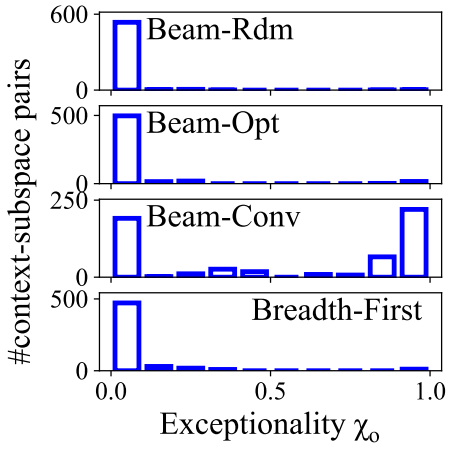


Figure 2.13: Score distributions of top-10 context-subspace pairs for 10 entities, 10 2-minute runs per entity.

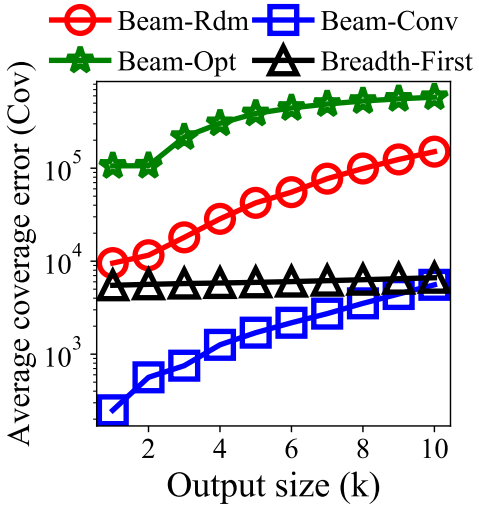


Figure 2.14: Average coverage error on 10 entities. Beam width 10.

defined by $Cov = \frac{1}{|H|}\sum_{(C,A)\in H} \text{rank}_{(C,A)}$. Fig. 2.14 reports the average coverage error of each method under varying output size k. Table 2.3 shows the average and median coverage errors under varying beam width w. In Fig. 2.14, the coverage error of Beam-Conv is less than other methods by orders of magnitude, which suggests that Beam-Conv found highly-scored context-subspace pairs. Table 2.3 shows that coverage error decreases when beam width increases. The reason is that a wider beam leads to more patterns visited at every level and thus a better coverage of patterns. It is especially beneficial when highly-scored pairs reside in patterns at lower levels.

2.6.4 Scalability

To test the scalability of Maverick, we generated synthetic graphs using a benchmark data generator *BSBM*[9] about products, vendors, consumers, and reviews. We varied the number of products from 100 to 2,000, which resulted in graphs of order $|V(G)|$ from 14,195 to 215,895. Fig. 2.15 shows how the execution time of all

[9] *http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/BenchmarkRules*

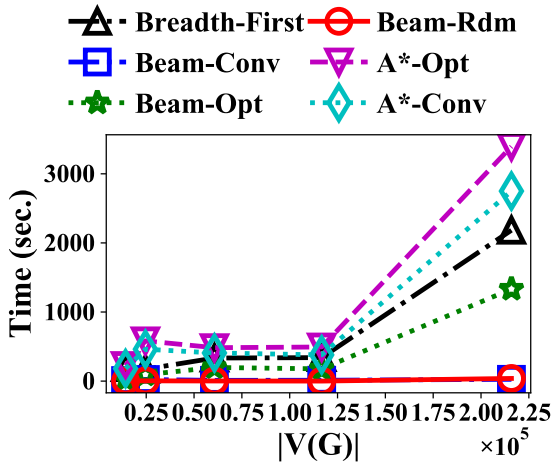


Figure 2.15: Execution time of enumerating all candidates up to level 2 by different orders of graphs.

Table 2.3: The effect of beam width (w) on the coverage errors of top-10 context-subspace pairs of 10 entities. In each cell: the average and the median coverage errors. Both numbers are the smaller the better.

w	Beam-Rdm	Beam-Opt	Beam-Conv	Breadth-First
3	3375.7/2636.9	2151.0/1071.5	**49.7/12.5**	383.0/390.5
4	3293.2/1607.3	2675.7/1622.1	**52.1/12.5**	383.0/390.5
5	2743.2/1871.2	2418.3/1550.8	**30.2/26.0**	383.0/390.5
6	2890.9/1809.2	2288.7/1259.7	**20.6/1.0**	383.0/390.5
7	2821.4/1398.9	1789.3/1259.7	**21.8/1.0**	383.0/390.5
8	2646.4/1818.6	1721.5/1168.8	**78.3/3.0**	383.0/390.5
9	2262.8/1653.4	1365.5/1107.3	**36.6/4.2**	383.0/390.5
10	2720.8/1619.9	1365.5/1107.3	**58.4/22.1**	383.0/390.5

algorithms increased along with the order of the graphs. Given that Beam-Conv and Beam-Rdm need less computation in candidate selection, both scaled much more gracefully than others. The figure also shows the execution time of two A* algorithms guided by h_{opt} and h_{conv}, respectively. The results confirmed that computing h_{opt} is more expensive than h_{conv}. This is because computing h_{opt} requires examining all the variables in a pattern, while computing h_{conv} only requires that for context-defining variables in the parent. Due to the overhead of computing h scores, the A* algorithms are shown more expensive than Breadth-First.

2.6.5 User Study for Comparing Exceptionality Scoring Functions

We conducted a user study to assess the quality of four different exceptionality scoring functions χ_f (one-of-the-few, Section 2.2), χ_o (outlyingness, Section 2.4.2), χ_i (isolation score, Section 2.4.2), and *frequency rank* (Eq. (2.1)). The measure *frequency rank* is simply to rank the exceptionality of entities in context C with respect to subspace A based on their attribute value frequency $P_{v.A}$, as defined in Eq. (2.1). The lower the frequency, the more exceptional it is. We compared the rankings of exceptional facts based on these functions as well as actual user preferences.

The user study participants were asked to choose a fact from a pair of facts that they deemed to be more exceptional. For quality assurance we manually crafted a set of trivial facts that are clearly non-exceptional or dull. We then formulated test pairs, of which each is composed of a regular fact and a trivial fact. The participants are expected to choose the regular fact as more exceptional. A participant's quality is thus gauged by their accuracy on the test pairs, which were mixed together with regular pairs without disclosure to the participants.

We used 10 regular facts and thus 45 pairs of these facts. We randomly selected 10 entities from graph OscarWinners, and run Maverick using Beam-Conv to discover exceptional facts of the entities. The exceptionality scoring function used in the discovery was outlierness (χ_o). Among all the facts discovered by Maverick, 10 facts are picked so that the scores of facts are roughly evenly distributed in $[0, 1]$. For each selected fact, χ_f, χ_i, and *frequency rank* are also calculated. We crafted 8 trivial facts and formulated 8 test pairs by pairing up these trivial facts with regular ones. Hence a participant responded to at most 53 pairs. The facts were presented to the participants in their natural language descriptions which we manually generated in the form of one-of-the-few facts. The study was conducted on line, for which the

Table 2.4: User study results at different participant quality levels.

# of pairs correct ($\geq$)	# of participants	χ_i	χ_f	χ_o	*frequency rank*
0	84	0.370	0.564	0.295	0.429
1	84	0.370	0.564	0.295	0.429
2	78	0.392	0.585	0.317	0.412
3	74	0.410	0.597	0.335	0.400
4	66	0.449	0.642	0.378	0.338
5	53	0.491	0.649	0.422	0.377
6	43	0.614	0.730	0.558	0.283
7	22	0.738	0.831	0.696	0.061
8	9	0.750	0.864	0.711	-0.007

participants were solicited from computer science graduate students in the authors' institution. $4,212$ responses from 84 participants were recorded in total.

For each exceptionality scoring function, we constructed a vector X of 45 values corresponding to the 45 pairs of regular facts. For each pair, the value in X is the difference between the two facts' ranks according to the scoring function. We also constructed another vector Y, in which a value is the difference between how many participants favored one fact versus another in the corresponding pair. The correlation between the scoring function and the participants is calculated using the Pearson Correlation Coefficient (PCC) which is defined as $(\mathrm{E}(XY) - \mathrm{E}(X)\mathrm{E}(Y))/(\sqrt{\mathrm{E}(X^2) - (\mathrm{E}(X))^2}\sqrt{\mathrm{E}(Y^2) - (\mathrm{E}(Y))^2})$. A PCC value in the ranges of [0.5,1.0], [0.3,0.5) and [0.1,0.3) indicates a strong, medium and small positive correlation, respectively [45].

The results of the user study are presented in Table 2.4, in which each row shows the PCC values calculated using participants at a different quality level (measured by number of test pairs the participants got correct, i.e., the first column in the table). We can make a few observations on the results. 1) As the quality level increases the number of participants that are accounted for at that level decreases,

showing that the test pairs were successful in filtering out low performing participants. 2) For scoring functions χ_i, χ_f and χ_o, the correlation with participants steadily increases when the participants' quality increases. 3) In general, the scoring function that performed the best was χ_f, followed by χ_i and χ_o. The results show a strong correlation between these three functions and high-quality human participants, which suggests these functions are effective in ranking the facts. The observation that χ_f performed the best could be due to a bias: the natural language descriptions of the facts were in the form of one-of-the-few facts. (On a side note, this suggests a strength of χ_f related to usability, as there is no clear way of directly expressing facts in line with χ_i and χ_o.) 4) On the contrary, *frequency rank* displayed a decrease in correlation as participant quality increases and its correlation was never strong. We reason that this could be due to low performing participants directly using frequency to hastily assess whether a fact is exceptional or not, without carefully examining the nature of the fact. The fact that *frequency rank* attains stronger correlation with lower-quality participants verifies that it cannot be used as a robust exceptionality scoring function, as explained in Section 2.2.

2.6.6 Case Study

To illustrate the effectiveness of Maverick, we present below some examples of exceptional facts discovered by Maverick in both graph WCGoals and graph Oscar-Winners.

◯ **Goal(46683) is the only own goal in Brazil's World Cup history.**

Exceptionality $\chi_o = 0.986$

Subspace $\{(\textit{awared-to}, \rightarrow)\}$

Context $C^{P,x_0}_{\text{Goal(46683)}}$, where $P = \{(x_0, \textit{scored-by}, x_1), (x_1, \textit{play-for}, \text{BRA})\}$

Indeed, among all the 221 goals that were scored by Brazil players in the *FIFA World Cup Finals* tournaments, Goal(46683), which was awarded to Croatia, was the only goal not awarded to Brazil. This exceptional fact has a very high score.

◯ **Among all the crew members of Oscar winning films, Paul J. Franklin (FilmCrew(7674)) is the only crew member with role Computer Animation.**

Exceptionality $\chi_f = 0.784$

Subspace $\{(\textit{film-crew-role}, \rightarrow)\}$

Context $C^{P_0,x_0}_{\text{FilmCrew(7674)}}$

This example demonstrates the utility of Maverick in revealing data errors, as motivated in Section 2.1. While investigating why this entity is exceptional, an analyst will realize the exceptional fact is due to a data error. An edge mistakenly links from node Paul J. Franklin to a genre node Computer Animation which is incorrectly used in this case as a role node. The correct crew role node should have been Computer Animator.

◯ **Goal(23464) is the only goal awarded to Paraguay, among all the goals scored in matches hosted in Mexico City that had at least two goals.**

Exceptionality $\chi_f = 0.983$

Subspace $\{(\textit{awared-to}, \rightarrow)\}$

Context $C^{P,x_1}_{\text{Goal(24227)}}$, where $P = \{(x_0,\ \textit{goal},\ x_1),\ (x_0,\ \textit{goal},\ x_2),\ (x_0,\ \textit{venue},\ \text{Mexico City})\}$

There are in total 62 goals scored in matches hosted in Mexico City, among which 58 were scored in 18 multiple-goal matches. These 58 goals were awarded to 12 different teams. Paraguay is the only team that was awarded only one of the 58 goals.

◯ **Game(899) is one of the only two games in which the home team was Senegal, among all the games where there was a player wearing the number 21 shirt.**

Exceptionality $\chi_f = 0.959$

Subspace $\{(\textit{home}, \rightarrow)\}$

Context $C^{P,x_1}_{\text{Game(899)}}$, where $P = \{(x_0, \textit{bibnum}, \text{Bibnum(21)}), (x_0, \textit{participate-in}, x_1)\}$

In 761 games some player wore number 21. Game(899) is one of the only two such games in which the home team was Senegal.

◯ **Among the Oscar winning films produced in the United States, The Lord of the Rings: The Return of the King (Film(31768)) is one of the only 7 films that were also produced in New Zealand.**

Exceptionality $\chi_o = 0.676$

Subspace $\{(\textit{country}, \rightarrow)\}$

Context $C^{P,x_0}_{\text{Film(31768)}}$, where $P = \{(x_0, \textit{country}, \text{USA})\}$

There are in total 662 Oscar winning films produced in the United States, of which 545 were produced solely in the United States. Only 7 of the co-produced films were co-produced in New Zealand. However, the score of this fact is not as high as that of the last two facts, because China, Brazil, and a few other countries co-produced even less films.

2.7 Related work

In exceptional fact discovery, the output context-subspace pairs can be viewed as a way of explaining outliers. Most conventional outlier detection solutions, including those for graphs, focus on finding outliers but do not explain why they are outly-

ing. For example, CODA [18] finds a list of community outliers, and FocusCO [19] clusters an attributed graph and then discovers outliers in the clusters. Besides the limitation that both approaches are only suitable for homogeneous graphs, it is up to users to figure out the explanations of the outliers. Although these two systems make such explanations easier by providing the communities or clusters in which the outliers reside, it still requires substantial expertise to summarize the communities/clusters' characteristics. A few works improve the interpretation of outliers' outlyingness [20, 46, 47]. For instance, systems such as [46] and [47] use visualization to help users identify outliers and potentially discover their outlying aspects.

Although most existing outlying aspects mining approaches focus on finding global outlying aspects and do not consider contexts [24, 25], there are a few attempts to find contextual outlying aspects [22, 21, 10, 23]. The general Maverick framework allows users to adopt any exceptionality measure in the literature such as outlierness [21]. Although Maverick focuses on categorical attributes at this stage, it can be extended for numerical attributes so that measures such as skyline points [10], promotiveness [23], outlierness [22], outlyingness rank [24], and z-score [25] can be adopted in the framework.

Trummer et al. [48] developed the SURVEYOR system to mine the dominant opinion on the Web regarding whether a subjective property (e.g., "safe cities") applies to an entity. This is useful for populating a knowledge base with ground truth for answering subjective queries. While they focus on deriving entities' hidden properties which may or may not be exceptional, Maverick focuses on finding exceptional entities using existing data in knowledge graphs.

CHAPTER 3

Discovering General Prominent Streaks in Sequence Data

3.1 Introduction

This chapter is on the problem of *prominent streak discovery* in sequence data. A piece of sequence data is a series of values or events. This includes time-series data, in which the data values or events are often measured at equal time intervals. Sequence and time-series data is produced and accumulated in a rich variety of applications. Examples include stock quotes, sports statistics, temperature measurement, Web usage logs, network traffic logs, Web clickstream, customer transaction sequence, social media statistics. Given a sequence of values, a *prominent streak* is a long consecutive subsequence consisting of only large (small) values. Examples of such prominent streaks include consecutive days of high temperature, consecutive trading days of large stock price oscillation, consecutive games of outstanding performance in professional sports, consecutive hours of high volume of TCP traffic, consecutive weeks of high flu activity, consecutive days of frequent mentioning of a person in social media, and so on.

It is insightful to investigate prominent streaks since they intuitively and succinctly capture extraordinary subsequences of data. Consider several example application scenarios: (1) Business analysts may be interested in prominent streaks in social media usage logs, e.g. streaks of re-tweeting a tweet, streaks of hashtagging a topic, and so on. (2) A security auditing may be performed after a streak of excessive login attempts is detected. (3) A cooling system can be started when a streak of days with high temperature has been discovered. (4) For disease outbreak detection,

we can identify prominent streaks in time series of aggregated disease case counts. Previous works on outbreak detection focus on conventional data mining tasks such as clustering and regression [49]. The concept of prominent streaks has not been studied before.

Prominent streak discovery can be particularly useful in helping journalists to identify newsworthy stories when data sequences evolve, investigators to find suspicious phenomena, and news anchors and sports commentators to bring out attention-seizing factual statements. Therefore it will be a key enabling technique for *computational journalism* [50]. In fact, we witness the mentioning of prominent steaks in many real-world news articles as shown in Chapter 1, where statements 4—7 indicate that general prominent streaks can have a variety of constraints. A streak can be on multiple dimensions (e.g., $\langle$point, rebound, assist$\rangle$), its significance can be with regard to a certain period (e.g., "since June 2009") or a certain comparison group (e.g., "the month of July"), and we may be interested in not only the most prominent streaks but also the top-k most prominent ones (e.g., "LeBron James joined Michael Jordan and Kobe Bryant as the only players", which means LeBron James's scoring streak mentioned above is among the top-3 streaks.)

Given its real-world usefulness and variety, the research on prominent streaks in sequence data opens a spectrum of challenging problems. In an earlier work [51], we proposed the concept of prominent streak and studied the problem of discovering the simplest kind of prominent streaks, i.e., those without the aforementioned constraints. In this chapter, we extend the work to discovering general multi-dimensional and top-k prominent streaks from multiple sequences, which shall substantially broaden the applicability of our study in real-world scenarios, as evidenced by the above stories in news articles.

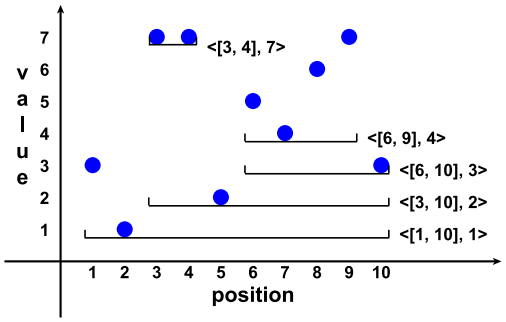


Figure 3.1: A Data Sequence and its Prominent Streaks.

3.1.1 Problem Definition

Definition 10 (Streak and Prominent Streak)**.** Given an n-element sequence $P = (p_1, \cdots, p_n)$, a *streak* is an interval-value pair $\langle [l, r], v \rangle$, where $1 \leq l \leq r \leq n$ and $v = \min_{l \leq i \leq r} p_i$.

Consider two streaks $s_1 = \langle [l_1, r_1], v_1 \rangle$ and $s_2 = \langle [l_2, r_2], v_2 \rangle$. We say s_1 *dominates* s_2, denoted by $s_1 \succ s_2$ or $s_2 \prec s_1$, if $r_1 - l_1 \geq r_2 - l_2$ and $v_1 > v_2$, or $r_1 - l_1 > r_2 - l_2$ and $v_1 \geq v_2$. For example, $\langle [1, 2], 3 \rangle \prec \langle [4, 7], 6 \rangle$ and $\langle [1, 2], 3 \rangle \prec \langle [3, 4], 5 \rangle$, while $\langle [1, 2], 3 \rangle$ and $\langle [7, 8], 3 \rangle$ do not dominate each other.

With regard to $P = (p_1, ...p_n)$, the set of all possible streaks is denoted by $\mathcal{S}_P$. A streak $s \in \mathcal{S}_P$ is a *prominent streak* if it is not dominated by any streak in $\mathcal{S}_P$, i.e., $\nexists s'$ s.t. $s' \in \mathcal{S}_P$ and $s' \succ s$. The set of all prominent streaks in P is denoted by $\mathcal{PS}_P$.

Problem Statement: The *prominent streak discovery problem* is to, given a sequence P, produce $\mathcal{PS}_P$.

Figure 3.1 is our running example which shows the assists made by an NBA player in 10 consecutive games $P = (3, 1, 7, 7, 2, 5, 4, 6, 7, 3)$. There are 5 prominent streaks in P– $\langle [1, 10], 1 \rangle$, $\langle [3, 10], 2 \rangle$, $\langle [6, 10], 3 \rangle$, $\langle [6, 9], 4 \rangle$, $\langle [3, 4], 7 \rangle$. Each streak is represented by a horizontal segment, which crosses the minimal-value points in the

streak and runs from the left end to the right end of the corresponding interval. For instance, $\langle[6, 9], 4\rangle$ is a prominent streak of minimal value 4, whose interval is from p_6 to p_9. It captures the fact that the NBA player made at least 4 assists in 4 consecutive games (game 6 $-$ game 9). $\langle[1, 10], 1\rangle$, the whole data sequence, is also a trivial prominent streak because no other streak can possibly dominate the sequence itself. The streak $\langle[8, 9], 6\rangle$ is an instance of non-prominent streaks because it is dominated by $\langle[3, 4], 7\rangle$.

Definition 10 focuses on the simplest type of prominent streaks. The concept of prominent streak can be extended in several ways. First, we may be interested in top-k prominent streaks which are dominated by less than k other streaks. Second, we may need to compare streaks from not only the same sequence but also multiple different sequences (e.g., sequences corresponding to different NBA players, cities, stocks, etc.) Third, the data points in a sequence can be multi-dimensional, leading to the pursuit of multi-dimensional prominent streaks. We have seen examples of all such general prominent streaks at the beginning of Section 3.1 and their combinations naturally exist. The focus of our following discussion will first be on the simplest prominent streak discovery problem. In Section 3.5, we discuss how to discover general prominent streaks.

Definition 10 and the problem statement focus on finding streaks of large values. To find streaks of small values (e.g., a stock index below 10000 for 12 consecutive weeks, described in the aforementioned second news article), two changes should be made. First, a streak should be captured by its interval length and the maximal value (instead of the minimal value) in the interval, i.e., $v = \max_{l \le i \le r} p_i$. Second, the dominance relation between streaks should be defined to prefer smaller values. More specifically, s_1 *dominates* s_2 if $r_1 - l_1 \ge r_2 - l_2$ and $v_1 < v_2$ (instead of $v_1 > v_2$), or $r_1 - l_1 > r_2 - l_2$ and $v_1 \le v_2$ (instead of $v_1 \ge v_2$). Given that the new definition would be

exactly symmetric to Definition 10, finding streaks of large and small values become the same problem. Hence, we only consider finding streaks of large values in the rest of this chapter.

3.1.2 Overview of the Solution

A brute-force method for discovering prominent streaks is not appealing. One can enumerate all possible streaks and decide if each streak is prominent by comparing it with every other streak. Given a sequence P with length n, there are $|\mathcal{S}_P| = \binom{n+1}{2}$ streaks in total. Thus the number of pair-wise streak comparison would be $\binom{|\mathcal{S}_P|}{2} = \frac{n^4+2n^3-n^2-2n}{8}$. Given a sequence of length 10000, the brute-force approach enumerates 10^8 streaks and performs 10^{16} comparisons. Many real-world sequences can be quite long. The sequence of daily closing prices for a stock with 40-year history has about 10000 values. A one-year usage log for a Web site has 8760 values at hourly interval.

Prominent streaks are in fact skyline points [52] in two dimensions– streak interval length $(r-l)$ and minimum value in the interval (v). A streak is a prominent streak (skyline point) if it is not dominated by any point, i.e., there exists no streak that has both longer interval and greater minimum value.

Based on this observation, our solution hinges upon the idea to separate the two steps of prominent streak discovery– *candidate streak generation* and *skyline operation* over candidate streaks. In candidate generation, we prune a large portion of non-prominent streaks without exhaustively considering all possible streaks. For skyline operation, we apply efficient algorithms from the rich literature on this topic, e.g., [52, 53, 54, 55]. The effectiveness of pruning in the first step is critical to overall performance, because execution time of skyline algorithms increases superlinearly by the number of candidate points [52].

Candidate streak generation

Algorithm 6: Baseline Method

Input: Data sequence P=$(p_1, ..., p_n)$

Output: Prominent streaks *skyline*

```
1 skyline ← empty
2 for r = 1 to n do
3     min_value ← ∞
4     for l = r downto 1 do
5         min_value ← min(p_l, min_value)
6         s ← ⟨[l, r], min_value⟩ // candidate streak
7         skyline ← skyline_update(skyline, s)
```

We considered three methods with increasing pruning power in candidate generation– a baseline method, a non-linear LPS (local prominent streak)-based method, and a linear LPS-based method. The baseline method exhaustively enumerates $\mathcal{S}_P$, all possible streaks in a sequence P, by a nested-loop over the values in P. Thus, the baseline method does not have pruning power. The sketch of this method is in Algorithm 6. It produces quadratic ($\frac{n(n+1)}{2}$) candidate streaks. We then propose the concept of *local prominent streak* (LPS) for substantially reducing the number of candidate streaks (Section 3.3). The intuition is, given a prominent streak s, there cannot be a super-sequence of s with greater or equal minimal value. In other words, s must be locally prominent as well. Hence we only need to consider LPSs as candidates. The algorithm sequentially scans the data sequence and maintains possible LPSs. The non-linear LPS-based method finds a superset of LPSs as candidates, while the linear LPS-based method guarantees to find only LPSs.

Skyline operation

Algorithm 7: Update Dynamic Skyline (*skyline_update*)

Input: Dynamic skyline *skyline*, new candidate streak $s = \langle [l, r], v \rangle$

Output: Updated dynamic skyline *skyline*

```
1 Find the largest i in skyline s.t. v_i ≤ v
2 if s ≺ s_i or s ≺ s_{i+1} then
3     return skyline
4 while s ≻ s_i and i > 0 do
5     Delete s_i from skyline
6     i ← i − 1
7 Insert s into skyline
8 return skyline
```

To couple candidate streak generation with skyline operation, Algorithm 6 maintains a dynamic skyline and updates it whenever a new candidate streak is produced. The updating procedure *skyline_update* is in Algorithm 7.

Our focus is not to compare various skyline algorithms. Many existing algorithms can be adopted. What matters is the number of candidate streaks produced by the candidate generation step. This is also verified by our experiments which show that, under various skyline algorithms, the candidate streak generation methods in Section 3.3 perform and compare consistently.

We can use a sorting-based method for finding the skyline points in a two-dimensional space [52]. If the candidate streak generation step does not prune streaks effectively, we cannot hold all candidate streaks in memory. The memory overflow can be addressed by external-memory sorting.

Another approach is to progressively update a dynamic skyline with candidate streaks, based on the nested-loop method in [52]. The outline of this approach is shown in Algorithm 7. We use *skyline* to denote the dynamic skyline. When a new candidate streak s is generated, s is inserted into *skyline* if it is not dominated by any point in *skyline*. The algorithm also checks if some points in *skyline* are dominated by s and eliminates them from *skyline*.

The dominance relationship can be efficiently checked, given that the streaks have only two dimensions– interval length $(r - l)$ and minimum value (v). The key idea is that the lengths of streaks monotonically decrease as their minimal values increase (except that there can be identical points, i.e., streaks with equal lengths and equal minimal values.) Hence the streaks in *skyline* are ordered by v (or by $r - l$). Suppose the candidate streak is $s = \langle [l', r'], v' \rangle$. We find in *skyline* a pivoting streak $s_i = \langle [l_i, r_i], v_i \rangle$ such that i is the largest index with $v_i \leq v'$, i.e., $v_i \leq v' < v_{i+1}$. The following Property 1 says that s must be dominated by s_i or s_{i+1} if it is dominated by any point in *skyline* and Property 2 says that s can only dominate s_i and its immediate neighbors with smaller v values. (For concise presentation, in these properties, we omit the discussion of boundary cases, i.e., $i = 0$ or $i = |skyline|$.) For quickly finding s_i and its neighbors, we use a balanced binary search tree (BST) on v to store *skyline*. (Thus we call it the BST-based skyline method.)

Property 1. A candidate streak $s = \langle [l', r'], v' \rangle$ is dominated by some points in *skyline* if and only if s is dominated by s_i or s_{i+1}, in which $s_i = \langle [l_i, r_i], v_i \rangle$ and i is the largest index such that $v_i \leq v'$, i.e., $v_i \leq v' < v_{i+1}$.

Proof. We first prove that, if there exists $j < i$ such that $s_j = \langle [l_j, r_j], v_j \rangle \succ s$, then $s_i \succ s$. Since i is the largest index such that $v_i \leq v'$, we have $v_j \leq v_i \leq v'$. Given

that $s_j \succ s$, we know $v_j = v_i = v'$ and $r_j - l_j > r' - l'$. From $v_j = v_i$, we know that $r_j - l_j = r_i - l_i$, otherwise they cannot both exist in *skyline*. Therefore $s_i \succ s$.

We then prove that, if there exists $j > i+1$ such that $s_j = \langle [l_j, r_j], v_j \rangle \succ s$, then $s_{i+1} \succ s$. Since the points in *skyline* are ordered by v, $v_{i+1} \leq v_j$ and $r_{i+1} - l_{i+1} \geq r_j - l_j$. We already know that $v' < v_{i+1}$ and $r_j - l_j \geq r' - l'$ (since $s_j \succ s$). Therefore $s_{i+1} \succ s$. ■

Property 2. If $s = \langle [l', r'], v' \rangle$ dominates totally k streaks in *skyline*, then the k streaks are s_i, s_{i-1}, ..., s_{i-k+1}.

Proof. Since the points in *skyline* are ordered by v, we know that $v_i \leq v_j$ and $r_i - l_i \geq r_j - l_j$ if $i < j$. s cannot dominate any s_j such that $j > i$. The reason is that $v' < v_{i+1} \leq v_j$. If s dominates s_i, then $v' \geq v_i$ and $r' - l' \geq r_i - l_i$. Since v_i decreases by i and $r_i - l_i$ increases by i, the k streaks dominated by s must be consecutively ordered. ■

In comparison with the sorting-based method, the above BST-based skyline method saves both memory space and execution time. It avoids memory overflow because the number of streaks in the dynamic skyline in most cases remains small enough to fit in memory. Hence no streak needs to be read from/written to secondary memory. The small size of dynamic skyline in real data is verified by our experiments in Section 3.6. After all, prominent streaks (and skyline points in general) are supposed to be minority, otherwise they cannot stand out to warrant further investigation. Furthermore, even if the dynamic skyline grows large, a method such as the block nested-loop based method in [52] can be applied to fall back on secondary memory. The small size of dynamic skyline also means small number of streak comparisons. Intuitively, given c candidate streaks, a fast comparison-based sorting algorithm (say quicksort) requires $O(c \log c)$ comparisons, while the BST-based method only requires

$O(c \log s)$ comparisons, where s is the maximal size of the dynamic skyline during computation. Experiments in Section 3.6 show that s is typically much smaller than c.

Monitoring Prominent Streaks

A desirable property of a prominent streak discovery algorithm is the capability of monitoring new data entries as the sequence grows continuously and always keeping the prominent streaks up-to-date. The aforementioned algorithms naturally fit into such monitoring scenario, with only minor modification. The details are given in Section 3.4.

3.1.3 Summary of Contributions and Outline

To summarize, our work makes the following contributions:

- We define the problem of prominent streak discovery. The simple concept is useful in many real-world applications. To the best of our knowledge, there has not been study along this line except our prior work [51].
- We propose the solution framework to separate *candidate streak generation* and *skyline operation* during prominent streak discovery. Under this framework, we designed efficient algorithms for candidate streak generation, based on the concept of local prominent streak. Both the non-linear LPS-based method (NLPS) and the linear LPS-based method (LLPS) produce substantially less candidate streaks than the quadratic number of candidates produced by a baseline method. LLPS further guarantees a linear number of candidate streaks.
- We extend the solution framework to discovering general prominent streaks. While the extensions to top-k and multi-sequence prominent streaks are simple, the extension to multi-dimensional prominent streak is non-trivial. These extensions significantly broaden the real-world application scenarios of the work.

- We conduct experiments over multiple real datasets. The results verified the effectiveness of our methods and showed orders of magnitude performance improvement over the baseline method. We also showed some insightful prominent streaks discovered from real data, to highlight the practicality of this work.

The rest of the chapter is organized as follows. In Section 3.2 we review related work. Section 3.3 presents the NLPS and LLPS methods for candidate streak generation. Section 3.4 discusses how to adapt the algorithms to monitor prominent streaks when data sequence continuously grows. Section 3.5 extends the concept of prominent streak and the algorithms for finding general prominent streaks. Experiment setup and results are reported in Section 3.6.

3.2 Related Work

Data mining on sequence and time-series data has been an active area of research, where many techniques are developed for similarity search and subsequence matching in sequence and time-series databases [56, 57, 58, 59], finding sequential patterns [60, 61, 62, 63, 64], classification and clustering of sequence and time-series data [65, 66, 67, 68], biological sequence analysis [69, 70], etc. However, we are not aware of prior work on the prominent streak discovery problem proposed in this chapter.

The skyline of a set of tuples is the subset of tuples that are not dominated by any tuple. A tuple dominates another tuple if it is equally good or better on every attribute and better on at least one attribute. The notion of skyline is useful in several applications, including multi-criteria decision making. Skyline query has been intensively studied over the last decade. Kung et al. [71] first proposed in-memory algorithms to tackle the skyline problem, which they called the "maximal vector problem". Börzsönyi et al. [52] considered the problem in database context

and integrated skyline operator into database system. They also invented a block-nested-loop algorithm (BNL) and extended the divide-and-conquer algorithm (DC) from [71]. Chomicki et al. presented the Sort-Filter-Skyline algorithm (SFS) [72], which improves upon BNL by pre-sorting tuples with a function compatible with the skyline criteria. We apply skyline algorithms over candidate streaks but our methods are orthogonal to specific choices of skyline algorithms.

A dataset may have too many skyline tuples, especially when the dimensionality of the data is high. Various approaches have been proposed to alleviate this problem. For example, Pei et al. [73] and Tao et al. [74] proposed to perform skyline analysis in subspaces instead of the original full space. Several methods were designed to find the representatives among a large number of skyline points [75, 76, 77, 78].

Progressive skyline algorithms optimize the efficiency in returning initial skyline points while producing more results progressively. Various algorithms developed along this line include the bitmap-based algorithm and the index-based algorithm [53], the nearest neighbor search algorithm [54], and the branch-and-bound skyline algorithm (BBS) [55]. Other variants of skyline queries have also been studied, including skyline cube which aims to answer skyline queries over any combination of dimensions [73, 79].

Jiang et al. [80] studied the problem of interval skyline queries on time-series. Given a set of time series and a time interval, they find the time series that are not dominated by others in the interval. A time series dominates another one if its value at every position is at least equal to the corresponding value in the other time series and it is at least larger at one position. The point-by-point equi-length interval comparison is clearly different from our problem.

The plateau of a time series is the time interval in which the vlaues are close to each other (within a given threshold) and are no smaller than the values outside

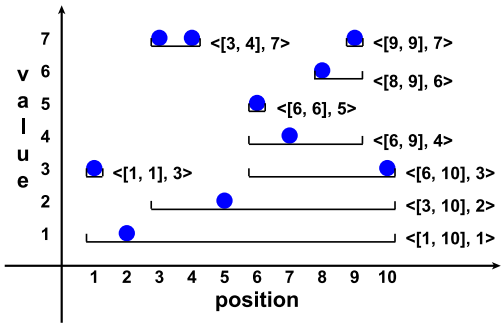


Figure 3.2: Local Prominent Streaks.

the interval [81]. The plateau problem is not concerned about comparing different intervals.

3.3 Discovering Prominent Streaks from Local Prominent Streaks

For an n-element sequence P, the baseline method (Algorithm 6) produces $\frac{n(n+1)}{2}$ candidate streaks. In this section, based on the concept of *local prominent streak* (LPS) we propose the non-linear LPS-based (NLPS) and linear LPS-based (LLPS) methods. Both drastically reduce the number of candidate streaks in practice. LLPS further guarantees only a linear number of candidate streaks.

3.3.1 Local Prominent Streak (LPS)

Definition 11 (Local Prominent Streak)**.** Given a sequence of data values $P = (p_1, \cdots, p_n)$, we say a streak $s = \langle [l, r], v \rangle \in \mathcal{S}_P$ is a *local prominent streak* (LPS) or *locally prominent* if there does not exist any other streak $s' = \langle [l', r'], v' \rangle \in \mathcal{S}_P$ such that $[l', r'] \supset [l, r]$ and $s' \succ s$. (I.e., there does not exist such s' that $[l', r'] \supset [l, r]$ and $v' \geq v$.) The symbol $\supset$ denotes the subsumption check between two intervals, i.e., $[l', r'] \supset [l, r]$ if and only if $l' \leq l \wedge r' > r$ or $l' < l \wedge r' \geq r$. We denote the set of local prominent streaks in sequence P as $\mathcal{LPS}_P$.

Figure 3.2 shows all the local prominent streaks found in our running example. All other streaks are not locally prominent. For example, $\langle [6,8], 4\rangle$ is not locally prominent since it is dominated by $\langle [6,9], 4\rangle$ and $[6,9] \supset [6,8]$. In the following we give several important properties of local prominent streaks.

Property 3. Every prominent streak is also a local prominent streak, i.e., $\mathcal{PS}_P \subseteq \mathcal{LPS}_P$.

Proof. Suppose there is a prominent streak that is not locally prominent, i.e., $\exists s \in \mathcal{PS}_P$ s.t. $s \notin \mathcal{LPS}_P$. By Definition 11, there exists some streak s' such that $[l', r'] \supset [l, r]$ and $s' \succ s$. That is contradictory to Definition 10 which says s is not dominated by any other streak. Therefore a streak cannot be prominent if it is not even locally prominent. ■

The above Property 3 is illustrated by Figure 3.2, as all the prominent streaks in Figure 3.1 also appear in Figure 3.2. However, the reverse of Property 3 does not hold– local prominent streaks are not necessarily prominent streaks. For example, $\langle [8,9], 6\rangle$ is an LPS but is dominated by $\langle [3,4], 7\rangle$ and therefore is not in Figure 3.1.

Lemma 1. Suppose $s = \langle [l, r], v\rangle$ and $s' = \langle [l', r'], v'\rangle$ are two different local prominent streaks in P, i.e., $s, s' \in \mathcal{LPS}_P$, $l \neq l'$ or $r \neq r'$. For any $k \in argmin_{i\in[l,r]} p_i$ and $k' \in argmin_{i\in[l',r']} p_i$, we have $k \neq k'$. I.e., $argmin_{i\in[l,r]} p_i \cap argmin_{i\in[l',r']} p_i = \varnothing$.

Proof. If $[l, r] \cap [l', r'] = \varnothing$, i.e., the two intervals do not overlap, it is obvious that $k \neq k'$. Now consider the case when $[l, r] \cap [l', r'] \neq \varnothing$, i.e., $l \leq l' \leq r$ or $l' \leq l \leq r'$. By definition of $argmin$, $p_k = v = min_{i\in[l,r]} p_i$ and $p_{k'} = v' = min_{i\in[l',r']} p_i$. Suppose there exist such k and k' that $k = k'$. Thus $v = v' = p_k$. By Definition 10, we have $p_i \geq v$ for every $i \in [l, r]$ and every $i \in [l', r']$. Since the two intervals $[l, r]$ and $[l', r']$

overlap, their combined interval corresponds to a new streak $s'' = \langle [l, r] \cup [l', r'], v \rangle$. [1] It is clear $s'' \succ s$ and $s'' \succ s'$. That is a contradiction to the precondition that both s and s' are LPSs. Thus, this lemma holds. ■

Lemma 1 indicates that two different LPSs cannot reach their minimal values at the same position. Therefore each value position in sequence P can correspond to the minimal value of at most one LPS. What immediately follows is that there are at most n LPSs in an n-element sequence. Formally we have the following property.

Property 4. $|\mathcal{LPS}_P| \leq |P|$.

From Property 3 we know that $\mathcal{LPS}_P$ is a sufficient candidate set for $\mathcal{PS}_P$, i.e., we can guarantee to find all prominent streaks if we only consider local prominent streaks. Property 4 further shows how small $\mathcal{LPS}_P$ is and thus how good it is as a candidate set. Specifically, the size of $\mathcal{LPS}_P$ is at most $|P|$, the length of the sequence, in contrast to the all $\frac{|P|(|P|+1)}{2}$ possible streaks considered by the baseline method (Algorithm 6). Thus, $\mathcal{LPS}_P$ helps to prune most streaks from further consideration. In the following sections we present efficient algorithms for computing a superset of $\mathcal{LPS}_P$ and $\mathcal{LPS}_P$ itself exactly.

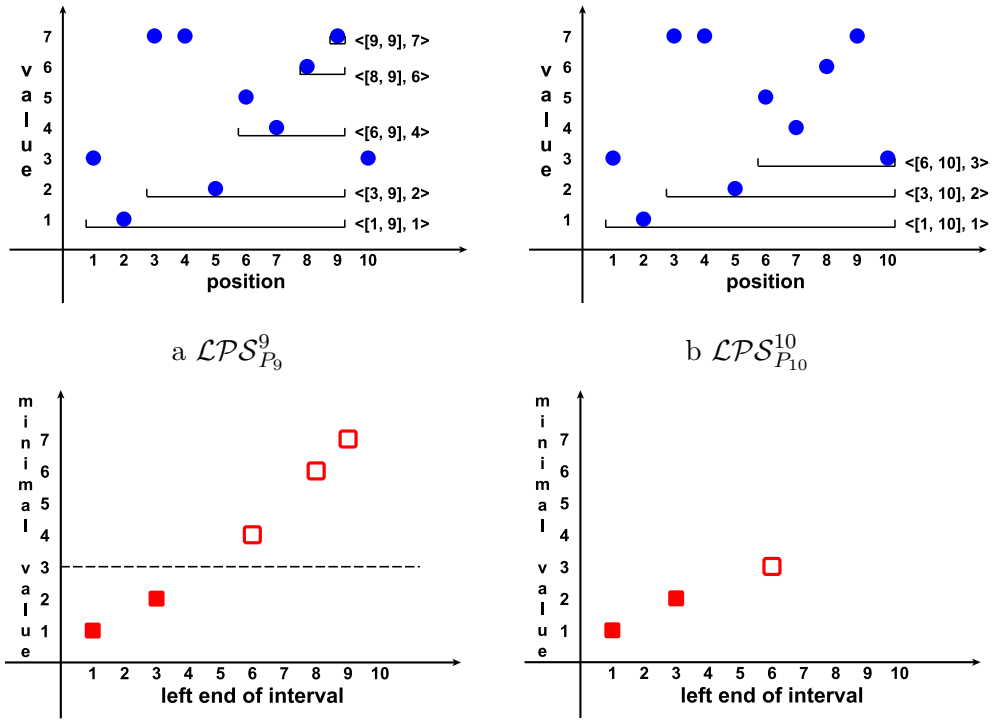
3.3.2 $\mathcal{LPS}_P^k$ and $\mathcal{LPS}_{P_k}^k$

To facilitate our discussion, we first define a new notation, $\mathcal{LPS}_P^k$.

Definition 12. $\mathcal{LPS}_P^k$ is the set of local prominent streaks in P that end at position k, i.e., $\mathcal{LPS}_P^k = \{s | s \in \mathcal{LPS}_P \text{ and } s = \langle [l, k], v \rangle\}$.

There are two key components in the definition of $\mathcal{LPS}_P^k$. The first is the upper script k, which fixes the right end of every interval in the set. It is clear that $\mathcal{LPS}_P^1$, $\mathcal{LPS}_P^2$, ..., $\mathcal{LPS}_P^{|P|}$ is a natural partition of $\mathcal{LPS}_P$. We use this partition scheme in

[1] The two intervals can overlap in four different ways. Thus $[l, r] \cup [l', r'] = [l, r]$ or $[l, r']$ or $[l', r]$ or $[l', r']$.



a $\mathcal{LPS}^9_{P_9}$ b $\mathcal{LPS}^{10}_{P_{10}}$

c l-v plot of $\mathcal{LPS}^9_{P_9}$ d l-v plot of $\mathcal{LPS}^{10}_{P_{10}}$

Figure 3.3: From $\mathcal{LPS}^9_{P_9}$ to $\mathcal{LPS}^{10}_{P_{10}}$.

the design of our algorithms. Specifically, we show how each $\mathcal{LPS}^k_P$ in this partition is calculated in a sequential and progressive way.

The second key component in the definition of $\mathcal{LPS}^k_P$ is the lower script P, which provides the scope for local prominent streaks. By generalizing this component we define $\mathcal{LPS}^k_{P_k}$. We denote the sequence with the first k entries of P as P_k. Then $\mathcal{LPS}_{P_k}$ is the set of local prominent streaks with regard to sequence P_k (instead of P) and $\mathcal{LPS}^k_{P_k}$ are those LPSs in $\mathcal{LPS}_{P_k}$ that end at k. Due to the change of scope, $\mathcal{LPS}^k_{P_k}$ is a superset of $\mathcal{LPS}^k_P$. Formally, we have the following property.

Property 5. $\mathcal{LPS}^k_P \subseteq \mathcal{LPS}^k_{P_k}$.

Proof. Consider any streak $s \in \mathcal{LPS}^k_P$. By Definition 12, $s = \langle [l, k], v \rangle$ and $s \in \mathcal{LPS}_P$. Therefore by Definition 11, there does not exist any $s' = \langle [l', r'], v' \rangle$ in P such that

$s' \succ s$ and $[l', r'] \supset [l, k]$. Since P_k is a prefix of P, i.e., the first k values in P, it follows that there does not exist any such s' in P_k either. Therefore $s \in \mathcal{LPS}^k_{P_k}$. ■

Consider the running example again. Figure 3.3a shows $\mathcal{LPS}^9_{P_9}$, including $\langle[1,9],1\rangle, \langle[3,9],2\rangle, \langle[6,9],4\rangle, \langle[8,9],6\rangle, \langle[9,9],7\rangle$. As shown in Figure 3.2, $\mathcal{LPS}^9_P$ contains $\langle[6,9],4\rangle, \langle[8,9],6\rangle, \langle[9,9],7\rangle$. Streaks $\langle[1,9],1\rangle$ and $\langle[3,9],2\rangle$ do not belong to $\mathcal{LPS}_P$, thus do not belong to $\mathcal{LPS}^9_P$, since they are locally dominated by $\langle[1,10],1\rangle$ and $\langle[3,10],2\rangle$, respectively. By contrast, $\langle[1,9],1\rangle$ and $\langle[3,9],2\rangle$ are part of $\mathcal{LPS}^9_{P_9}$ because they are not locally dominated by any streak of P_9, which only contains the first 9 values of P.

3.3.3 Non-linear LPS Method

By Property 5 and the fact that $\mathcal{LPS}^1_P, \cdots, \mathcal{LPS}^{|P|}_P$ is a partition of $\mathcal{LPS}_P$, we have

$$\mathcal{LPS}_P = \bigcup_{1 \le k \le |P|} \mathcal{LPS}^k_P \subseteq \bigcup_{1 \le k \le |P|} \mathcal{LPS}^k_{P_k} \tag{3.1}$$

Thus, we can use $\bigcup_{1 \le k \le |P|} \mathcal{LPS}^k_{P_k}$ as our candidate set for prominent streaks. Although its size can be greater than that of $\mathcal{LPS}_P$, in practice it does substantially reduce the size of candidate streaks, verified by the experimental results in Section 3.6.

Along this line, Algorithm 8 presents the method to compute candidate streaks. Since the number of candidates may be super-linear to the length of data sequence, we call it the *non-linear LPS method* (NLPS). The algorithm iterates k from 1 to $|P|$, progressively computes $\mathcal{LPS}^k_{P_k}$ from $\mathcal{LPS}^{k-1}_{P_{k-1}}$ when the k-th element p_k is visited, and includes them into candidate streaks. The details of updating from $\mathcal{LPS}^{k-1}_{P_{k-1}}$ to $\mathcal{LPS}^k_{P_k}$ are in Algorithm 9, which is based on the following Lemma 2. For convenience of discussion, we first define the right-end extension of a streak and a streak set.

Algorithm 8: Non-linear LPS Method (NLPS)

Input: Data sequence $P = (p_1, \cdots, p_n)$

Output: Prominent streaks *skyline*

```
1 skyline ← empty
2 for k = 1 to n do
3     Compute LPS^k_{P_k} by Algorithm 9
4     for each streak s in LPS^k_{P_k} do
5         skyline ← skyline_update(skyline, s)
```

Definition 13. If $s = \langle [l, r], v \rangle$ is a streak in an n-element data sequence P and $r < n$, the right-end extension of s is streak $\langle [l, r+1], v' \rangle$, where $v' = \min\{v, p_{r+1}\}$. The extension of a streak set S is the set which consists of extensions of all the streaks in S.

Lemma 2. If $s_1 = \langle [l, k], v_1 \rangle \in \mathcal{LPS}^k_{P_k}$ and $l \neq k$, then the streak $s_2 = \langle [l, k-1], v_2 \rangle \in \mathcal{LPS}^{k-1}_{P_{k-1}}$.

Proof. First, note that $v_2 = \min_{l_1 \leq i \leq k-1} p_i$ and $v_1 = \min\{v_2, p_k\}$. We prove by contradiction. Suppose $s_2 = \langle [l_1, k-1], v_2 \rangle \notin \mathcal{LPS}^{k-1}_{P_{k-1}}$. By Definition 12, $s_2 \notin \mathcal{LPS}_{P_{k-1}}$. Further by Definition 11, there exists $s_3 = \langle [l_3, r_3], v_3 \rangle \in \mathcal{S}_{P_{k-1}}$ such that $[l_3, r_3] \supset [l_1, k-1]$ and $s_3 \succ s_2$. Given any $s = \langle [l, r], v \rangle \in \mathcal{S}_{P_{k-1}}$, we have $r \leq k-1$. Therefore $r_3 = k-1$, $l_3 < l_1$ and $v_3 \geq v_2$. The right-end extension of s_3 is $s_4 = \langle [l_3, k], v_4 \rangle$, where $v_4 = \min\{v_3, p_k\} \geq \min\{v_2, p_k\} = v_1$. Therefore $s_4 \succ s_1$, which contradicts with the pre-condition that $s_1 \in \mathcal{LPS}^k_{P_k}$. The property holds. ∎

Lemma 2 indicates that, except $\langle [k, k], p_k \rangle$, for each streak in $\mathcal{LPS}^k_{P_k}$, its prefix streak is in $\mathcal{LPS}^{k-1}_{P_{k-1}}$. Hence, to produce $\mathcal{LPS}^k_{P_k}$, we only need to consider the right-

Algorithm 9: Progressive Computation of $\mathcal{LPS}^k_{P_k}$

Input: $\mathcal{LPS}^{k-1}_{P_{k-1}}$ and p_k

Output: $\mathcal{LPS}^k_{P_k}$

// When it starts, stack *lps* consists of streaks in $\mathcal{LPS}^{k-1}_{P_{k-1}}$.

```
1  pivot ← null
2  while ! lps.isempty() do
3      if lps.top().v < p_k then
4          break
5      else
6          pivot ← lps.pop()
7  if pivot == null then
8      lps.push(⟨[k, k], p_k⟩)
9  else
10     pivot.v ← p_k
11     lps.push(pivot)
```

// Now, *lps* contains all the streaks in $\mathcal{LPS}^k_{P_k}$.

end extension of $\mathcal{LPS}^{k-1}_{P_{k-1}}$. Beyond that, we only need to consider one extra streak $\langle [k,k], p_k \rangle$ since it may belong to $\mathcal{LPS}^k_{P_k}$ as well.

In order to articulate how to derive $\mathcal{LPS}^k_{P_k}$ from $\mathcal{LPS}^{k-1}_{P_{k-1}}$, we partition $\mathcal{LPS}^{k-1}_{P_{k-1}}$ into two disjoint sets, namely,

$$\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{<} = \{s | s = \langle [l, k-1], v \rangle \in \mathcal{LPS}^{k-1}_{P_{k-1}}, v < p_k\}, \tag{3.2}$$

$$\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\geq} = \{s | s = \langle [l, k-1], v \rangle \in \mathcal{LPS}^{k-1}_{P_{k-1}}, v \geq p_k\}. \tag{3.3}$$

It is clear that $\mathcal{LPS}^{k-1}_{P_{k-1}}$ is the disjoint union of these two sets, i.e., $\mathcal{LPS}^{k-1}_{P_{k-1}} = \mathcal{LPS}^{k-1}_{P_{k-1}}{}^{<} \cup \mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\geq}$, and $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{<} \cap \mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\geq} = \varnothing$. Use the running example

again. For $\mathcal{LPS}^9_{P_9}$ in Figure 3.3a, since $p_{10} = 3$, the two sets are $\mathcal{LPS}^{9\ <}_{P_9} = \{\langle[1,9],1\rangle, \langle[3,9],2\rangle\}$, $\mathcal{LPS}^{9\ \geq}_{P_9} = \{\langle[6,9],4\rangle, \langle[8,9],6\rangle, \langle[9,9],7\rangle\}$.

We consider how to extend streaks in $\mathcal{LPS}^{k-1\ <}_{P_{k-1}}$ and $\mathcal{LPS}^{k-1\ \geq}_{P_{k-1}}$, respectively. For simplicity of presentation, we omit the formal proofs when we make various statements below.

- $\mathcal{LPS}^{k-1\ <}_{P_{k-1}}$: We use $S1$ to denote the right-end extension of $\mathcal{LPS}^{k-1\ <}_{P_{k-1}}$. Since every streak in $\mathcal{LPS}^{k-1\ <}_{P_{k-1}}$ has a minimal value less than p_k, the corresponding extended new streak has the same minimal value. Hence all the new streaks belong to $\mathcal{LPS}^k_{P_k}$. For the running example, corresponding to $\mathcal{LPS}^{9\ <}_{P_9}$, we have $S1 = \{\langle[1,10],1\rangle, \langle[3,10],2\rangle\}$.
- $\mathcal{LPS}^{k-1\ \geq}_{P_{k-1}}$: We use $S2$ to denote the right-end extension of $\mathcal{LPS}^{k-1\ \geq}_{P_{k-1}}$. Since every streak in $\mathcal{LPS}^{k-1\ \geq}_{P_{k-1}}$ has a minimal value greater than or equal to p_k, the minimal value of every streak in $S2$ equals p_k. Hence, the longest streak in $S2$, denoted as $S2^*$, dominates all other streaks in $S2$ and it is the only streak in $S2$ that belongs to $\mathcal{LPS}^k_{P_k}$. In other words, we only need to extend the longest streak in $\mathcal{LPS}^{k-1\ \geq}_{P_{k-1}}$ to form a new candidate streak. Furthermore, since every streak in $S2$ has the same r value (the right end of the interval), i.e., k, $S2^*$ is the streak with the minimal l value (the left end of the interval) in $S2$. Clearly there cannot be another streak in $S2$ with the same length. For the running example, corresponding to $\mathcal{LPS}^{9\ \geq}_{P_9}$, we have $S2 = \{\langle[6,10],3\rangle, \langle[8,10],3\rangle, \langle[9,10],3\rangle\}$. The longest streak in $S2$ is $\langle[6,10],3\rangle$. It is clear that $\langle[6,10],3\rangle$ dominates other streaks in $S2$. Hence it belongs to $\mathcal{LPS}^{10}_{P_{10}}$.
- $\mathcal{LPS}^{k-1\ \geq}_{P_{k-1}} = \varnothing$: If $\mathcal{LPS}^{k-1\ \geq}_{P_{k-1}}$ is empty, a new streak $\langle[k,k],p_k\rangle$ belongs to $\mathcal{LPS}^k_{P_k}$. (Otherwise, it is dominated by $S2^*$.)

The above discussion is captured by the following Property 6.

Property 6. $\mathcal{LPS}^k_{P_k} = S1 \cup \{S2^*\}$ if $S2 \neq \varnothing$ and $\mathcal{LPS}^k_{P_k} = S1 \cup \{\langle [k,k], p_k \rangle\}$ if $S2 = \varnothing$.

We use Figure 3.3 to explain the above procedure of producing $\mathcal{LPS}^k_{P_k}$ from $\mathcal{LPS}^{k-1}_{P_{k-1}}$. Figure 3.3a and 3.3b show $\mathcal{LPS}^9_{P_9}$ and $\mathcal{LPS}^{10}_{P_{10}}$, respectively. Figure 3.3c and 3.3d also show $\mathcal{LPS}^9_{P_9}$ and $\mathcal{LPS}^{10}_{P_{10}}$, by using a different presentation– l-v plot. All the streaks $\langle [l,r], v \rangle$ in $\mathcal{LPS}^{k-1}_{P_{k-1}}$ share the same value of r, which is $k-1$. Therefore we plot the streaks by l (x-axis) and v (y-axis). In Figure 3.3c, the 5 points represent the 5 streaks in $\mathcal{LPS}^9_{P_9}$: $\langle [1,9], 1 \rangle$, $\langle [3,9], 2 \rangle$, $\langle [6,9], 4 \rangle$, $\langle [8,9], 6 \rangle$, $\langle [9,9], 7 \rangle$. The dotted line represents the 10-th data entry $p_{10} = 3$. It bisects $\mathcal{LPS}^9_{P_9}$ into $\mathcal{LPS}^{9\,\geq}_{P_9}$ (3 hollow points above the line) and $\mathcal{LPS}^{9\,<}_{P_9}$ (2 filled points below the line). We produce new candidate streaks $\mathcal{LPS}^{10}_{P_{10}}$ by extending the right ends of streaks in $\mathcal{LPS}^9_{P_9}$ to 10. The streaks extended from $\mathcal{LPS}^{9\,<}_{P_9}$ all belong to $\mathcal{LPS}^{10}_{P_{10}}$. They are the 2 filled points in Figure 3.3d, corresponding to $\langle [1,10], 1 \rangle$ and $\langle [3,10], 2 \rangle$. Among the streaks extended from $\mathcal{LPS}^{9\,\geq}_{P_9}$, only the one with the smallest l (the longest one) belongs to $\mathcal{LPS}^{10}_{P_{10}}$. It is the hollow point in Figure 3.3d, corresponding to $\langle [6,10], 3 \rangle$. Hence $\mathcal{LPS}^{10}_{P_{10}} = \{\langle [1,10], 1 \rangle, \langle [3,10], 2 \rangle, \langle [6,10], 3 \rangle\}$.

The details of the algorithm are shown in Algorithm 9. We use a stack *lps* to maintain $\mathcal{LPS}^k_{P_k}$. Since the streaks $\langle [l,r], v \rangle$ in $\mathcal{LPS}^k_{P_k}$ have the same r value which equals k, we do not need to store r in *lps*. Hence each item in *lps* has two data attributes, v and l. The items in the stack are ordered by v (and l). More specifically, their v and l values both strictly monotonically increase, from the bottom of the stack to the top. The monotonicity on l is obvious since they are different streaks of the same r value. The monotonicity on v thus is also clear because their lengths monotonically decreases due to monotonically increasing l and they must not dominate each other. In fact, Figure 3.3c and 3.3d visualize all items in *lps*, before

and after p_{10} is encountered, respectively. In each figure, the leftmost point denotes the bottom of the stack (with the smallest v), while the rightmost point denotes the top of the stack (with the largest v). After data entries p_1, ..., p_{k-1} are encountered, *lps* contains $\mathcal{LPS}^{k-1}_{P_{k-1}}$. Given data entry p_k, we popped from the stack all the streaks whose v values are greater than or equal to p_k. Among the popped streaks, the leftmost one (with the smallest l and v) is pushed back into the stack, with v value replaced by p_k and r extended from $k-1$ to k. (Again, the r value is not explicitly stored in the stack.) If no streak was popped, then $\langle [k,k], p_k \rangle$ is pushed into the stack. The remaining streaks in the original stack are kept, with their v and l values unchanged and r extended from $k-1$ to k.

Algorithm 8 computes candidate streaks for an n-element sequence P. It invokes Algorithm 9 n times.[2] In each invocation, exactly 1 item is pushed into the stack. Therefore in total there are n insertions and thus at most n deletions. Hence, the amortized time complexity of Algorithm 9 is $O(1)$.

In each iteration of Algorithm 8, we compute $\mathcal{LPS}^k_{P_k}$ and include them into candidate streaks. Thus, for an n-element sequence, the total number of candidate streaks considered is $\sum_{k=1}^{n} |\mathcal{LPS}^k_{P_k}|$. In the worst case, we may have a strictly increasing sequence and the candidate streaks include all possible streaks. This is as bad as the exhaustive baseline method in Algorithm 6. For example, given sequence (10, 20, 30), we have $\mathcal{LPS}^1_{P_1} = \{\langle [1,1], 10\rangle\}$, $\mathcal{LPS}^2_{P_2} = \{\langle [1,2], 10\rangle, \langle [2,2], 20\rangle\}$ and $\mathcal{LPS}^3_{P_3} = \{\langle [1,3], 10\rangle, \langle [2,3], 20\rangle, \langle [3,3], 30\rangle\}$.

[2]With regard to the first data element p_1, $\langle [1,1], p_1\rangle$ is pushed into the stack. It is the only prominent streak and local prominent streak for P_1.

Algorithm 10: Linear LPS Method(LLPS)

Input: Data sequence $P = (p_1, \ldots p_n)$

Output: Prominent streaks *skyline*

```
1 skyline ← empty
2 for k = 1 to n do
3 |   Compute LPS_P^{k-1} and LPS_{P_k}^k by Algorithm 11
4 |   for each streak s in LPS_P^{k-1} do
5 |   |   skyline ← skyline_update(skyline, s)
6 LPS_P^n ← LPS_{P_n}^n
7 for each streak s in LPS_P^n do
8 |   skyline ← skyline_update(skyline, s)
```

3.3.4 Linear LPS Method

Now we present the linear LPS (LLPS) method (Algorithm 10), which guarantees to produce a linear number of candidate streaks even in the worst case. Similar to Algorithm 8, this method iterates through the data sequence and computes $\mathcal{LPS}_{P_k}^k$ from $\mathcal{LPS}_{P_{k-1}}^{k-1}$ when the k-th data entry is encountered, for k from 1 to n. However, different from Algorithm 8, it also computes $\mathcal{LPS}_P^{k-1}$ from $\mathcal{LPS}_{P_{k-1}}^{k-1}$. Computation of both $\mathcal{LPS}_{P_k}^k$ and $\mathcal{LPS}_P^{k-1}$ is done in Algorithm 11, which is a simple extension of Algorithm 9. It is worth noting that, since $P_n = P$, $\mathcal{LPS}_P^n$ and $\mathcal{LPS}_{P_n}^n$ are identical.

To produce $\mathcal{LPS}_P^{k-1}$ from $\mathcal{LPS}_{P_{k-1}}^{k-1}$ given the k-th entry p_k, Algorithm 11 is based on the following Property 7. Its intuition is as follows. Recall that the minimal value of any streak in $\mathcal{LPS}_{P_{k-1}}^{k-1\,\geq}$ (Equation (3.3)) is not smaller than p_k. It follows that if the minimal value of a streak in $\mathcal{LPS}_{P_{k-1}}^{k-1\,\geq}$ is greater than p_k, the streak cannot grow into a longer local prominent streak without changing the minimal value. Hence,

Algorithm 11: Computing $\mathcal{LPS}_P^{k-1}$ and $\mathcal{LPS}_{P_k}^k$

Input: $\mathcal{LPS}_{P_{k-1}}^{k-1}$ and p_k

Output: $\mathcal{LPS}_P^{k-1}$ and $\mathcal{LPS}_{P_k}^k$

// Insert the following line before Line 1 in Algorithm 9.

1 $\mathcal{LPS}_P^{k-1} \leftarrow \varnothing$

// Insert the following two lines after Line 6 in Algorithm 9, in the same **else** branch as Line 6.

2 **if** $pivot.v > p_k$ **then**

3 $\quad \mathcal{LPS}_P^{k-1} \leftarrow \mathcal{LPS}_P^{k-1} \cup \{pivot\}$

the streak itself is a local prominent streak. To summarize, $\mathcal{LPS}_P^{k-1}$ is the same as $\mathcal{LPS}_{P_{k-1}}^{k-1\ \geq}$. The only exception is the longest streak in $\mathcal{LPS}_{P_{k-1}}^{k-1\ \geq}$, i.e., the streak with the smallest l and thus the smallest minimal value v. If its minimal value is equal to p_k, then it does not belong $\mathcal{LPS}_P^{k-1}$, because it can be right-extended and included in $\mathcal{LPS}_P^{k'}$ for some $k' \geq k$.

Lemma 3. For an n-entry sequence P, a streak $s = \langle [l, r], v \rangle$ is a local prominent streak if and only if ($l = 1$ or $v > p_{l-1}$) and ($r = n$ or $v > p_{r+1}$).

Proof. We prove by contradiction. Consider $l > 1$. If $v \leq p_{l-1}$, then s is dominated by $\langle [l-1, r], v \rangle$, which contradicts with s being a local prominent streak. Consider $r < n$. Similarly if $v \leq p_{r+1}$, then s is dominated by $\langle [l, r+1], v \rangle$, which contradicts with s being locally prominent. ■

Property 7. Given an n-entry sequence P, for any position $1 < k \leq n$, $\mathcal{LPS}_P^{k-1} = \{s | s = \langle [l, k-1], v \rangle \in \mathcal{LPS}_{P_{k-1}}^{k-1\ \geq}$ and $v > p_k\}$.

Proof. Proof of the equality from left to right: Suppose streak $s = \langle [l, k-1], v \rangle \in \mathcal{LPS}_P^{k-1}$. By Property 5, $s \in \mathcal{LPS}_{P_{k-1}}^{k-1}$, and by Lemma 3 $v > p_k$. By the concept of $\mathcal{LPS}_{P_{k-1}}^{k-1\ \geq}$ in Equation (3.3), $s \in \mathcal{LPS}_{P_{k-1}}^{k-1\ \geq}$.

Proof of the equality from right to left: Suppose streak $s = \langle [l, k-1], v \rangle$ satisfies $s \in \mathcal{LPS}_{P_{k-1}}^{k-1\ \geq}$ and $v > p_k$. Then s is a local prominent streak in the scope of $\mathcal{LPS}_{P_{k-1}}^{k-1}$, which means, by Lemma 3, $l = 1$ or $v > p_{l-1}$. Since $v > p_k$, by Lemma 3 s is a local prominent streak in P. Therefore $s \in \mathcal{LPS}_P^{k-1}$. ■

Continue the running example. $\mathcal{LPS}_P^9 = \mathcal{LPS}_{P_9}^{9\ \geq} = \{\langle [6, 9], 4 \rangle, \langle [8, 9], 6 \rangle, \langle [9, 9], 7 \rangle\}$. Note that $\mathcal{LPS}_{P_9}^{9\ \geq}$ and $\mathcal{LPS}_P^9$ are identical because the minimal values for the streaks in $\mathcal{LPS}_{P_9}^{9\ \geq}$ are all greater than p_{10}.

Similar to Algorithm 9, Algorithm 11 has an amortized time complexity of $O(1)$. With regard to candidate streaks, LLPS is different in that it only needs to consider the streaks in $\mathcal{LPS}_P^{k-1}$ as candidates. Consequently, LLPS reduces the total number of candidate streaks to $\sum_{k=1}^{n} |\mathcal{LPS}_P^k|$, i.e., $|\mathcal{LPS}_P|$ (Equation (3.1)). By Property 4, $|\mathcal{LPS}_P|$ is n at most, thus LLPS guarantees to produce only a linear number of candidate streaks even in worst case.

3.4 Monitoring Prominent Streaks

One desirable property of a prominent streak discovery algorithm is the capability of monitoring new data entries as the sequence grows continuously and always keeping the prominent streaks up-to-date. For example, a network administrator may check the prominent streaks in the network traffic of a Web server till any particular moment. Formally, given a continuously growing data sequence P (such as a data stream), the k-th data entry that has just come is denoted by p_k and the sequence

so far is denoted by P_k. At this moment, if the user requests $\mathcal{PS}_{P_k}$, the prominent streaks of P_k, our method should efficiently discover them.

With regard to skyline operation, the BST-based method progressively updates the dynamic skyline with new candidate streaks, thus can be applied for monitoring prominent streaks without modification.

With regard to candidate streak generation, all three methods (baseline, NLPS, LLPS) use one-pass sequential scan of the data sequence, therefore they all naturally fit into the monitoring scenario. Specifically, the new data point p_k corresponds to the next iteration of the outer loop in Algorithm 6, 8, and 10. The baseline method exhaustively lists all streaks ending at p_k and updates the skyline with these streaks. The NLPS method updates $\mathcal{LPS}^{k-1}_{P_{k-1}}$ to $\mathcal{LPS}^{k}_{P_k}$, and updates the skyline with the streaks in $\mathcal{LPS}^{k}_{P_k}$.

The adaptation of LLPS is a bit more complex, as shown in Algorithm 12. This algorithm records the last position when the user requested the prominent streaks. When p_k arrives, $\mathcal{LPS}^{k-1}_{P}$ and $\mathcal{LPS}^{k}_{P_k}$ are dynamically computed by Algorithms 11. The skyline is updated with the candidate streaks in $\mathcal{LPS}^{k-1}_{P}$, only if $\mathcal{PS}_{P_{k-1}}$ was not requested by the user when p_{k-1} was visited. Note that if $\mathcal{PS}_{P_{k-1}}$ was requested, the skyline has already been updated with the streaks in $\mathcal{LPS}^{k-1}_{P_{k-1}}$. Since $\mathcal{LPS}^{k-1}_{P} \subseteq \mathcal{LPS}^{k-1}_{P_{k-1}}$, we do not need to update the skyline with $\mathcal{LPS}^{k-1}_{P}$ again. Finally, if the user requests $\mathcal{PS}_{P_k}$, the skyline has to be updated with $\mathcal{LPS}^{k}_{P_k}$ since all the local prominent streaks (with regard to P_k) ending at p_k must be considered. In Section 3.6 we will show the significant superiority of this adaptation of LLPS over other methods.

Note that this algorithm degrades to NLPS (Algorithm 8) if the user requests the prominent streaks at every data entry. On the other hand, if the prominent

Algorithm 12: Continuous Monitoring of Prominent Streaks

Input: The new data entry p_k

```
1 Compute LPS_P^{k-1} and LPS_{P_k}^k by Algorithms 11
2 if last_requested_position < k − 1 then
3     for each streak s in LPS_P^{k-1} do
4         skyline ← sklyine_update(skyline, s)
5 if PS_{P_k} is requested then
6     for each streak s in LPS_{P_k}^k do
7         skyline ← sklyine_update(skyline, s)
8     last_requested_position ← k
9     // Now, skyline contains all prominent streaks in PS_{P_k}
```

streaks are only requested at p_n, i.e., the last entry in the sequence, it becomes the same as LLPS (Algorithm 10).

3.5 Discovering General Prominent Streaks

In this section, we extend the concept of prominent streak and the algorithms introduced in previous sections to general cases. Specifically, we investigate how to discover top-k, multi-sequence, and multi-dimensional prominent streaks.

3.5.1 Top-k Prominent Streaks

Definition 14 (Top-k Prominent Streak)**.** With regard to a sequence $P = (p_1, ...p_n)$ and its local prominent streaks $\mathcal{LPS}_P$, a streak $s \in \mathcal{LPS}_P$ is a *top-k prominent streak* if it is not dominated by k or more streaks in $\mathcal{LPS}_P$, i.e., $\big|\{s' | s' \in \mathcal{LPS}_P \text{ and } s' \succ s\}\big| < k$. The set of all top-$k$ prominent streaks in P is denoted by $\mathcal{KPS}_P$. Note that there can be more than k top-k prominent streaks.

Top-k prominent streaks are those local prominent streaks dominated by less than k other local prominent streaks, by Definition 14. This definition has two implications. First, a top-k prominent streak must be locally prominent. For instance, a streak does not qualify even if it is only dominated by 1 subsuming streak and $k > 1$. Second, a streak can qualify even if it is dominated by k or more other streaks, as long as less than k of those dominating streaks are local prominent streaks.

Consider a sequence $P = (20, 30, 25, 30, 5, 5, 15, 10, 15, 5)$, corresponding to the points made by a basketball player in all his games. The streak $\langle[3, 4], 25\rangle$, though only dominated by $\langle[2, 4], 25\rangle$, is a sub-streak of the latter, and hence is not a top-2 prominent streak. The intuitive explanation is that, $\langle[3, 4], 25\rangle$ is within the interval of $\langle[2, 4], 25\rangle$, therefore we do not consider it important. On the other hand, the streak $\langle[7, 9], 10\rangle$ is a top-2 prominent streak. Although it is dominated by 3 streaks $\langle[1, 4], 20\rangle$, $\langle[1, 3], 20\rangle$, and $\langle[2, 4], 25\rangle$, the dominating streaks are all from the same period and only 1 of the 3 is a local prominent streak.

The candidate streak generation methods discussed in previous sections are applicable in discovering top-k prominent streaks. We only need several small changes on skyline operation. For LLPS, since the candidates produced are guaranteed to be local prominent streaks only, we simply need to maintain a counter for each current skyline point in the dynamic skyline. The counter of a point records the number of its dominators in the skyline. When a candidate is compared against current skyline points, it is inserted into the skyline if it has less than k dominators. A current skyline point is removed if its counter reaches k. With regard to the baseline method and NLPS, they may produce candidates that are not local prominent streaks. A candidate must be pruned if another candidate streak dominates it and subsumes it. (Note that they both produce candidates with the same right-end of interval at the

same time. Therefore a candidate cannot be locally dominated by existing points in the current skyline.)

3.5.2 Multi-sequence Prominent Streaks

Definition 15 (Multi-sequence Prominent Streak)**.** Given multiple sequences $\mathcal{P} = \{P^1, ..., P^m\}$ and their corresponding sets of streaks $\mathcal{S}_{P^1}$, ..., $\mathcal{S}_{P^m}$, a streak $s \in \mathcal{S}_{P^i}$ is a *multi-sequence prominent streak* in $\mathcal{P}$ if there does not exist a streak in any sequence that dominates s. More formally, $\nexists s', j$ s.t. $s' \in \mathcal{S}_{P^j}$, and $s' \succ s$. The set of all multi-sequence prominent streaks with regard to $\mathcal{P}$ is $\mathcal{PS}_{\mathcal{P}}$.

As an example, consider 3 sequences corresponding to the points made by 3 basketball players in all their games—$P_1 = (20, 30, 25, 30, 5, 5, 15, 10, 15, 5)$, $P_2 = (10, 5, 30, 35, 21, 25, 5, 15, 5, 25)$, and $P_3 = (5, 10, 15, 5, 25, 10, 20, 5, 15, 10)$. The streak $\langle [1, 4], 20 \rangle$ of P_1 is a prominent streak within P_1 itself, but is dominated by $\langle [3, 6], 21 \rangle$ in P_2. Hence it is not a multi-sequence prominent streak.

The extension from single-sequence algorithms (baseline, NLPS, LLPS) to multi-sequence algorithms is simple. We process individual sequences separately by the single-sequence algorithms and use a common dynamic skyline to maintain their prominent streaks. That is, when a local prominent streak within a sequence P_i is identified, it is compared with current streaks in the dynamic skyline, which contains prominent streaks from all sequences.

3.5.3 Multi-dimensional Prominent Streaks

Definition 16 (Multi-dimensional Prominent Streak)**.** In an n-entry d-dimensional sequence $P = (\vec{p}_1, \cdots, \vec{p}_n)$, a point $\vec{p}_i$ is a d-dimensional vector of data values. A streak s in P is an interval-vector pair $\langle [l, r], \vec{v} \rangle$, where

$$\vec{v} = (\min_{l \le i \le r} \vec{p}_i[1], \cdots, \min_{l \le i \le r} \vec{p}_i[d]), \tag{3.4}$$

$\vec{p}_i[j]$ is the j-th dimension of $\vec{p}_i$, and $1 \le l \le r \le n$.

A d-dimensional vector $\vec{v} = (\vec{v}[1], \cdots, \vec{v}[d])$ dominates another vector $\vec{v'} = (\vec{v'}[1], \cdots, \vec{v'}[d])$, denoted by $\vec{v} \succ \vec{v'}$, if and only if $\vec{v}(1) \ge \vec{v'}[1], \cdots, \vec{v}[d] \ge \vec{v'}[d]$ and $\exists j$ such that $\vec{v}[j] > \vec{v'}[j]$. Moreover, we use $\vec{v} \succeq \vec{v'}$ to denote the case when $\vec{v}$ dominates or equals $\vec{v'}$.

A streak $s = \langle [l, r], \vec{v} \rangle$ dominates another streak $s' = \langle [l', r'], \vec{v'} \rangle$, denoted by $s \succ s'$, if and only if $r - l \ge r' - l'$ and $\vec{v} \succ \vec{v'}$, or $r - l > r' - l'$ and $\vec{v} \succeq \vec{v'}$.

The set of all possible streaks is denoted by $\mathcal{S}_P$. A streak $s \in \mathcal{S}_P$ is a *prominent streak* if it is not dominated by any streak in $\mathcal{S}_P$, i.e., $\nexists s'$ s.t. $s' \in \mathcal{S}_P$ and $s' \succ s$. The set of all multi-dimensional prominent streaks in P is denoted by $\mathcal{PS}_P$.

For a running example in this section, consider a two-dimensional sequence $P =$ ((10, 10),(40, 20),(40, 30),(30, 40),(50, 30),(20, 30)). By the above definition, there are 8 prominent streaks in P– $\langle [1, 6], (10, 10) \rangle$, $\langle [2, 3], (40, 20) \rangle$, $\langle [2, 5], (30, 20) \rangle$, $\langle [2, 6], (20, 20) \rangle$, $\langle [3, 5], (30, 30) \rangle$, $\langle [3, 6], (20, 30) \rangle$, $\langle [4, 4], (30, 40) \rangle$, $\langle [5, 5], (50, 30) \rangle$. Other streaks are not prominent. For instance, $\langle [2, 4], (30, 20) \rangle$ is dominated by $\langle [3, 5], (30, 30) \rangle$.

In finding prominent streaks from a d-dimensional sequence, skyline operations perform dominance relationship test on $d + 1$ dimensions—d dimensions for data values and one special dimension for streak length. We maintain a KD-tree [82, 83] on current skyline points. Given a candidate streak, we use a range query on the KD-tree to efficiently find its dominating points in the current skyline and another ranger

Algorithm 13: Update Dynamic Skyline for Multi-Dimensional Sequences (*skyline_update*)

Input: Dynamic skyline *skyline*, new candidate streak $s = \langle [l, r], \vec{v} \rangle$

Output: Updated dynamic skyline *skyline*

```
1 dominating ← Find streaks in skyline that dominate s, by a range query
    on the KD-tree over skyline
2 if dominating ≠ ∅ then
3     return skyline
4 dominated ← Find streaks in skyline that are dominated by s, by another
    range query on the KD-tree
5 Remove dominated from skyline
6 Insert s into skyline
7 return skyline
```

query to find its dominated points in the current skyline. Specifically, Algorithm 7 is replaced by Algorithm 13 for multi-dimensional sequences. We do not further discuss how to answer range queries by multi-dimensional index structures such as KD-tree since it is well studied.

With regard to candidate streak generation, the brute-force baseline method does not require change, except that *min_value* and its calculation in Algorithm 6 are replaced according to the definition of vector $\vec{v}$ in Equation (3.4). Our focus in the rest of this section is to extend the concept of local prominent streak and its properties, in order to adapt NLPS and LLPS for multi-dimensional data sequence. Note that Property 3, Property 5 and Lemma 2 still hold, and can be proven in the same way as for single-dimensional sequence. We thus will use the result directly

without tediously showing the proof. With the adaptation of NLPS and LLPS for multi-dimensional sequences, the continuous monitoring approach in Algorithm 12 works in the same way.

Definition 17. For a multi-dimensional sequence P, a streak $s = \langle [l, r], \vec{v} \rangle \in \mathcal{S}_P$ is a *local prominent streak* (LPS) if and only if there does not exist any other streak $s' = \langle [l', r'], \vec{v'} \rangle \in \mathcal{S}_P$, s.t. $[l', r'] \supset [l, r]$ and $s' \succ s$. (I.e., there does not exist such s' that $[l', r'] \supset [l, r]$ and $\vec{v'} \succeq \vec{v}$.) We use $\mathcal{LPS}_P$ to denote the set of all local prominent streaks in P.

For a multi-dimensional sequence, Property 3 still holds. Hence, every prominent streak in a multi-dimensional sequence P is also a local prominent streak, i.e., $\mathcal{PS}_P \subseteq \mathcal{LPS}_P$, and thus we can still find $\mathcal{LPS}_P$ and use it as the set of candidate streaks. Computing local prominent streaks in a multi-dimensional sequence is quite similar to that in a single-dimensional sequence. The concepts of $\mathcal{LPS}_P^k$ and $\mathcal{LPS}_{P_k}^k$ remain the same, except that the $\vec{v}$ in each streak $\langle [l, r], \vec{v} \rangle$ is a multi-dimensional vector instead of a single numeric value. Property 5 also holds. Therefore, the essential ideas of NLPS and LLPS algorithms remain unchanged. NLPS iterates k from 1 to $|P|$, progressively computes $\mathcal{LPS}_{P_k}^k$ from $\mathcal{LPS}_{P_{k-1}}^{k-1}$ when the k-th element $\vec{p}_k$ is visited, and includes $\mathcal{LPS}_{P_k}^k$ into candidate streaks. LLPS does not immediately include all of $\mathcal{LPS}_{P_k}^k$ into candidate streaks. Instead, it waits till seeing $\vec{p}_{k+1}$, then computes $\mathcal{LPS}_P^k$ (in addition to $\mathcal{LPS}_{P_{k+1}}^{k+1}$) from $\mathcal{LPS}_{P_k}^k$, and only includes $\mathcal{LPS}_P^k$ into candidate streaks. Hence, LLPS only considers local prominent streaks ($\mathcal{LPS}_P = \bigcup_{k=1}^{n} \mathcal{LPS}_P^k$) as candidates, while NLPS needs to consider more candidates ($\bigcup_{k=1}^{n} \mathcal{LPS}_{P_k}^k$), since $\mathcal{LPS}_P^k$ is subsumed by $\mathcal{LPS}_{P_k}^k$ according to Property 5.

3.5.3.1 Key Ideas

Our following discussion focuses on how to compute $\mathcal{LPS}^{k}_{P_k}$ and $\mathcal{LPS}^{k-1}_{P}$ from $\mathcal{LPS}^{k-1}_{P_{k-1}}$, when the k-th element $\vec{p}_k$ arrives. To facilitate the discussion, we partition $\mathcal{LPS}^{k-1}_{P_{k-1}}$ into two disjoint sets $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\preceq}$ and $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq}$, as shown below, which are similar to $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{<}$ and $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\geq}$ in Equations (3.2) and (3.3). $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\preceq}$ is the set of streaks, for which the value at any dimension of the vector $\vec{v}$ is not greater than the corresponding value in $\vec{p}_k$. $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq}$ is the set of streaks, for which $\vec{v}$ is greater than $\vec{p}_k$ on at least one dimension.

$$\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\preceq} = \{s | s = \langle [l, k-1], \vec{v} \rangle \in \mathcal{LPS}^{k-1}_{P_{k-1}}, \vec{v} \preceq \vec{p}_k\}, \tag{3.5}$$

$$\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq} = \{s | s = \langle [l, k-1], \vec{v} \rangle \in \mathcal{LPS}^{k-1}_{P_{k-1}}, \exists j \in [1, d] \ s.t. \ \ \vec{v}[j] > \vec{p}_k[j]\}. \tag{3.6}$$

For the running example, $\mathcal{LPS}^{5}_{P_5}$ is divided into $\mathcal{LPS}^{5}_{P_5}{}^{\preceq}$=$\{s_1$=$\langle [1,5], (10,10) \rangle\}$ and $\mathcal{LPS}^{5}_{P_5}{}^{\not\preceq}$=$\{s_2$=$\langle [2,5], (30,20) \rangle$, s_3=$\langle [3,5], (30,30) \rangle$, s_4=$\langle [5,5], (50,30) \rangle\}$.

• *Compute* $\mathcal{LPS}^{k-1}_{P}$ *from* $\mathcal{LPS}^{k-1}_{P_{k-1}}$:

We can prove that $\mathcal{LPS}^{k-1}_{P}$ is equivalent to $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq}$, given by the following property.

Property 8. $\mathcal{LPS}^{k-1}_{P} = \mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq}$.

Proof. Since Property 5 still holds, $\mathcal{LPS}^{k-1}_{P} \subseteq \mathcal{LPS}^{k-1}_{P_{k-1}}$. Furthermore, $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\preceq}$ and $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq}$ disjointly partition $\mathcal{LPS}^{k-1}_{P_{k-1}}$, i.e., $\mathcal{LPS}^{k-1}_{P_{k-1}} = \mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\preceq} \cup \mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq}$ and $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\preceq} \cap \mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq} = \varnothing$. Therefore we only need to prove that (1) none of the streaks in $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\preceq}$ is in $\mathcal{LPS}^{k-1}_{P}$ and (2) all streaks in $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq}$ are in $\mathcal{LPS}^{k-1}_{P}$.

(1) $\forall s \in \mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\preceq}$, $s \notin \mathcal{LPS}^{k-1}_{P}$. Suppose $s = \langle [l, k-1], \vec{v} \rangle$. Its right-end extension is $s' = \langle [l, k], \vec{v'} \rangle$, where $\vec{v'}[j] = min(\vec{v}[j], \vec{p}_k[j])$ for $j \in [1, d]$. Since $\vec{v} \preceq \vec{p}_k$

(by Equation (3.5)), it follows that $\vec{v'} = \vec{v}$ and thus $s' \succ s$. Hence, s cannot be a local prominent streak in P.

(2) $\forall s \in \mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq}$, $s \in \mathcal{LPS}^{k-1}_{P}$. We prove this by contradiction. Suppose $s = \langle [l, k-1], \vec{v} \rangle$. Assume $s \notin \mathcal{LPS}^{k-1}_{P}$, i.e., there exists $s' \succ s$ such that $s' = \langle [l', r'], \vec{v'} \rangle$, $[l', r'] \supset [l, k-1]$ and $\vec{v'} \succeq \vec{v}$. By Equation (3.6), $\exists j \in [1, d]$ such that $\vec{v}[j] > \vec{p}_k[j]$. Therefore $r' = k-1$, otherwise $r' = k$ and $\vec{v'}[j] <= \vec{p}_k[j] < \vec{v}[j]$, which contradicts with $\vec{v'} \succeq \vec{v}$. From $[l', r'] \supset [l, k-1]$ and $r' = k-1$, we get $l' < l$ which, along with $s' \succ s$, contradicts with $s \in \mathcal{LPS}^{k-1}_{P_{k-1}}$. The contradictions prove that $s \in \mathcal{LPS}^{k-1}_{P}$. ■

• *Compute* $\mathcal{LPS}^{k}_{P_k}$ *from* $\mathcal{LPS}^{k-1}_{P_{k-1}}$:

We note that Lemma 2 still holds under multi-dimensional sequence, i.e., except $\langle [k, k], \vec{p}_k \rangle$, for each streak in $\mathcal{LPS}^{k}_{P_k}$, its prefix streak is in $\mathcal{LPS}^{k-1}_{P_{k-1}}$. Hence, to produce $\mathcal{LPS}^{k}_{P_k}$, we only need to consider the right-end extension of $\mathcal{LPS}^{k-1}_{P_{k-1}}$ and one extra streak $\langle [k, k], \vec{p}_k \rangle$ which may belong to $\mathcal{LPS}^{k}_{P_k}$ as well. Again, we consider the two disjoint partitions of $\mathcal{LPS}^{k-1}_{P_{k-1}}$, $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\preceq}$ and $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq}$, respectively.

(1) The right-end extensions of all streaks in $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\preceq}$ belong to $\mathcal{LPS}^{k}_{P_k}$, by the property below.

Property 9. $\forall s \in \mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\preceq}$, its right-end extension $s' \in \mathcal{LPS}^{k}_{P_k}$.

Proof. We prove by contradiction. Suppose $s = \langle [l, k-1], \vec{v} \rangle$. Its right-end extension is $s' = \langle [l, k], \vec{v'} \rangle$, where $\vec{v'}[j] = min(\vec{v}[j], \vec{p}_k[j])$ for $j \in [1, d]$. Since $s \in \mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\preceq}$, $\vec{v} \preceq \vec{p}_k$. Therefore $\vec{v'} = \vec{v}$. If $s' \notin \mathcal{LPS}^{k}_{P_k}$, then there exists $s'' = \langle [l'', k], \vec{v''} \rangle$ such that $s'' \succ s'$, i.e., $l'' < l$, and $\vec{v''} \succeq \vec{v'}$. Since s'' and s' have the same right end of interval and $l'' < l$, $\vec{v''} \preceq \vec{v'}$. Therefore $\vec{v''} = \vec{v'} = \vec{v}$. Consider $s''' = \langle [l'', k-1], \vec{v'''} \rangle$, i.e., s'' is the right-end extension of s'''. $\vec{v'''} \succeq \vec{v''}$ by definition of right-end extension. Therefore $\vec{v'''} \succeq \vec{v}$ and thus $s''' \succ s$ (since $l'' < l$). This contradicts with $s \in \mathcal{LPS}^{k-1}_{P_{k-1}}$. ■

(2) Given a streak in $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq}$, its right-end extension does not always belong to $\mathcal{LPS}^{k}_{P_k}$.

For a single-dimensional sequence, $\mathcal{LPS}^{k-1}_{P_{k-1}}$ was similarly partitioned into $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{<}$ and $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\geq}$. Among the streaks in $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\geq}$, the right-end extension of the longest streak belongs to $\mathcal{LPS}^{k}_{P_k}$. If $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\geq}$ is empty, then $\langle [k,k], p_k \rangle$ belongs to $\mathcal{LPS}^{k}_{P_k}$.

For a multi-dimensional sequence, multiple but not necessarily all streaks in $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq}$ can be right-extended to streaks in $\mathcal{LPS}^{k}_{P_k}$. This can be simply proven by using the running example. Recall that $\mathcal{LPS}^{5}_{P_5}{}^{\preceq} = \{s_1 = \langle [1,5], (10,10) \rangle\}$ and $\mathcal{LPS}^{5}_{P_5}{}^{\not\preceq} = \{s_2 = \langle [2,5], (30,20) \rangle,\ s_3 = \langle [3,5], (30,30) \rangle,\ s_4 = \langle [5,5], (50,30) \rangle\}$. Since $\vec{p_6} = (20,30)$, the right-end extensions of s_2, s_3 and s_4 are $s'_2 = \langle [2,6], (20,20) \rangle$, $s'_3 = \langle [3,6], (20,30) \rangle$ and $s'_4 = \langle [5,6], (20,30) \rangle$, respectively. It is clear $s'_2, s'_3 \in \mathcal{LPS}^{6}_{P_6}$ and $s'_4 \notin \mathcal{LPS}^{6}_{P_6}$ since $s'_3 \succ s'_4$.

3.5.3.2 Efficient Computation

Based on the discussion in Section 3.5.3.1, in computing $\mathcal{LPS}^{k}_{P_k}$ and $\mathcal{LPS}^{k-1}_{P}$ from $\mathcal{LPS}^{k-1}_{P_{k-1}}$, the key is to partition $\mathcal{LPS}^{k-1}_{P_{k-1}}$ into $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\preceq}$ (which equals $\mathcal{LPS}^{k-1}_{P}$) and $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq}$. The right-end extensions of all streaks in $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\preceq}$ belong to $\mathcal{LPS}^{k}_{P_k}$, and all remaining streaks in $\mathcal{LPS}^{k}_{P_k}$ are formed by right-end extensions of streaks in $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq}$. Below we discuss an efficient method of partitioning $\mathcal{LPS}^{k-1}_{P_{k-1}}$ and identifying streaks in $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq}$ that should be extended to streaks in $\mathcal{LPS}^{k}_{P_k}$.

• *Partition* $\mathcal{LPS}^{k-1}_{P_{k-1}}$ *into* $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\preceq}$ *and* $\mathcal{LPS}^{k-1}_{P_{k-1}}{}^{\not\preceq}$:

Suppose there are m streaks in $\mathcal{LPS}^{k-1}_{P_{k-1}}$, which are $s_1 = \langle [l_1, k-1], \vec{v}_1 \rangle$, ..., $s_m = \langle [l_m, k-1], \vec{v}_m \rangle$, where $l_1 < \cdots < l_m$. We can prove that there exists t such that

$\mathcal{LPS}^{k-1\preceq}_{P_{k-1}} = \{s_1, \ldots, s_t\}$ and $\mathcal{LPS}^{k-1\npreceq}_{P_{k-1}} = \{s_{t+1}, \ldots, s_m\}$. (Two special cases are $\mathcal{LPS}^{k-1\preceq}_{P_{k-1}} = \varnothing$ (i.e., $t = 0$) and $\mathcal{LPS}^{k-1\npreceq}_{P_{k-1}} = \varnothing$ (i.e., $t = m$).) The proof is sketched as follows. Since $l_1 < \cdots < l_m$, for any dimension j, the value $\vec{v}_i[j]$ monotonically increases by i (not necessarily strictly increasing), i.e., $\vec{v}_1[j] \leq \vec{v}_2[j] \leq \cdots \leq \vec{v}_m[j]$. It follows that $\vec{v}_1 \prec \vec{v}_2 \prec \cdots \prec \vec{v}_m$. (Note that $\vec{v}_i \neq \vec{v}_{i+1}$ for any i, otherwise s_i would dominate s_{i+1}, which contradicts with that they all are local prominent streaks in P_{k-1}.) Given $s_{i_1} \in \mathcal{LPS}^{k-1\preceq}_{P_{k-1}}$ and $s_{i_2} \in \mathcal{LPS}^{k-1\npreceq}_{P_{k-1}}$, it must be that $i_1 < i_2$, otherwise $i_1 > i_2$, $\vec{v}_{i_1} \succ \vec{v}_{i_2}$ and thus $\forall j, \vec{v}_{i_1}[j] \geq \vec{v}_{i_2}[j]$, which contradicts with Equations (3.5) and (3.6).

• *Identify streaks in $\mathcal{LPS}^{k-1\npreceq}_{P_{k-1}}$ that should be extended to streaks in $\mathcal{LPS}^{k}_{P_k}$:*

To find all those right-end extensions of streaks in $\mathcal{LPS}^{k-1\npreceq}_{P_{k-1}}$ that belong to $\mathcal{LPS}^{k}_{P_k}$, consider the aforementioned partitioning of $\mathcal{LPS}^{k-1}_{P_{k-1}}$ into $\mathcal{LPS}^{k-1\preceq}_{P_{k-1}} = \{s_1, \ldots, s_t\}$ and $\mathcal{LPS}^{k-1\npreceq}_{P_{k-1}} = \{s_{t+1}, \ldots, s_m\}$, where the m streaks $s_m, \ldots, s_1$ are decreasingly ordered by the left ends of their intervals. For each $s_i = \langle [l_i, k-1], \vec{v}_i \rangle \in \mathcal{LPS}^{k-1\npreceq}_{P_{k-1}}$, its right-end extension is $s'_i = \langle [l_i, k], \vec{v'}_i \rangle$. The following important property tells us that if $s'_i \not\prec s'_{i-1}$, then s'_i belongs to $\mathcal{LPS}^{k}_{P_k}$.

Property 10. For each streak $s_i = \langle [l_i, k-1], \vec{v}_i \rangle \in \mathcal{LPS}^{k-1\npreceq}_{P_{k-1}}$, its right-end extension is $s'_i = \langle [l_i, k], \vec{v'}_i \rangle$. $s'_i \in \mathcal{LPS}^{k}_{P_k}$ if and only if $s'_i \not\prec s'_{i-1}$.

Proof. It is apparent that if $s'_i \prec s'_{i-1}$ then $s'_i \notin \mathcal{LPS}^{k}_{P_k}$. Thus our focus is to prove $s'_i \in \mathcal{LPS}^{k}_{P_k}$ if $s'_i \not\prec s'_{i-1}$, by contradiction. Assume $s'_i \not\prec s'_{i-1}$ but $s'_i \notin \mathcal{LPS}^{k}_{P_k}$. Hence, $\exists j < i-1$ and $s'_j \succ s'_i$ (and thus $\vec{v'}_j \succeq \vec{v'}_i$). Since $l_j < l_i$, $\vec{v'}_j \preceq \ldots \preceq \vec{v'}_{i-1} \preceq \vec{v'}_i$. Therefore $\vec{v'}_j = \ldots = \vec{v'}_{i-1} = \vec{v'}_i$. Hence $s'_{i-1} \succ s'_i$ which contradicts with $s'_i \not\prec s'_{i-1}$. ∎

Based on the properties discussed in Section 3.5.3.1 and 3.5.3.2 so far, we design an efficient method to compute $\mathcal{LPS}^{k}_{P_k}$ and $\mathcal{LPS}^{k-1}_{P}$ from $\mathcal{LPS}^{k-1}_{P_{k-1}}$. The current sky-

line points (prominent streaks) after the $(k-1)$-th element is encountered are stored in the aforementioned KD-tree index structure. The streaks in $\mathcal{LPS}^{k-1}_{P_{k-1}}$, $s_m, \ldots, s_1$, are stored in memory by the decreasing order of the left ends of their intervals. Since they have the same right ends of intervals, only the left ends and the corresponding vectors are stored. When the k-th element $\vec{p}_k$ arrives, this method considers the streaks s_i and their right-end extensions s'_i, starting from $i = m + 1$ and iteratively decreasing i by 1. (For $i = m + 1$, the special streak in consideration is $s'_{m+1} = \langle [k, k], \vec{p}_k \rangle$.) According to Property 10, the method only requires comparing s'_i with its predecessor s'_{i-1}. If $s'_i \prec s'_{i-1}$, then s_i is removed from the memory. Otherwise s'_i belongs to $\mathcal{LPS}^k_{P_k}$ and thus s_i is updated to s'_i in memory. More specifically, the vector $\vec{v}_i$ of s_i needs to be updated to $\vec{v'}_i$, by $\vec{v'}_i[j] = min(\vec{v}_i[j], \vec{p}_k[j])$ for $j \in [1, d]$. The method goes on until $i = t$ such that $\vec{v}_t \preceq \vec{p}_k$. At that moment, the method will take the following actions.

- The streaks scanned so far $(s_m, \ldots, s_{t+1})$ form $\mathcal{LPS}^{k-1\ \not\preceq}_{P_{k-1}}$ which is equivalent to $\mathcal{LPS}^{k-1}_P$. All remaining streaks in $\mathcal{LPS}^{k-1}_{P_{k-1}}$ $(s_t, \ldots, s_1)$ form $\mathcal{LPS}^{k-1\ \preceq}_{P_{k-1}}$.
- The streaks in $\mathcal{LPS}^{k-1}_P$ are candidate prominent streaks. They are compared with current skyline points by the aforementioned range queries over the KD-tree on the skyline points. Non-dominated candidates are inserted into the KD-tree.
- For all remaining streaks in the memory (i.e., $\mathcal{LPS}^{k-1\ \preceq}_{P_{k-1}}$), their right-end extensions belong to $\mathcal{LPS}^k_{P_k}$. Since their vectors are all dominated by or equivalent to $\vec{p}_k$, their corresponding vectors do not need to be updated. At this moment, all streaks of $\mathcal{LPS}^k_{P_k}$ are stored in memory by the decreasing order of the left-ends of their intervals.

More concretely, Algorithms 8 and 10 remain unchanged, and Algorithms 9 and 11 are replaced by Algorithms 14 and 15, respectively.

Algorithm 14: Progressive Computation of $\mathcal{LPS}^k_{P_k}$ on Multi-Dimensional Sequences

Input: $\mathcal{LPS}^{k-1}_{P_{k-1}}$ and $\vec{p}_k$

Output: $\mathcal{LPS}^k_{P_k}$

// When it starts, stack *lps* consists of streaks in $\mathcal{LPS}^{k-1}_{P_{k-1}}$.

1 $temp\_stack \leftarrow$ an empty stack
2 **while** ! $lps.isempty()$ **do**
3 **if** $lps.top().\vec{v} \preceq \vec{p}_k$ **then**
4 **break**
5 **else**
6 $s = \langle [l_s, k-1], \vec{v}_s \rangle \leftarrow lps.pop()$
7 $s' \leftarrow \langle [l_s, k], \vec{v'}_s \rangle$, where $\vec{v'}_s = (min(\vec{v}_s[1], \vec{p}_k[1]), \ldots, min(\vec{v}_s[d], \vec{p}_k[d]))$ //right-end extension of s
8 **if** $lps.isempty()$ **then**
9 $temp\_stack.push(s')$
10 **else**
11 $q = \langle [l_q, k-1], \vec{v}_q \rangle \leftarrow lps.top()$
12 $q' \leftarrow \langle [l_q, k], \vec{v'}_q \rangle$, where $\vec{v'}_q = (min(\vec{v}_q[1], \vec{p}_k[1]), \ldots, min(\vec{v}_q[d], \vec{p}_k[d]))$ //right-end extension of q
13 **if** $q' \not\succ s'$ **then**
14 $temp\_stack.push(s')$
15 **while** ! $temp\_stack.isempty()$ **do**
16 $lps.push(temp\_stack.pop())$
17 **if** $lps.isempty()$ *or* $lps.top().\vec{v} \not\preceq \vec{p}_k$ **then**
18 $lps.push(\langle [k, k], \vec{p}_k \rangle)$

// Now, *lps* contains all the streaks in $\mathcal{LPS}^k_{P_k}$.

Algorithm 15: Computing $\mathcal{LPS}_P^{k-1}$ and $\mathcal{LPS}_{P_k}^k$ on Multi-Dimensional Sequences

Input: $\mathcal{LPS}_{P_{k-1}}^{k-1}$ and $\vec{p}_k$

Output: $\mathcal{LPS}_P^{k-1}$ and $\mathcal{LPS}_{P_k}^k$

// Insert the following line before Line 1 in Algorithm 14.

1 $\mathcal{LPS}_P^{k-1} \leftarrow \varnothing$

// Insert the following line after Line 6 in Algorithm 14.

2 $\mathcal{LPS}_P^{k-1} \leftarrow \mathcal{LPS}_P^{k-1} \cup \{s\}$

3.5.3.3 A Note on "Curse of Dimensionality"

For a single-dimensional sequence with n elements, LLPS produces at most n candidates (i.e., local prominent streaks), according to Property 4. This upper-bound guarantees LLPS to be an efficient linear-time algorithm. However, the same property does not hold for multi-dimensional sequences. Consider an extreme case which is a 2-dimensional n-element sequence $(\vec{p}_1, \ldots, \vec{p}_n)$, where $\vec{p}_i = (i, n-i)$. It is not hard to prove that all $\frac{n(n+1)}{2}$ possible streaks in this sequence are prominent streaks and thus automatically local prominent streaks. This represents the worst case, in which nothing beats the brute-force baseline method.

While the worst case indicates the rather notorious "curse of dimensionality", our empirical results on multiple datasets are much more encouraging. The results show that the number of prominent streaks and the execution time of LLPS do not increase exponentially by the dimensionality of data. This is mainly due to that data values fluctuate and are correlated. We investigate these results in more details in Section 3.6.

Table 3.1: Data Sequences Used in Experiments on Basic Prominent Streak Discovery.

name	length	# prominent streaks	description
Gold	1074	137	Daily morning gold price in US dollars, 01/1985-03/1989.
River	1400	93	Mean daily flow of Saugeen River near Port Elgin, 01/1988-12/1991.
Melb1	3650	55	The daily minimum temperature of Melbourne, Australia, 1981-1990.
Melb2	3650	58	The daily maximum temperature of Melbourne, Australia, 1981-1990.
Wiki1	4896	58	Hourly traffic to *http://en.wikipedia.org/wiki/Main_page*, 04/2010-10/2010.
Wiki2	4896	51	Hourly traffic to *http://en.wikipedia.org/wiki/Lady_gaga*, 04/2010-10/2010.
Wiki3	4896	118	Hourly traffic to *http://en.wikipedia.org/wiki/Inception_(film)*, 04/2010-10/2010.
SP500	10136	497	S&P 500 index, 06/1960-06/2000.
HPQ	12109	232	Closing price of HPQ in NYSE for every trading day, 01/1962-02/2010.
IBM	12109	198	Closing price of IBM in NYSE for every trading day, 01/1962-02/2010.
AOL	132480	127	Number of queries to AOL search engine in every minute over three months.
WC98	7603201	286	Number of requests to World Cup 98 web site in every second, 04/1998-07/1998.

3.6 Experiments

We report and analyze experimental results in this section. The algorithms were implemented in Java. The experiments were conducted on a server with four 2.00GHz Intel Xeon E5335 CPUs running Ubuntu Linux. The limit on the heap size of Java Virtual Machine (JVM) was set at 512MB. We discuss the results on basic and general prominent streak discovery in Section 3.6.1 and Section 3.6.2, respectively.

3.6.1 Experimental Results on Basic Prominent Streak Discovery

We used multiple real-world datasets, including time series data library,[3] Wikipedia traffic statistics dataset,[4] NYSE exchange data,[5] AOL search engine log,[6] and FIFA World Cup 98 web site access log.[7] These datasets cover a variety of application scenarios, including meteorology, hydrology, finance, web log, and network traffic.

[3] *http://robjhyndman.com/TSDL/*

[4] *http://dammit.lt/wikistats/*

[5] *http://www.infochimps.com/datasets/nyse-daily-1970-2010-open-close-high-low-and-volume*

[6] *http://gregsadetsky.com/aol-data/*

[7] *http://ita.ee.lbl.gov/html/contrib/WorldCup.html*

Table 3.1 shows the information of 12 data sequences from these data sets that we used in experiments. For each data sequence, we list its name, length, and the number of prominent streaks in the sequence. Each data sequence was stored in a data file.

Examples of Interesting Prominent Streaks Discovered:

From 1985 to 1989, there had been more than one thousand consecutive trading days with morning gold price greater than \$300. During this period, there had been a streak of four hundred days with price more than \$400, though the \$500 price only lasted two days at most.

In Melbourne, Australia, during the years between 1981 and 1990, the weather had been pleasant. There had been more than two thousand days with minimal temperature above zero, and the streak was not ending. (We do not have data beyond 1990.) The longest streak during which the temperature hit above 35 degrees Celsius is six days. It was in the summer of the year 1981.

More than half of the prominent streaks we found in the traffic data of the Lady Gaga Wikipedia page were around September 12th, when she became a big winner in the MTV Video Music Awards (VMA) 2010. During that time, the page had been visited by at least 2000 people in every hour for almost four days.

Number of Candidate Streaks:

The three algorithms for candidate streak generation, namely Baseline (Algorithm 6), NLPS (Algorithm 8), and LLPS (Algorithm 10), differ by the ways they produce candidates and thus the numbers of produced candidates. Table 3.2 shows the total number of candidate streaks considered by each algorithm on each data sequence. The baseline algorithm produces an extremely large number of candidates since it enumerates all possible streaks, e.g., $\binom{7603202}{2}=2.89 \times 10^{13}$ for WC98. By contrast, NLPS only needs to consider $\bigcup_{1\leq k\leq |P|} \mathcal{LPS}^k_{P_k}$, which is a superset of the real prominent streaks $\mathcal{PS}_P$ but a much smaller subset of all possible streaks $\mathcal{S}_P$. For

Table 3.2: Number of Candidate Streaks, Basic Prominent Streak Discovery.

name	Baseline	NLPS	LLPS
Gold	5.77×10^5	6.04×10^4	1.05×10^3
River	9.81×10^5	2.18×10^4	1.33×10^3
Melb1	6.66×10^6	4.47×10^4	3.50×10^3
Melb2	6.66×10^6	4.28×10^4	3.49×10^3
Wiki1	1.20×10^7	7.16×10^4	4.79×10^3
Wiki2	1.20×10^7	5.77×10^4	4.75×10^3
Wiki3	1.20×10^7	7.31×10^4	4.70×10^3
SP500	5.14×10^7	1.69×10^6	9.98×10^3
HPQ	7.33×10^7	5.24×10^5	1.08×10^4
IBM	7.33×10^7	6.97×10^5	1.13×10^4
AOL	8.78×10^9	3.53×10^6	1.20×10^5
WC98	2.89×10^{13}	1.78×10^8	6.69×10^6

instance, the number of candidate streaks by NLPS is 1.78×10^8 for WC98, which is 5 orders of magnitude smaller than what Baseline considers. LLPS further significantly educes the number of candidates by only considering LPSs. For example, there are 6.69×10^6 LPSs in WC98, which is about 30 times smaller than 1.78×10^8. Note that the number of LPSs for LLPS is bounded by sequence length (Property 4), which is verified by Table 3.2.

Execution Time:

The number of candidate streaks directly determines the efficiency of our algorithms. In Table 3.3 we report the execution time of our algorithms using the three candidate streak generation methods (Baseline, NLPS, LLPS), for all 12 data sequences. For skyline operation, we implemented the sorting-based, external-memory sorting-based, and BST-based skyline methods mentioned in Section 3.1. Under these different skyline methods, Baseline, NLPS, and LLPS perform and compare consistently. Therefore in Table 3.3 we only report the results for implementations based

Table 3.3: Execution Time (in Milliseconds), Basic Prominent Streak Discovery.

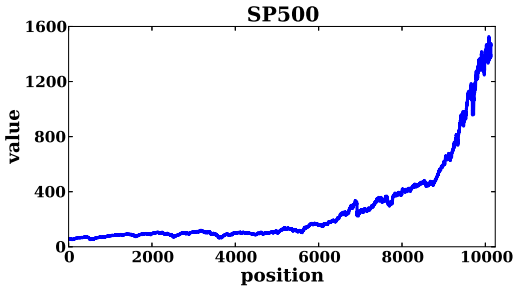
name	Baseline	NLPS	LLPS
Gold	183	122	13
River	126	84	19
Melb1	385	101	36
Melb2	384	101	35
Wiki1	670	105	46
Wiki2	646	97	46
Wiki3	632	126	48
SP500	4453	789	116
HPQ	6285	338	101
IBM	4228	377	135
AOL	290744	752	201
WC98	$>$ 1 hour	38999	3012

on the BST-based skyline method, due to space limitations. The reported execution time is in milliseconds and is the average of five runs.

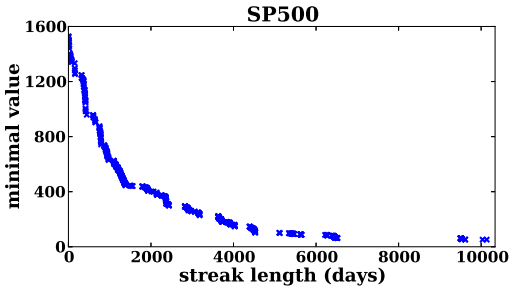
When reporting the execution time of these algorithms, we excluded data loading time, i.e., the time spent on just reading each data file. This is because data loading time is dominated by processing time of the algorithms once the data file gets large. In our experiments, WC98 cost 1 second to load while the loading time of all other datasets was below 30ms.

In Table 3.3 we use '>1 hour' to denote the execution time when an algorithm could not finish within one hour (i.e., 3600000ms). This lower bound is sufficient in showing the performance difference of the various algorithms.

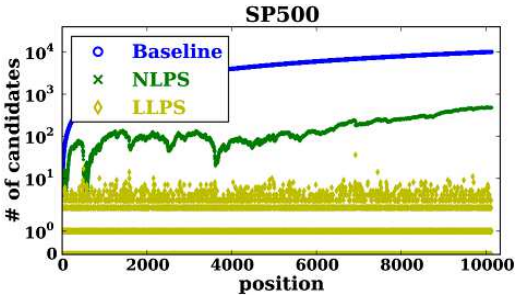
With regard to the comparison of Baseline, NLPS, and LLPS, it is clear from Table 3.3 that LLPS outperforms NLPS, and both NLPS and LLPS are far more efficient than Baseline. This is exactly due to the large gap in the number of candidate streaks (shown in Table 3.2), which in turn determines the number of comparisons performed during skyline operations.



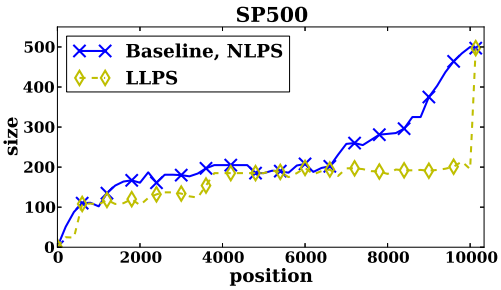
a Data Sequence



b Prominent Streaks

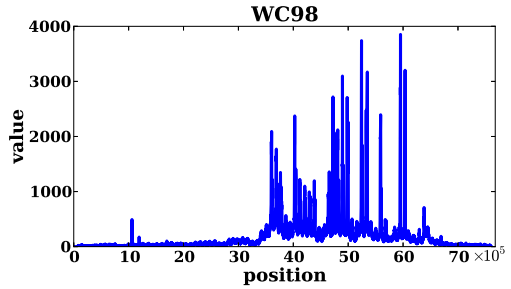


c Number of Candidate Streaks

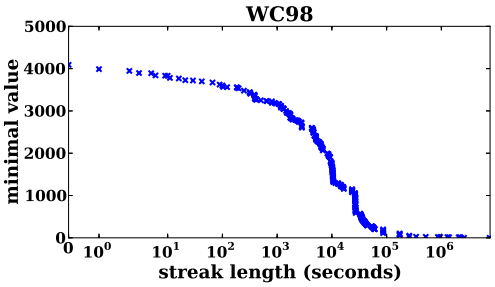


d Size of Dynamic Skyline

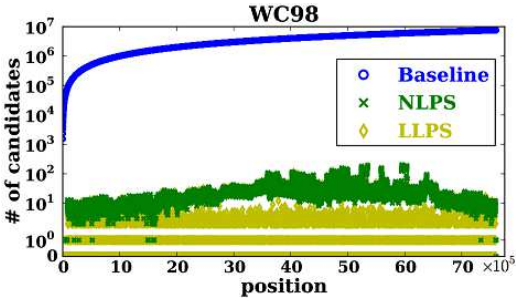
Figure 3.4: Detailed Results on SP500, Basic Prominent Streak Discovery.



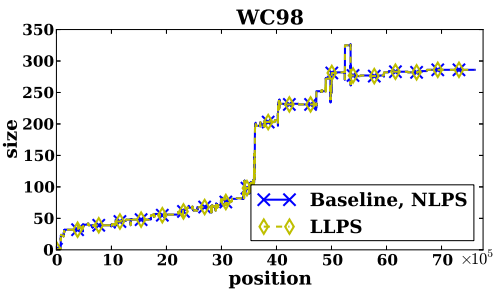
a Data Sequence



b Prominent Streaks



c Number of Candidate Streaks



d Size of Dynamic Skyline

Figure 3.5: Detailed Results on WC98, Basic Prominent Streak Discovery.

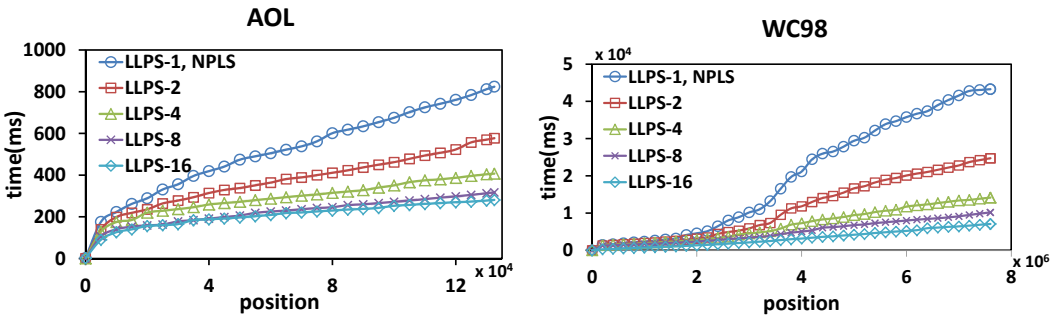
A Closer Look:

To have a better understanding of the experimental results, we take a close look at the SP500 data sequence. Figure 3.4a shows the data sequence itself. We see that the sequence is almost monotonically increasing at the coarse grain level. Due to that, the number of prominent streaks found in SP500 (497, as shown in Table 3.1) is the most among all the data sequences. We also visualize the prominent streaks in SP500 in Figure 3.4b, where the x-axis is for interval length and the y-axis is for minimal value in the interval.

In Table 3.2 we have seen the huge difference among Baseline, NLPS and LLPS in total number of candidate streaks. These three algorithms all generate candidates progressively. Therefore in Figure 3.4c we show, for each algorithm, the number of new candidate streaks produced at every value position of the data sequence. The figure clearly shows the superiority of LLPS since it always generates orders of magnitude less candidates at each position.

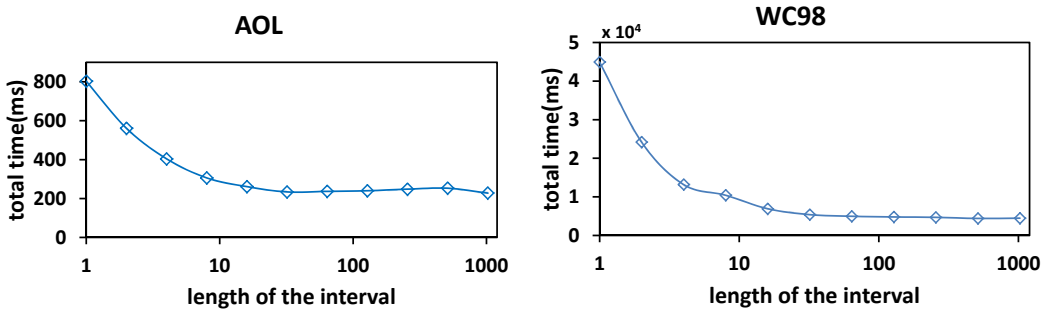
The BST-based skyline method maintains a dynamic skyline, as a binary search tree, in memory. The size of this tree affects the efficiency of tree operations, such as inserting and deleting a streak. Figure 3.4d shows the size of the dynamic skyline along the sequence of SP500 by each algorithm. The curves for Baseline and NLPS overlap since they both store $\mathcal{PS}_{P_k}$, at every position k, in the dynamic skyline. On the contrary, LLPS does not need to store some streaks in $\mathcal{PS}_{P_k}$, hence the tree size is much smaller than that for Baseline/NLPS when the sequence is almost constantly growing in the second half of SP500.

In Figure 3.5 we show the detailed results on WC98 data, which are similar to the results on SP500 but are also different on several aspects. The data sequence fluctuates. Hence there are less candidate streaks by NLPS and LLPS, which makes the gap between them and Baseline much bigger. For the same reason, the size of the



a AOL b WC98

Figure 3.6: Cumulative Execution Time at Various Positions, for Different Reporting Frequencies, Basic Prominent Streak Discovery.



a AOL b WC98

Figure 3.7: Total Execution Time by Reporting Frequencies, Basic Prominent Streak Discovery.

dynamic skyline is almost identical for the three algorithms. Note that Figure 3.5b uses logarithmic scale on x-axis, because the very long streaks would otherwise make most other streaks cluttered to the left if linear scale is used.

Monitoring Prominent Streaks:

In Section 3.4 we discussed how to monitor the prominent streaks as a data sequence evolves and new data values come. The adaptation of LLPS for monitoring purpose was shown in Algorithm 12. This algorithm can control at which positions the prominent streaks (so far) need to be reported.

Take AOL and WC98 as examples. Figure 3.6 shows the execution time of Algorithm 12. The x-axis represents the sequence position, and the y-axis is for the total execution time by that position. There are five curves in each figure, corresponding to five different frequencies of reporting prominent streaks. For instance, LLPS-1 means that, whenever a new data entry comes, all the prominent streaks so far are reported; LLPS-16 means the prominent streaks are requested at every 16 data entries. As discussed in Section 3.4, LLPS-1 is identical to NLPS (Algorithm 8), and LLPS-n is identical to LLPS (Algorithm 10), where n is the sequence length when it does not evolve anymore. Figures 3.6a and 3.6b clearly show that the total execution time of LLPS-i increases as the reporting frequency increases (i.e., reporting interval i decreases). Figures 3.7a and 3.7b further show how the total execution time changes along different reporting intervals. We can see that the execution time drops rapidly at the beginning and quickly reaches near-optimal value even when the frequency is still fairly high (e.g., reporting the prominent streaks at every 16 entries.)

3.6.2 Experimental Results on General Prominent Streak Discovery

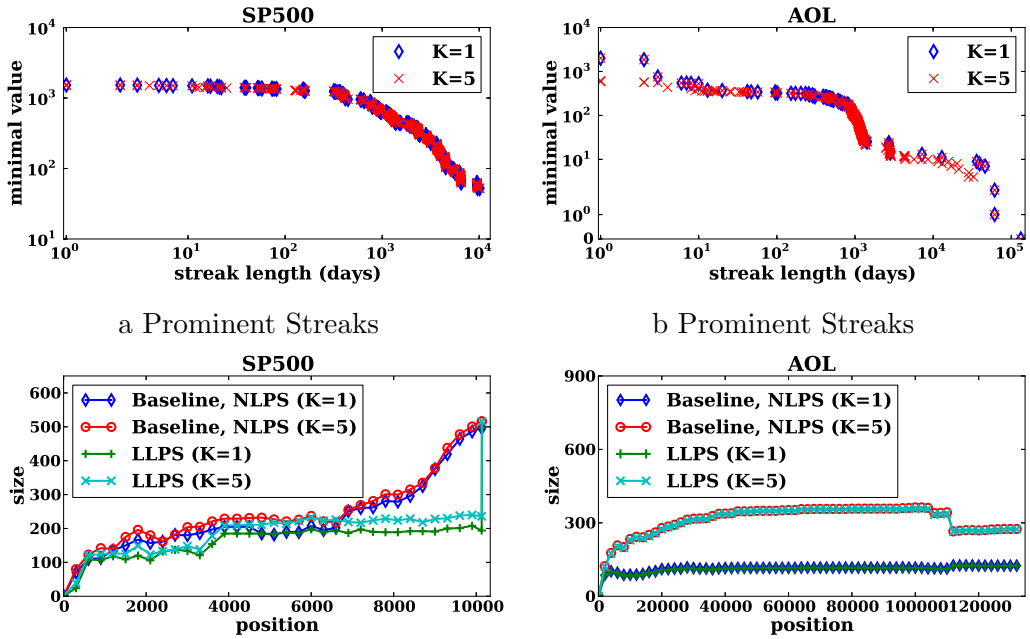
In this section, we discuss the results on top-k, multi-sequence, and multi-dimensional prominent streak discovery. At the end of this section, we also present the results from an experiment that put together these different extensions.

Top-k Prominent Streaks:

The experiments on top-k prominent streaks were conducted on the same datasets discussed in Section 3.6.1. For each dataset, Table 3.4 shows the number of top-5 prominent streaks (i.e., $\mathcal{KPS}_P$ in Definition 14) and the execution time of the extended Baseline, NLPS and LLPS algorithms. Note that the number of candidate streaks shown in Table 3.2 remains the same, since the same candidate streak generation methods are used for top-k prominent streaks, as discussed in Section 3.5.1.

Table 3.4: Number of Prominent Streaks and Execution Time (in Milliseconds), Top-5 Prominent Streaks.

name	# prominent streaks	Baseline	NLPS	LLPS
Gold	147	1884	348	44
River	144	6.81×10^4	275	57
Melb1	160	6.06×10^7	572	96
Melb2	160	3.01×10^6	445	150
Wiki1	181	3.68×10^7	1369	140
Wiki2	115	1.88×10^7	565	172
Wiki3	172	1.05×10^6	473	136
SP500	516	7.09×10^6	13700	270
HPQ	251	$>$ 10 hours	3211	178
IBM	232	$>$ 10 hours	5914	229
AOL	250	$>$ 10 hours	26000	798
WC98	409	$>$ 10 hours	$>$ 10 hours	13300



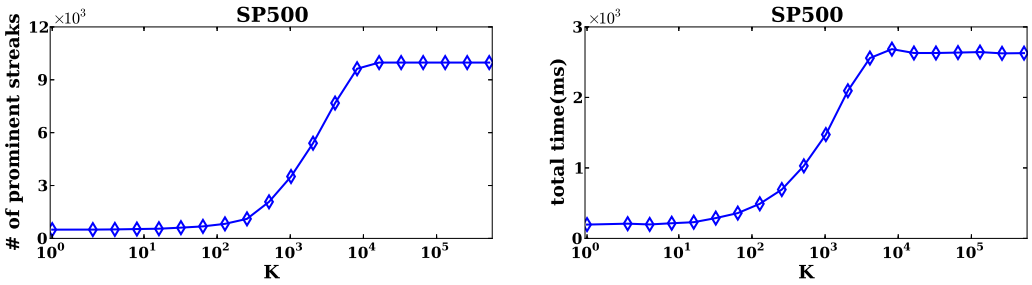
a Prominent Streaks

b Prominent Streaks

c Size of Dynamic Skyline

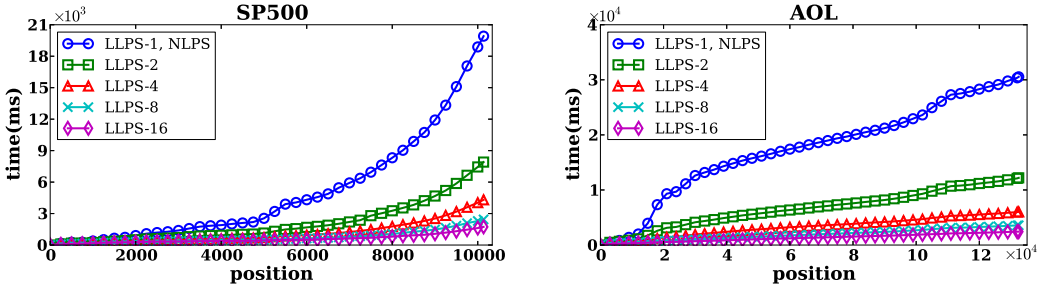
d Size of Dynamic Skyline

Figure 3.8: Detailed Results on SP500 and AOL, Top-1 vs. Top-5 Prominent Streaks.



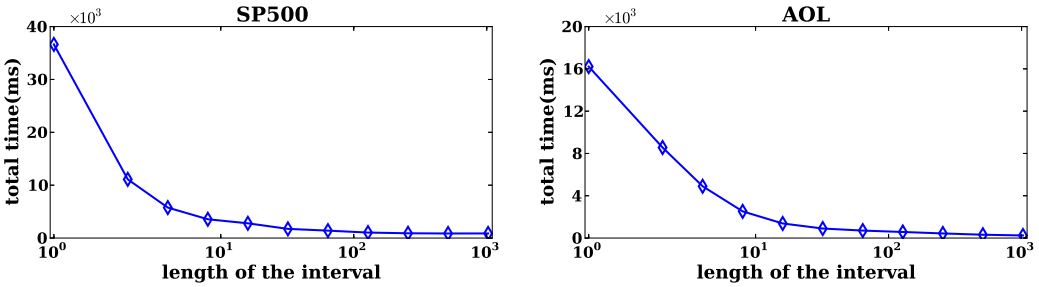
a Number of Prominent Streaks b Execution Time

Figure 3.9: Number of Prominent Streaks and Execution Time, LLPS on SP500, Top-k Prominent Streaks, Varying k.



a SP500 b AOL

Figure 3.10: Cumulative Execution Time at Various Positions, for Different Reporting Frequencies, Top-5 Prominent Streak Discovery.



a SP500 b AOL

Figure 3.11: Total Execution Time by Reporting Frequencies, Top-5 Prominent Streak Discovery.

As Table 3.4 shows, in comparison with the execution time in Table 3.3 (i.e., the time of discovering top-1 prominent streaks), the execution time of Baseline increased by one or more orders of magnitude, while the performance of NLPS and LLPS was degraded by less than one order of magnitude in most cases. This is explained as follows. Finding top-k prominent streaks incurs higher cost of skyline operation than finding top-1 prominent streaks. More specifically, the cost of skyline operation is determined by the number of dominance comparisons between candidate streaks and streaks in the dynamic skyline. Therefore the number of comparisons increases by both the number of candidate streaks and the size of the dynamic skyline. In comparison with top-*1*, finding top-k prominent streaks requires maintaining local prominent streaks with as many as $k-1$ dominators, which increases the size of dynamic skyline and thus incurs larger cost. For example, a dominator search for a candidate cannot terminate until the number of dominators reaches k, whereas the search terminates immediately in top-1 algorithms once a dominator is found. The more candidate streaks there are, the larger the increment of skyline operation cost (from top-1 to top-k) grows. This further explains why the performance of Baseline was degraded the most.

Figure 3.8 shows some interesting detailed results on two different sequences. Since sequence SP500 increases almost monotonically, a local prominent streak that is not globally prominent most likely has a relatively large number of dominators. Hence, the size of dynamic skyline in top-*5* prominent streak discovery is only slightly larger than that in top-*1*. This explains Figure 3.8c. On the contrary, for sequence AOL, the size of dynamic skyline for top-*5* is about twice the size for top-*1* (igure 3.8d). This is because sequence AOL fluctuates. The prominent streaks have different right ends of intervals due to the fluctuation. This also explains why the sizes of dynamic skylines in Baseline, NLPS and LLPS do not differ much from each other in this sequence.

Table 3.5: Data Sequences Used in Experiments on Multi-sequence Prominent Streak Discovery.

name	# sequences	average length	# prominent streaks	description
NBA1	1225	281	28	Points scored by all NBA players from 1991-2004
Wiki	8	14454	59	Hourly traffic to the Wikipedia pages of Ivy League universities

Table 3.6: Number of Candidate Streaks, Multi-sequence Prominent Streak Discovery.

name	Baseline	NLPS	LLPS
NBA1	9.41×10^7	1.23×10^6	3.31×10^5
Wiki	8.36×10^8	1.23×10^6	1.86×10^5

However, as Table 3.4 shows, their differences on execution time are still significant because NLPS and LLPS generate much less candidates than Baseline does.

By Definition 14, $\mathcal{PS}_P \subseteq \mathcal{KPS}_P$, i.e., all prominent streaks are also top-k prominent streaks. This is clearly shown in Figures 3.8a and 3.8b. Furthermore, $\mathcal{KPS}_P \subseteq \mathcal{LPS}_P$, i.e., top-$k$ prominent streaks must be local prominent streaks too. Therefore, the set $\mathcal{KPS}_P$ grows by k and stops growing after k reaches a certain value, when all streaks in $\mathcal{LPS}_P$ are included in $\mathcal{KPS}_P$. This is demonstrated by Figure 3.9a in which the number of prominent streaks in sequence SP500 increases by k until k reaches about $10,000$. As a result, total execution time also changes in sync with the number of prominent streaks, as shown in Figure 3.9b.

We also experimented with monitoring top-k prominent streaks. The results are shown in Figure 3.10 and Figure 3.11, which exhibit patterns of execution time similar to the patterns in Figure 3.6 and Figure 3.7 for monitoring top-1 prominent streaks.

Multi-sequence Prominent Streaks:

Table 3.7: Execution Time (in Milliseconds), Multi-sequence Prominent Streak Discovery.

name	Baseline	NLPS	LLPS
NBA1	3436	310	292
Wiki	33537	275	190

Table 3.8: Distribution of Players by Number of Prominent Streaks.

number of prominent streaks	number of players
0	1215
1	6
3	1
4	1
5	1
10	1

Table 3.9: Multi-sequence Prominent Streaks in Datast NBA1.

length	minimal value	players
1	71	David Robinson
2	51	Allen Iverson; Antawn Jamison
4	42	Kobe Bryant
9	40	Kobe Bryant
13	35	Kobe Bryant
14	32	Kobe Bryant
16	30	Kobe Bryant
17	27	Michael Jordan
27	26	Allen Iverson
34	24	Tracy McGrady
45	21	Allen Iverson
57	20	Allen Iverson
74	19	Shaquille O'Neal
94	18	Shaquille O'Neal
96	17	Karl Malone
119	16	Karl Malone
149	15	Karl Malone
159	14	Karl Malone
263	13	Karl Malone
357	12	Karl Malone
527	11	Karl Malone
575	10	Karl Malone
758	7	Karl Malone
858	6	Shaquille O'Neal
866	2	Karl Malone
932	1	John Stockton
1185	0	Jim Jackson

Table 3.10: Data Sequences Used in Experiments on Multi-dimensional Prominent Streak Discovery.

name	length	# prominent streaks	# dimensions	description
Malone	986	640	6	1991-2004 game log of Karl Malone (minutes, points, rebounds, assists, steals, blocks)
Crashes	3287	1493	5	2003-2011 Texas Motor Vehicle Crash Statistics (Crashes and Injuries by Date)
AAPL	6411	2616	3	NASDAQ stock data of Apple Inc. from 1970 to 2010, on daily values of opening price, change ratio $((open - close)/open \times 100\%)$ and trading volume

We used two datasets for experiments on multi-sequence prominent streak discovery. One (Wiki) is the hourly traffic to Ivy League universities' Wikipedia pages,[8] one sequence per university. The other dataset (NBA1) contains 1225 sequences, one sequence per NBA player. Each sequence lists the scores of a player in all the games he played from 1991 to 2004. [9] The characteristics of these two datasets are shown in Table 3.5, including the number of sequences, average sequence length and the number of prominent streaks. Tables 3.6 and 3.7 show the number of candidate streaks and the execution time, respectively, for Baseline, NLPS and LLPS. The results are very similar to that in Tables 3.2 and 3.3. This is because the process of multi-sequence prominent streak discovery is not very different from its single-sequence counterpart.

For dataset NBA1, Table 3.8 shows the distribution of players by the number of prominent streaks contributed by them. All 29 prominent streaks, i.e., NBA scoring records in the period of 1991 to 2004, come from merely 10 different players. Table 3.9 shows the detailed records. One interesting observation from the table is that Karl Malone and John Stockton, two of the healthiest NBA players, had scored in two longest streaks of games. Another example is that Allen Iverson is the only one who scored at least 20 points in more than 50 consecutive games.

Multi-dimensional Prominent Streaks:

[8] *http://dammit.lt/wikistats/*

[9] *http://www.databasebasketball.com/index.htm*

Table 3.11: Number of Candidate Streaks, Multi-dimensional Prominent Streak Discovery.

name	Baseline	NLPS	LLPS
Malone	4.87×10^5	1.27×10^4	4.47×10^3
Crashes	5.40×10^6	6.95×10^5	1.82×10^4
AAPL	2.06×10^7	4.77×10^5	4.08×10^5

Table 3.12: Execution Time (in Milliseconds), Multi-dimensional Prominent Streak Discovery.

name	Baseline	NLPS	LLPS
Malone	4575	336	180
Crashes	1.08×10^5	1113	326
AAPL	5.65×10^5	9997	557

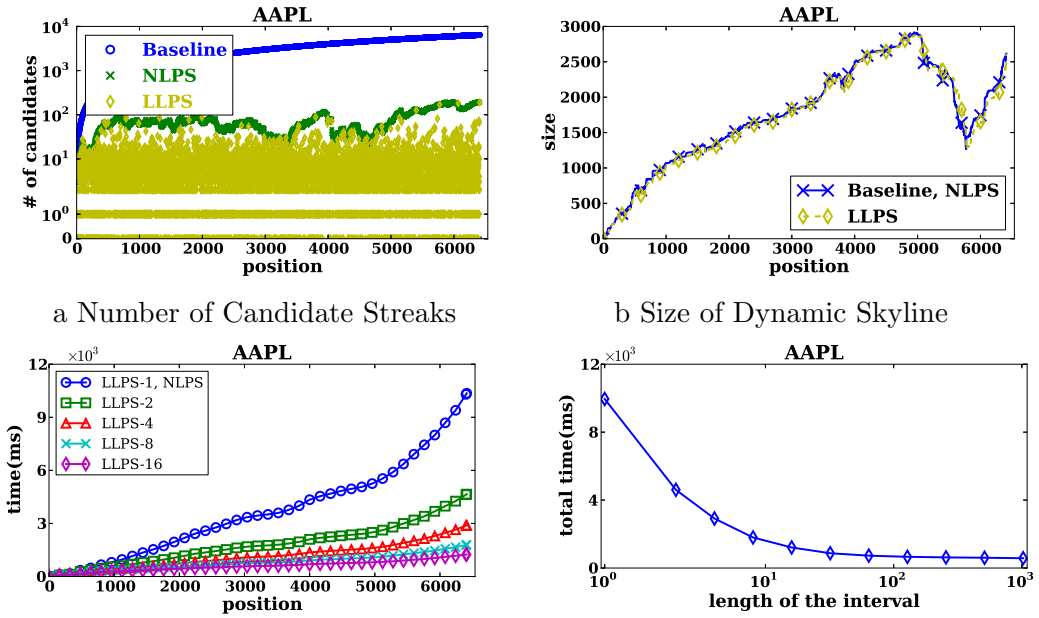
We used three datasets for experiments on multi-dimensional prominent streak discovery, listed in Table 3.10. The first dataset is the game log of NBA player Karl Malone, from 1991 to 2004 seasons.[10] This is a sequence of 986 elements, each of which represents Malone's performance in a game on 6 performance dimensions. The second dataset is the 2003-2011 Texas Motor Vehicle Crash Statistics,[11] a 5-dimensional sequence of 3287 elements, where each element is for one day and represents the daily counts of crashes, injuries, fatalities, and so on. The last dataset is the historical NASDAQ stock data of Apple Inc. from 1970 to 2010.[12] In this 6411-element sequence, each element is for a trading day and contains the opening price, change ratio, and trading volume of the stock of Apple Inc. on that day.

The number of candidate streaks and the execution time by Baseline, NLPS, and LLPS are shown in Tables 3.11 and 3.12. Figure 3.12 further shows detailed

[10] *http://www.databasebasketball.com/index.htm*

[11] *http://www.dot.state.tx.us/txdot_library/drivers_vehicles/publications/crash_statistics/default.htm*

[12] *http://www.infochimps.com/datasets/nasdaq-exchange-daily-1970-2010-open-close-high-low-and-volume*



a Number of Candidate Streaks

b Size of Dynamic Skyline

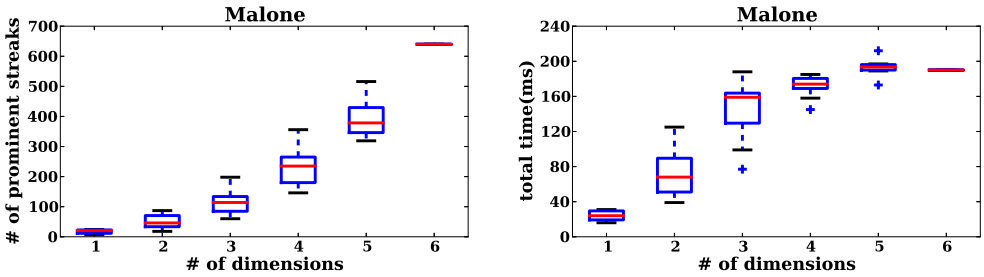
c Cumulative Execution Time at Various Positions, for Different Reporting Frequencies

d Total Execution Time by Reporting Frequencies

Figure 3.12: Detailed Results on AAPL, Multi-dimensional Prominent Streak Discovery.

experimental results on dataset AAPL. The observations made on these results are similar to those for basic, top-k, and multi-sequence prominent streak discovery.

We also investigated how number of prominent streaks and total execution time of LLPS increase by the dimensionality of data, using dataset Malone. As the boxplots in Figure 3.13 show, these measures do not increase exponentially by data dimensionality, at least under small dimensionality such as 6. This indicates that, while the "curse of dimensionality" can raise concerns, the empirical results are much more encouraging. It is partly due to that data values fluctuate and thus the appearance of a small value terminates many prominent streaks. Furthermore, data values are correlated, which practically reduces data dimensionality. Finally, the results are for at most 6 dimensions. We note that arguably the prominent streaks



a Number of Prominent Streaks b Execution Time of LLPS

Figure 3.13: Experiments on Increasing Dimensionality.

Table 3.13: Data Sequences Used in Experiments on Top-5 Multi-sequence Multi-dimensional Prominent Streak Discovery.

name	# sequences	average length	# dimensions	# prominent streaks	description
NBA2	1185	290	6	10867	1991-2004 game log of all NBA players (minutes, points, rebounds, assists, steals, blocks)

found in the real world, such as the ones in Section 3.1, mostly would not have more than 6 dimensions.

Putting it Together: Top-k Prominent Streaks on Multiple Multi-dimensional Sequences:

We also used dataset NBA2 (Table 3.13) for experiments on discovering top-k prominent streaks from multiple multi-dimensional sequences. This dataset contains 1185 6-dimensional sequences, each of which corresponds to the game log of an NBA player from 1991 to 2004. One of the sequences is the aforementioned dataset Malone.

Figure 3.14 shows that distribution of prominent streaks by length. It is clear that the distribution follows the power law. The reason is that the minimal value

Table 3.14: Number of Candidate Streaks, Top-5 Multi-sequence Multi-dimensional Prominent Streak Discovery.

name	Baseline	NLPS	LLPS
NBA2	9.41×10^7	2.98×10^6	8.76×10^5

Table 3.15: Execution Time (in Milliseconds), Top-5 Multi-sequence Multi-dimensional Prominent Streak Discovery.

name	Baseline	NLPS	LLPS
NBA2	1.39×10^7	4.33×10^5	1.14×10^5

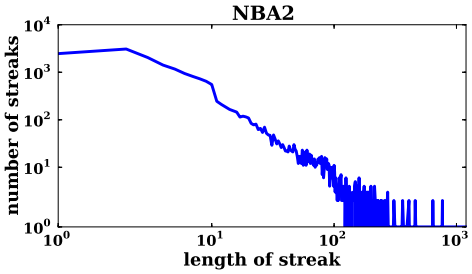
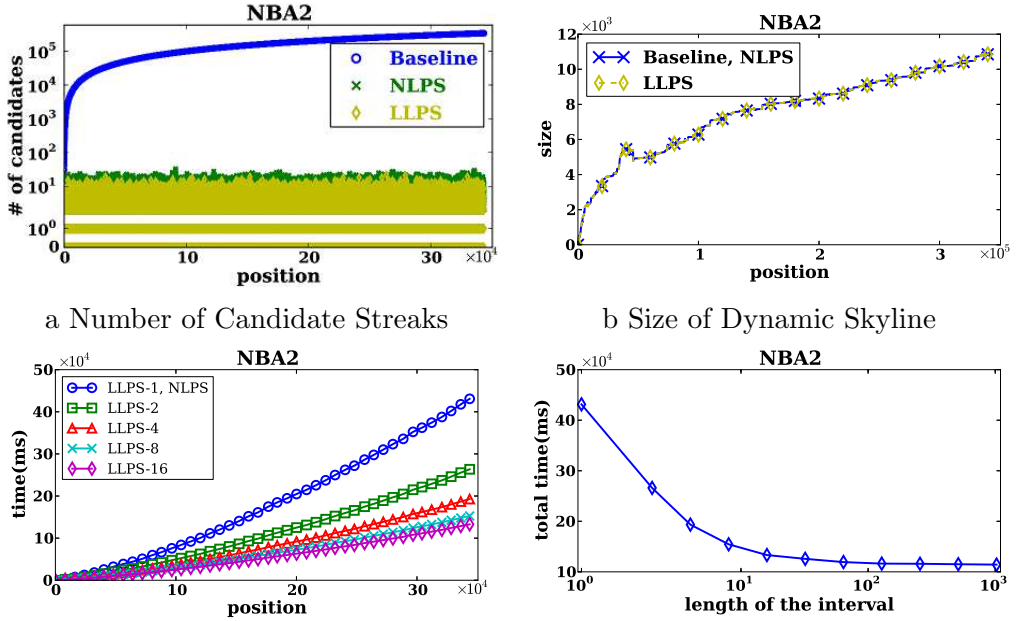


Figure 3.14: Distribution of Prominent Streaks by Length.



a Number of Candidate Streaks

b Size of Dynamic Skyline

c Cumulative Execution Time at Various Reporting Positions, for Different Reporting Frequencies

d Total Execution Time by Reporting Fre- quencies

Figure 3.15: Detailed Results on NBA2, Top-5 Multi-sequence Multi-dimensional Prominent Streak Discovery.

vector for a streak takes the minimal value on each dimension from all elements. The longer a streak is, the smaller the values in its minimal value vector become. Therefore it is difficult for a long streak to stand out as prominent. Figure 3.15 shows detailed experimental results on this dataset which show similar patterns to those observed for aforementioned experiments.

CHAPTER 4

Data In, Fact Out: Automated Monitoring of Facts by FactWatcher

4.1 Introduction

Computational journalism [84, 85] is a young field that assists journalism using computing. One of its objectives is to find news leads backed up by hard, factual data. In the last several years, our research in this thrust has been focused on automatic and algorithmic fact finding by database and data mining techniques [86, 87, 88, 89]. Specifically, we studied how to monitor three types of facts that can be expressed as the following factual statements:

Situational fact [88] "The social world's most viral photo ever generated 3.5 million likes, 170,000 comments and 460,000 shares by Wednesday afternoon." (*http://www.cnbc.com/id/49728455*) A situational fact is about a *contextual skyline* object within a certain context (e.g., all photos posted to Facebook) with regard to several measures (e.g., number of "likes", number of "comments", and number of "shares"), i.e., the object is not *dominated* by any object in the context when they are compared by the measures.

One-of-the-few [87] "Victor Oladipo scored 30 points and handed out 14 assists ... only three other rookies have recorded at least 30 points and 14 assists in a game ..." (*http://espn.go.com/espn/elias?date=20140222*) This statement is about a one-of-the-four object, which is only dominated by at most three other objects.

Prominent streak [86, 89] "This month the Chinese capital has experienced 10 days with a maximum temperature in around 35 degrees Celsius—the most for the

month of July in a decade." (*http://www.chinadaily.com.cn/china/2010-07/27/content_11055675.htm*) A prominent streak is a long consecutive subsequence (e.g., 10 days of temperature) consisting of only large (small) values (e.g., all above a value close to 35 degrees) within a sequence of values (e.g., daily maximum temperature of Beijing).

Automatic fact finding is helpful in multiple news domains, as factual statements similar to the above ones can be found from not only social media, sports and weather data, but also criminal records, government records and stock data. Several more examples are 1) situational fact: "There were 35 DUI arrests and 20 collisions in city C yesterday, the first time in 2013." 2) one-of-the-few: "Rick Perry is one of the only three candidates in the 2012 US federal election cycle to have received at least $600k from 'lawyers & lobbyists' (an interest group that is usually pro-Democrat) and $400k from 'energy & natural resources' (usually pro-Republican)." and 3) prominent streak: "The Nikkei 225 closed below 10000 for the 12th consecutive week, the longest such streak since June 2009."

This chapter demonstrates FactWatcher, a system for automated monitoring of the aforementioned three types of facts. Figure 4.1 illustrates the components of FactWatcher. Given an ever-growing database, upon the arrival of a new tuple t, FactWatcher checks if t triggers any new situational facts, one-of-the-few facts, and prominent streaks. It is impossible (especially with a manual approach) to exhaustively check all possible facts upon the arrival of every new tuple in a large database, due to the large search space. The systematic and efficient algorithms in FactWatcher thus enable a practical tool that assists journalists in identifying newsworthy stories. This is particularly valuable when we consider the diminishing readership and resources that traditional news media is facing.

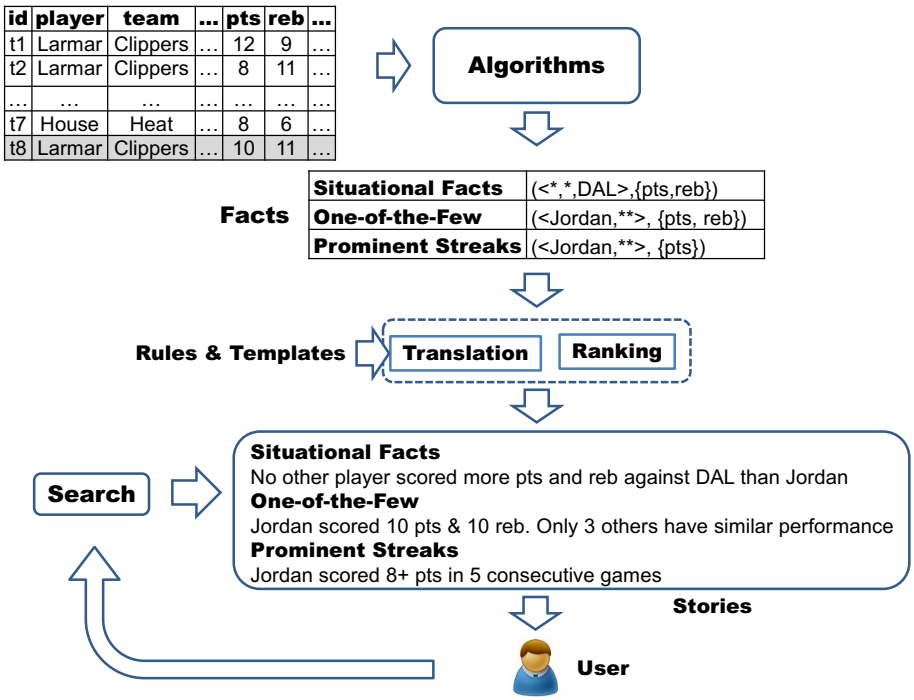


Figure 4.1: FactWatcher System Architecture

FactWatcher is an integrated system beyond the piecemeal algorithms in [86, 87, 88, 89]. It incorporates all three types of facts into a unified suite of data model, algorithm framework and fact ranking measure. It enables monitoring one-of-the-few facts in all different subspaces of dimension and measure attributes, which was not considered in [87]. It also supports a novel measure for ranking all types of facts by the elapsed time since their last comparable facts were discovered. Furthermore, FactWatcher provides multiple features in striving for an end-to-end system. By using rules and templates, the discovered facts are translated into textual news leads and presented to users; it allows users to customize the system by choosing which attributes in the database to consider and which measures to employ in ranking facts; it also supports keyword-based search of facts.

Table 4.1: A Data Table for the Running Example

id	*player*	*team*	*opposition team*	*pts*	*ast*	*reb*
t_1	*Lamar Odom*	*Clippers*	*Nets*	12	9	13
t_2	*Lamar Odom*	*Clippers*	*Lakers*	8	11	6
t_3	*Lamar Odom*	*Clippers*	*Lakers*	9	9	7
t_4	*Eddie House*	*Heat*	*Nets*	9	7	8
t_5	*Lamar Odom*	*Heat*	*Nets*	10	11	12
t_6	*Eddie House*	*Clippers*	*Nets*	10	11	10
t_7	*Eddie House*	*Heat*	*Wizards*	8	6	9
t_8	*Lamar Odom*	*Clippers*	*Lakers*	10	11	11

4.2 Concepts

Consider an append-only table $R(\mathcal{D}; \mathcal{M})$, where $\mathcal{D}=\{d_1, \ldots, d_n\}$ is a set of *dimension attributes* and $\mathcal{M} = \{m_1, \ldots, m_q\}$ is a set of *measure attributes*. A *constraint* C is a conjunctive expression of the form $d_1=v_1 \wedge \ldots \wedge d_n=v_n$ (also written as $\langle v_1, v_2, \ldots, v_n \rangle$ for simplicity), where $v_i \in dom(d_i) \cup \{*\}$ and $dom(d_i)$ is the value domain of dimension attribute d_i. The set of all possible constraints is denoted $\mathcal{C}$. Given a constraint $C \in \mathcal{C}$, $\sigma_C(R)$ is the relational algebra expression that chooses all tuples in R that satisfy C.

Given a *measure subspace* $M \subseteq \mathcal{M}$ and two tuples $t, t' \in R$, t *dominates* t' with respect to M, denoted by $t \succ_M t'$ or $t' \prec_M t$, if t is not worse than t' on any measure attribute in M and t is better than t' on at least one measure attribute. Let $\delta_M(R, t)$ denote the number of tuples in R that dominate t with regard to M. The *k-skyband* ($k \geq 1$) of R in M, denoted $\lambda^k_M(R)$, is the set of tuples in R dominated by fewer than k other tuples, i.e., $\lambda^k_M(R)=\{t \in R \mid \delta_M(R, t) < k\}$. The 1-skyband ($\lambda^1_M(R)$, or simply $\lambda_M(R)$) is known as the *skyline* of R. Given a user-specified threshold $\tau \geq 1$, the *top-τ skyband* of R in M, denoted $\tau_M(R)$, is the $\hat{k}$-skyband where $\hat{k}=\max\{k \mid \tau \geq |\lambda^k_M(R)|\}$, i.e., $\hat{k}$ is the largest such integer that the $\hat{k}$-skyband has no more than τ tuples. (For rigor, we say $\lambda^0_M(R)=0$.)

When a new tuple t is appended to R, FactWatcher discovers three types of interesting facts about t, as follows.

Situational fact FactWatcher finds S^t=$\{(C, M) \mid C \in \mathcal{C}, M \subseteq \mathcal{M}, t \in \lambda_M(\sigma_C(R))\}$—the set of constraint-measure pairs (C, M) such that t is in the corresponding *contextual skyline*, i.e., the skyline of those objects satisfying C with regard to M. Consider Table 4.1, where $\mathcal{D}$=$\{$*player, team, opposition team*$\}$ and $\mathcal{M}$=$\{$*pts, ast, reb*$\}$. The last appended tuple t_8 belongs to the contextual skylines for several constraint-measure pairs, including ($\langle$**, *, Lakers*$\rangle$, $\{$*pts, ast*$\}$) and ($\langle$*Lamar Odom, Clippers, **$\rangle$, $\{$*ast*$\}$).

One-of-the-few FactWatcher discovers F^t=$\{(C, M) \mid C \in \mathcal{C}, M \subseteq \mathcal{M}, t \in \tau_M(\sigma_C(R))\}$—the set of such constraint-measure pairs (C, M) that t is in the corresponding top-τ skyband. Consider Table 4.1 again. For constraint-measure pair ($\langle$*Lamar Odom, *, **$\rangle$, $\{$*pts, reb*$\}$), the skyline, 2-skyband and 3-skyband are $\{t_1\}$, $\{t_1, t_5\}$ and $\{t_1, t_5, t_8\}$, respectively. Hence, t_8 belongs to the top-3 skyband but not the top-2 skyband of this constraint-measure pair.

Prominent streak Given a set of object identification attributes $I \subseteq \mathcal{D}$, $\mathcal{G}$=$\{I \cup S \mid S \in 2^{\mathcal{D}-I}\}$ defines all considered ways of grouping tuples. In Table 4.1, if I=$\{$*player*$\}$, then $\mathcal{G}$=$\{\{$*player*$\}$, $\{$*player, team*$\}$, $\{$*player, opposition team*$\}$, $\{$*player, team, opposition team*$\}\}$. Given grouping attributes $G \in \mathcal{G}$, the corresponding group-by and aggregation operation is denoted *seq* $\gamma_G(R)$. *seq* is an aggregate function that, for each distinct value g of G (i.e., a group), produces an *ordered sequence* P_g=$t_{e_1} \ldots t_{e_u}$ consisting of all tuples in the group, i.e., $\forall 1 \leq i \leq u$, $t_{e_i}[G]$=g. The tuples are ordered by their unique timestamps—$\forall 1 \leq i < j \leq u$, t_{e_i} was inserted into R before t_{e_j}, i.e., the real-world event for t_{e_i} happened before that for t_{e_j}.

A *streak* s in a sequence P_g=$t_{e_1} \ldots t_{e_u}$ is any consecutive subsequence $t_{e_l} \ldots t_{e_r}$. We use $s.len$ to denote the length of s (i.e., $r-l+1$). We use $s.\vec{v}$ to denote the vector containing the minimal value in s on every measure attribute, i.e., $s.\vec{v}$=$(\min_{l \leq i \leq r} t_{e_i}[m_1]$, $\ldots$, $\min_{l \leq i \leq r}\ t_{e_i}[m_q])$, where $\{m_1, ..., m_q\}$ forms the set of all measure attributes $\mathcal{M}$.

Given a set of streaks S and a measure subspace $M \subseteq \mathcal{M}$, $s \in S$ is *prominent* if s is not dominated by any other streak (i.e., s is a skyline streak in S), where the *dominance* relation is based on streaks' lengths and the projections of their minimal value vectors on M. Hence, we use $\lambda_{\{len\} \cup M}(S)$ to denote the set of all prominent streaks in S with regard to M.

Upon arrival of the latest tuple t, FactWatcher discovers $P^t = \{(G, M) \mid G \in \mathcal{G}, M \subseteq \mathcal{M}, \exists\ s \in \lambda_{\{len\} \cup M}(S(P_{t[G]}))$ s.t. $s = \ldots t\}$, where $S(P_{t[G]})$ denotes the set of all streaks in sequence $P_{t[G]}$. In other words, with regard to G and M, the arrival of t establishes one or more new prominent streaks that end at t. Suppose the tuples in Table 4.1 are ordered by their ids. Consider $I = \{$*player*$\}$, $G = \{$*player, team*$\}$ and group g for *player=Lamar Odom, team = Clippers*. Before arrival of t_8, $P_g = t_1t_2t_3$ and thus $S(P_g) = \{t_1, t_2, t_3, t_1t_2, t_2t_3, t_1t_2t_3\}$. The prominent streaks with regard to $M = \{pts, ast\}$ are $\lambda_{\{len,pts,ast\}}\ (S(P_g)) = \{t_1, t_2, t_1t_2t_3\}$. Note that steak $s = t_1t_2$ ($s.len = 2$, $s.\vec{v} = (8, 9)$) is not prominent because it is dominated by streak $s' = t_1t_2t_3$ ($s'.len = 3$, $s'.\vec{v} = (8, 9)$). Upon arrival of t_8, $P_g = t_1t_2t_3t_8$, and the prominent streaks $\lambda_{\{len,pts,ast\}}\ (S(P_g))$ become $\{t_1, t_8, t_1t_2t_3t_8\}$. There are more than one new prominent streaks ending at t_8, corresponding to facts related to t_8.

4.3 User Interface

Figure 4.2 shows FactWatcher's customized GUI for NBA (National Basketball Association) data, where new tuples—players' statistics in individual games—come into the database when a game is over. The GUI's structure is dataset-agnostic as long as the data table is modeled by $R(\mathcal{D}; \mathcal{M})$ as given in Section 4.2. For instance, for data analytics of a social network service, suppose the dimension attributes are $\mathcal{D} = \{$*user age, user city, post type, timestamp*$\}$ and the measure attributes are $\mathcal{M} = \{$*number of likes, number of comments, number of shares*$\}$. FactWatcher finds facts such as "no

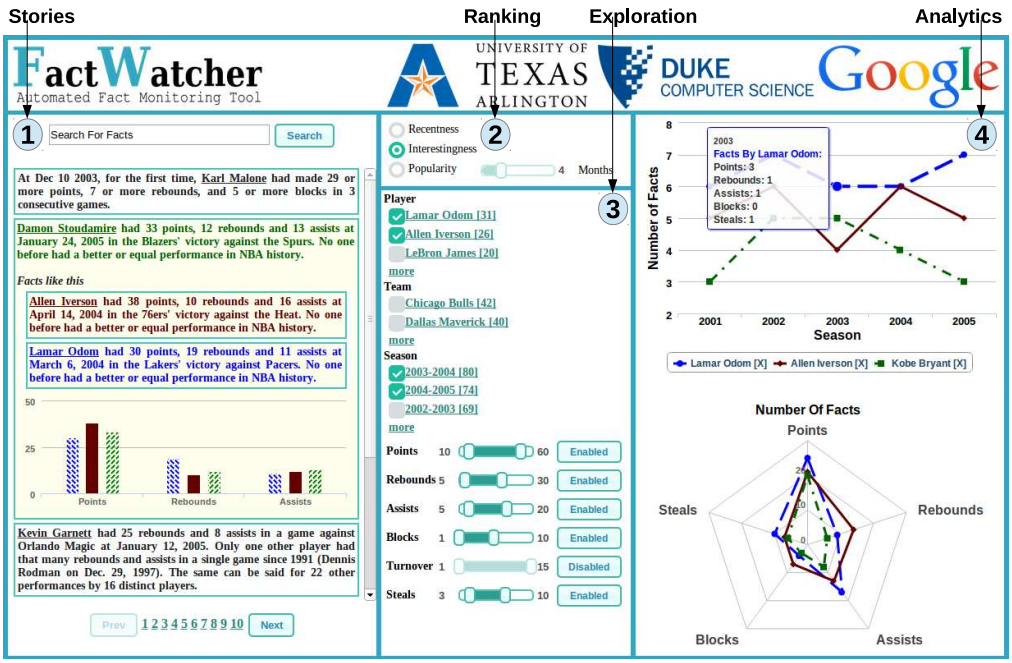


Figure 4.2: FactWatcher User Interface

one in Dallas has posted a photo that gets as many likes, comments and shares." The GUI consists of four areas, as follows.

1. Stories This area shows a ranked list of textual news leads (stories) translated from facts that have ever been discovered and are still valid. It also allows users search for translated stories by keywords. The translation is guided by a set of templates and rules. For example, if t_8 in Table 4.1 triggers a situational fact with regard to constraint-measure pair $(\langle *, \textit{Clippers}, \textit{Lakers}\rangle, \{\textit{pts}, \textit{ast}, \textit{reb}\})$, the story is "Lamar Odom had 10 points, 11 assists and 11 rebounds to become the first Clippers player with a 10/11/11 (points/ assists/ rebounds) game against the Lakers."

If a story is clicked, FactWatcher shows below it more stories for the same constraint-measure pair (C, M) or grouping-measure pair (G, M), as illustrated in Figure 4.2. It also presents bar charts to compare the stories by their values on M.

2. Ranking FactWatcher allows users to choose from several ways of ranking facts and their corresponding stories.

Recentness This default option simply orders facts by their triggering tuples' timestamps.

Popularity This option ranks facts by the frequencies of facts appearing in search results within the last x months, where x can be controlled using a slidebar.

Interestingness This option ranks facts by the elapsed time since their last comparable facts were discovered. Suppose tuple t triggers a new situational fact f_1 with regard to constraint-measure pair (C, M) and a new prominent streak f_2 with regard to grouping-measure pair (G, M). The interestingness of f_1 (f_2) is the elapsed time since the last fact was discovered in (C, M) ((G, M)). The longer the elapsed time is, the more interesting a fact is, since a long elapsed time indicates the fact does not come by easily.

3. Exploration This area presents a faceted interface for exploring the stories. Each facet corresponds to a dimension or measure attribute. Under the facet for a dimension attribute, the attribute values are associated with and ordered by numbers, which indicate how many facts involve the values. For instance, Figure 4.2 shows that there are 31 facts for such (C, M) that the constraint C has a conjunct *player=Lamar Odom*. FactWatcher places a checkbox beside each attribute value. A user can select/unselect the checkboxes across multiple facets. The selected values within one facet correspond to a disjunctive condition, and the disjuncts from different facets form a conjunctive condition. They together correspond to multiple constraints. Each fact (story) displayed in the "stories" area must satisfy one such constraint.

Beside the facet for a measure attribute, FactWatcher presents a slidebar and a button. A user can click the button to enable or disable a measure attribute. Accordingly the "stories" area displays such stories whose corresponding measure subspaces M only involve one or more enabled measure attributes. The user can also use the slidebar to set the minimum and maximum desired values on an enabled

attribute m_i. Accordingly the displayed facts (stories) must have values on m_i within the range.

4. Analytics This area visualizes statistics on facts related to objects selected by a user. The "stories" area highlights the objects (values on object identification attributes) in stories, e.g., *Allen Iverson, Lamar Odom*, etc. in Figure 4.2. When a user clicks on an object, it is added to the object list in the middle of the "analytics" area. The user can remove an object by clicking the crossmark beside it. The top part of the "analytics" area is a line chart, which shows one line per selected object that represents the number of facts (among the displayed facts in the "stories" area) triggered by the object over each time period. When the user hovers the mouse on a data point, a pop-up box shows, for each measure attribute, the number of facts whose measure subspaces contain the measure attribute. The bottom part of this area is a radar chart, which shows one polygon per selected object that represents how many facts triggered by the object are related to each measure attribute.

4.4 Algorithms

Situational fact

In finding situational facts upon the arrival of a new tuple t, a brute-force approach would compare t with all existing tuples to determine whether t is dominated, repeatedly for each constraint C satisfied by t in each measure subspace M. This approach is clearly exhaustive due to comparison with every tuple, for every constraint, and in every measure subspace. The algorithms in FactWatcher respond to these inefficiencies by the following ideas.

Tuple reduction Instead of comparing t with every previous tuple, it is sufficient to only compare with current skyline tuples. This is based on the simple property that, if any tuple dominates t, there must exist a skyline tuple that also dominates t. For

example, in Table 4.1, if the context is the whole table (i.e., no constraint) and the measure subspace M=$\{$*pts, reb*$\}$, the contextual skyline has one tuple—t_1. When the new tuple t_8 comes, with regard to the same constraint-measure pair, it suffices to compare t_8 with t_1 only.

Constraint pruning If new tuple t is dominated by another tuple t' in a measure subspace M, then t does not belong to the contextual skyline of (C, M) for any constraint C satisfied by both t and t'. For example, since t_8 is dominated by t_1 in the full measure space $\mathcal{M}$, it is not in the contextual skylines of ($\langle$*Lamar Odom, Clippers, **$\rangle$, $\{\mathcal{M}\}$), ($\langle$*Lamar Odom, *, **$\rangle$, $\{\mathcal{M}\}$), ($\langle$**, Clippers, **$\rangle$, $\{\mathcal{M}\}$), and ($\langle$**, *, **$\rangle$, $\{\mathcal{M}\}$). Based on this, FactWatcher examines the constraints satisfied by t in a certain order, and comparisons of t with skyline tuples associated with already examined constraints are used to prune remaining constraints.

Sharing computation across measure subspaces FactWatcher considers the full measure space $\mathcal{M}$ first and prunes various constraints for measure subspaces. For instance, after comparing t_8 with t_6 in $\mathcal{M}$, it realizes that t_8 has equal value on *pts* and smaller value on *ast* and thus t_8 is dominated by t_6 in $\{$*pts, ast*$\}$ and $\{$*ast*$\}$ under the constraints satisfied by both tuples.

Based on these ideas, the algorithms in FactWatcher efficiently maintain contextual skylines for all constraint-measure pairs. Upon the arrival of a new tuple t, for all measure subspaces starting from $\mathcal{M}$, constraints satisfied by t (which form a lattice based on subsumption relation between constraints and their corresponding contexts) are visited in either a bottom-up or a top-down order. The new tuple is compared with current skyline tuples associated with the constraints. Various constraints are pruned based on above ideas. Skylines for all constraint-measure pairs are maintained to include t and/or purge current skyline tuples if necessary.

One-of-the-few

While situational facts are about skyline objects, one-of-the-few facts are about top-τ skyband objects. For each constraint-measure pair (C, M), the algorithms maintain the $\hat{k}$-skyband $\lambda_M^{\hat{k}}(\sigma_C(R))$ for such a dynamic $\hat{k}$ that $\lambda_M^{\hat{k}}(\sigma_C(R))$ equals the top-τ skyband $\tau_M(\sigma_C(R))$, i.e., $\hat{k}$=max$\{k \mid \tau \geq |\lambda_M^k(\sigma_C(R))|\}$. A dominated tuple cannot be immediately discarded. Instead, a counter should be maintained to record $\delta_M(\sigma_C(R), t)$ for tuple t. Below is the core idea of maintaining top-τ skyband for a particular (C, M).

Let R' denote the relation after inserting a new tuple t' into relation R. Suppose t' satisfies constraint C. For any $t \in \sigma_C(R)$, t' may increase $\delta_M(\sigma_C(R), t)$ by at most 1. Hence, if $\tau_M(\sigma_C(R))$ is equal to $\lambda_M^{\hat{k}}(\sigma_C(R))$, $\tau_M(\sigma_C(R'))$ must be (i) $\lambda_M^{\hat{k}}(\sigma_C(R'))$, (ii) $\lambda_M^{\hat{k}-1}(\sigma_C(R'))$, or (iii) $\lambda_M^{\hat{k}+1}(\sigma_C(R'))$. To support incremental computation, we maintain $\lambda_M^{\hat{k}+1}(\sigma_C(R))$ (instead of $\lambda_M^{\hat{k}}(\sigma_C(R))$) and compute $\lambda_M^{\hat{k}+1}(\sigma_C(R'))$ from $\lambda_M^{\hat{k}+1}(\sigma_C(R))$. There are two cases, depending on how many tuples in $\sigma_C(R)$ dominate t':

I. $\delta_M(\sigma_C(R), t') \geq \hat{k}$. By definition, $t' \notin \tau_M(\sigma_C(R'))$, as it cannot dominate any tuple $t \in \tau_M(\sigma_C(R))$. In this case, $\tau_M(\sigma_C(R'))= \lambda_M^{\hat{k}}(\sigma_C(R'))=\lambda_M^{\hat{k}}(\sigma_C(R))$.

II. $\delta_M(\sigma_C(R), t') < \hat{k}$. In this case, update $\delta_M(\sigma_C(R'), t)$ for $t \in \lambda_M^{\hat{k}+1}(\sigma_C(R))$, and check if $\hat{k}'$ should be $\hat{k}$, $\hat{k}-1$, or $\hat{k}+1$ for $\tau_M(\sigma_C(R'))$.

Prominent streak

Upon a new tuple t, FactWatcher discovers new prominent streaks for all grouping-measure pairs (G, M). To share computation across different M, a data-cube style data structure and exploration space is adopted. Below we outline the key ideas of how to incrementally monitor prominent streaks for a particular (G, M).

Our solution hinges upon the idea to separate two steps—*candidate streak generation* which generates a small number of candidate streaks ending at the new tuple without exhaustively considering all possible streaks, and *skyline operation*

which maintains a dynamic set of prominent streaks by performing dominance comparison between existing prominent streaks and candidate streaks. The effectiveness of pruning in the first step is critical to overall performance, because execution time of skyline algorithms increases superlinearly by number of candidates.

A brute-force approach to candidate streak generation would enumerate all $\frac{n(n+1)}{2}$ possible streaks in an n-tuple sequence as candidates. We proposed the concept of *local prominent streak* (LPS) for substantially reducing candidate streaks. The intuition is, given a prominent streak s, there cannot be a super-sequence of s whose minimal value vector dominates $s.\vec{v}$. In other words, s must be locally prominent as well. Hence we only need to consider LPSs as candidates, the number of which is at most n—the length of the sequence. The algorithm incrementally maintains possible LPSs while new tuples keep getting appended to the database.

4.5 Usage Scenarios

FactWatcher is currently built upon several datasets, including an NBA dataset and a weather dataset. The NBA dataset has 317,371 tuples of NBA box scores from 1991-2004, on 8 dimension and 7 measure attributes. The weather dataset has 7.8 million daily weather forecast records for 5,365 locations of UK from Dec. 2011 to Nov. 2012. It has 7 dimension attributes and 7 measure attributes. When we explain the usage scenarios below, we refer to the GUI in Figure 4.2 and its corresponding NBA scenario.

Stories When a user visits FactWatcher, FactWatcher shows a list of stories in area "stories" of Figure 4.2. The user enters a keyword query in the search box. The list of stories will be updated. The faceted interface in area "exploration" and the line chart and radar chart in area "analytics" will change accordingly. The user

then clicks a particular story. Similar stories will be shown below it, with bar charts to compare the stories.

Ranking By default, the stories are ordered by recentness. The user explores other ranking schemes by choosing the radio button for *interestingness* or *popularity* in area "ranking". When *popularity* is chosen, the user further uses the slidebar beside it to control the period for assessing popularity of stories.

Exploration The user uses the faceted interface in area "exploration" to explore stories. The user checks *Lamar Odom*, *Allen Iverson* and some others under *player* and *2003-2004*, *2004-2005* under *season*. The area "stories" will show stories related to any of the selected players when they played for or against any team (as she did not select anything under *team*) during *2003-2004* or *2004-2005* season. The user further uses the slidebars for measure attributes to adjust the ranges of values on these attributes. The area "stories" will only show those stories whose measure attribute values do not fall out of the ranges. If the user wants to exclude a measure attribute from the filtering criteria, she can click the button beside its slidebar to disable it, e.g., *Turnover* in Figure 4.2.

Analytics When the user reads the stories, she can click on any underlined objects, i.e., players. After a while, the user has clicked on multiple objects, which are shown in the box in the middle of area "analytics". The top line chart and bottom radar chart in that area visualize the statistics on facts related to these objects, as described in Section 4.3. If the user is not interested in an object anymore, she can remove the object from comparison by clicking the "X" beside the object in the middle box.

CHAPTER 5

Conclusion and Future Plans

In this thesis, we study the problem of discovering exceptional facts about entities in knowledge graphs and discovering prominent streaks in sequence data. Each exceptional fact consists of a pair (*context*, *subspace*). To tackle the challenge of exploring the exponential large search spaces of both contexts and subspaces, we propose a beam search based framework, Maverick, which applies a set of heuristics during the discovery. A prominent streak is a long consecutive subsequence consisting of only large (small) values. We propose efficient methods based on the concept of local prominent streaks (LPS). The experiment results show that the proposed framework Maverick and the LPS based methods are both efficient and effective for discovering exceptional facts and prominent streaks, respectively.

Interesting future work on Maverick can be pursued along several directions on both *efficiency* and *usability*. With regard to *efficiency*, the current system focuses on finding exceptional facts given a specific entity. What is more appealing is a discovery mode in which the system automatically finds facts for all entities. Straightforwardly applying the system on a large knowledge graph will thus lead to exhaustive and repetitive computations for a huge number of entities. Devising algorithms for sharing computations across different entities can significantly increase the system's capability over large knowledge graphs. For example, entities of the same type usually share context-subspace pairs. It is feasible to avoid repetitive computation by materializing the intermediate results such as frequencies of subspace values and contexts. Furthermore, the system currently considers a static knowledge graph

which in reality constantly evolves and grows. To produce up-to-date facts, one has to repeatedly apply the system, which is not practical given the sheer size and change frequency of real-world knowledge bases. Hence, another substantial improvement of the system will be adding incremental exceptional fact discovery algorithms. With regard to *usability*, how to present exceptional facts poses intriguing challenges related to user interface, data visualization, and exceptionality measures. For instance, it is appealing for the system to produce natural language descriptions of the facts. While exceptionality measures such as one-of-the-few may lend itself to simple template-based translations, it is much more challenging to precisely convey facts ranked by more complex measures such as outlyingness and isolation score. Moreover, while our user study helps gain insights into different exceptionality scoring functions, to thoroughly understand their strengths and limitations, a more comprehensive and larger scale user study is worth doing.

Prominent streak discovery provides insightful data patterns for data analysis in many real-world applications and is an enabling technique for computational journalism. Given its real-world usefulness and complexity, the research on prominent streaks in sequence data opens a spectrum of challenging problems. Here we briefly outline several future directions. (1) More general concept of prominent streak can be pursued. For instance, finding conditional prominent streaks is about discovering constraints that make streaks prominent, e.g. "since June 2009" and "the month of July" for the motivating example streaks in Section 3.1. (2) Prominent streaks can be incorporated with the model of data cube [90]. Specifically, given a multi-dimensional sequence, the goal is to discover prominent streaks in not only the full space but also all possible subspaces. For example, given the NBA2 dataset used in our experiments, we may want to find prominent streaks in spaces (points, rebounds), (points, assists, blocks), and so on. (3) When there are many prominent streaks, it is important to

rank them by their interestingness, so that a user can focus on the top-ranked prominent streaks. Some important ranking criteria to consider include streak length, the number of similar prominent streaks in the dataset, values of prominent streaks and so on. FactWatcher has made some attempts to address the challenge. For example, it takes into account the rarity of the values of prominent streaks to compute the interestingness score for a multi-dimensional prominent streak. However, the solution is still less-than-ideal based on our observation.

REFERENCES

[1] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni, "Open information extraction from the web." in *IJCAI*, vol. 7, 2007, pp. 2670–2676.

[2] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "DBpedia: A nucleus for a web of open data," in *ISWC*, 2007.

[3] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *SIGMOD*, 2008, pp. 1247–1250.

[4] D. Vrandečić and M. Krötzsch, "Wikidata: a free collaborative knowledgebase," *Communications of the ACM*, vol. 57, no. 10, pp. 78–85, 2014.

[5] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: a core of semantic knowledge," in *WWW*, 2007, pp. 697–706.

[6] S. Cohen, J. T. Hamilton, and F. Turner, "Computational journalism," *Communications of the ACM*, vol. 54, no. 10, pp. 66–71, Oct. 2011.

[7] S. Cohen, C. Li, J. Yang, and C. Yu, "Computational journalism: A call to arms to database researchers," in *CIDR*, 2011, pp. 148–151.

[8] X. Jiang, C. Li, P. Luo, M. Wang, and Y. Yu, "Prominent streak discovery in sequence data," in *KDD*, 2011, pp. 1280–1288.

[9] Y. Wu, P. K. Agarwal, C. Li, J. Yang, and C. Yu, "On "one of the few" objects," in *KDD*, 2012, pp. 1487–1495.

[10] A. Sultana, N. Hassan, C. Li, J. Yang, and C. Yu, "Incremental discovery of prominent situational facts," in *ICDE*, 2014, pp. 112–123.

[11] G. Zhang, X. Jiang, P. Luo, M. Wang, and C. Li, "Discovering general prominent streaks in sequence data," *TKDD*, vol. 8, no. 2, pp. 9:1–9:37, June 2014.

[12] N. Hassan, A. Sultana, Y. Wu, G. Zhang, C. Li, J. Yang, and C. Yu, "Data in, fact out: Automated monitoring of facts by FactWatcher," *PVLDB, demonstration description*, vol. 7, no. 13, pp. 1557–1560, 2014.

[13] Y. Wu, P. K. Agarwal, C. Li, J. Yang, and C. Yu, "Computational fact checking through query perturbations," *TODS*, vol. 42, no. 1, pp. 4:1–4:41, Jan. 2017.

[14] N. Hassan, B. Adair, J. T. Hamilton, C. Li, M. Tremayne, J. Yang, and C. Yu, "The quest to automate fact-checking," in *Proceedings of the 2015 Computation+Journalism Symposium*, 2015.

[15] N. Hassan, G. Zhang, F. Arslan, J. Caraballo, D. Jimenez, S. Gawsane, S. Hasan, M. Joseph, A. Kulkarni, A. K. Nayak, *et al.*, "Claimbuster: The first-ever end-to-end fact-checking system," *Proceedings of the VLDB Endowment (PVLDB), demonstration description*, vol. 10, no. 7, 2017.

[16] G. Zhang, D. Jimenez, and C. Li, "Discovering exceptional facts from knowledge graphs." in *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2018, p. To appear.

[17] X. Wang and I. Davidson, "Discovering contexts and contextual outliers using random walks in graphs," in *ICDM*, 2009, pp. 1034–1039.

[18] J. Gao, F. Liang, W. Fan, C. Wang, Y. Sun, and J. Han, "On community outliers and their efficient detection in information networks," in *KDD*, 2010, pp. 813–822.

[19] B. Perozzi, L. Akoglu, P. I. Sánchez, and E. Müller, "Focused clustering and outlier detection in large attributed graphs," in *KDD*, 2014.

[20] H. Tong and C.-Y. Lin, "Non-negative residual matrix factorization: problem definition, fast solutions, and applications," *Statistical Analysis and Data Mining*, vol. 5, no. 1, pp. 3–15, 2012.

[21] F. Angiulli, F. Fassetti, and L. Palopoli, "Detecting outlying properties of exceptional objects," *TODS*, vol. 34, no. 1, pp. 7:1–7:62, Apr. 2009.

[22] F. Angiulli, F. Fassetti, G. Manco, and L. Palopoli, "Outlying property detection with numerical attributes," *DMKD*, pp. 1–30, 2016.

[23] T. Wu, D. Xin, Q. Mei, and J. Han, "Promotion analysis in multi-dimensional space," *PVLDB*, vol. 2, no. 1, pp. 109–120, 2009.

[24] L. Duan, G. Tang, J. Pei, J. Bailey, A. Campbell, and C. Tang, "Mining outlying aspects on numeric data," *DMKD*, vol. 29, no. 5, pp. 1116–1151, 2015.

[25] N. X. Vinh, J. Chan, S. Romano, J. Bailey, C. Leckie, K. Ramamohanarao, and J. Pei, "Discovering outlying aspects in large datasets," *DMKD*, vol. 30, no. 6, pp. 1520–1555, 2016.

[26] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee, "Building an efficient RDF store over a relational database," in *SIGMOD*, 2013.

[27] R. Rymon, "Search through systematic set enumeration," *Technical Reports MS-CIS-92-66, Department of Computer and Information Science, University of Pennsylvania*, 1992.

[28] Y. Xu and A. Fern, "On learning linear ranking functions for beam search," in *ICML*, 2007, pp. 1047–1054.

[29] R. Cyganiak, D. Wood, and M. Lanthaler, "Rdf 1.1 concepts and abstract syntax," *W3C Recommendation*, 2014.

[30] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of sparql," *TODS*, vol. 34, no. 3, p. 16, 2009.

[31] S. Harris and A. Seaborne, "Sparql 1.1 query language," *W3C recommendation*, vol. 15, 2013.

[32] L. Geng and H. J. Hamilton, "Interestingness measures for data mining: A survey," *ACM Computing Surveys (CSUR)*, vol. 38, no. 3, p. 9, 2006.

[33] H.-P. Kriegel, P. Kröger, and A. Zimek, "Outlier detection techniques," in *KDD*, 2010.

[34] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," in *NIPS*, 2013, pp. 2787–2795.

[35] Z. Wang, J. Zhang, J. Feng, and Z. Chen, "Knowledge graph embedding by translating on hyperplanes." in *AAAI*, 2014, pp. 1112–1119.

[36] Y. Lin, Z. Liu, M. Sun, Y. Liu, and X. Zhu, "Learning entity and relation embeddings for knowledge graph completion." in *AAAI*, 2015, pp. 2181–2187.

[37] M. Zhou, C. Zhang, X. Han, Y. Ji, Z. Hu, and X. Qiu, "Knowledge graph completion for hyper-relational data," in *International Conference on Big Data Computing and Communications*, 2016, pp. 236–246.

[38] D. Brickley and R. Guha, "Rdf schema 1.1," *W3C Recommendation*, 2014.

[39] M. Dean, G. Schreiber, S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein, "Owl web ontology language reference," *W3C Recommendation*, 2004.

[40] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach (3rd Edition)*. Prentice Hall, 2010.

[41] R. Bisiani, "Beam search," in *Encyclopedia of Artificial Intelligence*, 2nd ed., S. C. Shapiro, Ed. Wiley-Interscience, 1992, pp. 1467–1468.

[42] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *ICDM*. IEEE, 2008, pp. 413–422.

[43] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, "Grami: Frequent subgraph and pattern mining in a single large graph," *Proceedings of the VLDB Endowment*, vol. 7, no. 7, pp. 517–528, 2014.

[44] G. Tsoumakas, I. Katakis, and I. Vlahavas, "Mining multi-label data," in *Data mining and knowledge discovery handbook*. Springer, 2009, pp. 667–685.

[45] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988.

[46] L. Akoglu, M. McGlohon, and C. Faloutsos, "Oddball: Spotting anomalies in weighted graphs," in *PAKDD*, 2010, pp. 410–421.

[47] U. Kang, J.-Y. Lee, D. Koutra, and C. Faloutsos, "Net-ray: Visualizing and mining billion-scale graphs," in *PAKDD*, 2014, pp. 348–361.

[48] I. Trummer, A. Halevy, H. Lee, S. Sarawagi, and R. Gupta, "Mining subjective properties on the web," in *SIGMOD*, 2015, pp. 1745–1760.

[49] W.-K. Wong, "Data mining for early disease outbreak detection," Ph.D. dissertation, Pittsburgh, PA, USA, 2004.

[50] S. Cohen, C. Li, J. Yang, and C. Yu, "Computational journalism: A call to arms to database researchers," in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011, pp. 148–151.

[51] X. Jiang, C. Li, P. Luo, M. Wang, and Y. Yu, "Prominent streak discovery in sequence data," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '11. New York, NY, USA: ACM, 2011, pp. 1280–1288. [Online]. Available: *http://doi.acm.org/10.1145/2020408.2020601*

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

[53] K.-L. Tan, P.-K. Eng, and B. C. Ooi, "Efficient progressive skyline computation," in *Proceedings of the 27th International Conference on Very Large Data Bases*, ser. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 301–310. [Online]. Available: *http://dl.acm.org/citation.cfm?id=645927.672217*

[54] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: an online algorithm for skyline queries," in *Proceedings of the 28th international conference on Very Large Data Bases*, ser. VLDB '02. VLDB Endowment, 2002, pp. 275–286. [Online]. Available: *http://dl.acm.org/citation.cfm?id=1287369.1287394*

[55] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 41–82, Mar. 2005. [Online]. Available: *http://doi.acm.org/10.1145/1061318.1061320*

[56] R. Agrawal, C. Faloutsos, and A. Swami, "Efficient similarity search in sequence databases," pp. 69–84, 1993. [Online]. Available: *http://dx.doi.org/10.1007/3-540-57301-1_5*

[57] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast subsequence matching in time-series databases." College Park, MD, USA: University of Maryland at College Park, 1993.

[58] R. Agrawal, K.-I. Lin, H. S. Sawhney, and K. Shim, "Fast similarity search in the presence of noise, scaling, and translation in time-series databases," in *Proceedings of the 21th International Conference on Very Large Data Bases*, ser. VLDB '95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 490–501. [Online]. Available: *http://dl.acm.org/citation.cfm?id=645921.673155*

[59] B.-K. Yi, H. Jagadish, and C. Faloutsos, "Efficient retrieval of similar time sequences under time warping," in *Data Engineering, 1998. Proceedings., 14th International Conference on*, 1998, pp. 201–208.

[60] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proceedings of the Eleventh International Conference on Data Engineering*, 1995, pp. 3–14.

[61] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," pp. 1–17, 1996. [Online]. Available: *http://dx.doi.org/10.1007/BFb0014140*

[62] M. Zaki, "Spade: An efficient algorithm for mining frequent sequences," *Machine Learning*, vol. 42, no. 1-2, pp. 31–60, 2001. [Online]. Available: *http://dx.doi.org/10.1023/A:1007652502315*

[63] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, "Mining sequential patterns by pattern-growth: The prefixspan approach," *IEEE Trans. on Knowl. and Data Eng.*, vol. 16, no. 11, pp. 1424–1440, Nov. 2004. [Online]. Available: *http://dx.doi.org/10.1109/TKDE.2004.77*

[64] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining closed sequential patterns in large datasets," in *Proceedings of SIAM International Conference on Data Mining*, 2003, pp. 166–177.

[65] P. Smyth *et al.*, "Clustering sequences with hidden markov models." Citeseer, 1997, pp. 648–654.

[66] T. Oates, L. Firoiu, and P. Cohen, "Clustering time series with hidden markov models and dynamic time warping," in *Proceedings of the IJCAI-99 Workshop on Neural, Symbolic and Reinforcement Learning Methods for Sequence Learning*, 1999, pp. 17–21.

[67] T. W. Liao, "Clustering of time series data-a survey," *Pattern Recogn.*, vol. 38, no. 11, pp. 1857–1874, Nov. 2005. [Online]. Available: *http://dx.doi.org/10.1016/j.patcog.2005.01.025*

[68] Y.-I. Shin and D. Fussell, "Parametric kernels for sequence data analysis," in *Proceedings of the 20th international joint conference on Artifical intelligence*, ser. IJCAI'07. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, pp. 1047–1052. [Online]. Available: *http://dl.acm.org/citation.cfm?id=1625275.1625445*

[69] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[70] L. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[71] H. T. Kung, F. Luccio, and F. P. Preparata, "On finding the maxima of a set of vectors," *J. ACM*, vol. 22, no. 4, pp. 469–476, Oct. 1975. [Online]. Available: *http://doi.acm.org/10.1145/321906.321910*

[72] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting," in *Data Engineering, 2003. Proceedings. 19th International Conference on*, 2003, pp. 717–719.

[73] J. Pei, Y. Yuan, X. Lin, W. Jin, M. Ester, Q. Liu, W. Wang, Y. Tao, J. X. Yu, and Q. Zhang, "Towards multidimensional subspace skyline analysis," *ACM Trans. Database Syst.*, vol. 31, no. 4, pp. 1335–1381, Dec. 2006. [Online]. Available: *http://doi.acm.org/10.1145/1189769.1189774*

[74] Y. Tao, X. Xiao, and J. Pei, "Subsky: Efficient computation of skylines in subspaces," in *Proceedings of the 22nd International Conference on Data Engineering*, ser. ICDE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 65–. [Online]. Available: *http://dx.doi.org/10.1109/ICDE.2006.149*

[75] Z. Zhang, X. Guo, H. Lu, A. K. H. Tung, and N. Wang, "Discovering strong skyline points in high dimensional spaces," in *Proceedings of the 14th ACM international conference on Information and knowledge management*, ser. CIKM '05. New York, NY, USA: ACM, 2005, pp. 247–248. [Online]. Available: *http://doi.acm.org/10.1145/1099554.1099610*

[76] C.-Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang, "On high dimensional skylines," in *Proceedings of the 10th international conference on Advances in Database Technology*, ser. EDBT'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 478–495. [Online]. Available: *http://dx.doi.org/10.1007/11687238_30*

[77] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang, "Selecting stars: The k most representative skyline operator," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, 2007, pp. 86–95.

[78] Y. Tao, L. Ding, X. Lin, and J. Pei, "Distance-based representative skyline," in *Proceedings of the 2009 IEEE International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 892–903. [Online]. Available: *http://portal.acm.org/citation.cfm?id=1546683.1547325*

[79] T. Xia and D. Zhang, "Refreshing the sky: the compressed skycube with efficient support for frequent updates," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '06. New York, NY, USA: ACM, 2006, pp. 491–502. [Online]. Available: *http://doi.acm.org/10.1145/1142473.1142529*

[80] B. Jiang and J. Pei, "Online interval skyline queries on time series," in *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ser. ICDE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1036–1047. [Online]. Available: *http://dx.doi.org/10.1109/ICDE.2009.70*

[81] M. Wang and X. S. Wang, "Finding the plateau in an aggregated time series," in *Proceedings of the 7th international conference on Advances in Web-Age Information Management*, ser. WAIM '06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 325–336. [Online]. Available: *http://dx.doi.org/10.1007/11775300_28*

[82] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sept. 1975. [Online]. Available: *http://doi.acm.org/10.1145/361002.361007*

[83] ——, "Multidimensional binary search trees in database applications," *Software Engineering, IEEE Transactions on*, vol. SE-5, no. 4, pp. 333–340, 1979.

[84] S. Cohen, J. T. Hamilton, and F. Turner, "Computational journalism," *Commun. ACM*, vol. 54, no. 10, pp. 66–71, October 2011.

[85] S. Cohen, C. Li, J. Yang, and C. Yu, "Computational journalism: A call to arms to database researchers." in *CIDR*, 2011, pp. 148–151.

[86] X. Jiang, C. Li, P. Luo, M. Wang, and Y. Yu, "Prominent streak discovery in sequence data," in *KDD*, 2011, pp. 1280–1288.

[87] Y. Wu, P. K. Agarwal, C. Li, J. Yang, and C. Yu, "On one of the few objects," in *KDD*, 2012, pp. 1487–1495.

[88] A. Sultana, N. Hassan, C. Li, J. Yang, and C. Yu, "Incremental discovery of prominent situational facts," in *ICDE*, 2014, pp. 112–123.

[89] G. Zhang, X. Jiang, P. Luo, M. Wang, and C. Li, "Discovering general prominent streaks in sequence data," *ACM TKDD*, vol. 8, no. 2, pp. 9:1–9:37, June 2014.

[90] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Min. Knowl. Discov.*, vol. 1, no. 1, pp. 29–53, Jan. 1997. [Online]. Available: *http://dx.doi.org/10.1023/A:1009726021843*