CASM:
SEARCHING CONTEXT-AWARE SEQUENTIAL PATTERNS ITERATIVELY

BY
HENGZHI ZHONG

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:
Associate Professor Kevin Chen-Chuan Chang
Abstract

Many applications are interested in mining context-aware sequential patterns such as opinions, common navigation patterns, and product recommendations. However, traditional sequential pattern mining algorithms are not effective to mine such patterns. We thus study the problem of searching context-aware patterns on the fly. As a solution, we presented a variable-order random walk as the ranking model and developed two efficient algorithms GraphCAP and R³CAP. To show the effectiveness and efficiency of our solution, we conducted extensive experiments on real dataset. Lastly, we applied our solution to support opinion search, a novel application that significantly differs from traditional opinion mining and retrieval.
Acknowledgements

This project would not have been possible without the support of many people. Many thanks to my adviser, Kevin Chen-Chuan Chang, who read my numerous revisions and helped make some sense of the confusion. Also, thanks to Kevin and the University of Illinois for providing me with the financial means to complete this project. And finally, thanks to my parents and numerous friends who endured this long process with me, always offering support and love.
# Table of Contents

List of Tables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . v
List of Figures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . vi
List of Abbreviations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . vii
Chapter 1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
Chapter 2 Problem Definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
Chapter 3 Ranking Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
Chapter 4 Computation Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
  4.1 GraphCAP for Context-Aware Patterns . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
  4.2 Improvement: R³CAP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
  4.3 Complexity Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
Chapter 5 Handle Multiple Items . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
Chapter 6 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
Chapter 7 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
  7.1 Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
  7.2 Effectiveness Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
  7.3 Efficiency Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
Chapter 8 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
Chapter 9 Figures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
List of Tables

4.1 Time complexity summary. .............................................................. 12
7.1 A snapshot of laptop queries for App1. ........................................... 19
7.2 Top 5 patterns retrieved for App1. .................................................. 20
7.3 Average memory usage for each query. ........................................... 22
**List of Figures**

9.1 Example sequences. ................................................................. 25
9.2 Example context graphs. .......................................................... 26
9.3 Algorithm GraphCAP. .............................................................. 27
9.4 Reverse graph traversal. ......................................................... 27
9.5 Algorithm R³CAP. ................................................................. 28
9.6 Context graph $G_a$ with multi-item itemsets. .......................... 29
9.7 Algorithm MultiR³CAP extends R³CAP. .................................... 29
9.8 Experiment Results. ............................................................... 30
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>App1</td>
<td>Opinion Search.</td>
</tr>
<tr>
<td>App2</td>
<td>Navigation Analysis.</td>
</tr>
<tr>
<td>App3</td>
<td>Product Recommendation.</td>
</tr>
<tr>
<td>SPM</td>
<td>Sequential Pattern Mining.</td>
</tr>
<tr>
<td>CASM</td>
<td>Context-aware Sequence Mining.</td>
</tr>
<tr>
<td>MC</td>
<td>Markov Chain</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Many applications require mining sequential patterns. A sequential pattern is a sequence of itemsets (from now on, we refer sequential patterns as patterns). However, traditional sequential pattern mining algorithms are not effective when applications want to mine patterns that are context-aware. We illustrate some of these applications here:

Application 1 (App1): Opinion Search. As consumers often use product opinions to make purchasing decisions, App1 supports consumers to search opinions on any topic over a set of product reviews. For example, what are the opinions on HP’s battery life? Given pattern battery life as query and HP’s product reviews as sequences, App1 mines the sequential patterns that are most likely opinions, e.g. battery life...long and battery life...lasts.

Application 2 (App2): Navigation Analysis. In a product site, a business analyst assesses the importance of certain pages in the site. For example, what are the most likely pages users would visit after they have visited the FAQ page? Given FAQ as query and the site’s clickstreams as sequences, App2 mines the most likely navigation patterns, e.g. FAQ...Contact Us.

Application 3 (App3): Product Recommendation. The analyst also wants to recommend products to users. For example, what products should the analyst recommend next, given users have purchased laptop? Given laptop as query and users’ purchases as sequences, App3 mines most likely patterns as recommendation, e.g. laptop...battery and laptop...usb.

App1-3 are interested in context-aware patterns. Context-aware patterns are expanded and scored with respect to the query’s context. Given some pattern \( \alpha \) as query, e.g. battery life, FAQ, and laptop, App1-3 mine relevant patterns, e.g. battery life...long, FAQ...Contact Us, and laptop...battery. These patterns are expanded from \( \alpha \) by adding some item \( u \) to \( \alpha \).

To score which patterns are relevant to \( \alpha \), App1-3 examine \( \alpha \)’s context. As \( \alpha \) connects different items in various ways, these items affect each other’s dependence on \( \alpha \). For examples,

- to score which patterns are most likely opinions, e.g. battery life...long, App1 examines battery life’s context, i.e. the sequence of item connections between battery life and long. These connections indicate long is dependent (see \( s_1-s_4 \) in Figure 9.1) or independent (see \( s_7 \) in Figure 9.1) of battery life.

1The “...” notation separates itemsets in a pattern.
to score which patterns are most likely navigated, e.g. FAQ... Contact Us, App2 examines FAQ’s context, i.e. the sequence of intermediate pages users visited between FAQ and Contact Us. These intermediate pages indicate Contact Us’s dependence (or independence) on FAQ. Suppose sitemap is one of the intermediate pages. As sitemap contains direct links to all the pages, it’s possible users visit Contact Us independent of FAQ.

Just like App1-2, to score which patterns are most likely recommendations, e.g. laptop...battery, App3 examines laptop’s context, i.e. the purchases between laptop and battery. These intermediate purchases indicate whether or not battery is independent of laptop. Suppose camera is an intermediate purchase. If many users purchased battery after they have purchased camera, it’s likely battery is independent of laptop.

As these examples show, α’s connections to various items affect the dependence of the added item u on α, e.g. short affects long’s dependence on battery life, sitemap affects Contact Us’s dependence on FAQ, and camera affects battery’s dependence on laptop. As α’s context, these connections is a crucial indicator of dependence. Based on this context, App1-3 determines pattern relevancy.

As context-aware patterns are scored based on α’s context, traditional sequential pattern mining (SPM) cannot effectively mine them. Typically, as SPM mines frequent patterns, i.e. patterns whose occurrence is over a threshold, it ignores how items are connected within the patterns. Thus, SPM will filter out all the infrequent patterns even if their added item truly depends on α. For example, as very few reviewers voice the opinion battery life...long, SPM does not mine it as result. Thus, SPM produces a low recall for context-aware patterns. SPM also produces a low precision. For example, as reviewers often discuss battery life and screen in their reviews, battery life...screen is a frequent pattern and SPM outputs it as an opinion. Obviously, screen does not depend on battery life.

Thus, to support applications such as App1-3, we propose to search context-aware sequential patterns.

### Our Proposal

To search context-aware patterns on the fly, we propose a context-aware sequential pattern mining system. As input, users formulate query \( Q = \alpha \) to describe the patterns they are interested in. For example, to search the opinions on HP’s battery life, the query is \( Q_1 = \text{battery life} \).

As output, the context-aware sequential pattern system returns a ranked list of context-aware patterns. As these patterns are ranked by their relevance to the query, we also call them relevant patterns. The scoring function uses \( \alpha \)’s context to assess pattern relevancy.

As there are many relevant patterns of various length, it’s more efficient for users to explore the result space iteratively, based on their interest. For example, given query \( Q_1 \), the system returns patterns battery life...long and battery life...good. If users aren’t interested in battery life...good, the system shouldn’t waste any computation re-
sources to return longer-length patterns about battery life...good, e.g. battery life...good...all the time. Thus, given \(Q=\alpha\), each relevant pattern \(\beta\) is a length-(\(l+1\)) pattern expanded from \(\alpha\). If users are interested in exploring \(\beta\) further, they formulate a new query \(Q'=\beta\).

**Core Challenges**

For a context-aware sequential pattern mining system to be effective and efficient, there are several challenges to address:

1. As a retrieval mechanism, how can the system discover the context-aware patterns that are relevant to \(\alpha\)?
2. As a ranking mechanism, how can the system score these patterns in a principled way? As App1-3 motivate, we need a context-aware scoring function of a conditional nature. This conditional nature is similar to integrating confidence, a metric in association rule mining, into pattern mining. As context is a crucial indicator of item dependence, the scoring function must incorporate it in the conditional to assess pattern relevancy.
3. As an online search system, how can the system efficiently score patterns? As context-aware patterns differ from frequent patterns, they mandate a new computation model.

For the rest of the thesis, we address these challenges.
Chapter 2

Problem Definition

As Chapter 1 states, a context-aware sequential pattern mining system mines context-aware patterns. Thus, as a SPM problem defines, these patterns are mined from a sequence database $S$ where each sequence is an ordered list of transactions. A transaction is essentially an itemset. An itemset is a non-empty set of items. A pattern is a sequence of itemsets. We denote items and patterns with San Arif font.

For example, as App1 mines opinions, a sequence is a tokenized product review sentence. Each item is a keyword or phrase and an itemset consists of one item. Figure 9.1 shows an example. In this figure, within $<$ is a tokenized HP product review sentence or sequence. Items are underlined. Each letter denotes an item. As App2 mines navigation patterns, each item is a webpage and a sequence is the list of webpages visited by the same user, ordered by their visited time. As App3 mines purchase patterns, each item is a product and a sequence is all the products purchased by the same user, ordered by their purchase time.

As App1-3 illustrate, given some pattern $\alpha$ as query, context-aware patterns are patterns expanded from $\alpha$ by adding some itemset $u$ to $\alpha$. These patterns have three expansion forms: in a sequence $s_i \in S$,

- if $u$ appears after $\alpha$, $u$ is in $\alpha$’s forward context and forms pattern $\beta = \alpha \ldots u$;
- if $u$ appears before $\alpha$, $u$ is in $\alpha$’s backward context and forms pattern $\beta = u \ldots \alpha$;
- if $\alpha$ contains multiple itemsets, e.g. $t_1 \ldots t_2$, and $u$ appears within $\alpha$, $u$ is in $\alpha$’s in-between context and forms pattern $\beta = t_1 \ldots u \ldots t_2$.

For the rest of the thesis, we focus on mining $\alpha$’s forward context. The same techniques apply for $\alpha$’s backward and in-between context.

Context-aware patterns are scored with respect to $\alpha$’s context. Formally, we define $\alpha$’s context as a set of tuples $(u, C)$ where $C$ is the list of connections from $\alpha$ to $u$. For example, suppose $\alpha = a$. Its context is $\{(b, \{ab\}), (c, \{abc, ac, adbijc, aqc\}), \ldots\}$ (Figure 9.1). As Chapter 1 illustrates, this context indicate how likely various items depend on $\alpha$.

To quantify the ease of connectivity between items, we assume we are given a function $W: E \rightarrow \Omega$ where $E$ is the set of connections between two items and $\Omega \in [0, 1.0]$ is the set of connectivity values. As different applications are interested in different kinds of connections, $W$ is application specific.
We now formally define the problem, *Context-Aware Sequence Mining* (CASM). Given a sequence database $S$ and a weight function $W$, the input is query $Q=\alpha$ where $\alpha$ is a pattern of length-$l$ (i.e. $\alpha$ has $l$ itemsets). To focus on $\alpha$, patterns are mined from $S_\alpha$, the set of sequences that contain $\alpha$. Note that $S_\alpha \subseteq S$. The output of CASM is a list of length-$(l+1)$ context-aware patterns ordered by their relevance score. Let $Score$ be a pattern’s relevance score. Suppose one pattern is $\beta=\alpha \ldots u$. Its $Score$ indicates how easily $\alpha$ connects item $u$, given $\alpha$'s context in $S_\alpha$. If $u$ most likely depends on $\alpha$, $u$ easily connects $\alpha$. Thus, $\beta$ has a large $Score$ and is highly relevant to the query $Q=\alpha$.

As Chapter 1 proposes, CASM should be iterative. So after the initial search, if users are interested in some pattern $\beta=\alpha \ldots u$, they can explore it further by formulating a new query $Q'=\beta$. For the rest of the thesis, we refer the initial search as the first iteration and pattern explorations as subsequent iterations. Also, we refer the input pattern in the first iteration as the *query pattern*, the mined patterns as *expanded patterns*, and the input pattern in subsequent iterations as *exploring pattern*.

Given CASM as defined, we concretely materialize $Score$ in Chapter 3 and discuss how to compute it efficiently in Chapter 4 for itemsets that only contain one item. We generalize our solution for multiple items in Chapter 5.
Chapter 3

Ranking Model

In this chapter, we present CASM’s ranking model, assuming each itemset contains only one item. Thus, unless otherwise stated, we refer both item and itemset as item in this section.

As Chapter 2 motivates, the goal of CASM’s ranking model is to discover which items connect the input pattern $\alpha$ easiest and how easy they connect. Conceptually, items are connected in various ways in $S_\alpha$ such that their connections affect each other’s connectivity to $\alpha$. If some items truly depend on $\alpha$, other items’ connections will not take them away from $\alpha$. In other words, they easily connect $\alpha$. For example, battery life connects long easier than other items, e.g. screen, because there are more ways for battery life to connect long and most of their connections make sense (see Figure 9.1). Essentially, various item connections in $S_\alpha$ form a “network.” We refer this network as $G_\alpha$ and the connectivity from $\alpha$ to some item $u$ as reachability. As the ranking model evaluates the reachability from $\alpha$ to various items, the objective of this chapter is to formalize the ranking model as reachability evaluation on network $G_\alpha$.

Toward this objective, we formalize the reachability evaluation as a random walk on the network $G_\alpha$ from the source $\alpha$, to a set of sinks, i.e. $\alpha$’s reachable items. Under this framework, the random walk on $G_\alpha$ evaluates reachability as a Markov Chain (MC).

To evaluate the reachability from $\alpha$ to various items, the key lies with the construction of $G_\alpha$. As $G_\alpha$ is a network of item connections and $\alpha$’s context is defined as a set of item connections, we define $G_\alpha$ as $\alpha$’s context graph. For explanation clarity, we refer a sequence of item connections in $S_\alpha$ as a substring and in $G_\alpha$ as a path. To evaluate reachability accurately, $G_\alpha$ needs to capture the substrings in $S_\alpha$ such that they have an 1-to-1 mapping with the paths in $G_\alpha$.

Naively, we construct $G_\alpha$ as followed. Each itemset is a node. As we assume each itemset only contains one item, this is the same as saying each item is a node. Substrings are essentially paths in $G_\alpha$. To map the substrings into paths, we examine each sequence in $S_\alpha$. Suppose we are examining sequence $s_1$ (Figure 9.1). Starting from item $a$, we examine the following item $b$ and use the given weight function $W$ to evaluate how likely $b$ connects $a$. As App1 is interested in opinions, $W$ evaluates how likely $b$ is semantically related to $a$. As $a$ and $b$ are semantically related in $s_1$, $W$ outputs 1. As the connection from $a$ to $b$, i.e. substring $ab$, is the edge $ab$ in $G_\alpha$, the output 1 is $ab$’s edge weight. Next, we examine the following item $c$ and evaluate about how likely $c$ connects $b$. We continue our examination
process until there are no more items to examine in \(s_1\). We then examine another sequence. Figure 9.2(a) shows the mapped substrings after we have examined \(s_1\) and \(s_2\). After we have examined \(S_\alpha\), we have constructed \(G_\alpha\).

Our mapping process describes a Markovian process. In general, a MC of order \(k\) is defined as \(p(X_t=x_t|X_{t-1}=x_{t-1},...,X_{t-k}=x_{k-1})\) where \(X_t\) is a random variable and \(x_t\) is \(X_t\)'s realization. \(X_{t-1},...,X_{t-k}\) is \(X_t\)'s length-\(k\) history. If \(k\) is fixed, e.g. \(k=1\) always, MC is fixed \(k\)th-order; if \(k\) varies, MC is variable \(k\)th-order. As our naive mapping examines each connection to some item \(u\) based on \(u\)'s length-1 item, it describes a fixed first-order MC.

Our naive construction doesn’t produce an 1-to-1 mapping between substrings and paths. By following edges, path \(abijc\) exists in Figure 9.2(a) but its corresponding substring \(abijc\) doesn’t exist in \(s_1\) and \(s_2\). This is because a first-order MC doesn’t depend on longer-length history. As a result, substrings with length greater than 2 are not mapped accurately.

To map substrings of various length to paths accurately, our mapping process must describe a variable \(k\)th-order MC. As history, each item must remember the entire substring from \(\alpha\) to it. Correspondingly, each node in \(G_\alpha\) must remember all the paths that reach it from \(\alpha\). Given each unique path \(p\) that reaches node \(u\), \(u\) reaches next node \(v\) only if \(v\) can be reached via path \(pu\). The length of \(p\) varies.

Concretely, we construct \(G_\alpha\) as followed (see Figure 9.2(b)-9.2(c) for examples): each item in \(S_\alpha\) has multiple node instances in \(G_\alpha\) and each instance captures a unique path that reaches it. We use a superscript to denote each instance. So as item \(j\) appears in substrings \(\{abcj, abckij, ackij, abij\}\), it has nodes \(\{j^1, j^2, j^3, j^4\}\) reached via paths \(\{ab^1c^1j^1, ab^1c^3k^1i^1j^2, ac^2k^2i^2j^3, ad^1b^2i^3j^4\}\). When we refer nodes in a general way, we will not use the superscript.

As patterns are sequences of itemsets, they also have multiple instances. The substrings within the input pattern need to be captured as different nodes follow different pattern instances (see the blue-colored nodes in Figure 9.2(c)). An edge from node \(u\) to \(v\) indicates there’s a substring where \(u\) appears immediately before \(v\). Each edge has a weight. The given weight function \(W\) evaluates the weights.

Note that in subsequent iterations, their context graphs are “subgraphs” of their query pattern’s context graph. Suppose the query pattern, i.e. the input pattern of the first iteration, is \(\alpha\) and an exploring pattern, i.e. the input pattern in the \(n\)th iteration, is \(\beta=\alpha...u\). As \(\beta\)'s reachable items is a subset of \(\alpha\)'s reachable items, so are their reachable nodes (see the red-colored nodes in Figure 9.2(b) and 9.2(c)). In other words, the nodes and paths that contribute to the reachability scoring of \(\beta\) are a subset of the nodes and paths in \(G_\alpha\).

Under the random walk framework, reachability is a product of transition probabilities on the graph \(G_\alpha\). Suppose there is a path \(w_0w_1...w_{k-1}w_k\) where \(\alpha\) and \(u^i\) map to nodes \(w_0\) and \(w_k\) respectively. The reachability from node \(w_0\) to node \(w_k\) is

\[
p_r(w_k|w_0, G_\alpha) = \prod_{i=0}^{k-1} p_i(w_{i+1}|w_i, G_\alpha), \tag{3.1}
\]
where $p_t(w_{i+1}|w_i, G_\alpha)$ is the transition probability from node $w_i$ to $w_{i+1}$. Note that $p_r(w_i|w_i, G_\alpha)=1$.

The transition probability from node $w_i$ to $w_{i+1}$ is

$$p_t(w_{i+1}|w_i, G_\alpha) = \frac{\sum_{e \in E(w_i, w_{i+1})} W(e)}{\sum_{z \in \text{Out}(w_i)} \sum_{e \in E(w_i, z)} W(e)},$$

(3.2)

where $W(e)$ is the edge weight evaluated by the given weight function $W$, $E(w_i, w_{i+1})$ is the set of edges from $w_i$ to $w_{i+1}$, and $\text{Out}(w_i)$ is the set of $w_i$’s outgoing nodes, i.e. children.

Given reachability as defined, we define a pattern’s relevance score $\text{Score}$. Let pattern $\beta=\alpha...u$. As $\beta$’s $\text{Score}$ assesses how likely $\alpha$ reaches item $u$, it is the aggregated reachability from $\alpha$ to all of item $u$’s instances. Thus, $\beta$’s $\text{Score}$ is

$$\text{Score}(\beta|\alpha) = \sum_{u^i \in \text{u’s node instances}} p_r(u^i|\alpha, G_\alpha).$$

(3.3)

For example, $\text{Score}(a...j|a)=\sum_{j^i \in \{j^1,...,j^4\}} p_r(j^i|a, G_a)$. After every item from $\alpha$ is scored thusly, some items will emerge with a large $\text{Score}$, indicating they are more likely to be reached from $\alpha$ and thus, the patterns expanded with them are more relevant.

In the next chapter, we discuss how to compute $\text{Score}$ efficiently.
Chapter 4

Computation Model

In this chapter, we present CASM’s computation model, assuming each itemset contains only one item.

4.1 GraphCAP for Context-Aware Patterns

Naively, as context-aware patterns are sequential, we adapt the standard SPM algorithm PrefixSpan [12] to search context-aware patterns. PrefixSpan is iterative. In each iteration, given pattern $\alpha$ of length-$l$ and threshold $\text{min}_{\text{supp}}$, PrefixSpan mines frequent patterns of length-($l+1$) as followed:

1. Retrieve $S_{\alpha}$, the sequences that contain $\alpha$
2. Scan $S_{\alpha}$ to find length-1 items that appear after $\alpha$
3. FOR each such item $u$
4. Create $\beta = \alpha \ldots u$
5. IF $u$’s support $\geq \text{min}_{\text{supp}}$
6. Output $\beta$ as a frequent pattern

Line 5 applies the frequency threshold to score frequent patterns. To adapt PrefixSpan for context-aware patterns, it needs to score patterns with $\text{Score}$ instead of the frequency threshold.

PrefixSpan is inefficient for context-aware patterns. Like many SPM algorithms, PrefixSpan counts and stores each pattern’s frequency as it scans sequences. However, to score patterns with $\text{Score}$, PrefixSpan can’t simply count pattern frequency. When it scans $S_{\alpha}$, it must remember all the items reachable from $\alpha$ and for each item $u$, remember the substrings from $\alpha$ to $u$. This is a lot to remember and such bookkeeping is nontrivial (see Chapter 4.3).

As Chapter 3 states, patterns are scored based on their reachability on a context graph. Thus, to reduce the bookkeeping cost and facilitate pattern scoring, we propose GraphCAP, a CASM algorithm that actually materializes the context graph (Figure 9.3). Given query $Q=\alpha$, GraphCAP constructs the context graph $G_{\alpha}$ as it scans $S_{\alpha}$ (lines 1-3). Chapter 3 describes the construction process. Note that lines 1-3 are only executed in the first iteration. Once $G_{\alpha}$ is constructed, GraphCAP traverses it from $\alpha$ “top-down” to discover $\alpha$’s reachable nodes and score their reachability. Line 5 adds $\alpha$’s children to Queue. For example, suppose $\alpha=a\ldots j$. GraphCAP locates all of item $j$’s node instances
\{j^1, j^2, j^3, j^4\} and adds all their children to Queue. As different input pattern has different context, GraphCAP traverses \(G_\alpha\) per iteration.

### 4.2 Improvement: R³CAP

There are opportunities to improve GraphCAP. One, in each iteration, just as PrefixSpan scans sequences repeatedly, GraphCAP traverses the context graph \(G_\alpha\) repeatedly. This is quite expensive to be done online when users explore many patterns. Two, GraphCAP doesn’t score patterns efficiently. It recomputes reachability (Figure 9.3 line 9). In this chapter, we discuss how to capture these opportunities.

Toward this objective, we examine the reachability definition (Equation 3.1). We observe that paths can be decomposed into their prefix and suffix parts. Based on this observation, we can rewrite reachability in terms of its prefix and suffix reachability. Let path \(p\) be \(x...u...v\) and \(x, u,\) and \(v\) are nodes along this path. Suppose \(p\) is decomposed into prefix \(x...u\) and suffix \(u...v\),

\[ p_r(v|x...u, G_\alpha) = p_r(u|x, G_\alpha)p_r(v|u, G_\alpha), \]  

(4.1)

where \(p_r(u|x, G_\alpha)\) is \(u\)’s prefix reachability and \(p_r(v|u, G_\alpha)\) is \(u\)’s suffix reachability. Note that Equation 4.1 and Equation 3.1 are equivalent.

A node’s prefix and suffix reachabilities are relative to the node. That is, \(u\)’s prefix reachability \(p_r(u|x, G_\alpha)\) is also \(x\)’s suffix reachability and \(u\)’s suffix reachability \(p_r(v|u, G_\alpha)\) is also \(v\)’s prefix reachability. As a node has multiple ancestors and descendants, it has multiple prefix and suffix reachabilities. For example, as node \(c^1\) has ancestors \(\{a, b^1\}\), it has prefix reachabilities \(\{p_r(c^1|a, G_a), p_r(c^1|b^1, G_a)\}\); as \(c^1\) has descendants \(\{j^1, g^1, k^1, h^1, ...\}\), it has suffix reachabilities \(\{p_r(j^1|c^1, G_a), p_r(g^1|c^1, G_a), ...\}\).

Given Equation 4.1, we rewrite \(Score\) for subsequent iterations. Suppose \(\beta = \alpha...u\) and its expanded pattern \(\gamma = \beta...v\),

\[ Score(\gamma | \beta) = \sum_{u^j} \sum_{v^i} p_r(u^j|\alpha, G_\alpha)p_r(v^i|u^j, G_\alpha), \]  

(4.2)

where \(u^i\) is the node instance of the added item \(u\) in \(\beta\).

Equation 4.2 implies we can reuse reachability computations. By Equation 4.2, a node’s prefix and suffix reachability are part of its \(Score\). Thus, after the context graph \(G_\alpha\) is built and traversed in the first iteration, each node’s prefix and suffix reachability is computed. We can reuse them in subsequent iterations.

Concretely, suppose the query pattern, i.e. the input pattern of the first iteration, is \(\alpha\) and one of \(\alpha\)’s ex-
panded patterns is \( \beta = \alpha \ldots u \). Also, suppose one of \( \beta \)'s expanded patterns is \( \gamma = \beta \ldots v \). By Equation 4.2, \( \text{Score}(\gamma | \beta) = \sum_{u'} \sum_{v} p_r(u' | \alpha, G_\alpha) p_r(v | u', G_\alpha) \).

**Reuse a node’s prefix reachability:** As \( u' \)'s prefix reachability \( p_r(u' | \alpha, G_\alpha) \) is already computed in the first iteration, we can output it along with the result \( \text{List} \) tuple \((\beta, \text{Score}(\beta | \alpha))\). When \( \beta \) is the input pattern in the \( nth \) iteration, the prefix reachability \( p_r(u' | \alpha, G_\alpha) \) can then be given as part of the input and we can use it to compute \( \text{Score}(\gamma | \beta) \).

In other words, there’s no need to recompute \( p_r(u' | \alpha, G_\alpha) \).

**Reuse a node’s suffix reachability:** as \( p_r(u' | \alpha, G_\alpha) \) is provided in the input, only the suffix reachability \( p_r(v | u', G_\alpha) \) needs to be computed. As \( p_r(v | u', G_\alpha) \) is already computed in the first iteration, if we store it in \( G_\alpha \) somehow, we can reuse it to compute the suffix part of \( \text{Score}(\gamma | \beta) \). In other words, there’s no need to recompute \( p_r(v | u', G_\alpha) \) either.

The question is how to store suffix reachabilities. We observe that there are two reasons to traverse context graphs: one reason is to discover the reachable nodes; the other is to score their reachability. If at each node \( u \), we store \( u' \)'s reachable nodes and their reachability from \( u \), i.e. \( u' \)'s suffix reachabilities, we do not need to traverse context graphs anymore.

Thus, at each node \( u \), we propose to store its reachable nodes and suffix reachabilities. Specifically, each node \( u \) has a \( u.rTable \). \( u.rTable \) is a list of 2-argument tuples \((v, p_r(v | u, G_\alpha))\) where \( v \) is a reachable node from \( u \) and \( p_r(v | u, G_\alpha) \) is \( u' \)'s suffix reachability. Given this storage, instead of traversing \( G_\alpha \) and computing reachabilities from scratch per iteration, we can simply look up each \( u.rTable \) (we discuss the details later). As a result, we’ve improved pattern exploration and scoring.

Given the rTables as defined, the next question is how to populate them. To answer this question, we observe that a node’s reachable nodes is the union of its children’s reachable nodes, i.e.

\[
\text{u's reachable nodes} = \bigcup_{v \in \text{u's children}} \text{v's reachable nodes}.
\] (4.3)

As Equation 4.3 implies, we should populate the rTables by traversing \( G_\alpha \) “bottom-up” or in reverse. Specifically, we traverse \( G_\alpha \) from its leaf nodes to \( \alpha \) in the first iteration. For each node \( u \) visited, we populate \( u.rTable \) from its children’s rTables. Figure 9.4 shows an example. When we traverse from the leaf node \( j^1 \) and reach \( c^1 \), we add a tuple \((j^1, p_r(j^1 | c^1, G_\alpha))\) to \( c^1.rTable \). Next we reach \( b^1 \) and add \((c^1, p_r(c^1 | b^1, G_\alpha))\) to \( b^1.rTable \). As \( c^1 \)'s reachable nodes are also \( b^1 \)'s reachable nodes, we add them to \( b^1.rTable \) and compute the suffix reachabilities from \( b^1 \) to these nodes, e.g. \( p_r(j^1 | b^1, G_\alpha) \). By Equation 4.1, \( p_r(j^1 | b^1, G_\alpha) = p_r(c^1 | b^1, G_\alpha) p_r(j^1 | c^1, G_\alpha) \). As \( p_r(j^1 | c^1, G_\alpha) \) is already computed and stored in \( c^1.rTable \), we only need to compute \( p_r(c^1 | b^1, G_\alpha) \). By the time we reach \( \alpha \), all the rTables on the nodes along this path are populated. Note that as we compute each node’s reachability from its children’s
reachability, we’ve improved pattern scoring in the first iteration as well.

As Equation 4.1-4.3 motivate, we propose R^3CAP, a CASM algorithm that improves pattern exploration and scoring (Figure 9.5). R^3 stands for the three major concepts it leverages: Reachability reuse, rTable storage, and Reverse traversal. Given query Q=α, R^3CAP traverses G_α in reverse to populate rTables (lines 3-6). Instead of recomputing each node’s reachability from scratch, R^3CAP reuses reachability to score patterns (lines 7-15).

R^3CAP uses a Pattern object to facilitate pattern scoring. A Pattern is a 3-argument tuple (α, Score, pReach). pReach stores the prefix reachability. It’s a list of 2-argument tuples where the first argument is a node instance of the added item in α and the second argument is the reachability from α to this instance. In the first iteration, pReach from the input is empty. However, if α=a for example, as a only has one instance, its pReach=\{ (α, p_r(a|a, G_α)) \}. Note that p_r(a|a, G_α)=1. If α=a...j, as a...j has multiple instances, its pReach =\{ (j^1, p_r(j^1|a, G_α)), (j^2, p_r(j^2|a, G_α)), (j^3, p_r(j^3|a, G_α)), (j^4, p_r(j^4|a, G_α)) \}. Each p_r(j^i|a, G_α) can be found by looking up a.rTable, which is already populated after line 6 in Figure 9.5. In subsequent iterations, pReach is given from the input.

Concretely, we walk through R^3CAP with an example. Suppose the query pattern is a and in the first iteration, R^3CAP mines and scores various expanded patterns such as a...c and a...j. To explore a...j, for each item j’s node instance j^i, R^3CAP looks up each j^i.rTable (lines 10-11), e.g. looking up j^4.rTable obtains j^4’s reachable nodes and their reachability \{ (c^3, p_r(c^3|j^4, G_α)) \}. To score a...j...c, R^3CAP sums up all the contributing reachabilities (lines 12-15). One such reachability is p_r(c^3|a...j^4, G_α). By Equation 4.1. p_r(c^3|a...j^4, G_α)=p_r(c^3| j^4, G_α)p_r(j^4|a, G_α) where p_r(j^4|a, G_α^a_{Q_1}) is given from pReach in the input and p_r(c^3| j^4, G_α) is stored at j^4.rTable.

There’s a SPM algorithm STMFP [16] that only reads S_α once to construct a FP-tree and mines frequent patterns bottom-up on this tree. However, STMFP is not iterative and difficult to extend to an iterative framework. Also, as STMFP’s scoring function is the frequency threshold, it has the same scoring issue as PrefixSpan.

<table>
<thead>
<tr>
<th>Part 1: First Iteration</th>
<th>Retrieve S_α</th>
<th>Pattern Scoring</th>
<th>PT Ops</th>
<th>rTable Ops</th>
<th>PrefixSpan</th>
<th>GraphCAP</th>
<th>R^3CAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>O(1)</td>
<td>O(k^2</td>
<td>S_α</td>
<td>)comp(W)</td>
<td>0</td>
<td>0</td>
<td>O(1)</td>
</tr>
<tr>
<td>Iteration</td>
<td>O([Bu])</td>
<td>O([G_α])</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Part 2: Each Subsequent</td>
<td>Retrieve S_α</td>
<td>Pattern Scoring</td>
<td>PT Ops</td>
<td>rTable Ops</td>
<td>PrefixSpan</td>
<td>GraphCAP</td>
<td>R^3CAP</td>
</tr>
<tr>
<td>Iteration</td>
<td>O(1)</td>
<td>O(k^2</td>
<td>S_α</td>
<td>)comp(W)</td>
<td>0</td>
<td>0</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td>O([Bu])</td>
<td>O([G_α])</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Time complexity summary.
4.3 Complexity Analysis

We analyze the complexity of PrefixSpan, GraphCAP, and R³CAP. Let \( k \) be the maximum sequence length in \( S_\alpha \). In the worst case, every item in \( S_\alpha \) is unique. So every item only has one node instance in \( G_\alpha \).

Space Complexity

PrefixSpan needs to store all the items reachable from \( \alpha \) and all their substrings from \( \alpha \). Suppose PrefixSpan stores this information in a buffer \( Buf \). The buffer size \( |Buf| \) is \( O(k|S_\alpha|) \) items and \( O(k|S_\alpha|) \) substrings. GraphCAP stores context graphs. The graph size \( O(|G_\alpha|) \) is \( O(k|S_\alpha|) \) nodes and \( O(k|S_\alpha|) \) edges. R³CAP stores rTables; they contain \( O((k|S_\alpha|)^2) \) tuples total. This is because each node’s rTable has at most \( O(k|S_\alpha|) \) reachable nodes and there are \( O(k|S_\alpha|) \) nodes in \( G_\alpha \).

In summary, R³CAP requires the most space. PrefixSpan and GraphCAP store the same number of items and nodes, but as each edge is a length-2 path, PrefixSpan requires more space than GraphCAP.

Time Complexity

PrefixSpan, GraphCAP, and R³CAP perform several major steps. We assume an inverted index lookup retrieves sequences and takes \( O(1) \). The sequence scanning step essentially stores information in a buffer or graph, so its time is bounded by \( |Buf| \) or \( |G_\alpha| \). The pattern scoring step boils down to the number of transition probability (PT) and rTable operations. Let \( \text{comp}(W) \) be the complexity for computing each edge weight. As each PT operation examines at most \( |S_\alpha| \) edges, it takes \( O(|S_\alpha|\text{comp}(W)) \). If the rTables are implemented as a hashtable, each rTable operation takes \( O(1) \). Lastly, we assume arithmetic operations take \( O(1) \).

Given these assumptions, we examine the scoring step more closely. In each iteration, PrefixSpan does \( k-1 \) PT operations per item \( u \) as the maximum length from \( \alpha \) to \( u \) is \( k \). With \( k|S_\alpha| \) items, PrefixSpan does \( O(k^2|S_\alpha|) \) PT operations. GraphCAP does \( k-1 \) PT operations per node. So it does \( O(k^2|S_\alpha|) \) PT operations for all the nodes. In the first iteration, R³CAP computes PT on each edge only once (Traverse line 3 in Figure 9.5). So it does \( O(k|S_\alpha|) \) PT operations. However, R³CAP needs to populate rTables. This time is bounded by the total size of rTables, i.e. \( O((k|S_\alpha|)^2) \). In subsequent iterations, R³CAP doesn’t do any PT operations, but it does multiplications (Figure 9.5 line 13). As \( \alpha \) has \( O(k|S_\alpha|) \) reachable nodes, R³CAP does \( O(k|S_\alpha|) \) rTable lookups (Figure 9.5 lines 9-10). As each of \( \alpha \)’s reachable nodes has \( O(k|S_\alpha|) \) reachable nodes (Figure 9.5 line 11), R³CAP does \( O(k^2|S_\alpha|^2) \) multiplications (Figure 9.5 line 13).

Table 4.1 summarizes the time complexity. PrefixSpan is the least efficient. If \( W \) is complex, R³CAP is more efficient than GraphCAP.
Chapter 5

Handle Multiple Items

In Chapter 3 and 4, we discuss how to search context-aware patterns assuming each itemset consists of only one item. In this chapter, we generalize the solution for multiple items. We use () to notate itemsets, e.g. itemset ab indicates it contains items a and b. If an itemset only contains an item, we don’t use () to notate it. The order within () doesn’t matter.

To handle multi-item itemsets, we re-visit the ranking model. Intuitively, to support multi-item itemsets, each node in the context graph \( G_\alpha \) is an itemset instance. According to Chapter 3, we construct an example context graph in Figure 9.6 for multi-item itemsets. At each non-leaf node \( u \) is an \( u.rTable \), which is populated according to Chapter 4.2. For concept explanation purposes, instead of displaying \( u \)'s reachable nodes and their reachability in \( u.rTable \), Figure 9.6 only displays \( u \)'s reachable nodes and the sequences that contain the particular connections.

As itemsets now contain multiple items, just like a SPM algorithm, a CASM algorithm needs to mine the subsets of itemsets as well. For example, a CASM algorithm should mine pattern \( a...(bcd) \) and the patterns \( (bcd) \)'s subsets form, i.e. \{a...b, a...c, a...d, a...(bc), a...(bd), a...(cd)\}. However, as the context graph does not store an itemset's subsets as nodes, GraphCAP and R\(^3\)CAP mine an incomplete set of patterns, e.g. only mine \( a...(bcd) \). Furthermore, the nodes that the context graph does store don’t capture all the paths that connect them. For example, although \( (bc) \) comes from \( (bc) \) and \( (bcd) \), \( a.rTable \) only shows a connects bc from sequence \( s_3 \). Again, this is because the subset notion is not intuitive in the context graph.

So intuitively, we should extend the context graph to store each itemset's subsets as reachable nodes. However, this storage scheme will increase the number of graph nodes and rTable size exponentially. Thus, we construct \( G_\alpha \) according to Chapter 3 and populate rTables according to Chapter 4.2. Then, we enumerate subsets as they are needed during the pattern scoring step. For example, as the context graph stores the entire itemset, e.g. \( (bcd) \), we first score \( a...(bcd) \). Then we enumerate \( (bcd) \)'s subsets and score the patterns they form.

By \( Score \)'s definition, when an itemset \( u \) only contains one item, \( Score \) is the aggregated reachability from \( \alpha \) to each \( u \)'s node instance \( u^i \). As each itemset now may contain multiple items, \( u \) may be a subset in various itemsets in \( S_\alpha \). Thus, as node \( u^i \) comes from itemset node instances that contain it, \( Score \) aggregates the reachability from \( \alpha \) to
nodes that contain \( w^i \). Formally, let pattern \( \beta = \alpha \ldots u \) and node \( w^i \) be an instance that contains \( w^i \),

\[
Score(\beta | \alpha) = \sum_{w^i \in w^i \text{'s node instances}} p_r(w^i | \alpha, G_\alpha).
\] (5.1)

For example, \( Score(a \ldots (bcd) | a) = p_r((bcd)^l | a, G_a) \). As another example, \( Score(a \ldots (bc) | a) = p_r((bcd)^l | a, G_a) + p_r((bc)^l | a, G_a) \).

When \( \alpha \) contains multiple itemsets, e.g. \( \alpha = a \ldots (bc) \), the same logic applies. For example, as \( (bc) \) comes from \( (bc) \) and \( (bcd) \), \( Score(a \ldots (bc) \ldots g \ldots (bc)) = p_r((g^l | a \ldots (bcd)^l, G_a) + p_r((eg)^l | a \ldots (bc)^l, G_a) \).

As our observations motivate, we present MultiR^3CAP (Figure 9.7). MultiR^3CAP extends R^3CAP by modifying the scoring steps and adding an “enumerate and score” operation. Lines 13-15 in R^3CAP are modified into lines 13-18 in MultiR^3CAP to handle the subset enumeration and scoring.

We briefly analyze MultiR^3CAP’s complexity. MultiR^3CAP and R^3CAP have the same space complexity as both algorithms store itemsets as nodes. However, MultiR^3CAP’s time complexity differs. For each reachable node \( u \), MultiR^3CAP only needs to return the top \( N \) patterns, it can leverage \( Score \)'s monotonic property to improve scoring complexity. As Equation 5.1 shows, the reachability to an itemset \( u \) aggregates the reachability to all \( u \)’s supersets. Thus, subsets always score at least as large as their supersets, e.g. \( Score(a \ldots b | a) \geq Score(a \ldots (bc) | a) \geq Score(a \ldots (bcd) | a) \). In general,

\[
Score(\alpha \ldots u | \alpha) \geq Score(\alpha \ldots w | \alpha),
\] (5.2)

where \( u \) is a subset of \( w \).

As Equation 5.2 motivates, we propose topN-MultiR^3CAP to mine top \( N \) patterns as followed. Given \( \alpha \) and an integer \( N \), topN-MultiR^3CAP traverses and populates rTables as in Figure 9.7. However, topN-MultiR^3CAP no longer enumerates and scores exhaustively. Instead, for each reachable node in \( \alpha.rTable \), it enumerates the length-1 subsets first, and scores and stores the patterns they form in a priority queue. If the number of length-1 subsets is less than \( N \), then it enumerates the length-2 subsets for each reachable node in \( \alpha.rTable \), and scores and stores the patterns they form in the same priority queue. In other words, topN-MultiR^3CAP enumerates subsets in the order of their length. If the number of length-\( l \) subsets is less than \( N \), then topN-MultiR^3CAP continues to enumerate and score all the length-(\( l+1 \)) subsets. When the size of the priority queue is over \( N \), topN-MultiR^3CAP outputs the top \( N \) elements in the priority queue as the result List.
In the worst case, topN-MultiR^3CAP has the same time complexity as MultiR^3CAP. However, as $N$ tends to be small, e.g. $N=20$, enumerating up to length-2 subsets usually suffice. Under this assumption, topN-MultiR^3CAP enumerates $O((m + \binom{m}{2})|S_\alpha|)$ subsets total as there are $O(mk|S_\alpha|)$ length-1 subsets and $O(\binom{m}{2}k|S_\alpha|)$ length-2 subsets. Thus, topN-MultiR^3CAP usually has a far lower scoring complexity.
Chapter 6

Related Work

The objective of our work is to search context-aware patterns. To the best of our knowledge, we did not witness any similar investigations in this direction, especially over a dynamic collection of documents with varying quality.

CASM is related to sequential pattern mining (SPM) [2, 12, 13, 15, 16]. We discuss their differences in Chapter 3 and 4. In essence, CASM is a query-driven SPM problem that integrates the notion of application-specific, context-aware conditional probability in pattern mining. [3] mines the frequent patterns that are statistically significant. However, these patterns are neither context-aware nor application-specific.

To score patterns, our ranking model adapts a random walk framework and models variable kth-order Markov Chains (MC). [1] also models variable kth-order MC. It constructs a graph such that each node is selectively cloned to capture a first-order or second-order MC. Specifically, a node $u$ clones $u^1$ such that the path that reaches $u^1$ is a first-order MC; $u$ clones $u^2$ such that the path that reaches $u^2$ is a second-order MC. Unlike [1], every node in our context graph $G_\alpha$ captures a kth-order Markov where $k$ is the length of the path that reaches $u^i$ from $\alpha$.

CASM has wide applicability, one of which is App1, i.e. opinion search. App1 is related to opinion retrieval and opinion mining. App1 can be regarded as opinion-aware opinion retrieval. The opinion retrieval task in TREC is a special information retrieval task that searches opinionated documents such as blogs and orders them in a ranked list [7, 8, 9, 10]. As [9] states, one future direction of opinion retrieval is to move away from the page view and focus on the actual opinion entities. App1 is an attempt in this direction.

App1 differs from opinion mining in three ways. One, it’s query-driven. As the opinion mining survey [11] shows, the input to opinion mining is a product or the popular attributes of a product. Thus, opinion mining cannot handle scenarios Q2-3 (Chapter 7.2). Two, App1 complements opinion mining. [11] shows most opinion mining works compute sentiment scores from reviews. These scores tend to be numeric aggregations and thus, are not informative when users want to know why HP’s battery life is 0.7, for example. App1 can help explain the score by providing qualitative aggregations such as battery life...long. Third, opinion mining use various dictionary-based, deep natural language processing (NLP), and/or machine learning techniques [6, 14, 11]. App1 avoids heavy preprocessing and the dependence on linguistic knowledge by examining how various words position with respect to the input. Some opinion mining works [5, 17] cluster opinionated sentences and return the clustered labels as opinions. App1 can be
regarded as top-down clustering where the similarity measure, $Score$, specifically targets opinions.
Chapter 7

Experiments

In this chapter, we verify CASM’s effectiveness and efficiency. As there are real, publicly available data for App1 (opinion search) and App2 (navigation analysis), we run CASM on these data.

7.1 Setup

App1: Opinion Search

Sequence Database: we first crawled product reviews on laptops, digital cameras, and mp3 players from major review-hosting websites. We then removed stop words from these reviews and segmented them into sentences. Lastly, we tokenized sentences into sequences. There are 77,700+ product reviews and 615,500+ sequences total.

Features: for demonstrative purposes, we assume the given weight function \( W \) uses various features to quantify connectivity. It combines the features linearly, i.e. \( W(e) = \sum_i \lambda_i F_i \) where \( e \) is an edge and \( \lambda_i \) is the coefficient for feature \( F_i \). \( W \) uses the following features for opinions:

- **Punctuation**: we learned the correlation between punctuation and the semantic relatedness between two keywords. To do so, first we collected some true opinions. For each pair of keywords in these opinions, we recorded if it has a punctuation or not. With sufficient examples, we obtained an empirical distribution of how having punctuation between two words affects their semantic relatedness.

- **Quality**: we assume good quality sequences are more likely to be retrieved. As we are using Lucene\(^1\) to retrieve sequences \( S_\alpha \), a sequence’s quality is its query score in Lucene.

- **Dictionary**: we manually created a set of good keywords and phrases. Any opinions with them will be given a boost in their score.

\(^1\)http://lucene.apache.org/java/

<table>
<thead>
<tr>
<th>Input Pattern ( \alpha )</th>
</tr>
</thead>
<tbody>
<tr>
<td>battery life, screen, customerservice, price, gaming,</td>
</tr>
<tr>
<td>video quality, excellent, good, great, wonderful, worst, ...</td>
</tr>
</tbody>
</table>

Table 7.1: A snapshot of laptop queries for App1.
Table 7.2: Top 5 patterns retrieved for App1.

<table>
<thead>
<tr>
<th>$Q_1$=battery life (176)</th>
<th>$Q_2$=great (730)</th>
</tr>
</thead>
<tbody>
<tr>
<td>battery life...hours (33)</td>
<td>great...laptop (86)</td>
</tr>
<tr>
<td>battery life...good (16)</td>
<td>great...price (57)</td>
</tr>
<tr>
<td>battery life...short (11)</td>
<td>great...computer (51)</td>
</tr>
<tr>
<td>battery life...power (12)</td>
<td>great...not (43)</td>
</tr>
<tr>
<td>battery life...great (9)</td>
<td>great...screen (33)</td>
</tr>
</tbody>
</table>

Queries: we manually created a list of product components, features, and adjectives as query patterns. There are 79 queries total. Table 7.1 shows a snapshot.

App2: Navigation Analysis

Sequence Database: we use BMS-WebView, a dataset from the KDD Cup 2000[4]. There are 149,600+ transactions and 59,600+ sequences.

Features: there are many interesting features one could use, e.g. the elapsed time between page. However, as BMS-WebView only contains the webpages visited per user session and the webpages are already encoded into numeric strings, this dataset doesn’t provide much information that we can use as features. Limited by the dataset, we only use the source quality as feature.

Queries: we take each unique item from the dataset as an input pattern $\alpha$. There are 498 queries total.

7.2 Effectiveness Analysis

Qualitatively: User Studies

Qualitatively, we show CASM’s effectiveness with various scenarios of App1. We implemented a CASM system according to Chapter 4.2 and the weight function $W$ evaluated edge weights with all the features mentioned in Chapter 7.1.

The following scenarios are typical for a consumer who’s looking to buy a HP laptop:

Q1: What are the opinions on HP’s battery life?

Q2. What is great about HP laptops?

We queried the CASM system with query $Q_1$ and $Q_2$ on HP’s laptop reviews. Table 7.2 shows the top 5 patterns retrieved. The number in parenthesis is the pattern frequency. For $Q_1$, battery life...power is not relevant because power in battery life’s context doesn’t make sense. Note that while battery life...short is not as frequent as battery life...power, it ranks higher.

The consumer may also want to know
Q3. Which laptops have long battery life?

We queried the system with query $Q_3=$long battery life on all the laptop reviews. From the patterns retrieved, we returned the products whose reviews contribute a large reachability score for long battery life.

Quantitatively: Empirical Verification

Quantitatively, we verify CASM’s effectiveness. We implemented R$^3$CAP according to Chapter 4.2. To study the effects of features, we also implemented weight function $W$ with different feature sets. ALL uses all the features mentioned in Chapter 7.1. EW weighs each edge equally. Note that EW is the context-aware version of frequency. PUNCT uses the punctuation feature, and thus, it must examine the sequence content. DICT uses the dictionary feature. QUAL uses the source quality feature, and thus, it must look up the sequence’s query score. ALL is the most complex and EW is the least. To show the importance of context in sequential pattern mining, we use a standard sequential pattern mining (SPM) algorithm as the baseline. We denote the baseline as FREQ@1% and FREQ@2%, indicating the frequency threshold at 1% and 2% respectively.

As CASM searches relevant context-aware patterns, we use mean average precision (MAP) as the evaluation metric. MAP is a standard information retrieval metric that considers both recall and precision. It is the average of the average precision value for a set of queries. Average precision is the average of the precision values at which each relevant result is retrieved. For each retrieved pattern, we manually judged its relevancy. We then computed the MAP for the top 5, 10, 15, and 20 patterns retrieved. As the webpages in App2’s dataset are encoded into numeric strings, it’s impossible to judge whether a numeric pattern is relevant. So we only evaluate App1’s dataset.

Figure 9.8(a) shows the MAP results. It confirms that CASM is more effective than SPM for context-aware patterns. ALL has the best MAP results. Although EW uses no features, as the context-aware version of frequency, it still outperforms its context-unaware counterparts FREQ@1% and FREQ@2%. The sharp drop off in FREQ@1% and FREQ@2% at top 15 and 20 indicates many relevant patterns are filtered out as they are infrequent. FREQ@2% is worse than FREQ@1% because its 2% threshold filters more relevant patterns out. At top 5 and 10, the low MAP in both FREQ@1% and FREQ@2% is a result of ranking frequent but irrelevant patterns at the top.

To further contrast context-aware patterns with frequent patterns, we plotted a frequency vs relevancy graph for the top 20 retrieved patterns for all the queries. At the coordinate $(x, y)$, a pattern is ranked at the $x$th position by its frequency and at $y$th position by its relevancy to the query. We used ALL (for App1) and QUAL (for App2) to determine α’s relevancy. If frequent patterns are relevant and vice versa, they should be positively related.

Figure 9.8(b) shows the results. The red circles indicate frequency and relevancy are not always positively related. The green box shows the top 20 frequent patterns can rank pretty low by their relevancy. The pink box shows the top 20 relevancy-ranked patterns are usually not the most frequent. Thus, even without a gold standard for App2’s dataset,
we can see that frequency threshold is ineffective.

## 7.3 Efficiency Analysis

In this chapter, we study the efficiency of CASM’s computation model. To show that CASM requires a new computation model, we implemented PrefixSpan to search context-aware patterns as the baseline. As App1 and App2 deal with itemsets that contain only one item, we implemented GraphCAP and R³CAP to compare against PrefixSpan. We then queried these systems with the queries in Chapter 7.1 and recorded their space and time usage. We ran these algorithms on a single computer with a Core Duo CPU, 2.50GHz, and 1GB ram.

Table 7.3 shows the space usage. GraphCAP is the most space efficient and R³CAP is the least. This confirms our space analysis.

In terms of time usage, Figure 9.8(c) shows the first iteration run time for each query. GraphCAP is slightly more efficient than PrefixSpan. By sacrificing space efficiency, R³CAP is substantially more efficient than GraphCAP and PrefixSpan. All feature sets exhibit similar trend.

Figure 9.8(d) and 9.8(e) show the improvement of GraphCAP and R³CAP over PrefixSpan. We define improvement as

\[
\text{Improvement} = \frac{X's \ \text{run time} - Y's \ \text{run time}}{X's \ \text{run time}} \times 100,
\]

where \( X \) is PrefixSpan and \( Y \) is either GraphCAP or R³CAP. A positive y-value indicates GraphCAP (or R³CAP) is more efficient whereas a negative y-value indicates PrefixSpan is more efficient. In the first iteration, the positive y-value for both GraphCAP and R³CAP show PrefixSpan to be the least efficient. As R³CAP has a bigger y-value than GraphCAP, it’s more efficient than GraphCAP. Note that the ALL curves in the first iteration and the results in Figure 9.8(c) are the same but presented differently. As Figure 9.8(c) shows, R³CAP can improve PrefixSpan more than 75%.

In subsequent iterations, improvement gets more positive as the number of iterations increases. GraphCAP and R³CAP with features QUAL and EW have negative improvement in early subsequent iterations. This is because 1) \( |S_\alpha| \) in each subsequent iteration is much smaller than \( |S_\alpha| \) in the first iteration and 2) QUAL and EW are simple
features. Thus, PrefixSpan’s time on scanning sequences and populating buffer $Buf$ finish quickly. On the other hand, GraphCAP needs to spend time traversing the graph and $R^3$CAP needs to spend time on $r$Table operations. These operations dominate in pattern scoring. However, as the number of iterations increases, PrefixSpan’s time on repeatedly scanning sequences and populating buffer $Buf$ add up. The improvement plateaus because patterns tend to be sparse, so there may not be much to explore after exploring 15 or more patterns.

As the previous paragraph hints, feature complexity affects the complexity of the weight function $W$, which in turn affects efficiency. The legend Figure 9.8(d) and 9.8(e) orders features by their complexity. Algorithms with the more complex features, e.g. ALL and PUNCT, have larger positive improvement than the ones with less complex features, e.g. QUAL and EW. This is expected. As Table 4.1 shows, the presence (or lack) of $\text{comp}(W)$ has an effect on the algorithms.

Note that the features used in the experiments are for demonstrative purposes and the algorithms evaluated them online. Some features, i.e. PUNCT and DICT, can be evaluated offline, and thus, can make the algorithms run more efficiently. However, it’s not true for all features. An application may use more dynamic features that need to be evaluated online, e.g. item co-occurrence with respect to $\alpha$. Thus, assuming many features are complex and must be evaluated online, GraphCAP and $R^3$CAP outperform PrefixSpan.
Chapter 8

Conclusions

We have introduced the problem of searching context-aware patterns. It’s a novel problem with real, practical applications. As a solution, we presented a variable $k$th-order random walk as the ranking model and developed two efficient algorithms GraphCAP and R$^3$CAP. If space is a major concern or if the application’s goal is to return an exhaustive set of patterns (i.e. explore the entire pattern space) and features are very simple, GraphCAP may actually be more suitable. However, if the goal is to return relevant patterns effectively and efficiently, R$^3$CAP is more suitable as features may be more complex. We also discussed various extensions. MultiR$^3$CAP generalizes R$^3$CAP for multi-item itemsets and topN-MultiR$^3$CAP leverages $Score$’s monotonic property to mine top $N$ patterns. Lastly, as Chapter 7.2 shows, our solution is applicable for various applications. One of which is App1 (opinion search), which is a novel application different from traditional opinion mining and retrieval.
Chapter 9

Figures

$s_1$: Battery life can last long hours. <abcj>
$s_2$: Its battery life even with wifi can last five hours long. <adbijc>
$s_3$: Battery life long enough for five hours. <ackij>
$s_4$: Performance is slow, but battery life lasts long enough for five hours. <peabckij>
$s_5$: Battery life lasts thirty minutes. <abmf>
$s_6$: Battery life lasts long, decent screen. <abcgh>
$s_7$: Battery life short, long loading time. <aqcln>

Figure 9.1: Example sequences.
(a) First attempt to construct $G_{a,c}$, using sequences $s_1$ and $s_2$ in Figure 9.1.

(b) Context graph $G_{a,c}$, using sequences in Figure 9.1.

(c) Context graph $G_{a...c}$, using sequences in Figure 9.1.

Figure 9.2: Example context graphs.
Algorithm **GraphCAP**

- **Given:** A weight function $W$ and sequences $S = \{s_1, s_2, \ldots, s_n\}$
- **Input:** Sequential pattern $\alpha$, query $Q = \alpha$, and context graph $G_\alpha$
- **Output:** A ranked list of $\{((\beta_1, \text{Score}_1), (\beta_2, \text{Score}_2), \ldots)\}$

1: IF context graph $G_\alpha$ is null
2: Retrieve $S_\alpha$, the sequences that contain $\alpha$ /* Index lookup */
3: Read $S_\alpha$ to build $G_\alpha$
4: Initialize List to be $\emptyset$
5: Locate $\alpha$’s instances on $G_\alpha$ and add their children to Queue
6: WHILE Queue is not empty
7: $u^i = $Queue.pop()
8: Obtain itemset $u$ from its node instance $u^i$ and create $\beta = \alpha \cdots u$
9: Compute $p_r(u^i|\alpha, G_\alpha)$
10: IF List has $\beta$
11: Update $\text{Score}(\beta|\alpha) = p_r(u^i|\alpha, G_\alpha)$ in List
12: ELSE
13: $\text{Score}(\beta|\alpha) = p_r(u^i|\alpha, G_\alpha)$ and add $(\beta, \text{Score}(\beta|\alpha))$ to List
14: Add $u^i$’s children to Queue
15: Return List

---

**Figure 9.3:** Algorithm GraphCAP.

---

**Figure 9.4:** Reverse graph traversal.
Algorithm R³CAP

- **Given:** A weight function \( W \) and sequences \( S = \{s_1, s_2, ... s_n\} \)
- **Input:** Pattern \( Pat = (\alpha, Score, pReach) \), query \( Q = \alpha \), and \( RT \), all the rTables in \( G_\alpha \)
- **Output:** A ranked list of Patterns

1: Initialize \( List \) and \( queryPrefix \) to be \( \emptyset \)
2: Let itemset \( u \) be the last item in \( \alpha \) and \( x \) be the first item in \( \alpha \)
3: IF \( RT \) is null
4: Retrieve \( S_\alpha \), the sequences that contain \( \alpha \) /* Index lookup */
5: Read \( S_\alpha \) to build \( G_\alpha \)
6: Traverse(\( \alpha \), \( G_\alpha \))
7: For each node instance \( u_i \), add \( (u_i, p_r(u_i|x,G_\alpha)) \) to \( pReach \)
8: ELSE, set \( queryPrefix = pReach \), then set \( pReach = \emptyset \)
9: FOR each tuple \( (u_i, p_r(u_i|\alpha,G_\alpha)) \) in \( queryPrefix \)
10: Look up \( u_i.rTable \) to get \( u_i \)'s reachable nodes and their reachability \( \{(v_1, p_r(v_1|\alpha,G_\alpha)), (v_2, p_r(v_2|\alpha,G_\alpha)), ...\} \)
11: FOR each entry \( (v_j, p_r(v_j|\alpha,G_\alpha)) \) in \( u_i.rTable \)
12: Obtain itemset \( v \) from node instance \( v_j \) and create \( \beta = \alpha ... v \)
13: Compute \( p_r(v_j|\alpha,G_\alpha) = p_r(v_j|\alpha,G_\alpha) \cdot p_r(\alpha|G_\alpha) \)
14: IF \( List \) has \( \beta \), update \( Score(\beta|\alpha) += p_r(v_j|\alpha,G_\alpha) \) and \( pReach \) in \( List \)
15: ELSE, \( Score(\beta|\alpha) = p_r(v_j|\alpha,G_\alpha) \) and add Pattern \( (\beta, Score(\beta|\alpha), \{(v_j, p_r(v_j|\alpha,G_\alpha))\}) \) to \( List \)
16: Return \( List \)

Subroutine Traverse

- **Input:** Node \( x \) and context graph \( G_\alpha \)

1: Locate \( x \) on \( G_\alpha \)
2: FOR each children \( u \) of \( x \)
3: Compute \( p_r(u|x,G_\alpha) = p_r(u|x,G_\alpha) \)
4: Add \( (u, p_r(u|x,G_\alpha)) \) to \( x.rTable \)
5: Traverse(\( u,G_\alpha \))
6: FOR each entry \( (v, p_r(v|u,G_\alpha)) \) in \( u.rTable \)
7: Compute \( p_r(v|x,G_\alpha) = p_r(v|u,G_\alpha) \cdot p_r(u|x,G_\alpha) \)
8: Add \( (v, p_r(v|x,G_\alpha)) \) to \( x.rTable \)
9: IF \( x \) doesn’t have children, set \( x.rTable = \emptyset \)

Figure 9.5: Algorithm R³CAP.
Sequences
s_1: a(bcd)(ef)
s_2: a(bcd)g
s_3: a(bc)(eg)
s_4: acg

Algorithm MultiR³CAP

• Given: A weight function W and sequences S={s_1, s_2, ..., s_n}
• Input: Pattern Pat=(α, Score, pReach), query Q=α, and RT, all the rTables in G_α
• Output: A ranked list of Patterns

1: Initialize List and queryPrefix to be ∅
2: Let itemset u be the last item in α and x be the first item in α
3: IF RT is null
4: Retrieve S_α, the sequences that contain α / Index lookup
5: Read S_α to build G_α
6: Traverse(α, G_α)
7: FOR each node instance u^i, add (u^i, p_r(u^i|x^j, G_α)) to pReach
8: ELSE, set queryPrefix=pReach, then set pReach=∅
9: FOR each tuple (u^i, p_r(u^i|α, G_α)) in queryPrefix
10: Look up u^i.rTable to get u^i’s reachable nodes and their reachability {(v_1, p_r(v_1|u^i, G_α)), (v_2, p_r(v_2|u^i, G_α)), ...}
11: FOR each entry (v_j, p_r(v_j|u^i, G_α)) in u^i.rTable
12: Obtain itemset v from node instance v_j
13: Enumerate v’s subsets
14: FOR each subset w
15: Create β=α...w
16: Compute p_r(w|α, G_α)=p_r(w|u^i, G_α)p_r(u^i|α, G_α)
17: IF List has β, update Score(β|α)=p_r(v_j|α, G_α) and pReach in List
18: ELSE, Score(β|α)=p_r(v_j|α, G_α) and add Pattern (β, Score(β|α), (v_j, p_r(v_j|α, G_α))) to List
19: Return List

Figure 9.7: Algorithm MultiR³CAP extends R³CAP.
(a) MAP on top 5, 10, 15, and 20 patterns for App1.

(b) Ranking Difference.

(c) Efficiency per query using All features on App1.

(d) Efficiency on App1.

(e) Efficiency on App2.

Figure 9.8: Experiment Results.
References


