Clustering and Indexing of Experience Sequences for Popularity-driven Recommendations*

K. Selçuk Candan Mehmet E. Dönderler J. Ramamoorthy Jong W. Kim
CSE Department, Arizona State University,
Tempe, AZ, 85287-8809
{candan, mdonder, jaikannan, jong}@asu.edu

ABSTRACT

As part of our iCare efforts, we are developing mechanisms that provide guidance to individuals who are blind in diverse contexts. A fundamental challenge in this context is to represent and index *experiences* that can be used to provide recommendations. In this paper, we address the challenge of indexing *experiences* in order to retrieve them based on their *popularities*. In particular, we model experiences as sequences of propositional statements from a particular domain (daily life, web browsing, etc.). We then show that knowledge about domain constraints (such as commutativity between possible statements) need to be used for clustering and indexing experiences for popularity-search. We also highlight that `don't care`s (propositional statements not relevant to the user's query) make the task of popularity indexing challenging. Thus, we develop a canonical-sequence based approach that significantly reduces the experience sequence retrieval time in the presence of commutations. We introduce rule-compression, which helps achieve further reductions in the retrieval cost. We propose a novel two-level index structure, `EXP`*dex*, to efficiently answer wild-card (`don't care`) queries. We compare the proposed approach analytically and experimentally to a `don't care`-unaware solution, which does not take into account wild-cards in queries while constructing the popularity index. Experiments show that the proposed approach provides large savings in retrieval time when commutations between the elements of sequences are allowed.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Abstracting methods, indexing methods*; K.4.2 [**Computers and Society**]: Social Issues—*Assistive technologies for persons with disabilities*

General Terms

Algorithms, experimentation, human factors, performance

Keywords

Indexing experiences, generating navigational recommendations, assistive technology for blind users

*This work is supported by NSF ITR Grant, ITR-0326544, iLearn: IT-enabled Ubiquitous Access to Educational Opportunities for Blind Individuals.

CARPE'06, October 28, 2006, Santa Barbara, California, USA.

1. MOTIVATION

As part of our ongoing iCare project, we are developing assistive and rehabilitative technologies for individuals who are blind and visually impaired [16, 17, 23, 32, 34]. We note that it is much easier for individuals who are blind to get disoriented during navigation, whether in virtual or physical environments. Thus, our efforts under the iCare umbrella range from iCare-Assistant, a novel assistive system to help students who are blind in accessing web and electronic course materials, to iCare-Navigator, an RFID-based mobility device to help individuals who are blind navigate complex environments, such as campuses and libraries. Our ultimate goal is to provide guidance and recommendations to individuals who are blind in diverse contexts.

A fundamental challenge we face within the broad iCare efforts is to capture, represent, index, and retrieve *experiences* that can be used to provide recommendations to people who are blind. In this paper, we focus on this challenge.

1.1 Modeling Experiences

Modeling experiences of individuals is an important task in diverse contexts [1, 6, 14, 15, 18, 19, 21, 28, 33, 38]. For instance, in the area of user interface, modeling user experiences enables design of effective tools [18]. In the area of intrusion detection, models of *expected* behaviors enable detection of possible intrusions [6, 14, 15]. In the context of multimedia and web system design, models of expected user or population-behavior enable the design of prefetching and replication strategies for improved content delivery [28, 33].

Recording and indexing individual's various experiences also carry importance in personal information management [19] and experiential computing [20] contexts. [20, 21] lay the ground rules for research in the area of experiential computing area and highlight that, among others, user state and query context are highly relevant for experiential computing systems. [21] highlights the importance of logging and indexing of events in the form of *eChronicles* for experiential systems. [2] propose activity based data models for capturing user experience within the context of desktop management systems. [38] proposes various *user experience*

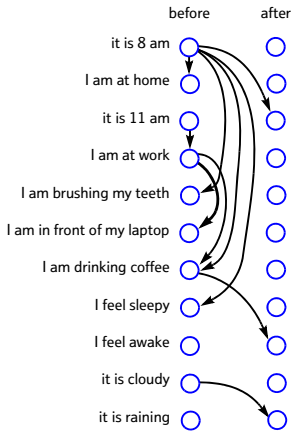


Figure 1: Sample from the LifeNet database (figure taken from [36]); arrows across columns denote before-after constraints between propositional statement; arrows within a column denote simultaneity

models applicable to different contexts in arts and multimedia domains. Since our goal within the iCare project is to provide effective recommendations to individuals who are blind, instead of focusing on the properties of a specific domain, in this paper we rely on a high-level, generic definition of user experiences. In particular, we model experiences as a sequence of propositional statements (picked from a particular domain):

DEFINITION 1.1 (USER EXPERIENCE). *Let $\mathcal{D}$ be a domain and $\mathcal{A}$ be a set of propositional statements from this domain. A user experience, e_i, is simply modeled as a finite sequence $e_{i,0} \cdot e_{i,1} \cdot \ldots \cdot e_{i,n}$, where $e_{i,j} \in \mathcal{A}$.* ◇

Naturally, capturing the appropriate proposition statements that form a particular domain and discovering the relationships between these statements is essential for any human-centric reasoning and recommendation system. A notable effort in understanding and describing the daily-life experiences is the LifeNet effort [36], which consists of 80,000 first person propositional statements and 400,000 temporal links that have been mined from the Open Mind Common Sense corpus [37]. Figure 1 presents a sample of propositional statements taken from LifeNet and the relationships between these statements mined and represented: the arrows across columns denote the observed before-after constraints, while arrows within a column denote simultaneity [36].

1.2 Popularity-based Recommendation Task

An experience-driven recommendation system should be able to capture the past states of the individual and the future states that the individual wishes to reach, while identifying certain proposition statements to recommend to the individual based on this context (past history and future goals). In other words, a recommendation task should capture the experiences (propositional statements) that are relevant for the context and allow the user to state what is not relevant (*don't cares*) through wildcard symbols. Therefore, we define recommendation task as follows:

DEFINITION 1.2 (RECOMMENDATION TASK). *Let $\mathcal{D}$ be a domain and $\mathcal{A}$ be a set of propositional statements from this domain. Let $\mathcal{E}$ be an experience collection (possibly representing experiences of a group of individuals).*

We define a popularity query as a sequence, q, of propositional statements and wildcard characters from $\mathcal{A} \cup \{\text{“}\star\text{”}\}$ executed over the database, $\mathcal{E}$.

The query processor (recommendation engine) returns an answer if a highly frequent (popular) match is found (“$\star$” is a wildcard symbol that matches any label in $\mathcal{A}$)[1] ◇

In other words, the query asks whether a sequence of propositional statements (possibly including `don't cares`) is popular or not.

Note that, in any experience domain, the order between certain propositional statements are irrelevant. For example, in the *real-life* model captured by LifeNet, any statements not connected with temporal constraints (arrows across columns) can be assumed to be commutative. In particular, order for those statements which are marked *simultaneous* with arrows within a column, is possibly irrelevant for a recommendation system.

1.3 Problem Statement: Indexing and Retrieval of Popular Experience Sequences in the Presence of Commutations and Wildcards

Based on the above discussion, in this paper, we formulate the problem of “experience-driven recommendation in the presence of commutations and wildcards” as follows: Given

- an alphabet, $\mathcal{A}$, consisting of proposition statements from a domain $\mathcal{D}$;
- a collection $\mathcal{E}$ of experiences, where each $p \in \mathcal{E}$ is a propositional sequence (of symbols from $\mathcal{A}$; in the rest of the paper, we refer to each position in the propositional sequence as a *node* in the sequence); and
- a set $\mathcal{C} = \{c_1, \ldots c_r\}$ of commutativity rules, specific to the domain $\mathcal{D}$, of the form
  - $c_i \equiv a \leftrightarrow b$, where $a, b \in \mathcal{A}$ (this denotes that nodes labeled “a” and “b” may be swapped provided that they are neighbors in the sequence)
- and a query sequence, q, of symbols from $\mathcal{A} \cup \{\text{“}\star\text{”}\}$,

find if q matches a frequent experience (sequence of statements) modulo the commutations allowed in the domain.

For example, Figure 2 provides (a) an alphabet and applicable commutativity rules, (b) a wildcard query, and (c,d) two example sequences matching the query based on the commutativity rules.

1.4 Contributions of this Paper

Sequence matching is a core operation in various domains, such as plan indexing, gene sequence matching, data compression, and used as a fundamental operation in the retrieval of tree-structured data where sequences correspond to paths (see Section 6 for the related work). As described above, in this paper, we also pose the experience-driven recommendation task as a sequence matching problem. *Yet, we*

[1]The wildcard “//” which corresponds to an arbitrary number of $\star$s is not considered in this paper. However, it could be implemented if an upper bound on the number of $\star$s is available.

$\mathcal{A} = \{A, B, C, D\}$
$\mathcal{C} = \{A \leftrightarrow C, D \leftrightarrow C, B \leftrightarrow C\}$

(a) Alphabet and allowed commutations

$q = \text{A} \cdot \star \cdot \text{B} \cdot \text{D} \cdot \text{C}$

(b) Query

$p_1 = \text{C} \cdot \text{A} \cdot \text{B} \cdot \text{C} \cdot \text{D}$
$p_2 = \text{A} \cdot \text{B} \cdot \text{B} \cdot \text{D} \cdot \text{C}$

(c,d) Possible matches

Figure 2: An example: (a) the alphabet, where each symbol corresponds to a propositional statement in a particular domain, and commutativity rules, (b) a wildcard query, and (c, d) two matching propositional sequences, p_1 and p_2. Note that although p_1 and p_2 are not the same experience, they match the query modula the commutation rules in the particular domain

note that, despite the extensive literature in precise or approximate string matching (Section 6), there is no work on indexing we are aware of where (a) an explicitly provided set of commutativity rules has to be considered and (b) queries contain wildcard symbols. Thus, in this paper, we propose a novel index structure (`EXP`*dex*) to efficiently retrieve popular sequences in the presence of commutations and wildcards.

1.5 Organization of the Paper

The organization of this paper is as follows: In Section 2, we discuss the problem of matching and show that using properly chosen *canonical sequences*, it is possible to retrieve commutative sequences efficiently. In Section 3, we highlight the challenge `don't care's` in queries pose and propose early wildcard replacement as an approach to tackle this problem. In Section 4, we introduce rule-compression which further improves the sequence matching performance, and present a novel two-level indexing mechanism to enable processing of *wildcard* queries. The result is the `EXP`*dex* index structure for indexing popular sequences. In Section 5, we experimentally show that the proposed index structure for sequence matching and retrieval in the presence of commutations works efficiently. In Section 6, we provide an overview of the related work and we draw our conclusions in Section 7.

2. SEQUENCE MATCHING IN THE PRESENCE OF COMMUTATIONS

In the previous section, we posed the experience-driven recommendation task as a sequence indexing problem in the presence of commutations and wildcards. In this section, we highlight the key observations that enable indexing of sequences that are equivalent under a set of commutativity rules.

2.1 Canonical Sequences

The first construct we introduce to deal with the existence of commutations is the *canonical sequence*:

DEFINITION 2.1 (CANONICAL SEQUENCE). *Let*

- $\mathcal{A}$ *be an alphabet;*
- $rank : \mathcal{A} \rightarrow Z^+$ *be a ranking function corresponding to a total (such as lexicographic) order of the elements of the alphabet* $\mathcal{A}$*;*
- $p = v_0 \cdot v_1 \cdot \ldots \cdot v_n$ *be a sequence and* $l : \{v_0, \ldots, v_n\} \rightarrow \mathcal{A}$ *be its node labeling,*
- $\mathcal{P}(p, \mathcal{C})$ *be the set of all permutations of sequence* p *allowed by the set* $\mathcal{C}$ *of commutativity rules; and*
- $order : \mathcal{P}(p, \mathcal{C}) \times \{v_0, \ldots, v_n\} \rightarrow Z^+$ *be a function which returns the position of a given node* v_i *in a given permutation of* p*.*

Then, the canonical form p^c *of the sequence* p *is a permutation of* p *in* $\mathcal{P}(p, \mathcal{C})$ *(i.e.,* $p^c \in \mathcal{P}(p, \mathcal{C})$*), such that*

$$\forall_{p' \in \mathcal{P}(p,\mathcal{C})} prop(p^c, p') = true,$$

where the proposition $prop(p^c, p')$ *is defined as follows: for all* $v_i \in p^c$ *and* $v_j \in p'$

$$\begin{aligned}(order(p^c, v_i) = 1 \wedge order(p', v_j) = 1) \Rightarrow \\ ((rank(l(v_i)) \leq rank(l(v_j))) \wedge \\ (rank(l(v_i)) = rank(l(v_j)) \Rightarrow prop(p^c_{tail}, p'_{tail})))\end{aligned}$$

Here, p_{tail} *is the tail (i.e., subsequence which excludes the first node) of the sequence* p*. Note that if the tails have no nodes (which is possible only with* $prop(p^c, p^c)$*), then prop returns true (in other words,* $prop(\perp, \perp) = true$*).* ◇

Intuitively, the canonical sequence is the permutation of a given sequence in which all commutativity rules are applied such that nodes with the lexicographically smaller labels are propagated far to the left.

EXAMPLE 2.1. *For example, given the sequence* $p = B \cdot A \cdot D \cdot C \cdot B$*, the commutativity rules* $\mathcal{C} = \{A \leftrightarrow B, D \leftrightarrow C, B \leftrightarrow D\}$*, and assuming* $rank(A) < rank(B) < rank(C) < rank(D)$*, we obtain the canonical sequence* $p^c = A \cdot B \cdot C \cdot B \cdot D$*.* ◇

2.2 Complexity of Computing Canonical Sequences

Given a sequence $p = v_0 \cdot v_1 \cdot \ldots \cdot v_n$, of length n, canonical form transformation can be achieved step by step by finding the first, the second, ..., and the last element (from left to right) in p^c applying the commutativity rules $\mathcal{C}$ on p. If there are more than one element for the first position, an element whose rank is less than or equal to all the candidates is chosen. This whole process takes $O(n^2)$ time. Having found the first (left-most) element of p^c, the second, the third, and the other elements in p^c can also be found applying the same process. That is, the entire process is repeated $(n-1)$ times, until all elements are placed in their respective positions in p^c. Thus, the complexity of identifying p^c is[2] $O(n^3)$.

[2]In the paper, we distinguish the cost of disk-based operations from the others using **bold-face**. Since it is not bold-face, $O(n^3)$ refers to an in-memory operation.

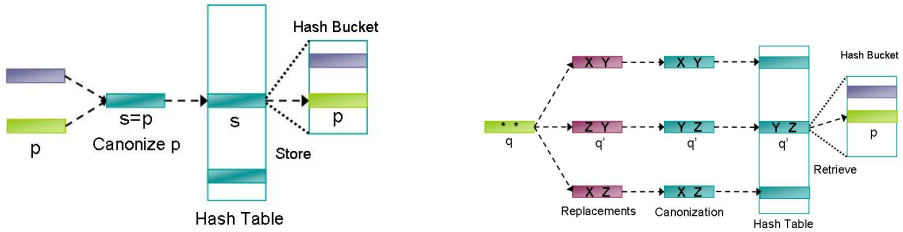


Figure 3: (a) Wildcard-unaware indexing; and (b) processing a 2-"$\star$" wildcard query on such an index.

2.3 Popularity Clustering of Propositional Sequences in the Presence of Commutations

Given a database, $\mathcal{E}$, of experiences (propositional sequences) the recommendation engine identifies popular (highly frequent) experiences. A particular challenge is that commutations allowed in the particular domain $\mathcal{D}$, have to be taken into account when identifying popularities.

Given the canonical sequence definition provided above, the task of popularity-clustering involves the following:

1. for all $p_i \in \mathcal{E}$, identify ${p_i}^c$
2. count and eliminate ${p_i}^c$ which are not frequent based on a given threshold

The resulting popularity clustered database, $\mathcal{E}^c$, is then used for answering queries and generating recommendations.

2.4 Indexing Sequences in the Presence of Commutations

Given a query sequence q and its canonical form, q^c, under a given set of commutativity rules, $\mathcal{C}$, a multitude of index structures (including hash tables or compact tries) can be used for retrieving those sequences which have the same canonical sequence, q^c. Thus, given a query sequence *without* any wildcard symbols, these index structures can be used for finding if the query matches under the commutativity rules.

On the other hand, when queries have wildcard symbols, naive approaches to popularity indexing are not appropriate or do not scale.

3. POPULARITY-INDEXING COMMUTATIVE SEQUENCES FOR QUERIES INVOLVING DON'T-CARES

As discussed earlier, a recommendation task (an experience sequence) may contain `don't-care` statements. In order to enable wildcards (`don't-care`s) in the popularity queries, on the other hand, we need to consider the following challenges underlying popularity indexing of commutative sequences:

- In the presence of `don't-care`s, early *popularity-counting* and pruning may be ineffective. For example consider the following sequences,

$$A \cdot B_1 \cdot C$$
$$A \cdot B_2 \cdot C$$
$$\dots$$
$$A \cdot B_{100} \cdot C$$

  Let us assume that each of these propositional sequences occurs once. If there are no wildcards in the queries, then none of the sequences are popular thus can all be eliminated. On the other hand, if popularity queries involve wildcards (e.g., $A \cdot * \cdot C$), then there are 100 matches to the query. Hence, wildcard-unaware pruning can lead to misleading recommendations.

- Since the data label that will match the "$\star$" is not know in advance, the exact set of commutativity rules that apply to "$\star$" labeled query symbols cannot be known. Therefore, it is not possible to create a single canonical query q^c that will match each possible data sequence p. Thus, multiple queries to the index structure may be needed to handle a single wildcard query.

These problems render naive approaches to popularity-indexing and retrieval ineffective.

3.1 Popularity-Indexing without Considering Wildcard's in Queries

An obvious naive solution is to index and retrieve experience sequences, based on their canonical forms, using existing index structures. Given a set of experiences, $\mathcal{E}$, we would first identify popular canonized sequences, $\mathcal{E}^c$ and index those that are frequent as described in Section 2.3 and Figure 3(a).

When a sequence query q containing k nodes labeled with the "$\star$" wildcard symbol is provided, each wildcard in the query is replaced with labels from the alphabet, $\mathcal{A}$, to obtain $|\mathcal{A}|^k$ possible queries. Each of these (potentially large number of) no-wildcard queries needs to be canonized separately and sent to the index to obtain matches. This is illustrated in Figure 3(b).

As we will illustrate formally in Section 4.5 and experimentally in Section 5, this solution suffers from performance challenges, especially when the alphabet size (i.e., the number of possible propositional statements in domain $\mathcal{D}$) is large. Furthermore, as described above, since the original popularity database is constructed *without considering* `don't-cares`, the recommendations are also prone to losses.

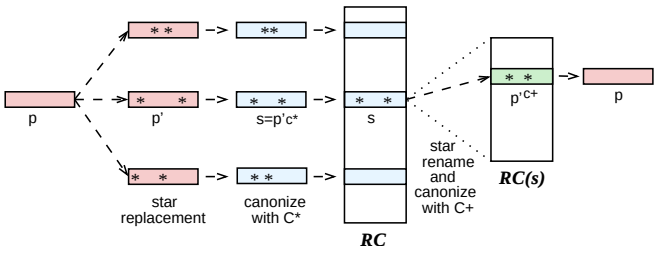


Figure 4: Sequence indexing using EXP*dex*

4. PROPOSED APPROACH: EARLY WILDCARD REPLACEMENT AND SEQUENCE CANONIZATION WITH RULE SET COMPRESSION

As stated above, naive approaches to popularity-based indexing suffer from early pruning and performance challenges. In this section, we build on the *sequence canonization* approach we introduced in the previous section to reduce both complexities. In particular, we propose an index structure with (a) early wildcard replacement and (b) sequence canonization with rule set compression. More specifically, we

- we propose to replace the propositional statements in experience sequences with `don't-care` symbols *before* canonization and popularity counting, and
- introduce a rule set compression scheme for efficient retrieval of sequences for $\star$ wildcard queries. Rule set compression identifies those sequences that can be clustered together; sequences in the same cluster have the same canonical form with respect to the given set of commutativity rules.

Relying on these, we develop an index structure, called EXP*dex*, that efficiently answers commutative queries containing "$\star$"s [3]. EXP*dex* has two-levels:

- In the first level, an index structure provides a *rough* form of clustering.
- The second level index structure refines the rough clusters formed in the first level, guiding the search to the correct sequence.

The first level of clustering provides enough commonalities among the sequences in a given rough cluster to enable us do searches (in the refinement level) using optimizations specifically formulated for the sequences in this cluster. Thus, as we will see, both of these levels are needed and that the efficiency of the proposed algorithm relies on the clustering degrees at both levels of the index structure. Figure 4 provides an overview of the two-phase EXP*dex* indexing process. Next, we explain this two-level clustering and indexing process in detail.

[3]Obviously, for queries without wildcard labels, the task of sequence matching is much simpler, and it follows the explanation given here for "$\star$" queries, ignoring the discussions related to "$\star$" wildcard labels.

Given a data sequence p to be inserted into $\mathcal{RC}$, we

1. create a new rule set $\mathcal{C}_\star$ containing three rules
$$c_i = \langle a \leftrightarrow b \rangle, \quad \langle a \leftrightarrow \text{“}\star\text{“} \rangle, \quad \text{and} \quad \langle b \leftrightarrow \text{“}\star\text{“} \rangle$$
for each rule c_i in $\mathcal{C}$. We call the additional rules, the $\star$-commutativity rules;
2. pick all possible combinations of k nodes in p and replace their labels with "$\star$"s. Let p' be one of the resulting sequences;
3. canonize each p' using $\mathcal{C}_\star$ to get $p'^{c\star}$;
4. store sequence p under each index $p'^{c\star}$ in $\mathcal{RC}$.

Figure 5: $\mathcal{RC}$ Indexing based on rough canonical sequences

1. Using the new rule set, $\mathcal{C}_\star$, we compute the canonical form, $q^{c\star}$, of the query q. Intuitively, this is the *rough* canonical form of q;
2. We, then, search for $q^{c\star}$ in $\mathcal{RC}$.

Figure 6: Querying $\mathcal{RC}$ based on rough canonical sequences

4.1 Early Wildcard Replacement

In order to implement *early wildcard replacement*, we extend the given domain alphabet, $\mathcal{A}$, to $\mathcal{A}_\star = \mathcal{A} \cup \{\text{“}\star\text{”}\}$, such that $\forall_{a \in \mathcal{A}} \ \text{“}\star\text{”} \equiv_l a$ Here, $\equiv_l$ denotes label equivalence. Furthermore, we define the rank of the wildcard symbol in such a way that $\forall_{a \in \mathcal{A}} \ rank(\text{“}\star\text{”}) < rank(a)$. Given this extension, we highlight that the following lemma holds:

LEMMA 4.1. *All legal permutations of a given sequence have the same canonical sequence. That is,*

$$\forall_{p' \in \mathcal{P}(p,\mathcal{C})} \quad p^c \equiv_l (p')^c$$

◇

EXAMPLE 4.1. *Given the sequence $p = B \cdot C \cdot A$ and the commutativity rules $\mathcal{C} = \{A \leftrightarrow B, B \leftrightarrow C\}$, the legal permutations of p (in addition to itself) under $\mathcal{C}$ are $p_1 = C \cdot A \cdot B$ and $p_2 = C \cdot B \cdot A$. Assuming that $rank(A) < rank(B) < rank(C)$, from Lemma 4.1, we see that p, p_1, and p_2 have the same canonical sequence:*

$$p^c \equiv_l p_1^c \equiv_l p_2^c = B \cdot C \cdot A.$$

◇

Although simple, this powerful lemma enables rewriting of sequences such that all legal permutations are accessible through a common popularity index.

4.2 Clustering, Indexing, and Querying Using Rough Canonical Sequences

The first level index structure, EXP, is created by identifying what we call the *rough* canonical sequences based on the labels in the input data.

Rough canonical form transformation of input sequences is done using the extended alphabet $\mathcal{A}_\star$ instead of the original alphabet, $\mathcal{A}$. The transformation is rough, in the sense that the original rule set, $\mathcal{C}$, is extended (into $\mathcal{C}_\star$) without paying

attention to what specific symbols " $\star$ " might be matching: in $\mathcal{C}_\star$, the wildcard is assumed to be able to commute with all symbols; thus acting transparently against symbol commutation rules (Figure 5). In general, a single rough canonical sequence (of length n) in $\mathcal{RC}$ indexes

- only 1 sequence, if $\mathcal{C}_\star$ is empty; and
- $n!$ sequences (assuming all these sequences are in the database) if $\mathcal{C}_\star$ contains all possible commutativity rules;

Given a query q with k wildcards (" $\star$ "s), $\mathcal{RC}$ is accessed as shown in Figure 6. Note that all sequences having the same rough canonical form as that of the query can be found using only a **single** query to $\mathcal{RC}$. Obviously, since the rule set used to establish `EXP` is *rough*, not all of the sequences indexed under $q^{c\star}$ match the query q. Therefore, we may need to further refine our search.

Note that, at this level, each sequence is considered for popularity multiple times (for each wild-card replacement). The rough canonization process, underlying $\mathcal{RC}$, clusters these wildcard-replaced sequences and maintains only those clusters that are popular (i.e., large enough). This popularity-based cluster elimination prevents the index become arbitrarily large.

4.3 Refining Rough Canonical Sequences

Let the rough cluster of sequences, corresponding to the input sequence s be denoted as $\mathcal{RC}(s)$. In the first level of the index structure, these are all indexed together. Thus, in order to search the canonical sequence q (subject to the commutativity rules in $\mathcal{C}$) within the rough cluster, $\mathcal{RC}(q^{c\star})$, we need a second index structure to refine the search. This index structure will rely on a scheme we call *rule set compression* that exploit the commonalities among the sequences in a given rough cluster. We create/maintain this second level index structure as follows: Let us assume that p is a sequence that is being indexed under the rough cluster $\mathcal{RC}(s)$.

Step 1. *Creation of a rule set specific to a given rough cluster:* The first step in the process is the creation of a rule set specific to the given rough cluster:

- *If p is the first sequence to be indexed under $\mathcal{RC}(s)$*, then we create a new rule set, $\mathcal{C}_+$, which contains

  $\langle a \leftrightarrow b\rangle, \langle a \leftrightarrow \text{“}\star_b\text{“}\rangle, \langle b \leftrightarrow \text{“}\star_a\text{“}\rangle, and \langle \text{“}\star_b\text{”} \leftrightarrow \text{“}\star_a\text{“}\rangle$

  for each rule $c_i = \langle a \leftrightarrow b\rangle \in \mathcal{C}$ where $a, b \in p$. In addition, $\forall_{a\in p}$, we include

  $$\langle a \leftrightarrow \text{“}\star_a\text{“}\rangle,$$

  into $\mathcal{C}_+$. We call the rules that apply to the new term $\star_a$, the *$\star_a$-commutativity rules*. Note that to enable wildcard matching

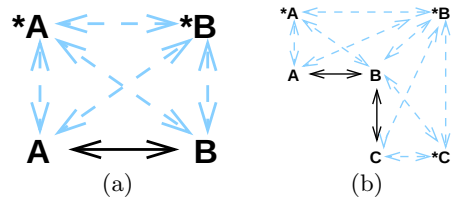
  $$\forall_{a\in\mathcal{A}} \ \text{“}\star_a\text{”} \equiv_l a,$$

  where $\equiv_l$ denotes label equivalence. Furthermore, to enable proper canonization we have

  $$\forall_{a,c\in\mathcal{A}} \ rank(\text{“}\star_a\text{”}) < rank(c)$$

  and

  $$rank(\text{“}\star_a\text{”}) < rank(\text{“}\star_b\text{”}) \ \ iff \ \ rank(a) < rank(b)$$



(a) (b)

Figure 7: Rule set compression examples: (a) The two labels can be represented under the same compressed rule; (b) none of the labels can be grouped under the same compressed rule.

- *If p is "**not**" the first sequence to be indexed under $\mathcal{RC}(s)$*, then we update the already existing $\mathcal{C}_+$ by inserting relevant rules that apply to p (if these rules are not in $\mathcal{C}_+$ already).

Step 2. *Incremental rule set compression:* For some labels a and b that appear in $\mathcal{C}_+$, $\star_a$- and $\star_b$-commutativity rules may essentially be identical to each other, meaning that both labels commute with the same set of labels that appear in $\mathcal{C}_+$. Since the number of pairwise rules that apply to a single cluster can be large, we compress $\mathcal{C}_+$ to reduce the number of rules.

For instance, for the labels a and b above, we merge the corresponding wildcards and rule sets. More specifically, if for all rules of the form $\langle \text{“}\star_a\text{”} \leftrightarrow x\rangle$ in $\mathcal{C}_+$, $\langle \text{“}\star_b\text{”} \leftrightarrow x\rangle$ is also in $\mathcal{C}_+$ (and vice versa), then we create a combined label " $\star_{a,b}$ ", remove " $\star_a$ " and " $\star_b$ ", and replace the above rules with

$$\langle \text{“}\star_{a,b}\text{”} \leftrightarrow x\rangle.$$

Furthermore, to ensure proper canonization, for the merged wildcard symbol, we set

$$rank(\text{“}\star_{a,b}\text{”}) = min\{rank(\text{“}\star_a\text{”}), rank(\text{“}\star_b\text{”})\}$$

where

$$rank(\text{“}\star_a\text{”}) < rank(\text{“}\star_b\text{”}) \ \ iff \ \ rank(a) < rank(b).$$

Figure 7(a) shows an example. In general, if there are m labels, a, b, ..., $m \in \mathcal{C}_+$, such that all of them commute with the same set of labels in $\mathcal{C}_+$, then a combined label " $\star_{a,b,..,m}$ " is created; all " $\star_\alpha$ "s that exist in $\mathcal{C}_+$ (where α may be any proper subset of labels a, b, ..., m) are removed; and all " $\star_\alpha \leftrightarrow x$" rules that exist in $\mathcal{C}_+$ are replaced with

$$\langle \text{“}\star_{a,b,c,..,m}\text{”} \leftrightarrow x\rangle$$

Similarly,

$$rank(\text{“}\star_{a,..,m}\text{”}) = min\{rank(\text{“}\star_a\text{”}), .., rank(\text{“}\star_m\text{”})\}.$$

Note that during the indexing process, each new rule inserted in $\mathcal{C}_+$ (for instance due to the insertion of a new sequence into the rough cluster) can either bring together two compressed rule sets (as with the rule between A and B in Figure 7(a)) or invalidates an existing compression. For example, the rule between B and C in Figure 7(b) destroys the A, B compression in Figure 7(b). Thus, if new rules are added to $\mathcal{C}_+$ (due to sequence p) and if they invalidate some of the existing compressed $\star$-rules in $\mathcal{C}_+$, then we need to *decompress* those affected $\star$-rules.

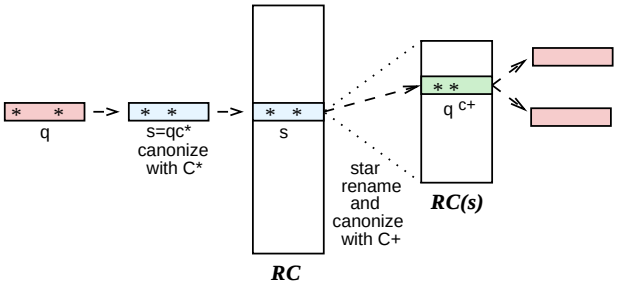


Figure 8: Querying and retrieval using EXP*dex*

At the end of the process, the total number of wildcard symbols is limited by a function of the original alphabet size and the rules, denoted as $f(|\mathcal{A}|, \mathcal{C})$. Note that $f(|\mathcal{A}|, \mathcal{C}) \leq |\mathcal{A}|$; the number of " $\star_\alpha$ "s in $\mathcal{C}_+$ for $\mathcal{RC}(s)$ can be at most $|\mathcal{A}|$, which may happen only if there exist no compression opportunities.

Step 3. *Star replacement:* For each selected node of p, we replace its label (e.g., a) with a corresponding wildcard (e.g., $\star_a$, $\star_{a,b}$, or $\star_{a,b,..,m}$). Let us denote the corresponding labeling p'. Note that there are multiple sequences p that are indexed under replacement sequence p'.

Step 4. *Canonization:* We canonize p' to get p'^{c+} under the rule set $\mathcal{C}_+$

Step 5. *Hashing and Counting:* Finally, we count the original sequence p under the hash index p'^{c+}.

At the end of the overall indexing process, those refined hash-indexes that are sufficiently large will be maintained and others (those refined clusters that are not popular) will be pruned from the index.

4.4 Retrieval

Given a query q, we first use the first level index structure to identify a rough cluster $\mathcal{RC}(q^{c\star})$ and then we use the second level index structure to find the matching sequences within the rough cluster:

1. We replace the original wildcard labels with the replacement wildcards in $\mathcal{C}_+$ (e.g., $\star_a$, $\star_{a,b}$, or $\star_{a,b,..,m}$) to get the individual queries. Let us use q' to denote one of these replacement queries.

2. Using $\mathcal{C}_+$, we compute the canonical form, q'^{c+} of the replacement query q'.

3. We, then, search q'^{c+} in the second level index.

Figure 8 depicts the retrieval process.

4.5 Complexities

Table 1 presents the notation we use in this section to describe query complexities. Table 2 uses this notation to list the query complexities of the naive and the proposed canonical-sequence based approach (EXP*dex*) via rule compression. We discuss these next.

First let us consider the alternative naive strategy. The query complexity for the alternative approach is $\mathbf{O}(|\mathcal{A}|^{\mathbf{k}})$ hash-table accesses, one for each sub-query to the hash-table. For EXP*dex*, on the other hand, the query complexity consists of several components:

Notation	
$\mathbf{O(1)}$	disk-based access cost
$O(1)$	in-memory access cost
D	number of input data sequences
l	average sequence length
k	number of $\star$ wildcards in the query
$\mathcal{A}$	alphabet of labels
cl_size_{rough}	average number of sequences in a cluster for a rough canonical sequence in the first level
cl_size_{ref}	average number of sequences in a cluster for a refined canonical sequence in the second level
$\frac{cl\_size_{rough}}{cl\_size_{ref}}$	average number of refined canonical sequences per rough canonical sequence. This gives the average size of $\mathcal{RC}(s)$; i.e. the number of refinement queries to the second level index.

Table 1: Notation used for computing complexities

	Alternative	EXP*dex*
Complexity	$\mathbf{O}(\lvert\mathcal{A}\rvert^{\mathbf{k}})$	$\mathbf{O}(\frac{\mathbf{cl\_size_{rough}}}{\mathbf{cl\_size_{ref}}})$

Table 2: Query complexity for the naive approach and EXP*dex*

- finding the rough canonical form for the query,$O(l^3)$,
- accessing the first-level index with the rough canonical form, $\mathbf{O(1)}$,
- refining the rough canonical form of the query by replacing $\star$ wildcards in the query by the $\star$ labels in $\mathcal{C}_+$ and canonizing these sub-queries using $\mathcal{C}_+$, $O(\frac{cl\_size_{rough}}{cl\_size_{ref}}) \times O(l^3)$, and
- accessing the second-level index for the sub-queries, $O(\frac{cl\_size_{rough}}{cl\_size_{ref}}) \times \mathbf{O(1)}$.

Therefore, the query complexity of EXP*dex* is

$$O(l^3) + \mathbf{O(1)} + O(\frac{cl\_size_{rough}}{\mathcal{C}_{s+}}) \times \left(O(l^3) + \mathbf{O(1)}\right).$$

The disk-based query complexities of the alternative naive approach and our proposed solution are summarized in Table 2. Our canonical-sequence based approach results in a better performance than the alternative approach, especially when cl_size_{rough} and cl_size_{ref} are of the same magnitude. This means that for query performance, the important parameters are not the absolute sizes of the clusters, but their sizes relative to each other. In the best case, there is only one refined canonical sequence for each general canonical sequence ($cl\_size_{rough} = cl\_size_{ref}$).

The storage complexity of both index structures depend on the number of popular clusters and the upper bound on the number of popular clusters the the recommendation system will choose to maintain.

5. EVALUATION

In this section, we present experiment results that validate the efficiency and effectiveness of the indexing algorithms

Data Set Properties	
# of experience sequences	1000...10000
Experience sequence length	10,20
Alphabet size (number of propositional statements in the domain)	15,50
Commutation rule sets	No rules (`None`) 50 rules (`50`) 5 commutation groups (`5C`) All commutations allowed (`All`)

Table 3: Experimented configurations

presented in this paper for popularity-indexing of experience sequences in the presence of commutative propositional statements in the domain.

- **Setup:** The data sets (Table 3) include data which have different degrees of allowed commutations: *No rules* (`None`) correspond to the case where there is no commutativity; *50 rules* (`50`) correspond to the case where there are 50 randomly created rules in the system; *5 intra-commutative groups* (`5C`) correspond to the case where the alphabet is split into 5 groups such that labels within each group commute, but they do not commute with labels in other groups; and *all applicable rules* (`All`) correspond to the extreme case where all labels are allowed to commute with each other. To observe the impact of the number of "$\star$" wildcards in queries, we also created 1- and 2-$\star$ index structures.

In order to observe and interpret the performance of the proposed optimization mechanisms, in addition to `EXP`*dex*, we indexed all input sequences after canonizing them, using a hash table. This corresponds to the *don't-care* unaware indexing scheme presented in Section 3.1.

The implementation of the algorithms was done in Java and ran on Redhat 7.2 Linux workstations, with 1 GB RAM, and 1.8 GHz Pentium IV processor. For disk-based hash implementations, we used BerkeleyDB.

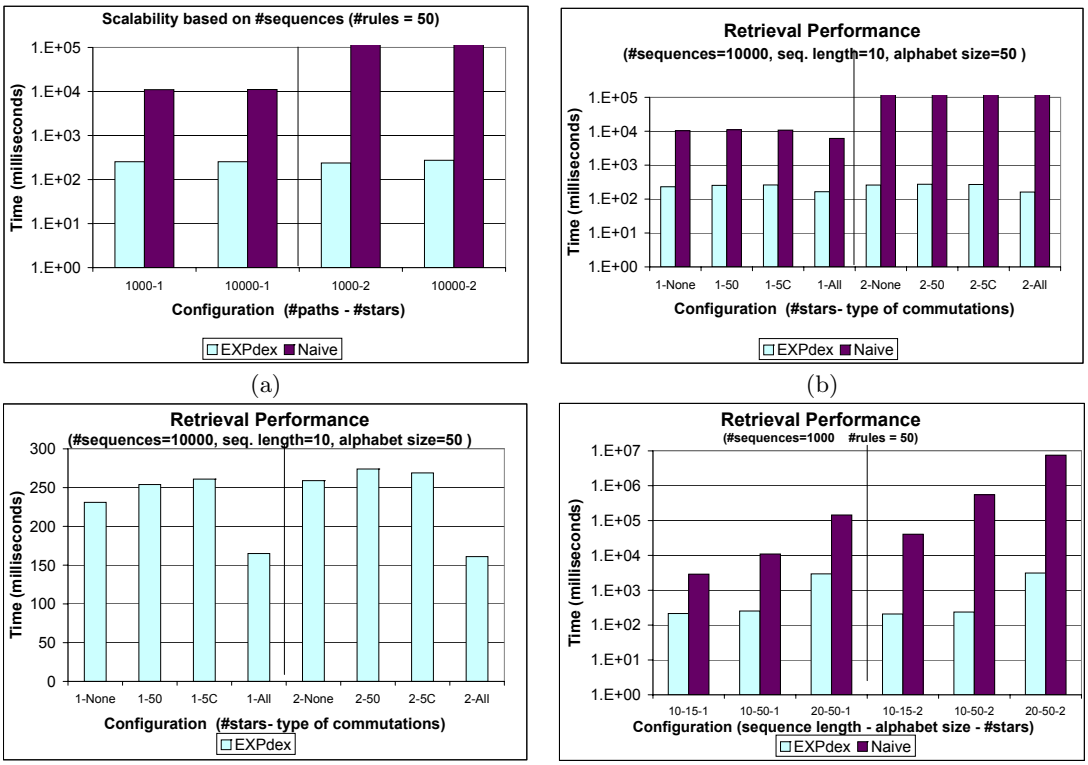
We provide our experiment results in Figures 9(a) through (d).

- **Performance:** In the experiments on synthetic data, `EXP`*dex* performed 10 to 1000 times faster than the naive approach as can be seen in Figures 9(a), (b), and (d).
- **Scalability for performance based on the number of sequences:** Figure 9(a) shows that, as expected (Section 4.5), for both 1-$\star$ and 2-$\star$ queries, there is no visible effect on the performance of `EXP`*dex* as the number of sequences indexed increases. The performance of the naive approach is also not affected by the increase in the number of sequences for 1-$\star$ or 2-$\star$ queries, as predicted.
- **Impact of the number of "$\star$"wildcards in queries:** Figure 9(a) also shows the impact of the number of wildcards on the performance. `EXP`*dex*, as expected, scales very well to the increase in the number of wildcards in the query as the performance is not significantly affected by the number of $\star$s. As predicted in Section 4.5, the naive approach is sensitive to the number of $\star$s, and gets 10 times slower when there are 2 wildcards. This is also visible in Figure 9(b), which shows a larger set of configurations.
- **Impact of the possible commutations:** Figure 9(c) shows the impact of rule set compression. As predicted in Section 4.5, in `EXP`*dex*, the query performance is not significantly affected by the amount of clustering possible. The only exception to this is the extreme case, when all possible commutations are allowed. Then, `EXP`*dex* performs faster than in the other cases. The reason for this is that in this particular case, the database size becomes very small due to large amount of clustering, and the hash becomes more effective (Figure 9(e)).
- **Impact of the sequence length and the alphabet size:** Figure 9(d) shows that the length of the indexed sequences impacts both the naive approach and `EXP`*dex*. The impact of the alphabet size is negligible for `EXP`*dex* compared to the impact of the sequence size. As predicted in Section 4.5, the naive approach is considerably affected by the increase in the alphabet size, and the performance degradation increases as the number of $\star$s increases.
- **Summary:** The experiments showed that the `EXP`*dex* indeed works efficiently and it scales much better than the naive approach when the number of clusters in the data as well as the number of wildcards in the query increase. In comparison, the alternative naive approach requires significantly larger amount of time to process recommendation queries than `EXP`*dex*.

6. RELATED WORK

The traditional string matching, or more generally sequence matching, problem is to find an occurrence of a pattern (a substring or a subsequence) in a string or a sequence, or to decide that none exists. This is a well-studied problem. Two of the best known algorithms for the problem of string matching are the Knuth-Morris-Pratt(KMP) [24] and Boyer-Moore (BM) [7] algorithms. KMP slides a window (of size m) over the data string (of size n), but skips unpromising window positions to achieve linear, $O(m+n)$, worst case execution time. BM allows linear-time and linear-space preprocessing to achieve *sublinear*, $O(nlog(m)/m)$, average search time; though its worst case behavior, $O(mn)$, is worse than that of the KMP. There are a number of index structures for efficiently accessing strings and sequences. These include tries [9] (which provides $O(l)$ access to data, where l is the length of the maximum *keyword* in the data string), Patricia tries [39] (where certain paths are compressed to reduce the number of nodes), and suffix array [27] and suffix tree [29,41] data structures (where each position in the data string is considered as a *suffix* and each such suffix is indexed using a trie or a patricia trie). Search time on a suffix tree is $O(m)$. Aho-Corasick trie [3] provides search for patterns by extending KMP with a trie-like data structure for all input patterns. It provides $O(n)$ search with $O(m)$ trie construction time. In a similar fashion, Commentz-Walter algorithm [10] extends BM with a trie of input patterns to provide simultaneous search for multiple patterns. A suffix automaton is similar to trie, in the sense that it recognizes all suffixes of a given string. BDM algorithm [12] creates a suffix automaton for the input pattern (and it lends itself to parallelizations [31]). Suffix automaton is also used in [13] to construct Directed Acyclic Word Graphs (DAWGs) of input words. In contrast to the above algorithms that work on the plain-text domain, to improve efficiency, Karp-Rabin algorithm [22] compares strings' hash values.

Unlike the above data structures which help search for exact matches, approximate string or sequence matching algorithms focus on finding patterns that are *not too different* from the ones provided by the users [35]. [26] presents a $O(kn)$ dynamic programming based algorithm for locating



(*Note that the performance numbers are in log scale for (a), (b), and (d)*)

Figure 9: (a) performance impact of the number of $\star$s and the number of sequences; (b) impact of clustering opportunities on the query performance for 1-$\star$ and 2-$\star$ queries ("`None`" corresponds to no commutation, "`50`" corresponds to 50 randomly selected rules, "`5C`" corresponds to 5 commutative groups, and "`All`" corresponds to the extreme case where all labels are allowed to commute); (c) the same as (b) but only for `EXP`*dex* plotted alone in milliseconds; (d) impact of the sequence length, the alphabet size, and the number of $\star$s.

matches with at most k errors. [40] constructs a (large) finite automaton which counts the number of errors observed during the matching process. [42] uses suffix trees for approximate string matching. The edit distance (such as Levenshtein or Hamming distances) between two strings is defined as the minimum number of character inserts, deletes, and changes needed to convert one string to the other; various algorithms have been developed to find strings that are closer in edit distance [11, 25]. [11] considers the case where swaps between consecutive symbols are allowed and develops an $O(nlog(n))$ dynamic programming based algorithm.

An interesting variant of non-exact string matching problem is when wildcard symbols are allowed [4,30]. In this case, matches that differ from each other can not be ordered with respected to an *edit distance* which favors some matches over others. In fact, regular-expression based frameworks further generalize this. For instance, the RE-tree data structure introduced in [8] enables efficient search of regular expressions in a given input string. [5] uses the Patricia tree as a logical model and presents algorithms with sublinear time for matching regular expressions. Although these techniques capture wildcards, the existence of commutativities in experiences render a straight-forward implementation of these algorithms ineffective for experience-driven recommendations.

7. CONCLUSION

In this paper, we recognize that an experience-driven recommendation system should be able to capture the past states of the individual and the future states that the individual wishes to reach, while identifying certain proposition statements to recommend to the individual based on this context (past history and future goals). Thus, we pose the experience-driven recommendation task as a sequence matching problem. In particular, we highlighted that popularity-indexing of the experiences has to be cognizant of the *commutation* rules underlying the domain as well as the possible *don't-care*s in recommendation tasks. We also showed that efficient indexing of sequences, which are commutative under a given rule set, can be achieved by *canonizing* the sequences in advance. We pointed out that if the recommendation tasks involve `don't cares`, performing *early wildcard replacement* is needed. To retrieve popular experience sequences (with commutations) efficiently, we further introduce a *sequence canonization with rule compression* technique for handling *wildcard* queries. These two techniques are brought together in a novel index structure `EXP`*dex*. We showed that alternative, `don't care`-unaware solutions are significantly slower than `EXP`*dex*.

Our future work will involve the application of this index structure for providing experience-driven recommendations to blind students and its evaluation in various navigational contexts, such as online navigation within educational web sites and/or physical navigation within real-world campus environments.

8. REFERENCES

[1] S. Adalı, B. Bouqata, A. Marcus, F. Spear, and B. Szymanski. A day in the life of a metamorphic petrologist. *ICDE Workshops*, 2006.

[2] S. Adalı and M. L. Sapino. An activity based data model. *Semantic Desktop Workshop* 2005.

[3] A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *CACM*, 18(6):333–340, June 1975.

[4] A. Amir, G.M. Landau, M. Lewenstein, and N. Lewenstein. Efficient special cases of pattern matching with swaps. *Information Processing Letters*, 68(3):125–132, 1998.

[5] R. Baeza-Yates and G. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *JACM*, 43(6):915–936, 1996.

[6] J. Blustein, C. Fu, and D. L. Silver. Information visualization for an intrusion detection system. *Hypertext* '05. pp. 278–279, New York, NY, 2005.

[7] R. Boyer and J. Moore. A fast string-searching algorithm. CACM, 20:762–772, 1977.

[8] C.-Y. Chan, M. Garofalakis, and R. Rastogi. Re-tree: An efficient index structure for regular expressions. *VLDB*'94.

[9] D. Comer and R. Sethi. The complexity of trie index construction. *JACM*, 24(3):428–440, July 1977.

[10] B. Commentz-Walter. A string matching algorithm fast on the average. *ICALP*, pages 118–132, 1979.

[11] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *SODA*'02.

[12] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12(4/5):247–267, Oct. 1994.

[13] M. Crochemore and R. Vrin. Direct construction of compact directed acyclic word graphs. *CPM'97*, pp 116–12. LNCS 1264, 1997.

[14] D.Dasgupta and F.A.Gonzalez. An intelligent decision support system for intrusion detection and response. *MMM-ACNS*'01.

[15] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31, :805–822, 1999.

[16] M.E. Dönderler, K.S. Candan, S.Wu, L.Peng, and J.W.Kim. Adaptive electronic course content delivery for students who are blind. *ASSETS* 2004.

[17] M.E. Dönderler, L.Peng, and K.S. Candan. Adaptive content delivery to assist blind students in accessing course materials. *Accessing Higher Ground Conference :Assistive Technology and Accessible Media in Higher Education*, 2003.

[18] G. Fischer. User modeling in human-computer interaction. In *User Modeling and User-Adapted Interaction*, 2001.

[19] J. Gemmell, G. Bell, and R. Lueder. Mylifebits: a personal database for everything. *CACM*, 49(1):88–95, 2006.

[20] R. Jain. Experiential computing. *CACM* 46(7): 48-55.

[21] R. Jain. Multimedia electronic chronicles. *IEEE MultiMedia*, 10(3):111–112, 2003.

[22] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res.*, 131(2), 1987.

[23] J.W. Kim, K. S. Candan, M. Dönderler. Topic segmentation of message hierarchies for indexing and navigation support. *WWW* 2005.

[24] D.Knuth, J.Morris, and V.Pratt. Fast pattern matching in strings. SIAM J. of Computing, 6(2):323–350, 1977.

[25] S. Kurtz. Approximate string searching under weighted edit distance. *WSP'96*, pp. 156–170, 1996.

[26] G. M. Landau and U. Vishkin. Fast string matching with k differences. *J. Comput. System Sci.*, 1988.

[27] U. Manber and E. Mayers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[28] C.B. Mayer, K.S. Candan, V. Sangam. Effects of user request patterns on a multimedia delivery system. *MTAP*, 24(3):233–251, 2004.

[29] J. I. Munro, V. Raman, and S. S. Rao. Space efficient suffix trees. *J. Algorithms,* 39:205–222, 2001

[30] S.Muthukrishnan and H.Ramesh. String matching under a general matching relation. *Inform. and Computation*, 122(1):140–148, 1995.

[31] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. CPM'98, pp. 14–33, 1998.

[32] Yan Qi and K. Selcuk Candan. CUTS: CUrvature-based development pattern analysis and segmentation for blogs and other Text Streams. *Hypertext* 2006.

[33] M.L. Sapino, K.S. Candan, and P. Bertolotti. Log-analysis based characterization of multimedia documents for effective delivery of distributed multimedia presentations. *DMS* 2006.

[34] M.L. Sapino, K.S. Candan, J.W. Kim, and F. Antonelli. Annotating educational discussion boards to help students who are blind. *Int. J. of Continuing Education and Life-Long Learning.* accepted 2006.

[35] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J.of Algorithms*, 1980.

[36] P. Singh, B. Barry, and H.Liu. Teaching machines about everyday life. *BT Technology Journal*, 22(4), 2004.

[37] P. Singh, T. Lin, E. Mueller, G. Lim, T. Perkins, and W. Zhu. Open mind common sense: Knowledge acquisition from the general public, LNCS2519, 2002.

[38] H. Sridharan, H. Sundaram, and T. Rikakis. Computational models for experiences in the arts, and multimedia. In *ETP '03: Proceedings of the 2003 ACM SIGMM workshop on Experiential telepresence*, pp. 31–44, 2003.

[39] W. Szpankowski. Patricia tries again revisited. *J. of the ACM*, 37(4):691–711, 1990.

[40] E. Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6:132–137, 1985.

[41] E. Ukkonen. Constructing suffix trees on-line in linear time. *Proc. Information Processing 92*, 1:484–492, 1992.

[42] E. Ukkonen. Approximate string matching over suffix trees. *Lecture Notes in Computer Science*, 684:228–242, 1993.