A Temporal Pattern Search Algorithm for Personal History Event Visualization

Taowei David Wang, Amol Deshpande, and Ben Shneiderman

Abstract—We present Temporal Pattern Search (TPS), a novel algorithm for searching for temporal patterns of events in historical personal histories. The traditional method of searching for such patterns uses an automaton-based approach over a single array of events, sorted by time stamps. Instead, TPS operates on a set of arrays, where each array contains all events of the same type, sorted by time stamps. TPS searches for a particular item in the pattern using a binary search over the appropriate arrays. Although it is considerably more expensive per item, it allows TPS to skip many unnecessary events in personal histories. We show that TPS's running time is bounded by $O(m^2 n \lg(n))$, where m is the number of items in a search pattern, and n is the number of events in a history. We also show that although the asymptotic running time of TPS is inferior to that of a non-deterministic finite automaton (NFA) approach $(O(mn))$, under our experimental conditions, TPS consistently outperforms NFA. Since the experimental conditions we describe here subsume the conditions under which users would typically use TPS (i.e. within an interactive visualization program), we argue that TPS is the appropriate design choice for us. Moreover, TPS and the strategy to split input data into several sorted arrays is generalizable to other applications.

Index Terms— Graphical user interfaces, human-centered computing, information visualization, pattern matching,

1 Introduction

Temporally ordered events can often reveal interesting information. Understanding whether a particular pattern occurs, how frequently certain pattern occur, and whether one pattern occurs more often than the others are common questions an analyst would pursue. There has been much attention from both the academia and the industry to develop analysis tools focused on temporal events and their temporal patterns in a variety of domains: health care and electronic health records (EHR) [20], [4], business intelligence [18], and web server log analysis [13]. In particular, electronic health records have gained much attention in recent years. Clinicians recruiting participants for clinical trials often screen candidates by reviewing their records. In the initial stages, they would issue queries to find participants with certain features. These features include patient attributes such as age and gender, but also often include a temporal pattern of event. Some query patterns are as simple as finding participants who "had a stroke after a heart attack". Some may involve negation, e.g. "diagnosed with breast cancer without having a prior mammogram", and some may be more complex, e.g. "with no prior history of heart problem, later diagnosed with hypertension, received no treatments, and finally experienced a heart attack".

While the typical model to do these tasks is to perform the search using a command line language such as SQL, and then review the results in an analysis tool, there are high costs associated with this approach. Because domain analysts often do not know the query language, they have to rely on an additional SQL expert to formulate the queries. The collaboration turnaround may take hours, if not days. In the situation where an analyst is iteratively refining a query for the purpose of exploratory analysis, the multiple turnarounds are detrimental to the process. Furthermore, executing these queries on personal histories that have hundreds or thousands of events with tens of thousands of histories in a database can take a long time. It would be ideal to issue simpler conjunctive queries to the database, and then allow the analyst to perform temporal pattern searches themselves in an analysis tool, where data can be further structured to better support their analyses. However, not all analysis tools provide the features to search for temporal patterns.

Lifelines2, our interactive visualization tool, is one that does. It visualizes each electronic health record as a horizontal stripe on a time line (Figure 1). Within each record, the events of the same type are placed in the same horizontal line. Event types are differentiated by color. To clarify, by *record* (as in *electronic health record*) we mean a collection of event histories associated with an entity (such as a patient in the context of an electronic health

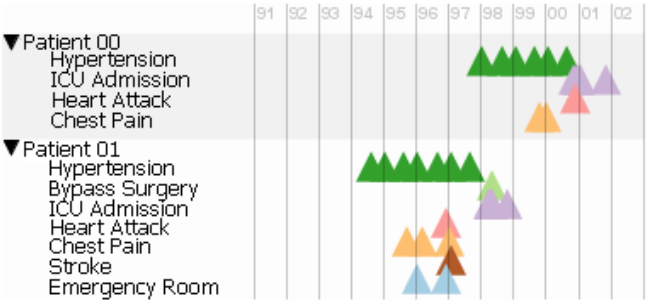

Fig. 1. A partial screen shot of Lifelines2 display. Two electronic health records for fictional heart disease patients are shown here. The color-coded triangles represent each individual event (diagnosis, intervention, chief complaints, etc.), and events of the same type are drawn in the same horizontal line within a record.

• *The authors are with the University of Maryland at College Park, College Park, MD 20742. E-mail:{tw7, amol, ben}@ cs.umd.edu*

Manuscript received (insert date of submission if desired). Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.

record) and not tuples as in the database literature.

Lifelines2 provides several ways for users to filter electronic health records by their features. One way is via an event sequence filter, where users specify a pattern that describes a sequence of temporally ordered events as search criteria to find patients. Previous research suggested that complex interfaces that support full control of all temporal constraints among items in a pattern often overwhelm users [6]. Instead, we decided on a simplified temporal pattern specification interface. Although more restricted, it can still specify all the aforementioned example medical queries. We allow users to specify a temporal pattern such that each item in the pattern can be a presence item or an absence item (negation). These items are selected from a list of combo boxes (Figure 2). The ordering of the combo boxes determines the temporal ordering. We do not allow other additional temporal constraints (e.g. a `Stroke` occurs within 3 days after a `Heart Attack`) among the items in the pattern. Once a pattern is specified, Lifelines2 finds all records that contain such pattern and visually filters out the rest.

We made the following design decisions on how we internally represent the event histories in Lifelines2. Each record is represented as a set of sorted arrays of events. There is one array for each event type. All events of the same type are sorted by their time stamps in their array. This decision comes from the following constraints:

1. **Data Constraint**: It seems most straight-forward to store all events on a single sorted array regardless of type. However, for events that have the same time stamp, this scheme can create conflicts. Using one array for each event type allows us to circumvent this problem. (However, we assume events that have the same type and the same time stamp are the same event).
2. **Drawing Constraint**: Lifelines2 visualization always displays the events of the same type on the same horizontal line, a separated and sorted data structure would let us efficiently access the desired events (no search required, only iterate).
3. **Interface Constraint**: One of the main features of Lifelines2 is to visually align the nth event of a specific type to allow users to examine what other events occur before or after relative that the alignment. We have shown that alignment can improve user performance to understand the data by over 60% in a previous study [20], so we make every effort to best support this feature. It is imperative to be able to quickly find the nth event of a given type. In addition, it is useful to analysts to hide event types that are not of interest. These interface features involve finding event data of a specific type. Separating events into different arrays by type would allow Lifelines2 to afford these features more efficiently.

These constraints present certain challenges to searching temporal event histories, but also present a unique opportunity to explore and exploit this data structure. This paper focuses on our exploration in designing a temporal pattern searching algorithm in the presence of these constraints, especially the latter two constraints that involve human performance and visual design. We first describe the problem more formally, provide related work, and present the Temporal Pattern Search algorithm. We then analyze the worst-case running time of the algorithm and evaluate its performance against existing common approaches.

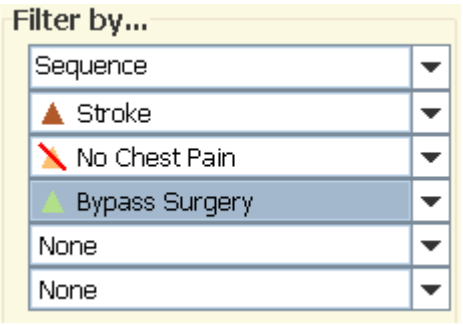

Fig. 2. This is the user interface widget in Lifelines2 for specifying temporal patterns. Each item in the pattern is specified via a combo box. The top-most combo box specifies the first item in the pattern, and the bottom-most specifies the last. This screen shot describes the pattern query: "find all patients who have experienced a stroke and followed by a bypass surgery without having complained about chest pain during that period." Note that the specification of the absence of events is done in the same fashion as the presence of events.

2 Problem Description

An event e is defined, and uniquely identified by an event type $T = T_e$ and a time stamp $t = t_e$. A personal record consists of a set of k sorted arrays. Each array corresponds to an event type T_{e_i}, where $i \in [1..k]$, and contains all events of that type in the person's history, sorted by their time stamps.

A temporal pattern contains temporally ordered events as items, and each item may be a presence or absence (negation) item. More formally, a temporal pattern P is $P = p_1 p_2 p_3 \ldots p_m$ where each p_i is an event type T_{p_i} or its negation $\bar{T}_{p_i}$. In this paper, we use the word negation and absence interchangeably. In the case that a temporal pattern contains only positive items (no negation), the pattern $P = p_1 p_2 p_3 \ldots p_m = T_{p_1} T_{p_2} T_{p_3} \ldots T_{p_m}$ matches a personal record if and only if $\exists e_1, e_2, e_3, \ldots, e_m$ such that $T_{e_i} = T_{p_i}$ and $t_{e_i} < t_{e_{i+1}} \; \forall i \in [1, m-1]$ in the record. In other words, the event types in the specified search pattern exist in the record, and that they occur in the order the pattern specifies.

If a negation exists, e.g. $p_1 p_2 p_3 = T_{p_1} \bar{T}_{p_2} T_{p_3}$, then the pattern matches a record if and only if $\exists e_1, e_3$ such that $T_{e_1} = T_{p_1}$ and $T_{e_3} = T_{p_3}$ and $t_{e_1} < t_{e_3}$ and $\nexists e_2$ such that

$T_{e_2} = T_{p_2}$ and $t_{e_2} \in (t_{e_1}, t_{e_3})$. If there exist contiguous negation items such as $p_1 p_2 p_3 p_4 = T_{p_1} \bar{T}_{p_2} \bar{T}_{p_3} T_{p_4}$, then the pattern matches a record if and only if $\exists e_1, e_4$ such that $T_{e_i} = T_{p_1}$ and $T_{e_4} = T_{p_4}$ and $t_{e_1} < t_{e_4}$ and $\nexists e_2, e_3$ such that $(T_{e_2} = T_{p_2}$ and $t_{e_2} \notin (t_{e_1}, t_{e_4}))$ or $(T_{e_3} = T_{p_3}$ and $t_{e_3} \notin (t_{e_1}, t_{e_4}))$. The semantics of the temporal patterns we discuss in this paper can be mapped into regular expressions, for example, $P = T_{p_1} \bar{T}_{p_2} \bar{T}_{p_3} T_{p_4}$ can be translated into an equivalent regular expression $.*T_{p_1}[\hat{}\,T_{p_2} T_{p_3}]*T_{p_4}.*$, where "." is the wildcard character, "*" the Kleene star, "^" the negation, and "[...]" a class of symbols. A straight application of the regular expression would be fine if our data were amendable to be structured as a single string. The problem we have is how to search for records that match the specified pattern under the three constraints we have discussed.

3 Related Work

At the first glance, string matching is an obviously related area. However, there are some fundamental differences to searching for temporal patterns. In classical string matching, a pattern is wholly specified ("abc" as opposed to "a" followed by "b" followed by "c"), no two characters can occur at the same position in a string, and, finally, negations are not usually considered. Nevertheless, it would be useful to examine these clever algorithms to see if their approaches can be generalized for our purposes: Knuth-Morris-Pratt (KMP) [11], Boyer-Moore (BM) [2], and Rabin-Karp (RK) [10]. KMP preprocesses a given pattern to build a prefix table that tells the algorithm when and how far ahead (in number of characters) it can skip, avoiding unnecessary scans. Likewise, BM builds two tables based on the pattern which tells the algorithm how much to safely skip ahead. Finally, RK utilizes the fact that a string can be viewed as a number, where each character is represented by a digit (in radix = size of the alphabet), and one can use modulo equivalences to speed up checking for matches.

These string matching algorithms all exploit the fact that a string is contiguous, every character (except for the first one and the last one) has only one predecessor and a successor, and the characters are indexable. This information, along with the known pattern, can be used to reduce the number of scans in the text. Sadri et al. used this fact to optimize sequential searches in SQL-TS over data streams by using a variant of KMP [17]. The drawback is that the items in the pattern must be contiguous as if they are characters in a string, limiting the expressiveness of their search patterns. To search for the type of temporal patterns as we have defined, we cannot simply apply these string matching algorithms.

Other string match approaches such as the Shift-And and Shift-Or algorithms and Backward Nondeterministic Dawg Matching algorithm can be more easily adapted to deal with classes of characters, optional and repeatable characters, or other extensions [15] [16]. These algorithms rely on clever use of bit parallelism to efficiently perform the search. However, these algorithms also assume the inputs to be a single string, and cannot be applied to sets of arrays as in our case.

The traditional method to match regular expressions in strings utilizes either deterministic or non-deterministic finite automaton (DFA or NFA) [19]. Given a text of length *n* and a pattern of length *m*, a DFA can perform a search in *O(m),* but building a DFA can take up to $O(m^2)$ space, which is only useful for applications where building such DFA is infrequent or can be done offline, such as network intrusion detection systems [21], [12], [7]. On the other hand, an NFA can compactly represent an equivalent automaton in space *O(m)*, but the time may take up to *O(mn)* time.

Given the tradeoff between DFAs and NFAs, many systems choose to use NFAs or extensions of NFAs [1], [5]. These approaches tend to have an expressive pattern language where negations, Kleene closures, and temporal constraints are included. They are more expressive than regular expressions. These systems are geared towards fast processing over continuous event streams, where an event is more complex, and contains additional attributes. Our approach focuses on a simpler problem where events do not have additional attributes, and this allows us to design simpler algorithms. For example, in [1], searching with negation of events is supported by first finding all positive events and then pruning off the results that contain negation events in the wrong temporal ordering. In contrast, our algorithm searches for negations in-place.

Harada et al. developed a query language and algorithm to search for patterns in multiple personal histories. Their approach assumes a grouping over a column of data (e.g. customer ID) and an ordering by a second column (e.g. time stamp) in the data structure, and performs pattern search algorithms over this structure [8], [9]. They do not use an NFA approach to perform this search. Instead, they developed an algorithm that resembles building a topological graph. The expressiveness of their language allows the specification of only limited negation. For example, let `a`, `b`, `c`, `d` be event types, their approach can define patterns that have the same semantics as the regular expression `.*a.*[^bd].*c.*`, but not `.*a[^bd]*c.*`, which our approach does. This limitation means that their algorithm never has to backtrack. There was no reported run time analysis for their algorithm.

4 TPS Algorithm Description

Code Listing 1 shows the pseudo code for the algorithm Temporal Pattern Search. `IS_A_MATCH(R, P)` takes a record `R`, and a temporal pattern `P`. A record `R` is a table of arrays, indexed by event types. Each array contains all events of the same type, and can be accessed by `R[T]`, where `T` is the event type. Each event `e` has a time stamp

CODE LISTING 1
STARTING FUNCTION AND INITIALIZATION OF GLOBAL VARIABLES

```
0  IS_A_MATCH(R,P)
1     β ← FALSE       //backtrack flag
2     T[P.length]     //matched times
3     Δ[P.length]     //last pos item
4     Φ[P.length+1]   //next neg item time
5     Π[P.length]     //if has neg item before
6     χ ← 0           //current index
7     τ ← -∞          //current time
8     δ ← -1          //last pos item
9
10    while (χ < P.length)
11       if (χ == -1)
12          return FALSE
13       if (P[χ].isNeg)
14          HANDLE_ABSENCE(R,P)
15       else
16          HANDLE_PRESENCE(R,P)
17    return TRUE;
```

CODE LISTING 2
THE HANDLE_ABSENCE FUNCTION

```
18 HANDLE_ABSENCE(R,P)
19    minTime ← NIL
20    numAbs ← 0
21    for (i←χ to P.length)
22       item ← P[i]
23       if (NOT item.isNeg) break
24       numAbs ← numAbs + 1
25       absEvent ← NEXT(R[item.Type], τ)
26       if (AbsEvent == NIL)
27          continue
28       minTime← MIN(absEvent.Time, minTime)
29    if (minTime == NIL)
30       if (δ > -1)
31          Φ[δ] ← ∞
32       else
33          Φ[Φ.length-1] ← ∞
34       χ ← χ + numAbs
35    else
36       if (χ + numAbs < P.length)
37          nItem ← P[χ+numAbs]
38          nEvent ← NEXT(R[nItem.Type,] τ)
39          Π[χ+numAbs] ← TRUE
40          if (nEvent == NIL OR
41              nEvent.Time > minTime)
42             χ ← δ
43             if (χ > 0)
44                δ ← Δ[χ]
45                τ ← minTime-1
46             β ← TRUE
47          else
48             T[χ + numAbs] ← nEvent.Time
49             Φ[δ] ← minTime
50             Δ[χ+numAbs] ← χ-1
51             δ ← χ + numAbs
52             χ ← χ + numAbs + 1
53             τ ← nEvent.Time
54       else
55          χ ← δ
56          δ ← Δ[χ]
57          τ ← minTime-1
58          β ← TRUE
```

`e.Time` and a type `e.Type`. `P` is an array of pattern items. Each item includes an event type `item.Type`, and a flag indicating whether it is a negation item `item.isNeg`. Table 1 shows a trace of the TPS algorithm with example `P` and `R`. Even though we have specifically defined a record to be a set of sorted array of events, for the ease of narration, we have represented the input record `R` as an array in Table 1 and refer to its i^{th} elements using the array notation `R[i]` in the following discussions.

In addition to the details that are shown, we assume that there exists the two following functions `NEXT(R[T],t)` and `PREV(R[T],t)`, where `R` is a record, `T` is an event type, and `t` is a time stamp. `NEXT(R[T],t)` returns an event e of type `T` in `R` such that `e.TIME > t` and that there is not another event `d` of type `T` in `R` such that `d.TIME < e.TIME`. If no such event e exists, the function returns `NIL`. In other words, `NEXT(R[T],t)` returns the event of type `T` that occurs after and closest to the given time stamp t, if it exists. `PREV(R[T],t)` is similar; it returns the event of type `T`, if exists, that has the largest time stamp, but still less than `t`. Since all event arrays in records are sorted by time stamps, we assume the `NEXT(R[T],t)` and `PREV(R[T],t)` functions use efficient binary searches.

4.1 Overview

The main loop `IS_A_MATCH(R,P)` processes an item from the pattern P one at a time. The variable χ (current index on the search pattern) keeps track of which item is current. When all items in the pattern have been found, and no constraints from negation events are violated, TPS returns `TRUE`. If χ is set to be -1, it means that there is no match and the main loop returns `FALSE`.

CODE LISTING 3
THE HANDLE_PRESENCE FUNCTION

```
56 HANDLE_PRESENCE(R,P)
57    event ← NEXT(R[P[χ].Type], τ)
58    if (event == NIL)
59       χ ← -1
60    else
61       backtrackMore ← FALSE
62       if (β AND Π[χ])
63          if (Δ[χ] < 0)
64             badTime ← Φ[Φ.length-1]
65          else
66             badTime ← Φ[Δ[χ]]
67          if (badTime < event.Time)
68             χ ← Δ[χ]
69             τ ← Φ[χ]-1
70             δ ← Δ[χ]
71             backtrackingMore ← TRUE
72       if (backtrackingMore)  return
73       τ ← event.Time
74       T[χ] ← event.Time
75       Δ[χ] ← δ
76       δ ← χ
77       χ ← χ+1
78       β ← FALSE
```

When processing an item in the pattern, if it is a presence item, `IS_A_MATCH(R,P)` calls `HANDLE_PRESENCE(R,P)`, which attempts to find an event that satisfies the current item. If it is an absence item, TPS calls `HANDLE_ABSENCE(R,P)`, which finds the next absence event, and checks to see if that absence event occurs between the previous presence item match and the next presence item match. If it does, then a constraint is violated, and the algorithm backtracks. Backtracking means TPS tries to look for an alternative to one or more of its previously made matches. The algorithm increments the variables χ (the current item on pattern) and τ (the current time) when processing the search. When backtracking occurs, TPS rolls back these variables (among others) appropriately to restart a previous search.

The first thing to notice about this description is that there would be no need for backtracking if there were no absence events. Furthermore, when it backtracks, it backtracks only to the previous closest presence item (never to an absence item). The characteristics on how the algorithm backtracks determine what states we need to keep track of.

We use Greek letters to denote global variables. Capital letters represent arrays, and lower case ones represent scalar variables. Aside from χ and τ , Code Listing 1 also shows the initialization of T, Δ, Φ, Π, β, and δ. They describe the states TPS is in when matching and save previously-done work to reduce unnecessary backtracking. β indicates whether we are backtracking. δ keeps track of the index of the previously matched presence item (with respect to χ). Δ is an array of length `[P.length]`, and it keeps track of the index of the last presence item for each item in the pattern. So `Δ[i]` indicates the index of the last presence item for the ith item in the pattern (Δ is an array of δ's). T is another array of the same length, and keeps track of the time stamps for all previously-matched presence items.

For each item in the pattern, Φ keeps track of the known time for the following absence event (if the following item is an absence event). This allows TPS to skip some fruitless matches during backtracking. Suppose we are searching for pattern $P = A\bar{B}C$ in *AAAAAAABAC*. *P[0]* will initially match the first *A* in the record. Upon encountering *B* in the record and backtracking to *P[0]* = *A*, the next search should start from the time stamp of the known next absence event *B* (instead of from the time stamp of the known bad *A*) to avoid the multiple, unnecessary matches to all of the *A*'s before *B* in the record. Φ has length *P*.`length+1`. The extra space at the end is used when a pattern starts with a negation, and needs a default place to remember that absence item. Finally, for each item in the pattern, Π keeps track of whether it has an absence item immediately before that item. If Π[i] = `FALSE`, it indicates that there is no absence item prior to the ith item in the pattern. Because there is no absence item immediately prior, when TPS finds an alternative match for the ith item in backtracking, the algorithm does not need to check whether this alternative match would violate an existing constraint. Π is not necessary, but makes our narration easier.

We introduce the term *absence block* in a search pattern. An absence block is a contiguous block of absence event items in the search pattern, generally surrounded by presence events on either end (except when the block is the leading or the ending part of the pattern). For example, the pattern $P = T_{p_1}\bar{T}_{p_2}\bar{T}_{p_3}T_{p_4}\bar{T}_{p_5}T_{p_6}$ consists of two absence blocks $\bar{T}_{p_2}\bar{T}_{p_3}$ and $\bar{T}_{p_5}$, of size 2 and 1 respectively. When the current item in the pattern `P[χ]` is an absence item, `HANDLE_ABSENCE(R, P)` (Code Listing 2) is called. In lines 21-28, TPS looks ahead in the pattern to determine what events are in the absence block. For each event type in the absence block, the algorithm finds the next event of that type in the record, and saves the minimum time stamp value ofver all such events as `minTime`. If `minTime` does not exist, then that means there are no events of these types in the record, and we can safely increment χ and remember that these events do not exist by saving this fact in Φ (lines 30-34). Otherwise, we look ahead for the next presence item, and find a matching presence event in the record and compare its time stamp with `minTime`. If it occurs before `minTime` (no violation), then we can move to the next item in the pattern and update our states (lines 48-53). However, if it occurs after `minTime`, we have to backtrack (lines 42-46).

When we backtrack, TPS set β to false to indicate that it is in backtrack mode. When in backtrack mode, any subsequently matched item will require an additional check to see if a prior constraint is violated. For example, in

TABLE 1
TRACE OF THE TPS ALGORITHM USING EXAMPLE RECORD R AND PATTERN P

index:	0	1	2	3	4	5	6	7	8	9
time:	0	10	20	30	40	50	60	70	80	90
Record **R** =	A	C	E	B	C	A	C	D	C	F

index:	0	1	2	3	4	5
Pattern **P** =	A	$\bar{B}$	C	$\bar{D}$	$\bar{E}$	F

	Current index χ and current time τ	Binary Search Performed in matching pattern with data	Code Line #	β	T	Δ	Φ	Π	χ	τ	δ
0		initialization		FALSE	[ ]	[ ]	[ ]	[ ]	0	-∞	-1
1	χ = 0, τ = -∞	matching P[0] = A … R[0]	(73-78)	FALSE	[0]←0	[0]← -1			1	0	0
2	χ = 1, τ = 0	matching P[1] = $\bar{B}$ … R[3] matching P[2] = C … R[1]	(48-53)		[2]←10	[2]←0	[0]←30	[2]←TRUE	3	10	2
3	χ = 3, τ = 10	matching P[3] = $\bar{D}$ … R[7] matching P[4] = $\bar{E}$ … R[2] matching P[5] = F … R[9]	(42-46)	TRUE				[5]←TRUE	2	19	0
4	χ = 2, τ = 19	matching P[1] = $\bar{B}$ … R[3] matching P[2] = C … R[4]	(68-70)	TRUE					0	29	-1
5	χ = 0, τ = 29	matching P[0] = A … R[5]	(73-78)	FALSE	[0]←50	[0]← -1			1	50	0
6	χ = 1, τ = 50	matching P[1] = $\bar{B}$ … NIL	(30-34)				[1]←∞		2		
7	χ = 2, τ = 50	matching P[2] = C … R[6]	(73-78)	FALSE	[2]←60	[2]←0			3	60	2
8	χ = 3, τ = 60	matching P[3] = $\bar{D}$ … R[7] matching P[4] = $\bar{E}$ … NIL matching P[5] = F … R[9]	(42-46)	TRUE				[5]←TRUE	2	69	0
9	χ = 2, τ = 69	matching P[2] = C … R[8]	(73-78)	FALSE	[2]←80	[2]←0			3	80	2
10	χ = 3, τ = 80	matching P[3] = $\bar{D}$ … NIL matching P[4] = $\bar{E}$ … NIL	(30-34)				[2]←∞		5		
11	χ = 5, τ = 80	matching P[5] = F … R[9]	(73-78)	FALSE	[5]←90	[5]←5			6	90	5

Table 1, the *C* in pattern $P = A\bar{B}C\bar{D}\bar{E}F$ is originally matched to *R[1]* = *C* in stage 2. A backtrack occurs in stage 3, forcing a C to be matched with *R[4]* in stage 4.

However, making that choice violates $A\bar{B}C$ because *R[3]* = *B*, causing another backtrack. This is the reason why this additional check is required in backtrack mode. When a violation like this occurs, TPS sets the local variable `backtrackMore` to be `TRUE` in `HANDLE_PRESENCE(R, P)`, line 71. This makes TPS to backtrack additionally to *P[0]* = *A* in stage 5 in Table 1, which eventually leads to matching *P[0]* to *R[5]* = *A*. Code listing for `HANDLE_PRESENCE(R, P)` shows how TPS deals with presence items and, additionally, backtracks when β is true.

When handling a presence item, TPS finds the first occurrence of the given event type after the current time τ (line 57). If there is no such event, TPS immediately sets the χ to -1 (lines 59), which subsequently causes `IS_A_MATCH(R, P)` to return `FALSE`, terminating the search. If there is a match and TPS is in backtrack mode, the algorithm checks to see if the match violates a previous constraint (line 63). This check is performed by comparing the matching event's time with the time of the closest known absence event in the future (`badTime`) that follows the match of the previous presence item (line 67). If the violation exists (matching event time is greater than `badTime`), TPS sets `backtrackingMore` to `TRUE`, and updates several variables to backtrack more (lines 68-71). If there is no violation (or that we are not in backtrack mode to begin with), TPS updates the variables to advance on the pattern (lines 73-78): current time τ is updated to the new matching event's time, and this new time is also stored in T for future reference. The current index χ is stored in last index δ, and Δ is accordingly updated. χ is incremented by 1 to advance on the pattern. Finally, β is set to FALSE to exit backtracking mode.

4.2 An Example

Table 1 shows a trace of TPS searching for the pattern P = A $\bar{B}$ C $\bar{D}\bar{E}$ F in the record R = ACEBCACDCF. Again, note that we are using an array notation to describe R for ease of narration even though R is indeed a set of sorted arrays of historical events. Each event in R is associated with a time stamp.

The first column shows the iteration number of the main loop in Code Listing 1. The second column shows the value of the two most important variables χ and τ at

the beginning of each loop. The third column shows the events in R that are being matched with item(s) in P in that loop. The fourth column indicates which lines of code are reached to assign the values to the state variables on the right portions of the table.

The first row of Table 1 shows 0th loop, where all variables are initialized. We describe the entire trace, but focus on loops 1-6 in detail. Loops 1 and 2 demonstrate how presence and absence items are handled if there are no backtracks. Loops 3 and 4 shows an example how negation items can trigger multiple backtracks. Loop 6 shows how TPS handles negation items when there are no such events in the record.

In the 1st loop, TPS attempts to find a match for the first item in the pattern P[0] = A. The event *A* at time 0 is matched. The current index on the pattern χ is incremented to indicate that we will try to match the second item in the pattern. The current time τ, from which the next search will begin, is also updated to 0, the time stamp of the matching event. δ is updated to remember the index (0) of a previously-matched presence item on P. β is kept as `FALSE` as we do not need to backtrack.

The 2nd loop shows how TPS deals with an absence event in P[1] = $\bar{B}$. TPS uses binary searches to find all first events of the absence block's event types (in this case there is only 1 ($\bar{B}$) and the first presence event (C) after this contiguous absence event types. Because the first `B` TPS finds in R[3] occurs after the first E in R[2], the absence event criteria are satisfied, and no backtrack is necessary (β is unchanged). TPS remembers the time stamp of the first *B* is found in Φ[0], where 0 is the index of the previous presence event. Π[2] is set to `TRUE` to indicate that the item P[2] follows at least one absence item.

In the 3rd loop, TPS again deals with an absence block and its following presence event: $\bar{D}\bar{E}$ F. This time, however, because both the matched F and E occur between the previously matched P[2] = F and the upcoming P[5] = F, β is set to `TRUE` to indicate that TPS will backtrack. χ is rolled back to 2 so TPS will find a new match for P[2]. τ is set to be right before the time stamp of the first known absence event of the absence block. In this case, it is R[2] = E, which has time stamp of 20. TPS sets τ to be 20 - 1 = 19, where 1 is the smallest time granularity in consideration (usually milliseconds). This way if a C in the record has the time stamp of 20, the binary search functions would not miss it.

In backtracking mode in the 4th loop, TPS attempts to match P[2] = C to R[4]. Since P[2] follows an absence block, TPS checks to see if matching P[2] to R[4] violates a constraint. In this case, it does violate A $\bar{B}$ C, for there exists a B in R[3]. This means TPS needs to backtrack again, to P[0]. TPS decrements χ, but increments τ to right before the time stamp of R[3] (29).

TPS finds a match for P[0] = A in R[5] in loop 5 and updates other variables normally as in loop 1, and sets β = `FALSE`. When processing P[1] = $\bar{B}$ in loop 6, TPS finds no B's in R after the current time τ = 50. This means the constraint of A $\bar{B}$ C will not be violated for any C that occurs after. This fact is stored in Φ[1]. χ is of course incremented by 1. Because the lack of B in the record, TPS only processes the absence block that contains P[1] in this loop. The following presence event is left for the next loop.

In the 7th loop, TPS finds R[6] = C to match P[2]. However, in the 8th loop, another backtrack occurs. TPS finds a R[7] = D between the previously found C and the next available F in R[9]. This forces TPS to re-search for another match for C. Variables are updated exactly in the same manner as in loop 3.

Fortunately, an alternative is found in R[8] = C in loop 9, and this match does not violate an existing constraint. In loop 10, TPS fails to find another E or D in the remainder of the record. Finally, TPS matches the last item in the pattern P[5] = R[9], and returns `TRUE`, terminating the execution.

5 Analysis

If the search pattern contains no negation: $P = p_1 p_2 p_3 \ldots p_m = T_{p_1} T_{p_2} T_{p_3} \ldots T_{p_m}$, its running time is $O(\lg(|T_{p_1}|) + \lg(|T_{p_2}|) + \ldots + \lg(|T_{p_m}|)$, where $|T_{p_1}|$ denotes the number of events of type T_{p_1} in a personal record. This is because the algorithm uses binary search to find an event when a presence item is encountered. If there are no events to be found, the algorithm immediately terminates and returns `FALSE`. On the other hand, if such a pattern exists, then the running time is bounded by $O(m\lg(n))$, where n is the total number of events (regardless of type) in the record.

When the search pattern contains absence items, TPS may need to perform backtrack when absence events are encountered, and the worst case analysis is based on (1) how many backtracks can there be, and (2) how much work is done when a backtrack occurs. In general, the input data triggers a backtrack when an *absence block* in the pattern is not satisfied, e.g. a T_{p_2} occurs between a T_{p_1} and a T_{p_4}. The maximum number of absence blocks in the search pattern is then $\frac{m}{2}$, where m is the length of the pattern. The algorithm works so that when an event that violates an absence block specified in the pattern, the event's time is recorded in the array Φ, and the algorithm will start at that event's time when backtracking. For this reason, we only have to consider non-overlapping presence-absence events in the data as potential backtrack points from the data. Therefore, for a record containing n events, there can be at most $\frac{n}{2}$ non-overlapping presence-absence event patterns, where 2 is the length of the smallest possible presence-absence pattern. For each such pattern, it can cause up to $\frac{m}{2}$ backtracks. Putting everything

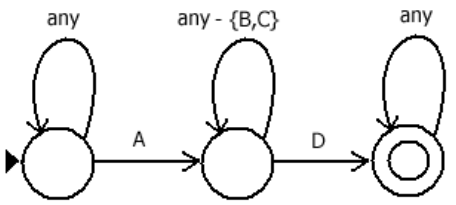

Fig. 3. An NFA corresponding to .*A[^BC]*D.*, which is the regular expression equivalent to the search pattern $A\bar{B}\bar{C}D$. The black triangle indicates the starting state, and the double circle indicates the accepting state. *any* represents the set of all characters.

together, the total number of backtracking opportunities is $\frac{m}{2}\cdot\frac{n}{2}=\frac{mn}{4}$.

The amount of extra work resulting from a backtrack is related to the size of the absence blocks. The pseudo code indicates that each backtrack results in a constant number of variable assignments, but the real work lies in the occasions when TPS re-performs its search on the failed part of the pattern. A backtrack is always triggered by an absence block, and a new search is performed to find a matching presence event immediately before the absence block, all absence events in the absence block, and the trailing presence event (if there is one specified in the pattern). This means $O((2+|ab|)\lg(n))$ work is performed, where *ab* is the size of the largest absence block. This expression is maximized when $|ab| = m$, resulting in $O(m\lg(n))$, where m is the length of the search pattern.

The worst case running time would then be the number of backtracking opportunities times the work done in a backtrack situation. However, a search pattern cannot both have the largest absence block and the most number of absence blocks at the same time. Let m be the size of the search pattern, and x be the largest absence block size. We want to find the x that maximizes the running time function $f(x)=(\frac{n}{2}\cdot\frac{m-x}{2})(x\ln(n))$, where m and n here are considered constants. Its second derivative $f''(x)=-1$ shows that *f(x)* is concave down. By the second derivative test, and the fact that its first derivative equals to zero when $x=\frac{m}{2}$, $x=\frac{m}{2}$ must be a global maximum. This means that the worst case patterns are the ones that have a largest absence block equaling half the length of the search pattern, and the other half of the pattern consists of only alternating (positive-negative) patterns to increase backtrack opportunities. Substituting $x=\frac{m}{2}$ in *f(x)*, we obtain the worst case bound $O(\frac{m^2n}{8}\lg(n))=O(m^2n\lg(n))$. In addition, TPS uses $O(m)$ space for the state-keeping arrays.

TABLE 2
EXPONENTIAL BEHAVIOR FOR `java.util.regex`

Search Pattern Length	Time (sec)
2	0.024
3	0.272
4	31.193
5	3099.675

Using `java.util.regex` to perform searches exhibits exponential behavior. The time reported is time taken to match a pattern over a single string. The input string size is 500, while the search pattern length varies. The strings are chosen so that there is no match for the patterns, thus requiring `java.util.regex` to exhaust all possibilities before returning false. This behavior is due to the fact that the patterns contain many `.*`, which can require exponential number of searches.

6 EVALUATION

There are several regular expression implementations we can compare TPS to. We considered comparing TPS with the default Java regular expression engine in `java.util.regex`, and a faster (though limited) implementation that uses deterministic finite automata: (`dk.brics.automaton`) [14]. However, initial tests revealed that the Java regular expression engine runs in time exponential to the length of the search pattern in the worst case. Wfe used a fixed input string of 500 characters, and search patterns of varying length (n = 2,3,4,5). The string is chosen so that the patterns will not match. The regular expression patterns are obtained by converting *positive-only* patterns described in our discussions to its equivalent regular expression form. For example, *ABCBD* is converted to `.*A.*B.*C.*B.*D.*`. Table 2 illustrates this exponential behavior of matching *positive-only* patterns (no negation) of varying length against a *single* string of size 500. This behavior is due to the fact that each `.*` presents a choice point for `java.util.regex`, and combinatorially, there are $O(2^m)$ choice points in a search pattern consisting of all positive items of length m [3]. For the particular application and type of patterns we consider, `.*` is unavoidable, and occurs very frequently (grows linearly with respect to the number of items in the search pattern in the worst case). Therefore the very poor performance of `java.util.regex` means that it is not a good choice for this comparison.

The DFA approach, on the other hand, has a similar problem. So long the search pattern sizes are kept reasonable, the performance is very good. However, when the search pattern grows longer than 30 items, the exponential space the DFA requires makes the DFA approach not feasible (it fails to construct the DFA when all memory has been allocated to the Java Virtual Machine).

Instead, we compared our Temporal Pattern Search algorithm to a simple NFA that handles the wild card character (`.`), Kleene Star (`*`) over single symbols, and nega-

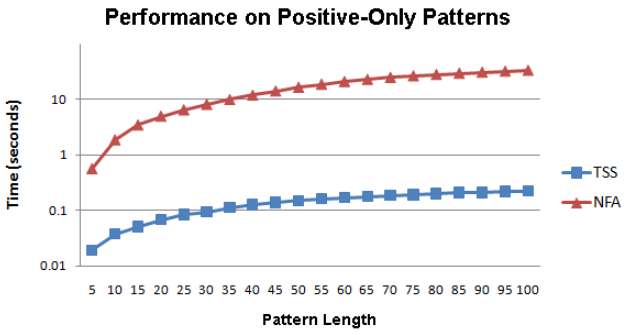

Fig. 4. Comparison of the TPS and NFA performance in search pattern length = 5, 10, …, 100 for *positive-only* patterns. The vertical axis is logarithmic, and it is time in seconds. TPS is in blue squares. NFA is in red triangles. TPS consistently performs one order of magnitude faster than NFA in this experiment.

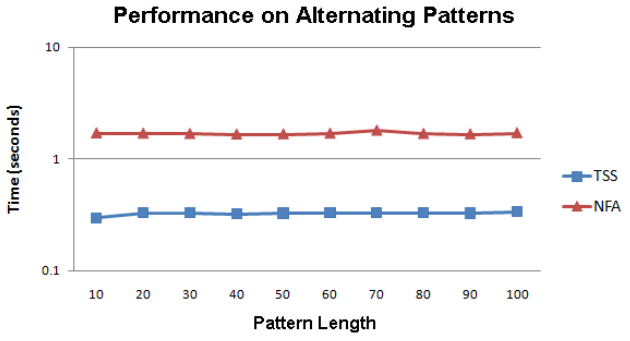

Fig. 5. Comparison of the TPS and NFA performance in search pattern length = 10, 20, …, 100 for *alternating patterns*. The vertical axis is logarithmic, and it is time in seconds. TPS in blue squares, and NFA is in red triangles. TPS consistently outperforms NFA for these search patterns.

tion over a set of symbols (`[^xyz]`, where `xyz` are symbols) (Figure 3).

The NFA keeps a set of states that it is currently in. Initially only the start state – indicated by the black triangle in the figure – is in the set of current states. As the NFA reads one symbol at a time from the input string, it checks with all the current states it is in and sees if any of state's transition rules fire to bring in a set of new current states. When all input are consumed, the NFA checks to see if the accepting state (indicated by the double circle in the figure), is in the set of current states. If it is, then the NFA returns `TRUE` to indicate that a match has been found; `FALSE` otherwise. For a search pattern of length m, there are at most m states in its corresponding NFA (each positive term translates to a state, and each negative block, regardless with length, also translates to one state). For an input string of size n, the running time for the NFA is bounded by $O(mn)$ because the NFA may be in all m states for each of the n symbols.

Due to the limited expressiveness of the search patterns under consideration here, the NFA does not need to wait until all inputs have been consumed to check for the accepting state. In our implementation, the NFA checks for the accepting state after each symbol has been consumed, except when the last item on the pattern is a negations (in which case, the NFA does need to consume all inputs to ensure that there are no violations). This small optimization increases the NFA performance by a significant amount in practice.

6.1 Method

The NFA most naturally works with a temporally ordered string of events. TPS works over a set of type-separated, temporally ordered arrays. The aim of this evaluation is to see if we had the most appropriate data support for each approach, whether TPS has any advantage over NFA. That is, we attempt the answer the question "if we had implemented our visualization tool in the most appropriate way to use an NFA search, would it perform faster?" We structure our evaluation as follows. (1) We guarantee that the input event history data does not have events that have the same time stamp. This means the NFA does not need to be modified to handle this case, and that the input data will be semantically equivalent in both approaches. (2) The input data is transformed into a single string for the NFA to process. (3) In all our reported results, we do not include the time it takes to build the NFA because it is linearly bounded by the size of the search pattern – although the time it takes to build an NFA would be reflected in the interface when users construct a pattern query.

Let k designate the number of event types in a set of records. We randomly generated a set of 5000 records for each k = 2,3,…,9 and k = 10, 20, 30, 40, 50. Each record in the set has 500 events. Each event in a record occurs with uniform probability. We also randomly generated a set of search patterns for each dataset, where each event type occurs in the pattern also with uniform probability. In the search pattern input file, there are three types of patterns: (1) patterns that contain only positive items, (2) patterns that contain alternating positive and negative item (half with leading positive terms, half with leading negative terms) and (3) patterns that represents the theoretical worst-case scenario, given length of the pattern m and size of the event types k. These worst-case scenario patterns are like the alternating patterns, but with a large absence block of size $\min(k,\frac{m}{2})$ inserted at a randomly chosen position according to a uniform distribution.

For each input set of records, the script parses the records, constructs the appropriate data structures for TPS and NFA, and constructs the NFA. Then it starts timing each approach's execution time. The performance numbers are obtained by running our script in Java 1.6-64 bit environment on Windows Vista 64-bit with a dual core CPU (2.5GHz) and 4GB of RAM.

6.2 Results

We look at the results from two perspectives, first, we inspect how the length of a search pattern impacts the performance of TPS and NFA. We do this by summing up the time taken to execute all patterns of one type

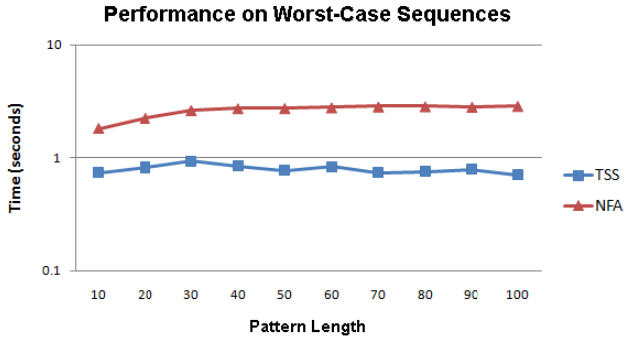

Fig. 6. Comparison of TPS and NFA performance in search pattern length = 10, 20, …, 100 for *worst-case* patterns. The vertical axis is logarithmic, and it indicates time in seconds. TPS is in blue squares, and NFA is in red triangles. TPS consistently outperforms NFA, but the differences is considerably smaller than in the comparison for *positive-only* patterns.

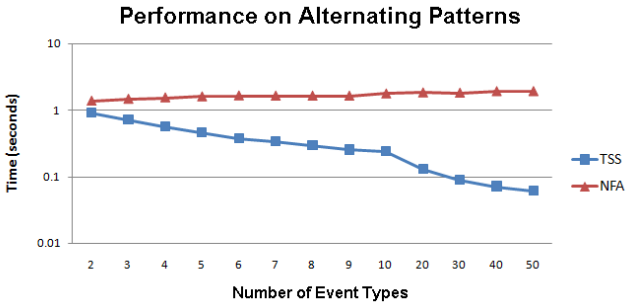

Fig. 7. Comparison of the TPS and NFA performance in number of event types k = 2, 3, …, 9, and 10, 20, … 50 over all lengths of search patterns for *alternating patterns*. The vertical axis is logarithmic, and it is time in seconds. TPS is in blue squares, and NFA is in red triangles. TPS performs similarly to NFA initially, but dramatically better as the number of event types grows.

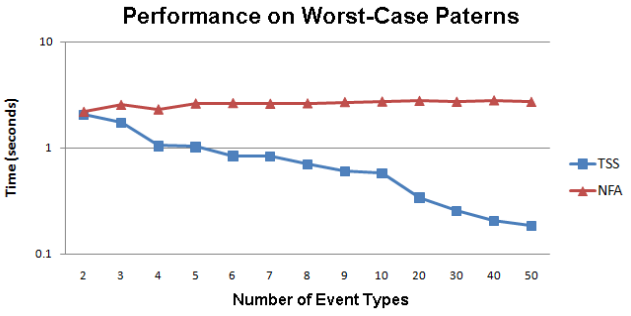

Fig. 8. Comparison of the TPS and NFA performance by varying number of event types k = 2,3, …, 9 and 10, 20, …, 50 over all lengths of search patterns for *worst-case* patterns. The vertical axis is logarithmic, and it is time in seconds. TPS is in blue squares, and NFA is in red triangles. TPS performs similarly to NFA initially, but dramatically better as the number of event types grows.

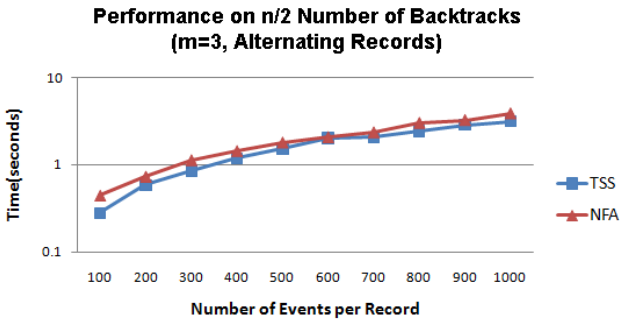

Fig. 9. Comparison of the TPS and NFA performance by varying the number of events in a record n = 100, 200, …, 1000. The search pattern is fixed to be $A\bar{B}C$, and the records are fixed to ABABAB…ABC. This guarantees n/2 backtracks for record. There are 5000 records for each n. The vertical axis is logarithmic, and it is time in seconds. TPS is in blue squares, and NFA is in red triangles. Although TPS still performs slightly better than NFA for most n shown here, the difference is negligible.

across all alphabet sizes and compute the average time it takes to perform a search on all records.

Figure 4 shows the performance graph of TPS and NFA when all search patterns include only *positive* items. The x-axis is the length of the search pattern, and the y-axis is time in seconds and in logarithmic scale. We notice a few things. First, TPS performs consistently one order of magnitude faster than NFA. Secondly, running time increases as the length of the pattern increases. The rate at which the two curves are increasing seem to slow down as the length increases to 100, though TPS's curve is growing at a slower rate. We had expected the TPS to outperform NFA on the *positive-only pattern* portion of the evaluation as TPS's search over only positive events is bounded by $O(m\lg(n))$, that is, this is the "best-case" scenario.

Next, we look at the results for the *alternating patterns*. Unlike the *positive-only patterns*, these patterns provide ample opportunities for backtracking, and we expect to see the TPS performance to be worse than the previous graph. Figure 5 shows the graph of TPS and NFA performance over a variety of search pattern lengths. We see that TPS still consistently outperforms NFA, but the gap is less than an order of magnitude. We also see that both approaches have a fairly flat performance curve even as the length of the search pattern grows. The reason why NFA's curve is much calmer than the *positive-only patterns* case is that the number of states has been reduced by half, and that the number of states currently active is reduced when an event matches a state that deals with negation. On the other hand, we are pleased and surprised that given many backtrack opportunities, TPS held its own this well for search patterns of these lengths.

In the *worst-case patterns* scenario, we expect TPS to lose most of its edge while NFA performs similarly to the alternating patterns scenario. NFA does indeed perform similarly to the *alternating patterns* scenario, except with an initial growth in the length = 10, 20, 30 parts, which TPS also experienced (Figure 6). In this scenario, TPS's performance gets closer the NFA's, taking nearly one second to perform a search over all the records. However, it still beats NFA in all of our test cases, and we do not observe a growth adhering to the m^2 term in the asymptotic bound.

To understand how TPS is outperforming NFA in our tests, we plot the average time it takes to perform a search over all records when the number of event types is varied. While the event type size does not affect the asymptotic bound, because TPS examines an event on a need-to basis, if the alphabet size is large compared to the number of event types present in the search pattern, TPS can ignore a lot of events. On the other hand, NFA is required to, in general, examine all the events. In Figure 7, we see that while NFA's performance remains roughly constant over event type size k = 2, 3,…, 5 and 10, 20,…, 50 in the *alternating patterns* scenario, TPS's performance improves as k increases. Figure 8 shows a similarly dramatic trend with *worst-case patterns* scenario.

On the flip side of the coin, if TPS is required to look at most events, TPS's performance is not expected to perform better than NFA. To show this, we fix the search pattern to be $A\bar{B}C$ (m = 3), and the events in records to be ABABAB…ABC. This combination results in a search that requires TPS to look at every event in the record, creating very large number of backtracks, and eventually returns `FALSE`. We vary n = 100, 200,…, 1000 to show the effects of n. Figure 9 shows the performance plot of TPS and NFA searching on 5000 records for each n. n affects the running time on TPS and NFA linearly for n = 100,…, 1000. TPS has a slight edge on NFA, but the difference is negligible. We also vary m while keeping the search pattern to be in the form $A\bar{B}A\bar{B}A\bar{B}...A\bar{B}C$, with m = 5, 7, 9, 11. The results are similar to the case of m = 3, where neither TPS nor NFA holds any advantage over another (graphs not shown here).

6.3 Discussion

While we have shown that the worst-case asymptotic bound for the TPS algorithm is worse than that of a NFA, the empirical evaluations show that TPS consistently outperforms the NFA approach under our experimental conditions with randomly generated inputs. The experimental conditions are chosen to reflect the situations under which we expect users to be using Lifelines2. In working with our collaborators in the medical field, the average number of events in a single patient record usually are in the low hundreds, with highest nearing a thousand, and lowest only a handful. The number of types of events is typically between 10-35. In practice, we do not encounter the cases where k is so small and TPS performs only marginally better than NFA. On the other hand, we also do not encounter cases when k is as large as 50 to have the TPS advantage greater than one order of magnitude (Figure 7).

For n ≤ 1000, the performance growth is roughly linear even in cases that the maximum number of searches and backtracks are required (Figure 9). We also do not observe a quadratic growth for $m \leq 100$, although the asymptotic bound for TPS contains a m^2 term. While we do not expect users to specify a pattern query of these extreme lengths, it is comforting to know that if users wished to do so, TPS would be able to handle it.

The implications of these results are that while we cannot recommend TPS for all situations, we argue it is the appropriate choice for our application Lifelines2. In addition, other analysis tools focused on temporal categorical data such as [4], [18], [13] can benefit from TPS to provide sequential search capabilities. In addition, TPS can be used as an external function in database applications where searching for temporal patterns is required.

7 Conclusions and Future Work

We present the novel algorithm Temporal Pattern Search, and show that its running time is bounded by $O(m^2 n \lg(n))$. Although the bound is less attractive to that of the NFA approach, TPS performs favorably in our experiments with random inputs against NFA. TPS utilizes binary searches over a set of time-sorted event arrays, and is able to skip many irrelevant events. We show that TPS saves significant amount of time in comparison to NFA when there are many event types. Since these experimental conditions we described here either reflect or subsume the conditions under which we expect users to be using our visualization tool, we expect the performances shown here to hold. Furthermore, we argue that using TPS in our application is a design success, and other applications that share similar constraints to ours may be able to leverage on TPS.

The future directions of TPS include how we can expand the expressiveness of TPS to further support visual exploratory tasks. One immediate extension to include specification of finer temporal constraints. Another would be to study how TPS can be modified to perform a search for multiple patterns at once or search for all instances of match events in a record.

References

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. immerman, "Efficient pattern matching over event streams," *Proceedings of ACM SIGMOD* 08, pages 147-160, 2008.

[2] R. S. Boyer and J. S. Moore, "A fast string-searching algorithm".volume 20, pages 762–772, 1977.

[3] R. Cox. "Regular expression matching can be simple and fast," http://swtch.com/rsc/regexp/regexp1.html, 2007.

[4] DataMontage. http://www.stottlerhenke.com/datamontage/.

[5] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White, "Towards expressive publish/subscribe systems," *Proceedings of the 10th International Conference on Extending Database Technology*, pages 627-644, 2006.

[6] J. Fails, A. Karlson, L. Shahamat, and B. Shneiderman, "A visual interface for multivariate temporal data: Finding patterns of events over time," *Proceedings of the IEEE Symposium of Visual Analytics Science and Technology* (VAST 06), pages 167-174, 2006.

[7] D. Ficara, S. Giodano, G. Procissi, F. Vitucci, G. Antichi, and A. D. Pietro, "An improved DFA for fast regular expression matching," *ACM SIGCOMM Computer Communication Review*, 38(5):29–40, 2008.

[8] L. Harada and Y. Hotta, "Order checking in a CPOE using event analyzer," *Proceedings of ACM CIKM*, pages 549-555, 2005.

[9] L. Harada, Y. Hotta, and T. Ohmori, "Detection of sequential patterns of events for supporting business intelligence solutions," *Proceedings of IDEAs 04*, pages 475-479, 2004.

[10] R. M. Karp and M. O. Rabin, "Efficient randomized patter matching algorithms," Technical Report TR-31-81, Aiken Computation Laboratory, Harvard University, 1981.

[11] D. E. Knuth, J. H. Moris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, 6(2):323-350, 1977.

[12] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," pages 155–164, New York, NY, USA, 2007. ACM.

[13] H. Lam, D. Russell, D. Tang, and T. Munzner, "Session viewer: Visual exploratory analysis of web session logs," *Proceedings of the IEEE Symposium of Visual Analytics Science and Technology* (VAST 07), pages 147-154, 2007.

[14] A. Møller, "dk.brics.automaton, an automaton/regexp library for Java", http://www.brics.dk/automaton/, 2001.

[15] G. Navarro and M. Raffinot, "Fast and flexible string matching by combining bit-parallelism and siffix automata," *ACM Journal of Experimental Algorithmics* (JEA), 5(4), 2000.

[16] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings*. Cambridge, UK.: Cambridge University Press, pages 77-97, 2002.

[17] P. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi, "Expressing and optimizing pattern queries in database systems," *ACM Trans. on Database Systems*, 29(2):282–318, June 2004.

[18] M. Suntinger, H. Obweger, J. Schiefer, and M. E. Gröller, "The event tunnel: Interactive visualization of complex event streams for business process pattern analysis," pages 111-118, 2008.

[19] K. Thompson. Regular expression search algorithm. CACM,11(6):419-422, 1968.

[20] T. D. Wang, C. Plaisant, A. J. Quinn, R. Stanchak, S. Murphy, and B. Shneiderman, "Aligning temporal data by sentinel events: discovering patterns in electronic health records," pages 457-466, New York, NY, USA, 2008. ACM.

[21] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. H. Katz., "Fast and memory-efficient regular expression matching for deep packet inspection," *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, pages 93-102, 2006.

Taowei David Wang obtained his BS in computer science from Duke University in 2002 and his MS from the University of Maryland in 2005. He is currently working towards his PhD degree. His research interests include temporal categorical data visualization, human-computer interaction.

Amol Deshpande is an Assistant Professor of Computer Science at the University of Maryland at College Park. He received his PhD from UC Berkeley in 2004, and his bachelors degree from IIT Bombay in 1998. His research interests include adaptive query processing, data streams, sensor networks, and statistical modeling of data. He is a recipient of the National Science Foundation (NSF) CAREER Award.

Ben Shneiderman is a Professor in the Department of Computer Science and Founding Director (1983-2000) of the Human-Computer Interaction Laboratory at the University of Maryland. He was elected as a Fellow of the Association for Computing Machinery (ACM) in 1997 and a Fellow of the American Association for the Advancement of Science (AAAS) in 2001. He received the ACM SIGCHI Lifetime Achievement Award in 2001. His research interests are human-computer interaction, information visualization, and user interface design.