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 binary search 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. Although the asymptotic running time of TPS is inferior to that of a non-deterministic finite automaton (NFA) approach $(O(mn))$, TPS performs better than NFA under our experimental conditions. We also show TPS is very competitive with Shift-And, a bit-parallel approach, with real data. Since the experimental conditions we describe here subsume the conditions under which analysts would typically use TPS (i.e. within an interactive visualization program), we argue that TPS is an appropriate design choice for us.

Index Terms— Pattern matching, temporal event data, information visualization, graphical user interfaces

1 Introduction

Temporally ordered events can often reveal interesting information. Understanding whether a particular pattern occurs, how frequently certain patterns 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 academia and 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) [27], [4], business intelligence [25], web server log analysis [14], financial applications [29] and so on.

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 temporal event patterns. 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 even 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, and are unwilling to learn it, they have to rely on one of two approaches. First, they could partner with 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. Secondly, interfaces exist to circumvent the learning curve of query languages. In these situations, analysts can issue higher level commands and the interface translates them into SQL queries. However, more often than not, the interfaces reduce the expressiveness of the underlying language in favor of usability and learnability. In particular, searching for temporal patterns is often unsupported by the interface, in favor of the simpler Boolean or conjunctive queries. The inflexibilities often frustrate our physician/clinician collaborators using state-of-the art electronic health record database interfaces such as Amalga [16] or i2b2 [18].

Finally, searching temporal patterns on personal histories that have hundreds or thousands of events with tens of thousands of histories in a database can take a long time. Our experience in building a query interface extension for Amalga revealed some performance problems using SQL [15] [22]. A temporal pattern query in SQL is not feasible for the hospital's database of thousands of patients because of prohibitively high number of self-join operations. Only after building additional indices and preprocessing (which itself can take hours) could a temporal pattern query be managed [15]. Even so, the running time increases exponentially with the number of elements in the pattern. It is unrealistic to perform tailored optimization techniques for each potentially different temporal pattern search. Instead, we have found that it is more effective to break down a temporal pattern search into two stages. We would first issue a conjunctive SQL query to

- *The authors are with the University of Maryland, 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.

Digital Object Indentifier 10.1109/TKDE.2010.257

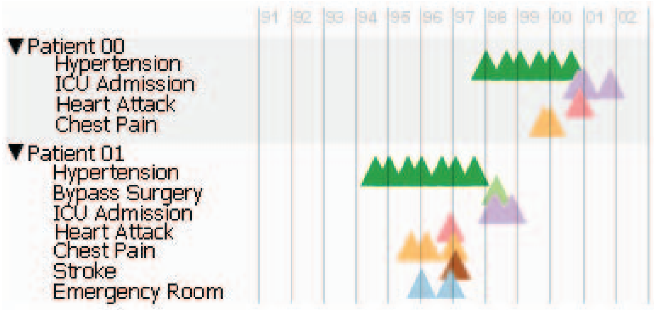

Fig. 1. A partial screen shot of Lifelines2 display. Two electronic health records for fictional heart disease patients are shown here. The color-coded triangles represent each individual event (diagnosis, intervention, chief complaints, etc.), and events of the same type are drawn in the same horizontal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

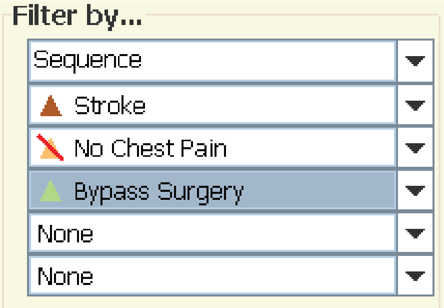

Fig. 2. This is the user interface widget in Lifelines2 for specifying temporal patterns. Each item in the pattern is specified via a combo box. The top-most combo box specifies the first item in the pattern, and the bottom-most specifies the last. This screen shot describes the pattern query: “find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

find patients who have events A, B, and C, which can be executed efficiently. Temporal pattern search (e.g., A followed by B followed by C) can then take place in an analysis tool, where data can be further structured to better support these searches. 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 on 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 record) and not tuples as in the database literature.

Lifelines2 provides several ways for analysts to filter records by their features. One way is via an event sequence filter, where analysts specify a pattern that describes a sequence of temporally ordered events as search criteria. Previous research suggested that complex interfaces that support full control of all temporal constraints among items in a pattern often overwhelm analysts [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 analysts 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`) in this interface. However, this kind of temporal constraints can be specified via temporal summaries in our interface [28]. Once a pattern is specified, Lifelines2 finds all records that contain this pattern and visually filters out the rest.

We made a few decisions on how we internally represent the event histories in Lifelines2. As these decisions were made to support existing features of the tool and before we designed our Temporal Pattern Search (TPS) algorithm, the constraints represent interesting design challenges. 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, mislead analysts, and produce wrong results. Using one array for each event type allows us to circumvent this problem. The frequency of an event sharing the same time stamp with another depends on datasets. Table 1 shows a list of sample clinical data we have been supplied with by our collaborators. The proportion of events that have the same time stamp can range from less than 0.5% to almost 50%. However, we assume events that have the same type and the same time stamp are the same event in order to merge events that are in fact the same but come from two siloed datasources. While this assumption is practical and reasonable for personal records, it may not apply to all temporal event data.
2. **Drawing Constraint**: Lifelines2 maintains a drawing order of events by event types. Events of the same type are drawn together (on the same horizontal space). Lifelines2 maintains the z-order by event types to avoid visual inconsistencies that can potentially disrupt analytical tasks. The separated arrays would allow the drawing algorithm an efficient way to access events of the same type.
3. **Interface Constraint**: While searching for temporal patterns is important, it is not all that Lifelines2 does. Other operators designed for exploratory analysis (such as *alignment*, *rank*, and *summarization*) benefit from this separate arrays approach. 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 most efficiently.

TABLE 1
SAMPLE DATASETS AND THEIR STATISTICS

Dataset Name	#Records	#Events	#Events w/same time stamp
Creatinine*	3598	32134	16
Heparin*	841	65728	31251
Heart attack	9361	196581	93610
Transfer	51006	207187	51318
Bipap	6583	135951	14650

* Dataset was reported in a previous publication [28].

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 search algorithm in the presence of these constraints, particularly the first constraint. We first describe the problem more formally, provide related work, and present the TPS algorithm. We then analyze the worst-case running time of the algorithm, discuss its extensibility to common situations, and evaluate its performance against existing common approaches.

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 $t_{e_1} < t_{e_2} < t_{e_3}$. 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 $T_{e_1} = T_{p_1}$ 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 $t_{e_1} < t_{e_2}$ and $t_{e_2} < t_{e_4}$) or ($T_{e_3} = T_{p_3}$ and $t_{e_1} < t_{e_3}$ and $t_{e_3} < t_{e_4}$). 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 the equivalent regular expression $.*T_{p_1}[^\wedge T_{p_2} T_{p_3}]*T_{p_4}.*$, where "." is the wildcard character, "*" the Kleene star, "^" the negation, and "[...]" indicates a class of symbols. A direct application of the regular expression would be fine if our data were amenable to be structured as a single string. The problem we have is how to search for records that match the specified pattern given our constraints.

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. There are three main differences in classical string matching problems: (1) a pattern is wholly specified ("abc" as opposed to "a" followed by "b" followed by "c"), (2) no two characters can occur at the same position in a string, and, (3), 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) [12], Boyer-Moore (BM) [2], and Rabin-Karp (RK) [11]. 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 it can 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 [24]. 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.

The traditional method to match regular expressions in strings utilizes either deterministic or non-deterministic finite automata (DFA or NFA) [26]. Given a text of length *n* and a pattern of length *m*, a DFA can perform a search in *O(n)* time, but building a DFA can take up to *O(2^m^)* space [23]. This makes the DFA approach useful for applications where building a DFA is infrequent or can be done offline, such as network intrusion detection systems [30], [13], [7]. On the other hand, an NFA can compactly represent an equivalent DFA in space *O(m)* [10]. Although the running time is bounded by *O(mn)*, the overall approach lends itself better for iterative query refinement in an interactive environment.

Given the tradeoff between DFAs and NFAs, many systems choose to use NFAs or extensions of NFAs, e.g., [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.

Newer string match approaches such as the Shift-And, Shift-Or, or the Backward Nondeterministic Dawg Matching algorithm can be more easily adapted to deal with classes of characters, optional and repeatable characters that classical string match algorithms cannot handle [19], [20], [21]. The basic idea behind these algorithms is to use bits to represent the states of an NFA. When a symbol is read, all states can be updated in parallel by cleverly using bitwise operators. When there are *w* bits in a computer word, these algorithms are expected to perform *w* times faster than an equivalent NFA. Today when most consumer machines are either 32 or 64 bits, this can be a significant performance advantage. When more than *w* states are required to represent the pattern, multiple words can be used. In this case, the performance of these algorithms suffers because of the overhead of using an array of integers instead of a single integer to represent the states. For example, the best worst case time for Shift-And becomes *O(mn/w)*. However, unlike real NFA or DFA approaches, these algorithms are difficult to extend to handle, for example, value constraints associated with input symbols, while our algorithm is easily amenable to these extensions. Finally, both the automaton and the bit-parallel approaches assume the inputs to be a single string, and cannot be applied to our use cases.

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.*`, whereas 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 TPS algorithm. `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 events of the same type `T`, and can be accessed by `R[T]`. Each event `e` has a time stamp `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 2 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 2 and refer to its i^{th} element using the array notation `R[i]` in the following discussions.

In addition to the details that are shown, we assume that there exists the following function `NEXT(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. Since all event arrays in records are sorted by time stamps, we assume the `NEXT(R[T],t)` uses efficient binary searches.

Code Listing 1
Starting Function and Initialization of Global Variables

```
IS_A_MATCH(R,P)
  β ← FALSE          //backtrack flag
  T[P.length]        //matched times
  Δ[P.length]        //last pos item
  Φ[P.length+1]      //next neg item time
  Π[P.length]        //if has neg item before
  χ ← 0              //current index
  τ ← -∞             //current time
  δ ← -1             //last pos item

  while (χ < P.length)
    if (χ == -1)
      return FALSE
    if (P[χ].isNeg)
      HANDLE_ABSENCE(R,P)
    else
      HANDLE_PRESENCE(R,P)
  return TRUE;
```

CODE LISTING 2
THE HANDLE_ABSENCE FUNCTION

```
HANDLE_ABSENCE(R,P)
  minTime ← NIL
  numAbs ← 0
  for (i←χ to P.length)
    item ← P[i]
    if (NOT item.isNeg) break
    numAbs ← numAbs + 1
    absEvent ← NEXT(R[item.Type], τ)
    if (absEvent == NIL)
      continue
    minTime← MIN(absEvent.Time, minTime)
  if (minTime == NIL)
    if (δ > -1)
      Φ[δ] ← ∞
    else
      Φ[Φ.length-1] ← ∞
    χ ← χ + numAbs
  else
    if (χ + numAbs < P.length)
      nItem ← P[χ+numAbs]
      nEvent ← NEXT(R[nItem.Type,] τ)
      Π[χ+numAbs] ← TRUE
      if (nEvent == NIL OR
        nEvent.Time > minTime)
        χ ← δ
        if (χ > 0)
          δ ← Δ[χ]
          τ ← minTime-1
        β ← TRUE
      else
        T[χ + numAbs] ← nEvent.Time
        Φ[δ] ← minTime
        Δ[χ+numAbs] ← χ-1
        δ ← χ + numAbs
        χ ← χ + numAbs + 1
        τ ← nEvent.Time
    else
      χ ← δ
      δ ← Δ[χ]
      τ ← minTime-1
      β ← TRUE
```

CODE LISTING 3
THE HANDLE_PRESENCE FUNCTION

```
HANDLE_PRESENCE(R,P)
  event ← NEXT(R[P[χ].Type], τ)
  if (event == NIL)
    χ ← -1
  else
    backtrackingMore ← FALSE
    if (β AND Π[χ])
      if (Δ[χ] < 0)
        badTime ← Φ[Φ.length-1]
      else
        badTime ← Φ[Δ[χ]]
      if (badTime < event.Time)
        χ ← Δ[χ]
        τ ← Φ[χ]-1
        δ ← Δ[χ]
        backtrackingMore ← TRUE
    if (backtrackingMore)  return
    τ ← event.Time
    T[χ] ← event.Time
    Δ[χ] ← δ
    δ ← χ
    χ ← χ+1
    β ← FALSE
```

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 keeps track of which item in the pattern 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`.

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 is that there would be no need for backtracking if there were no absence events. Furthermore, when TPS backtracks, it backtracks only to the closest previous 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 Τ, Δ, Φ, Π, β, and δ. They describe the states TPS is and save previously-done work to reduce unnecessary backtracking. β is a boolean, and indicates whether we are backtracking. δ represents the index of the previously matched presence item (while χ is the current index). 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 i^{th} item in the pattern (i.e., Δ is an array of δ's). Τ 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 next event if the next 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. Π[i] = `FALSE` indicates that there is no absence item prior to the *ith* item in the pattern. When this is the case, and TPS finds an alternative match for the *ith* item in backtracking, the algorithm need not to check whether this alternative match might 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 for events described 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 over 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, find a matching presence event in the record, and compare its time stamp with `minTime`. If the presence event 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 `TRUE` 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 2, 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 `backtrackingMore` to be `TRUE` in `HANDLE_PRESENCE(R, P)`, line 71. In our example, this causes TPS to backtrack additionally to *P[0]* = *A* in stage 5 in Table 2, 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 • 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 matched 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 newly matched 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 2 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 events of the same type. 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 2 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 are examples that show 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 it will try to match the second

TABLE 2
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	$\overline{B}$	C	$\overline{D}$	$\overline{E}$	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] = $\overline{B}$ … R[3] matching P[2] = C … R[1]	(48-53)		[2]←10	[2]←0	[0]←30	[2]←TRUE	3	10	2
3	0χ = 3, τ = 10	matching P[3] = $\overline{D}$ … R[7] matching P[4] = $\overline{E}$ … R[2] matching P[5] = F … R[9]	(42-46)	TRUE				[5]←TRUE	2	19	0
4	χ = 2, τ = 19	matching P[1] = $\overline{B}$ … 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] = $\overline{B}$ … 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] = $\overline{D}$ … R[7] matching P[4] = $\overline{E}$ … 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
1	χ = 3, τ = 80	matching P[3] = $\overline{D}$ … NIL matching P[4] = $\overline{E}$ … 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

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 matched 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] = $\overline{B}$. TPS uses binary searches to find the first event for each event type in the absence block (in this case there is only one ($\overline{B}$)) and the first presence event (C) after the previously matched positive event (A). Because the first B TPS finds in R[3] does not violate the absence event criteria, no backtrack is necessary (β is unchanged). TPS remembers the time stamp of the first *B* 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 trailing presence event: $\overline{D}\overline{E}F$. This time, however, because both the first D and E occur between the previously matched P[2] = C 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] trails an absence block, TPS checks to see if matching P[2] to R[4] violates a constraint. In this case, it does violate $A\overline{B}C$, 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] = $\overline{B}$ in loop 6, TPS finds no B's in R after the current time τ = 50. This means the constraint of $A\overline{B}C$ 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, TPS finds a R[7] = D between the previously found C and the next available F in R[9], and another backtrack occurs. 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 any existing constraints. 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 negations: $P = p_1 p_2 p_3 \ldots p_m = T_{p_1} T_{p_2} T_{p_3} \ldots T_{p_m}$, 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 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} for the pattern $P = T_{p_1} \overline{T}_{p_2} \overline{T}_{p_3} T_{p_4} \overline{T}_{p_5} T_{p_6}$. The maximum number of absence blocks in a search pattern is $\frac{m}{2}$, 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 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 required 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 $f(x) = (\frac{n}{2} \cdot \frac{m-x}{2})(x\lg(n))$, 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 $O(\frac{m^2 n}{8}\lg(n)) = O(m^2 n\lg(n))$. In addition, TPS uses $O(m)$ space for the state-keeping arrays.

6 Algorithm Extensions

The basic TPS algorithm presented can be extended to handle a variety search constraints. Here we present a straight-forward modification to TPS that handles temporal range constraints and value range constraints, and discuss extension involving event type inheritance.

Temporal range constraints specify how an event must temporally relate to others. Each item on the temporal pattern can have multiple temporal constraints. For example, for the pattern $P = p_1 p_2 p_3 \ldots p_m$ we can additionally specify that p_3 must occur between 5 to 10 days after p_1 and within 3 hours after p_2. It is sufficient to consider only temporal constraints of an item related to previously matched items as constraints related to yet-to-be matched items can be converted to this form. It is worth noting that the temporal constraints work for both positive and negative items. When temporal constraints are used on a negative item, they specify the time spans when event must *not* match.

CODE LISTING 4
MODIFIED NEXT FUNCTION

```
NEXT(A, t, M, D, V)
  idnex ← BIN_SEARCH(A, t)
  if (index == -1)
     return NIL
  else
     while(!(DURATION_CHECK(A[index],D)
           AND VALUE_CHECK(A[index],V)))
        index = index + 1
        if (index>A.length-1)
           return NIL
  return A[index]
```

The value range constraints, on the other hand, specify that the matching events must have values within a certain range in order to be considered a match. For example, physicians may look for patients who had a heart attack followed by a heart surgery followed by a systolic blood pressure reading of 140 or greater. More complex value range constraints can involve higher dimensional data, or data values relative to previously matched items.

These constraints change how TPS works because matching on event types is now conditional on their time stamp and values. The basic idea to handle these constraints is to selectively filter the events that can be matched in the steps of the algorithm. Since he basic structure of TPS guarantees the correct backtracking, the simplest way to handle these constraints is to extend NEXT(R[T],t). That is, after a binary search is performed on an event type array A, and an index is determined, the temporal constraints and the value constraints for that item are checked. If first event fails the check, subsequent events on the array can be tested until one passes or no more events can be tested.

Code Listing 4 shows the modified function, where M is the array of currently matched events. D and V are, respectively, arrays representing the temporal range and value range constraints. The modified function requires the binary search function as before, and additionally functions that check whether an event passes the constraints. This direct extension makes the worst case bound $O(m^2n^2\alpha)$, where α is the amount of work needed to run DURATION_CHECK(…) and VALUE_CHECK(…). These functions can have high complexity as there can be at most *i-1* duration constraints for each *i*th item on a pattern of length *m*. Additionally, the values may be multidimensional or of other complex data structures.

When the event type structure is a hierarchy instead of a list and search patterns include supertypes (types that have descendents), the NEXT(R[T],t) function needs to be further extended to use binary search on each of the subtype's array, and return the one closest to current time t. Caching of the results of the binary searches can improve performance. Since each NEXT(R[T],t) can take up to *O(k*lg*(n))* to perform, the over all worst-case bound for TPS becomes $O(km^2n\lg(n))$, where k is the number of event types.

TABLE 3
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 search paths.

7 EVALUATION

There are several regular expression implementations we considered comparing TPS to. We first tested the standard Java regular expression engine in java.util.regex, and a faster (though more limited) implementation that uses deterministic finite automata: (dk.brics.automaton) [17]. However, initial tests revealed that the standard Java regular expression engine runs in time exponential to the length of the search pattern in the worst case. We used a fixed input string of 500 characters, and search patterns of varying lengths (m = 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 3 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 our purposes or this comparison.

The DFA approach [17], on the other hand, has a similar problem. So long as 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 in preprocessing makes the DFA approach infeasible (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 more scalable approaches. We include, first, a

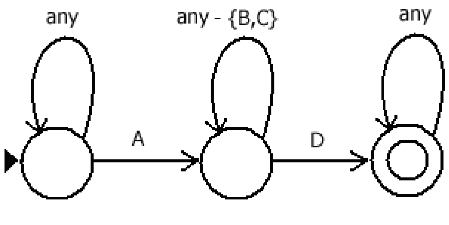

Fig. 3. An NFA corresponding to .*A[^BC]*D.*, which is the regular expression equivalent to the search pattern $A\overline{BC}D$. The black triangle denotes the starting state, and the double circle indicates the accepting state. *any* represents the set of all input symbols.

simple NFA that handles the wild card character (`.`), Kleene Star (`*`) over single symbols, and negation over a set of symbols (`[^xyz]`), where `xyz` are symbols) (Figure 3), and secondly, the bit-parallel Shift-And algorithm described in [19].

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.

The Shift-And algorithm used here differs from [19] in that it models the first state so that the first state is not assumed to be active at all times (needs to be inactive when the pattern starts with a negation, and that negated event is encountered in the search). Since we are using 32-bit Java Runtime Environment, the length of the computer word w = 32. The Shift-And used here is extended to handle patterns that produce more than 32 states. The extension uses an array of integers to represent more than 32 bits (e.g., 2 integers are used for up to 64 states), and handles the necessary bitwise operations (shift, and, or, xor, negation) correctly over the array. The extended version, however, carries an overhead for the array representation. In the evaluation below, when the number of states for the Shift-And algorithm remains under 32, the original algorithm is used. When the number of states is greater than 32, the extended algorithm is used. When the number of states for Shift-And is less than or equal to 32, the search algorithm performs in *O(n)*. If greater than 32, the algorithm performs in *O(mn/32)*. The implication is that on a 64-bit platform, the original Shift-And algorithm would be twice as fast as its 32-bit counterpart. The 64-bit version would also be able to represent up to 64 states, although beyond that, an extended version like our implementation needs to be used. Like the NFA implementation described above, the Shift-And algorithm also terminates immediately when a match has been found.

7.1 Method

The NFA and Shfit-And approaches most naturally work 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 or Shift-And. That is, we attempt the answer the question "if we had implemented our visualization tool in the most appropriate way to use an NFA (or Shift-And) 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 or Shift-And do not need to be modified to handle this case, and that the input data will be semantically equivalent in the evaluation. (2) The input data is transformed into a single string for the NFA and Shift-And approaches to process. (3) In all our reported results, we do not include the time it takes for preprocessing. That is, the time used to build the NFA and the bit masks for Shift-And is not included. The time required to build an NFA is *O(m)*, and the time required to for Shift-And preprocessing is O(*mk/w*), where k is the number of event types, and w is the number of bits in a computer word. The preprocessing time is fairly light although it would still be reflected in the interface when analysts 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 po-

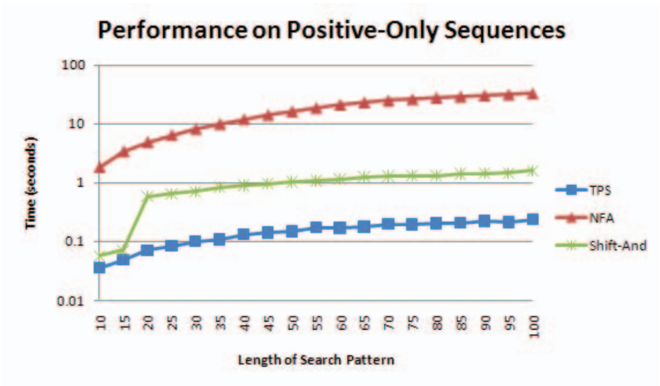

Fig. 4. Performance comparison of TPS, NFA, and Shift-And for search pattern lengths *m* = 10, 15, …, 100 for *positive-only* patterns. The vertical axis is logarithmic in time. TPS outperforms the other two approaches, and betters NFA consistently by one order of magnitude. The sharp increase of time required at *m* = 20 for Shift-And reflects the use of using multiple integers to represent more than 32 states.

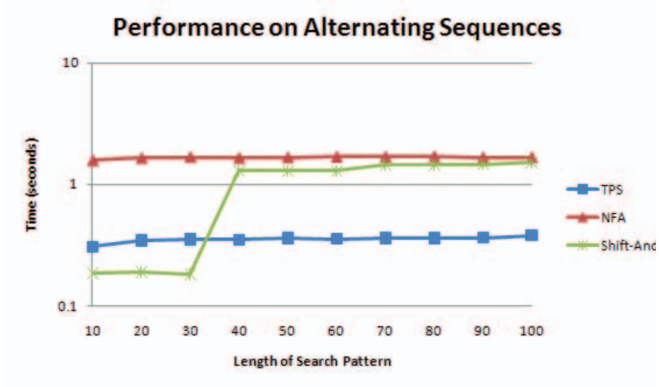

Fig. 5. Performance comparison of TPS, NFA, and Shift-And for search pattern lengths *m* = 10, 20, …, 100 for *alternating patterns*. The vertical axis is logarithmic in time. Similar to the case of *positive-only* patterns, TPS consistently outperforms NFA for these search patterns by roughly an order of magnitude. However, the Shift-And approach is faster when the *m* is at most 32.

sition according to a uniform distribution. These patterns are designed to illicit the worst performance in TPS.

For each input set of records, the script parses the records, constructs the appropriate data structures for TPS, NFA, and Shift-And, and constructs the NFA and Shift-And supporting data structures. Clock is then set to collect only the search time for each approach. Each pattern for each dataset is performed 10 times for each approach. The average is then presented here. The performance numbers are obtained by running our script in Java 1.6 32-bit environment on Windows Vista 64-bit with a dual core CPU (2.5GHz) and 4GB of RAM.

7.2 Results

We look at the results from two perspectives. First, we inspect how the length of a search pattern impacts the performance of the three algorithms. We do this by summing up the time taken to execute all patterns of one type across all alphabet sizes and compute the average time it takes to perform a search on all records.

Figure 4 shows the performance graph 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 on logarithmic scale. First, TPS performs consistently better than both NFA and Shift-And. TPS is more than one order of magnitude faster than NFA, and nearly one order of magnitude better than Shift-And when m>15. In the Shift-And algorithm, each .* is modeled as a separate state, so a pattern length of m in all-positive patterns has $2m$ states. This is why the performance hit occurs when m>16, as opposed to m>32 in other pattern types. The sharp decrease in performance represents the overhead required to support arbitrarily long search patterns. Secondly, the running time increases as the length of the pattern increases. We had expected the TPS to outperform NFA on the *positive-only pattern* portion of the evaluation as TPS's search over positive- only events is bounded by $O(m\lg(n))$, that is, this is the "best-case" scenario. However, we are a bit surprised by its (slight) advantage over Shift-And when the number of states is fewer than 32.

Next, we look at the results for the *alternating patterns*. Unlike the *positive-only patterns*, these patterns provide many opportunities for backtracking, and we expect to see the TPS performance to be worse than the previous graph. Figure 5 shows the performance graph 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. The Shift-And approach outperforms TPS for patterns whose length is at most 32, but has similar performance as NFA when the length is greater. We also see that both NFA and TPS 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.

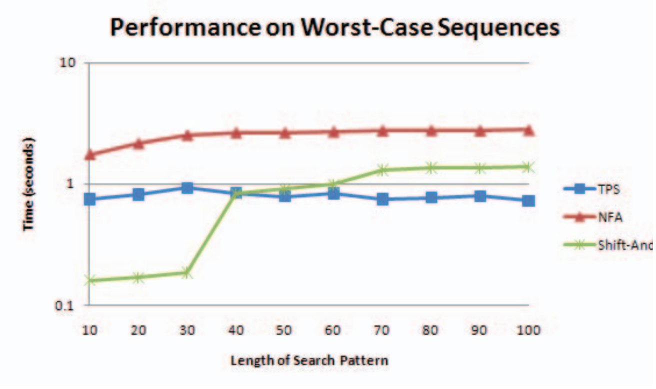

Fig. 6. Performance comparison of TPS, NFA, and Shift-And for search pattern lengths m = 10, 20, …, 100 for *worst-case* patterns. The vertical axis is logarithmic in time. Like two previous cases, TPS consistently outperforms NFA, but the differences are considerably smaller than in the comparison for *positive-only* patterns. Shift-And handily outperforms the other two for *m* < 40, but loses to TPS (significantly for *m* > 60) otherwise.

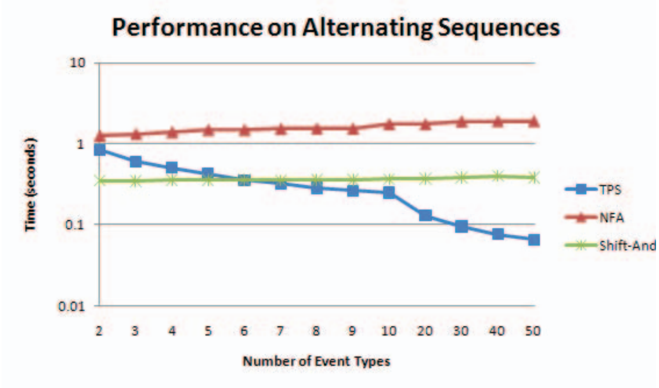

Fig. 7. Performance comparison of TPS, NFA, and Shift-And by varying the number of event types k = 2, 3, …, 9, and 10, 20, … 50 over all lengths of *alternating patterns*. The vertical axis is logarithmic in time in seconds. TPS performs similarly to NFA initially, but gets dramatically better thatn both NFA and Shift-And as the number of event types grows.

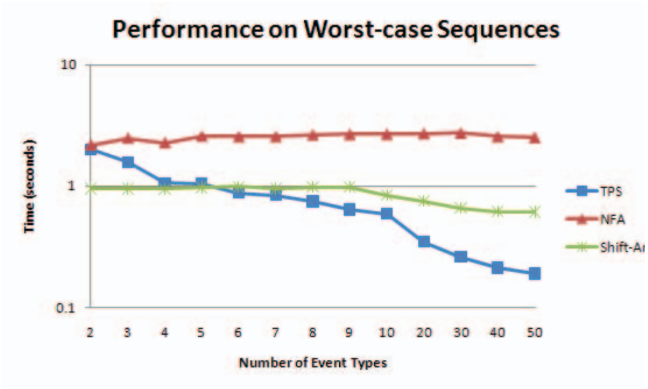

Fig. 8. Performance comparison of the three algorithms by varying the number of event types k = 2,3, …, 9 and 10, 20, …, 50 over all lengths of *worst-case* patterns. The performance graph is similar to Figure 7, with the only exception that Shift-And also improves in performance as the number of event types increases past 9.

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 when m grows from 10 to 30, which TPS also experiences (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 worst-case bound. Similar to the alternating cases, the Shift-And approach is very fast for m up to 30. However, it loses its performance edge to TPS with $m > 40$.

To study why TPS is performing relatively well when its worst case running time is comparatively bad, 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 number of events is large compared to the number of event types present in the search pattern, TPS can ignore most of events. On the other hand, both NFA and Shift-And are required to, in general, examine all the events. We focus on the two more difficult cases: alternating and worst-case. In Figure 7, we see that initially TPS and NFA have similar performance (m = 2). However, as k increases, both NFA's and Shift-And's performances remain roughly in the *alternating patterns* scenario. On the other hand, TPS's performance steadily improves as k increases. Figure 8 shows a similar trend with *worst-case patterns* scenario. The difference here is that the Shift-And approach also enjoys some performance gain when k is at least 10, though not as dramatic as TPS's.

On the flip side of the coin, if TPS is required to look at all of the events, TPS's performance is not expected to perform better than NFA. To show this, we fix the search pattern to be $A\bar{B}...A\bar{B}C$, 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. We performed this experiment for m = 3, 5, 7, 9, 11, 31, 41, 51, 61. Figure 9 shows the average performance for m = 3, 5, 7 for the three approaches on 5000 records for each n. n affects the running times linearly for n = 100,…, 1000. TPS and NFA have identical performance, while Shift-And outperforms them by an order of magnitude. When the same experiment is performed for larger m (m = 41, 51, 61), Shift-And's performance edge becomes very small (Figure 10).

7.3 Evaluation with Real Data

The experiments with random data above show the strengths and weaknesses of TPS. They give a reasonably clear picture on the situations where TPS is expected to perform well or badly. However, the evaluation would not be complete if these algorithms are not applied to real scenarios. We used the datasets listed in Table 1 and the search patterns our physician collaborators are interested in to evaluate how the three algorithms perform in real

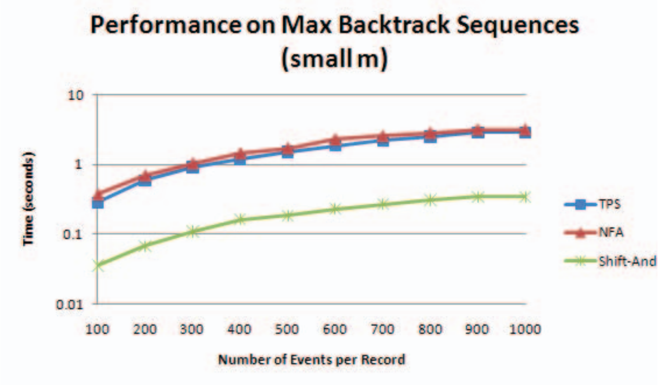

Fig. 9. Comparison of the performance of TPS, NFA and Shift-And by varying the number of events in a record n = 100, 200, …, 1000. The search pattern is fixed to be $A\bar{B}...A\bar{B}C$. The graph shows the average performance over small patterns:m = 3, 5, 7. The records are fixed to ABABAB…ABC. This guarantees n/2 backtracks for record. There are 5000 records for each n. TPS and NFA have similar performance, while the Shift-And algorithm outperforms the other two.

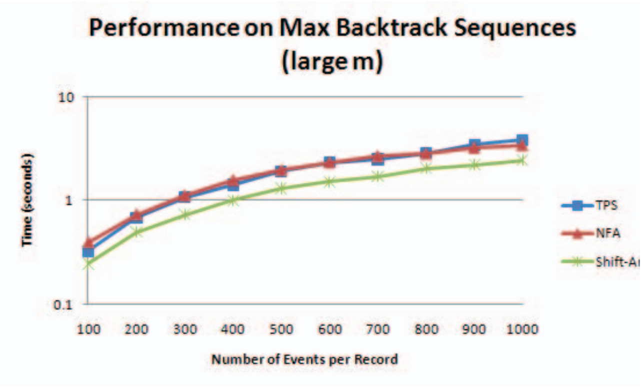

Fig. 10. Same comparison as Fig.9, except this graph shows the average performance over large pattern length: m = 41, 51, 61. The advantage of Shift-And is far less prominent when m is large.

scenarios. The datasets in Table 1 contain events that share the same time stamp. We had to modify these datasets to ensure that these events do not exist since neither the NFA nor the Shift-And approaches can handle them. We used the same method as described in Section 7.1 to make sure this is the case and converted the data to its equivalent and appropriate form for each of the three algorithms.

Table 4 shows the number of temporal patterns our collaborators are interested in for each dataset, the average length of the patterns, and how much time they took each algorithm to perform. We did not perform the evaluation for one of the datasets (Heart Attack) in Table 4 because our users had no temporal patterns they wanted to search for. TPS outperforms the other two approaches in all but the Creatinine dataset. Although Shift-And has its biggest advantage in randomized evaluation when the pattern lengths are small, TPS outperforms Shift-And here.

7.4 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 and in the real scenarios. The experimental conditions are chosen to reflect the situations under which we expect analysts to be using Lifelines2. In working with our collaborators in the medical field, the highest number of events in a single patient record is usually in the hundreds, and the lowest can be only a handful. The number of types of events is typically between 10-35 (although one dataset, not listed, has over 1000 event types). In practice, we do not encounter cases where k is so small and TPS performs only marginally better than NFA. We also do not encounter cases where TPS's advantage over NFA is greater than one order of magnitude because $k > 50$.

One the other hand, the Shift-And algorithm has a great speed advantage when m is at most 32 and consistently outperforms TPS in the experiments with random data. When it is not, its performance is slightly worse than that of TPS on average. TPS also has a slight performance edge in the evaluation with real data and search patterns. While the NFA approach and TPS are both easily extensible to handle additional constraints, the bit-parallel approaches are not. These approaches rely on bit masks built in the pre-processing stages so that when a symbol is read, the masks can be retrieved in constant time and applied. However, when these additional constraints are present, the masks can no longer be pre-built. The masks that need to be applied would be conditional upon the values in addition to each of the symbols read.

TABLE 4
REAL SCENARIO EVALUATION

Dataset Name	#Patterns	Ave. Length	Time (milliseconds)		
			TPS	NFA	Shift-And
Creatinine	5	3.2	51	325	20
Heparin	5	4.6	12	260	20
Heart attack	-	-	-	-	-
Transfer	5	3	93	981	99
Bipap	9	3.89	66	711	82

For $n \leq 1000$, the performance growth is roughly linear even in cases where the maximum number of searches and backtracks are required (Figure 9). In Figure 1, 2, and 3, we also did not observe a quadratic growth for $m \leq 100$, although the asymptotic bound for TPS contains an m^2 term. While we do not expect analysts to specify a pattern query of these extreme lengths, it is comforting to know that 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], [25], [14] 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.

8 Conclusions and Future Work

We present the novel algorithm TPS, and show that its running time is bounded by $O(m^2 n \lg(n))$. Although the bound is less attractive than that of the NFA approach, TPS performs favorably in our experiments against NFA. TPS also performs competitively with the Shift-And algorithm in real scenarios. 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, and that TPS is more easily extensible than bit-parallel algorithms such as Shift-And. Since the randomized experimental conditions we described here subsume the conditions under which we expect analysts to be using in our visualization tool, we expect the performances shown here to hold. Finally, we argue that using TPS in our application is a design success, and other similar applications may benefit from TPS.

The future directions of TPS include how we can expand the expressiveness of TPS to further support visual exploratory tasks. One immediate extension is to tie the pattern search with the visual operator alignment to enhance the experience for search and browse. Another would be to study how TPS can be modified to perform a search for multiple patterns at the same time or search for all instances of match events in a record.

ACKNOWLEDGMENT

We appreciate the support from NIH-National Cancer Institute grant RC1CA147489-02: Interactive Exploration of Temporal Patterns in Electronic Health Records.

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", *Communications of the Association for Computing Machinery (CACM),* 20(10), 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 (EDBT)*, pages 627-644, 2006.

[6] J. Fails, A. Karlson, L. Shahamat, and B. Shneiderman, "A Sisual 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] J.E. Hopcroft, R. Motwani, and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 2000.

[11] R.M. Karp and M.O. Rabin, "Efficient Randomized Patter Matching Algorithms," Technical Report TR-31-81, Aiken Computation Laboratory, Harvard University, 1981.

[12] D.E. Knuth, J.H. Moris, and V.R. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing*, 6(2):323-350, 1977.

[13] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia," *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications, Systems (ANCS)*, pages 155–164, New York, NY, USA, 2007. ACM.

[14] 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.

[15] S. Lam, "PatternFinder in Microsoft Amalga: Temporal Query Formulation and Result Visualization in Action", unpublished. http://www.cs.umd.edu/hcil/patternFinderInAmalga/PatternFinderS-HonorsPaper.pdf

[16] Microsoft Amalga, http://www.microsoft.com/amalga/, 2009

[17] A. Møller, "Regexp Library for Java", http://www.brics.dk/automaton/, 2001.

[18] S. Murphy, M. Mendis, K. Hackett, R. Kuttan, W. Pan, L. Phillips, V. Gainer, D. Berkowicz, J. Glaser, I. Kohane, H. Chueh, "Architecture of the Open-Source Clinical Research Chart from Informatics for Integrating Biology and the Bedside," *Proceedings of the American Medical Informatics Association Annual Symposium (AMIA '07)*, pages 548-552, 2007.

[19] G. Navarro, "Pattern Matching," *Journal of Applied Statistics*, 31(8), pages 925-949, 2004.

[20] G. Navarro and M. Raffinot, "Fast and Flexible String Matching by Combining Bit-Parallelism and Suffix Automata," *ACM Journal of Experimental Algorithmics* (JEA), 5(4), 4th article, 2000.

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

[22] C. Plaisant, S. Lam, B. Shneiderman, M. Smith, D. Roseman, G. Marchand, M. Gillam, C. Feied, J. Handler, and H. Rappaport, "Searching Electronic Health Records for Temporal Patterns in Patient Histories: A Case Study with Microsoft Amalga," *Proceedings of the American Medical and Informatics Association Annual Symposium (AMIA '08)*, pages 601-605, 2008.

[23] M.O. Rabin and D. Scott, "Finite Automata and Their Decision Problems," IMB Journal of Research and Development, volume 2, pages 114-125, 1959.

[24] P. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi, "Expressing and Optimizing Pattern Queries in Database Systems," *ACM Trans. on Database Systems*, 29(2), pages 282–318, June 2004.

[25] M. Suntinger, H. Obweger, J. Schiefer, and M. E. Gröller, "The Event Tunnel: Interactive Visualization of Complex Event Streams for Business Process Pattern Analysis," *Proceedings of the IEEE Pacific Visualization Symposium (PacificVIS '08)*, pages 111-118, 2008.

[26] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6), pages 419-422, 1968.

[27] 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," *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*, pages 457-466, 2008.

[28] T.D. Wang, C. Plaisant, B. Shneiderman, N. Spring, D. Roseman, G. Marchand, V. Mukherjee, and M. Smith, "Temporal Summaries: Supporting Temporal Categorical Aggregation and Comparison," *IEEE Transactions on Visualization and Computer Graphics*, 15(6), pages 1049-1056, 2009.

[29] H. Hu, B. Salzberg, and D. Zhang, "Online Event-Driven Subsequence Matching over Financial Data Sstream," Proceedings of the 2004 ACM SIGMOD International Conference, pages 23-34, 2004.

[30] 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 (2005) and PhD (2010) from the University of Maryland. 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.