An Integer Programming Approach to Temporal Pattern Matching Queries

Megan Monroe and Amol Deshpande
Department of Computer Science
University of Maryland
College Park, Maryland 20742
Email: madeyjay@umd.edu, amol@cs.umd.edu

***Abstract*—Around the globe, an increasing number of our actions and activities are being recorded and stored as categorized, timestamped events. This type of data, which comprises electronic health records, shipment tracking data, and process logs, is well-suited to the tabular structure of standard database relations. The difficulty, however, is that temporal pattern matching queries (one of the most common types of queries over this type of data) are not well-suited to standard relational query processing. This difficulty is exacerbated when the question is not only whether an event record matches a given query, but how best to match the events in the record to the events in the query when multiple options are present. In this paper, we demonstrate that temporal pattern matching queries can be formulated and solved as integer programs. This novel approach has distinct advantages over standard query processing techniques in that it puts no ordering constraints on the underlying event record, and can easily discern between multiple possible solutions using the objective function. Additionally, these integer programs can be constructed incrementally, facilitating an easy translation between the query specification interface and the underlying execution instructions. We show that integer programs can capture a wide range of temporal pattern matching constructs, including intervals, absences, repetition, and flexibility, and provide new insights on both the strengths and drawbacks of implementing this processing strategy in practice.**

I. Introduction

Categorical, temporal event data (event data) can be used to describe almost any historical occurrence. It serves as the cornerstone of electronic health records, shipment tracking data, and process logs. It is also being gleaned from an increasing number of sensors that track everything from traffic conditions to eye movements. Essentially, this type of data answers three questions:

- What happened?
- Who (or what) did it happen to?
- When did it happen?

The prevalence of event data can be attributed to its simplicity - its ability to represent complex, real-world activities as a small set of discreet values. A single event is typically represented by the following four fields:

Event Category	Record ID	Start Time	End Time

The latter two fields give rise to the two types of events that can be recorded: point events and interval events. Point events occur at a single point in time, and are represented using only a single timestamp (the start time). Interval events occur over a duration of time, and have both a start time and an end time. In both cases, these events are well-suited to the tabular structure of standard database relations, making it an ideal format for representing our daily life.

Event Category	Record ID	Start Time	End Time
Drug A	Patient X	1/10/13	1/20/13
Stroke	Patient X	1/15/13	

Event Category	Record ID	Start Time	End Time
Headache	Patient X	1/15/13	
Stroke	Patient X	1/15/13	

Fig. 1: Top: The patient's Stroke event is recorded after the interval event, even though it occurs *while* the patient was taking Drug A. Bottom: Ordering is imposed even though the events are concurrent.

The difficulty arises when event data must be queried. One of the most common types of queries over such data are *temporal pattern matching (TPM) queries*, which are formulated as a pattern of events, potentially with additional temporal constraints. The query should return all the records (such as patient records or shipment records) that contain the specified pattern. TPM queries are notoriously ill-suited to standard, relational query processing. Answering such a query with standard relational operators results in a lengthy series of sorting and large self-join operations. This strategy does not scale well as datasets get increasingly larger. Furthermore, relational storage structures can obscure the order of events in the record, either by representing events out of order, or by imposing an order when none exists (see Figure 1).

Ironically, TPM queries are very similar to matching queries over strings [1], a data type that, unlike event data, is unanimously *not* well-suited to standard relational storage. Thus, event data is perpetually in the awkward position of being conveniently stored in an environment that does not facilitate querying. The query strategies that have emerged from various data analytics fields address this with customized preprocessors and data structures that are tailored specifically to TPM queries, and run either within or outside of the database system.

Then things get complicated. It is not always enough to query event records based on whether they do or do not contain a certain pattern of events. In many cases, there will be multiple ways to match the query pattern to the events in a record. This is particularly true of longitudinal medical records that

contain multiple years of patient treatment history. In these cases, the query must not only determine whether there is a match, but must also determine the events that *best* match the query pattern (best-match TPM). This is a pressing need in real-world scenarios such as patient similarity and process optimization. For example, a medical researcher might want to find the shortest hospital stay for a given patient, during which a certain medication was administered. Current data mining techniques focus primarily on the existential aspect of queries, but do not determine best-match.

In this paper, we demonstrate that best-match TPM is NP-Hard, and use this complexity to motivate an integer programming (IP) approach to solving these problems. Integer programs are an appealing solution for best-match TPM because they:

1) Require no ordering of the underlying event record.
2) Can easily discern between multiple possible solutions using the objective function.
3) Allow query patterns to be constructed using as many or as few ordering constraints as necessary.
4) Can be constructed incrementally as queries are specified and reformulated.
5) Provides a natural metric for ranking results, again using the objective function.

This paper is organized as follows: related work is presented in Section II, followed by a description of the query specification interface in Section III. The hardness of best-match TPM is discussed in Section IV. A full description of how a best-match TPM is formulated into an integer program is presented in V, followed by a simple example of such a formulation in Section VI. In Section VII, we explore the feasibility of this approach in practice using three example queries. Finally, we discuss future work and conclude in Section VIII.

II. Related Work

Query over temporal event data have been addressed most prominently in two primary domains: temporal database querying and data mining. The former has focused its efforts on tailoring standard, relational query languages to better account for temporal data ([2], [3], [4], [5], [6]). The temporal database field, however, revolves around a state-based perspective of the database rather than an event-based perspective. That is, the foremost concern is the state of the database at any given moment, rather than the sequence of events that brought the database to that state. These adapted languages reflect this, and thus are not ideal for TPM queries.

Furthermore, temporal databases are still rooted in the relational data model. Query languages such as TSQL and TQuel ([7], [8]) have constructs for specifying temporal relationships, but they function primarily as shortcuts. The query processing remains tethered to the standard relational operators. The notion of best-match TPM has yet to be explicitly addressed in this domain.

The data mining community has also produced a large body of work on temporal event query ([9], [10], [11], [12], [13], [14]). Data mining approaches have been used to serve a variety of purposes, including determining patient similarity ([15], [16], [17], [18]). The focus of this work has been two-fold: finding frequently occurring event patterns and determining whether a record contains a certain event pattern. Best-match TPM, again, is typically omitted.

III. Query Specification

The integer programming strategies discussed in this paper are designed to serve as the back-end processing for a visualization and analytics tool called EventFlow (name changed for review). EventFlow employs a graphical query interface, described in detail in [19], [20], that allows users to draw out event patterns of interest (See Figure 2). The goal of the IP formulation presented here is to match the current functionality of this query specification interface. A listing of this functionality, as well as the corresponding graphics that will be used throughout this paper, can be found in Table I.

EventFlow is an ideal testing ground for an IP back-end due to the scalability constraints of the visualization component of the software. In order to produce a readable visualization, EventFlow operates on datasets that consist of thousands of records (as opposed to millions), with each record consisting of between 20-100 events. For larger datasets, EventFlow is loaded with a sample subset, which is typically sufficient for users to identify and analyze the predominant trends. Thus, the query processing strategy does not need to scale significantly beyond this "sweet spot."

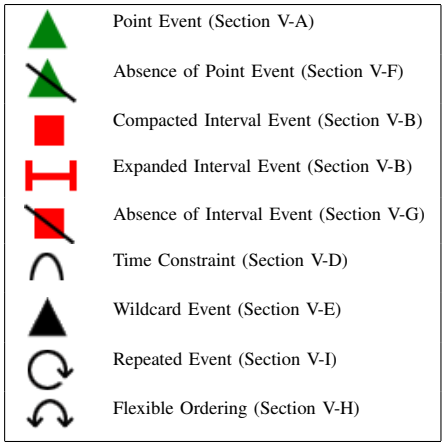

TABLE I: Query graphics, for reference throughout this paper. Different colors indicate different categories of events.

From a usability standpoint, a critical feature of search is the ability to rank results. Ranking provides structure for the result presentation, as well as helps users identify possible false positives due to errors in their query [19]. In EventFlow, the current metric for ranking results is the count of extraneous events in the record that were not matched to the query. In many scenarios, however, this is not an informative metric. In fact, the ideal ranking metric is in continuous flux. Users execute queries, evaluate the results, and generate new questions, which in turn lead to new queries or perhaps even a different perspective on the same query. The appropriate metric for ranking query results can change as quickly as these questions evolve.

It is appropriate then, to choose a query processing strategy that naturally produces a ranking metric that is both flexible

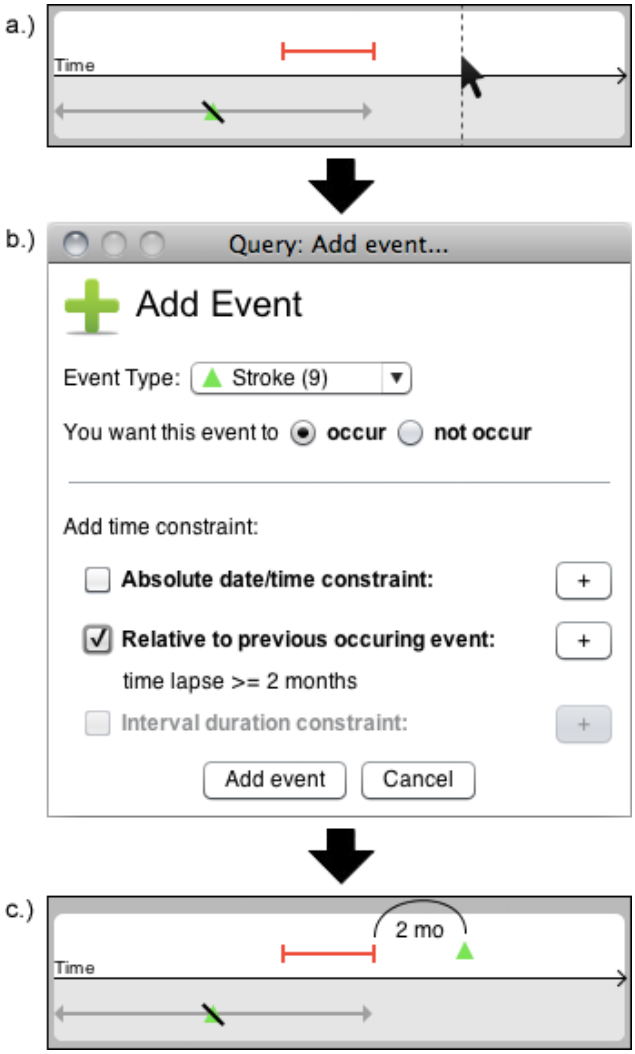

Fig. 2: EventFlow's graphical query system allows users to draw out a pattern of interest by clicking on the query canvas (a) and specifying the event attributes (b). The new event is then added to the query (c).

and informative. An IP back-end accomplishes this using the objective function. The objective function can not only distinguish between multiple possible matches within an event record, but also distinguish the best match across records. The final value of the objective function can be used to rank the matched search results, and can be tailored to the users' current focus of analysis.

Additionally, EventFlow allows users to construct queries incrementally, by adding each element individually to the query canvas. Elements can be both modified and removed, and new queries can evolve from making slight adjustments to the previous query. This differs from many command-based query systems, in which a change to a single query element can alter the structure of the entire query.

This incremental query specification interface translates nicely into an integer programming back-end, where constraints can be added, modified, and removed in the same, incremental fashion. The constraints are almost entirely independent of each other (absences, which consist of paired constraints, are the one exception), and thus changes to one constraint will not require changes to the entire set of constraints. For example, if a user modifies their query to require that there is at least a 3-day time lapse between two pre-existing events, the resulting back-end change would modify only the time constraint between those two events. All of the other constraints would remain as they were.

IV. The Hardness of Best-Match Event Query

The difficulty of selecting a back-end processing strategy for TPM, and specifically best-match TPM, is that the hardness of the problem is highly dependent on the query itself. Consider, for example, the query depicted in Figure 3. The query is for two consecutive point events, and we can assume that the *best* match is the one that minimizes the time lapse between these two events.

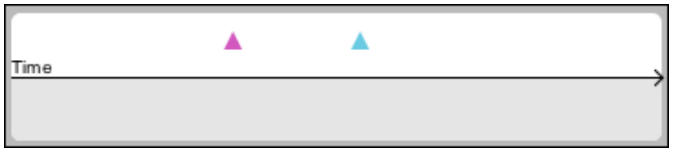

Fig. 3: Query for two consecutive point events.

Clearly this query can be processed against a given record by performing a linear scan across all of the events in that record. However, this strategy becomes less straightforward as the query gets increasingly complex. As query elements such as absences and repetition are specified, linear scan strategies must be modified to allow for backtracking when matches for the first elements in the query are invalidated by subsequent event patterns [13]. These queries can extend the back-end workload to quadratic and cubic complexity.

When overlapping interval events are integrated into both the event record and the query, the hardness of best-match TPM escalates even further. A series of overlapping interval events can be represented as an interval graph, where each vertex represents an event, and an edge is present between events that overlap each other (see Figure 5). Consider, for example, the query for three overlapping intervals, where the sequence of interval start and end points can occur in any order (Figure 4). When a query of overlapping intervals such as this is processed against an event record, the problem is equivalent to determining whether the interval graph of the query is a subgraph of the interval graph of the record (interval subgraph isomorphism).

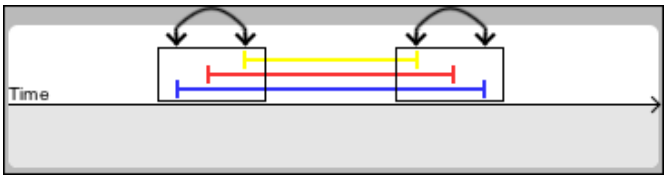

Fig. 4: A query for three overlapping intervals, where the start and end points can occur in any order.

Much like TPM queries, there are many varieties of the interval subgraph isomorphism problem, with equally varied hardness. Marx and Schlotter proved that induced subgraph isomorphism is NP-complete when graphs G and H are restricted to be interval graphs [21]. Heggernes et al. proved that when G is an interval graph and H is a connected

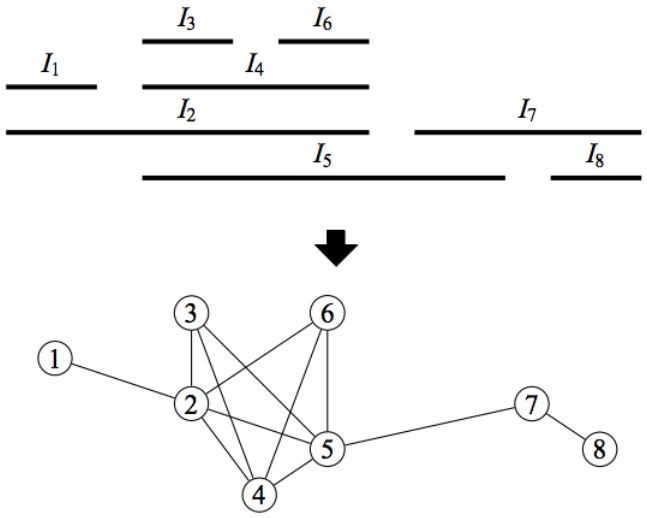

Fig. 5: A pattern of overlapping intervals across a linear timeline can be translated into a graph representation.

proper interval graph, the problem is solvable in polynomial time [22]. Kijima et al. demonstrated that spanning subgraph isomorphism is NP-Hard for proper interval graphs using a reduction of 3-partition [23]. A simple, temporal pattern matching query without absences or other complex constructs (but with overlapping intervals) is equivalent to this last problem, and hence is NP-Hard.

Thus, a back-end processing strategy for TPM queries must address a problem that can range from linear complexity to NP-Hardness. While many strategies strive to find a middle ground along this continuum, in this paper we take a top down approach by implementing an integer programming back-end, which is known to be NP-Hard. We apply this strategy to the more complex event queries, and then evaluate its ability to scale down to simpler queries.

V. An Integer Programming Approach

In this section, our focus is to find the events in a given record that *best match* given a temporal pattern query. We demonstrate how to incrementally construct an integer program to solve TPM queries. We begin with the most basic queries, those that involve only a sequence of point events, and then progress through increasing complex constraints involving interval events, absences, order independence, and repetition.

A. Point Events Only

Consider a single record, consisting of events $E = \{e_1, e_2, e_3, ...e_i\}$, where each event is represented by two dimensions: time and category, i.e. $e_x = \{e_{x\_t}, e_{x\_c}\}$. Similarly, in this initial case, a query pattern consists of elements $Q = \{q_1, q_2, q_3.....q_j\}$, where $q_y = \{q_{y\_c}\}$, i.e. for each query element, we are provided an event category. Since the query is specified across a single timeline, we have the additional constraint that $q_{y\_t} \leq q_{(y+1)\_t} \forall 1 \leq y < j$.

The query is processed by matching each element in Q to an event in E. This is done using an $i \times j$ table of binary variables $M = \{m_{1,1}, m_{1,2}, ...m_{i,j}\}$, where $m_{x,y} = 1$ if event e_x is matched to element q_y. Otherwise $m_{x,y} = 0$. This table will be referred to as the *match table*.

	q_1	q_2	q_3	$\cdots$	q_j
e_1	$m_{1,1}$	$m_{1,2}$	$m_{1,3}$	$\cdots$	$m_{1,j}$
e_2	$m_{2,1}$	$m_{2,2}$	$m_{2,3}$	$\cdots$	$m_{2,j}$
e_3	$m_{3,1}$	$m_{3,2}$	$m_{3,3}$	$\cdots$	$m_{3,j}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
e_i	$m_{i,1}$	$m_{i,2}$	$m_{i,3}$	$\cdots$	$m_{i,j}$

The match table is subject to three primary constraints:

$$\{\forall 1 \leq y < j\}, \sum_{x=1}^{i} m_{x,y} = 1 \tag{1}$$

$$\{\forall 1 \leq y < j\}, \sum_{x=1}^{i} m_{x,y} \cdot e_{x\_c} = q_{y\_c} \tag{2}$$

$$\{\forall 1 \leq y < j\}, \sum_{x=1}^{i} m_{x,y} \cdot e_{x\_t} < \sum_{x=1}^{i} m_{x,y+1} \cdot e_{x\_t} \tag{3}$$

The first constraint, specified for every column of the table, is that each query element must be matched to one, and only one event (Equation 1: **One-to-one Constraints**). The second constraint requires that a query element must be matched to an event of the same category (Equation 2: **Category Constraints**). Finally, the events must be matched in the correct temporal order (Equation 3: **Ordering Constraints**). For events that must occur at the same time, the "$<$" sign is replaced with an "$=$" sign. The timestamp, e_{x_t}, of the event that is matched to a given query element q_y will be referred to as q_{y_t}.

$$\sum_{x=1}^{i} m_{x,y} \cdot e_{x\_t} = q_{y\_t} \tag{4}$$

What is critical to point out is that, while some notion of order must be maintained across the query elements, there are no ordering requirements on the event record. A given record can be extracted from a database, and integrated into the IP as is. It does not need to be sorted or stored in any sort of custom data structure. Temporal relationships like *during* or *concurrently* are handled naturally by the IP constraints.

B. Points and Intervals

With the inclusion of interval events, there is the additional constraint that if an interval's start point is matched to a query element, it's end point must be matched to the corresponding end-point in the query. This can be accomplished by having duplicate entries in the match table.

Consider a small example in which (q_2, q_4) represents an interval's start and end point in the query sequence, and (e_1, e_3) represents an interval's start and end point in the event record. In this scheme, q_2 can be matched to e_1 if and only if q_4 can be matched to e_3. Using our previous table scheme, we have the following constraint:

$$m_{1,2} = m_{3,4} \tag{5}$$

However, we can eliminate this constraint by simply allowing $m_{1,2}$ to appear twice within the table (note that these duplicates will never appear in the same row or column):

	q_1	q_2	q_3	q_4
e_1	$m_{1,1}$	$[m_{1,2}]$	$m_{1,3}$	$m_{1,4}$
e_2	$m_{2,1}$	$m_{2,2}$	$m_{2,3}$	$m_{2,4}$
e_3	$m_{3,1}$	$m_{3,2}$	$m_{3,3}$	$[m_{1,2}]$
e_4	$m_{4,1}$	$m_{4,2}$	$m_{4,3}$	$m_{4,4}$

These duplicate entries in the match table occur for every pair of query/event record intervals.

**Note: For the sake of simplicity, the duplicate entries in the match table should be assumed, but will not be explicitly stated in subsequent equations.

C. Event Attributes

A timestamp and a category (e_{x_t} and e_{x_c}) are the fundamental features of each event. However, events can have additional attributes, denoted e_{x_a}, that can represent anything from a prescription dosage to a temperature. These attributes can be easily incorporated into constraints of the IP. For example, the following constraint specifies that the dosage should be less than 30mg.

$$\sum_{x=1}^{i} m_{x,3} \cdot e_{x\_\text{a: dosage}} < 30mg \tag{6}$$

D. Time Constraints

There are three primary time constraints that can be incorporated into a query: gaps between events, interval durations, and absolute dates (Figure 2). All of these can be incorporated into the IP using a modified version of the ordering constraints (Equation 3). For example, to specify that an event must occur at least 3 months after the preceding event, the ordering constraint would be augmented to:

$$q_{b\_t} - q_{a\_t} \geq 3 \text{ months} \tag{7}$$

E. Wild Card Elements

Similar to string matching, a wild card query element can be matched to an event of any category. For example, in a dataset of hospital transfers, a query could be formulated for patients who arrived to the Emergency Room, and then were transferred through at least one other department before being discharged (see Figure 6). Wild card events actually simplify the integer program. They require one-to-one and ordering constraints to be applied, similar to a normal point event, but do not require any category constraints.

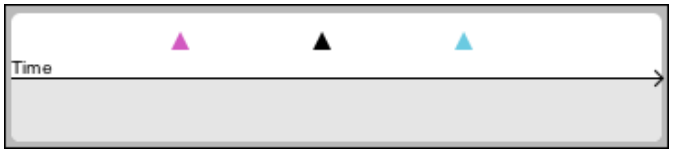

Fig. 6: Query for patients who arrived to the Emergency Room (in pink), and then were transferred through at least one other department before being Discharged (in blue). This wild card event (in the middle) is represented in black.

F. Point Event Absences

Ironically, the absence of a point event actually functions as an interval of time. Consider the query for two Stroke events, with no Diagnosis in between them. The implication is that there are no Diagnosis events for the entire interval of time between the two Stroke events. We refer to this as a "span" absence: the absence of the Diagnosis event implicitly spans out to the presence events on either side of it.

Fig. 7: Query for two stroke events (in green), with no diagnosis (in orange) in between.

In this query, the two Stroke events are represented by consecutive query elements q_y and q_{y+1}. For every Diagnosis event $e_d = \{e_{d\_t}, e_{d\_c}\}$ in the event record then, there must be a constraint specifying that e_{d_t} does not occur between the event times that are matched to q_y and q_{y+1}:

$$q_{y\_t} \geq e_{d\_t} \qquad \text{or} \qquad q_{(y+1)\_t} \leq e_{d\_t} \tag{8}$$

These constraints are specified by introducing a new variable $a \in [0,1]$ and a large constant K. For our purposes we use K= Long.MAX_VALUE, which represents that highest possible time. The absence is then specified using the following two constraints:

$$q_{y\_t} + \mathrm{K}a \geq e_{d\_t} \tag{9}$$

$$q_{(y+1)\_t} - \mathrm{K}(1-a) \leq e_{d\_t} \tag{10}$$

Using these constraints, if $q_{y\_t} \geq e_{d\_t}$, then $a = 0$ and both constraints are satisfied. Conversely, if $q_{(y+1)\_t} \leq e_{d\_t}$, then $a = 1$ and both constraints are still satisfied. Finally, if $q_{y\_t} \leq e_{d\_t} \leq q_{(y+1)\_t}$, then the constraints cannot be satisfied.

G. Interval Event Absences

The absence of an interval event can function similar to the absence of a point event. That is, it can span out to the presence events on either side of it. For example, one might formulate a query for two Stroke events, where the patient did not take his medication for the entire interval of time in between them.

Fig. 8: Query for two stroke events (in green), with no medication intervals (in red) in between.

In this case, the absence of the interval can be specified much in the same way as the absence of a point event. For every medication interval in the patient record, represented by two event points $e_{m-start} = \{e_{m-start\_t}, e_{m-start\_c}\}$ and $e_{m-end} = \{e_{m-end\_t}, e_{m-end\_c}\}$, the interval must either end before the first Stroke event, or begin after the second Stoke event:

$$q_{y\_t} \geq e_{m-end\_t} \qquad \text{or} \qquad q_{(y+1)\_t} \leq e_{m-start\_t} \tag{11}$$

However, it is also possible to query for the absence of an interval that occurs at a single point in time. We refer to this as a "point" absence. Consider, for example, the query for a patients who stopped taking their medication at some point between their two Stroke events.

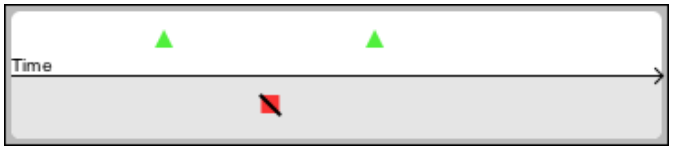

Fig. 9: Query for patients who stopped taking their medication (in red) at some point between their two Stroke events (in green).

A "point" absence can be handled by translating it into a "span" absence. This is done by framing the absence with two wild card query elements, and allowing the absence to span out to these elements. The ordering constraints are then modified to allow for equality. That is, the wild card elements are allowed to match to the same events that are matched to the Stroke elements.

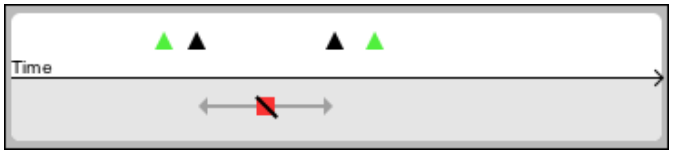

Fig. 10: Modified query for patients who stopped taking their medication (in red) at some point between their two Stroke events (in green).

H. Flexible Ordering

One of the most salient benefits of using an integer programming approach for solving best-match TPM is that ordering constraints can be added between any two events. Or perhaps better put, ordering constraints can be *not* added between any two events. Queries can be easily formulated such that certain events can appear in any order. For example, the following query requires the occurrence of three events. However, the first two events can occur in any order.

Using ordering constraints, this query can be specified such that the first two events must occur before the third, but no

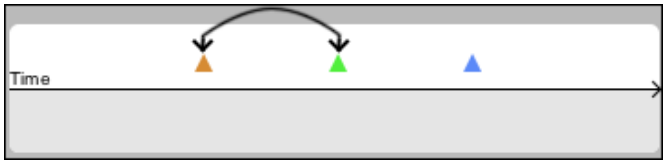

Fig. 11: Query for three events, where the first two events can occur in any order.

constraint is added to dictate the ordering of the first and second event.

$$\sum_{x=1}^{i} m_{x,1} \cdot e_{x\_t} < \sum_{x=1}^{i} m_{x,3} \cdot e_{x\_t} \tag{12}$$

$$\sum_{x=1}^{i} m_{x,2} \cdot e_{x\_t} < \sum_{x=1}^{i} m_{x,3} \cdot e_{x\_t} \tag{13}$$

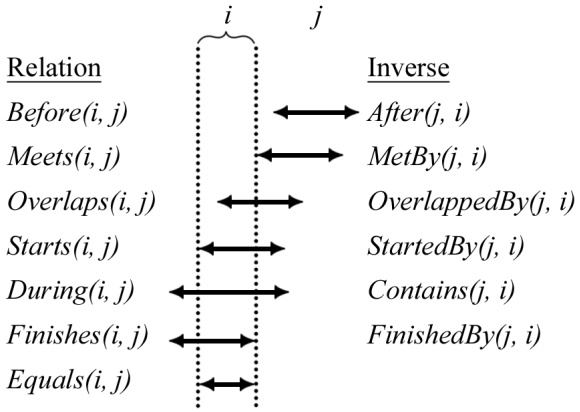

Fig. 12: Allen's 13 interval relationships [24].

This flexibility becomes even more useful with queries involving interval events. A common query is to find records in which two intervals overlap. Allen's seminal work on interval logic [24] described 13 different ways in which two intervals can occur in relation to one another (see Figure 12). Of these, 8 relationships involve an overlapping. This means that, without order independence, 8 different queries would be required to capture this basic relationship. However, an integer program can capture this relationship with only four ordering constraints, specifying only that the two interval start points must occur before the two interval end points.

$$\sum_{x=1}^{i} m_{x,1} \cdot e_{x\_t} < \sum_{x=1}^{i} m_{x,3} \cdot e_{x\_t} \tag{14}$$

$$\sum_{x=1}^{i} m_{x,2} \cdot e_{x\_t} < \sum_{x=1}^{i} m_{x,3} \cdot e_{x\_t} \tag{15}$$

$$\sum_{x=1}^{i} m_{x,1} \cdot e_{x\_t} < \sum_{x=1}^{i} m_{x,4} \cdot e_{x\_t} \tag{16}$$

$$\sum_{x=1}^{i} m_{x,2} \cdot e_{x\_t} < \sum_{x=1}^{i} m_{x,4} \cdot e_{x\_t} \tag{17}$$

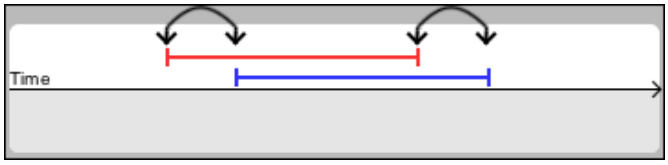

Fig. 13: Query for overlapping intervals.

Essentially, ordering constraints must only be added when they are necessary, and each element in the query can be ordered according to any other query elements. This allows queries to incorporate a large amount of flexibility, without introducing additional complexity in the query processing.

I. *Event Repetition*

Event repetition occurs when a given query element, or pattern of query elements can be matched to an indeterminate number of events in the record. For example, consider the query for patients who went to the Emergency Room at least 4 times and at most 5 times after having a Stroke.

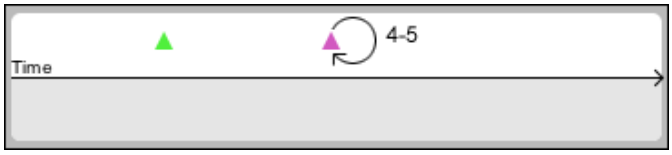

Fig. 14: Query for patients who went to the Emergency Room (in pink) either 4 or 5 times after having a Stroke (in green).

This can be accomplished relaxing the one-to-one constraint for the final query element, allowing for this element to be matched to 0 events in the patient record (Equation 18). Additionally, the category constraint must be modified to account for the scenario when this element is not matched (Equation 19). Finally, a similar consideration must be made for the ordering constraints (again, we use K to represent a large constant - Equation 20).

$$\sum_{x=1}^{i} m_{x,6} <= 1 \tag{18}$$

$$\sum_{x=1}^{i} m_{x,6} \cdot e_{x\_c} = q_{6\_c} \cdot \sum_{x=1}^{i} m_{x,6} \tag{19}$$

$$\sum_{x=1}^{i} m_{x,5} \cdot e_{x\_t} < \sum_{x=1}^{i} m_{x,6} \cdot e_{x\_t} + textrmK(1 - \sum_{x=1}^{i} m_{x,6}) \tag{20}$$

J. *The Objective Function*

While the above constraint strategies will determine whether a given event record contains the query sequence, the objective function will dictate which events in the record will be matched to the query when there are multiple possible options. The objective function is highly context dependent, however, the following examples should provide a sense of how the objective function can guide the selection of a best-match:

- Minimize/maximize the duration of the query sequence:
  MIN: $\sum_{x=1}^{i} m_{x,j} \cdot e_{x\_t} - \sum_{x=1}^{i} m_{x,1} \cdot e_{x\_t}$
- Match query elements to events close to the beginning/end of the record:
  MAX: $\sum_{x=1}^{i} m_{x,j} \cdot e_{x\_t}$
- Minimize/maximize the lapse in time between two elements of the query sequence:
  MIN: $\sum_{x=1}^{i} m_{x,5} \cdot e_{x\_t} - \sum_{x=1}^{i} m_{x,2} \cdot e_{x\_t}$
- Minimize/maximize the number of events matched to the query:
  MIN: $\sum_{x=1}^{i} \sum_{y=1}^{j} m_{x,y}$
- Minimize/maximize an attribute of one of the events:
  MAX: $\sum_{x=1}^{i} m_{x,j} \cdot e_{x\_a}$

VI. An Example

Consider the following query, being evaluated against the given event record. In this case we are looking for patients who had never had a Stroke prior to taking Drug A, but then had a Stroke after starting their prescription. We would like to ensure that the patient took the prescription for at least one month, and finally, we would like minimize the time lapse between the start of the prescription and the Stroke. The query evaluation process begins by breaking the query and the event record into their respective sets $Q = \{q_1, q_2, q_3\}$ and $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ (Figure 15).

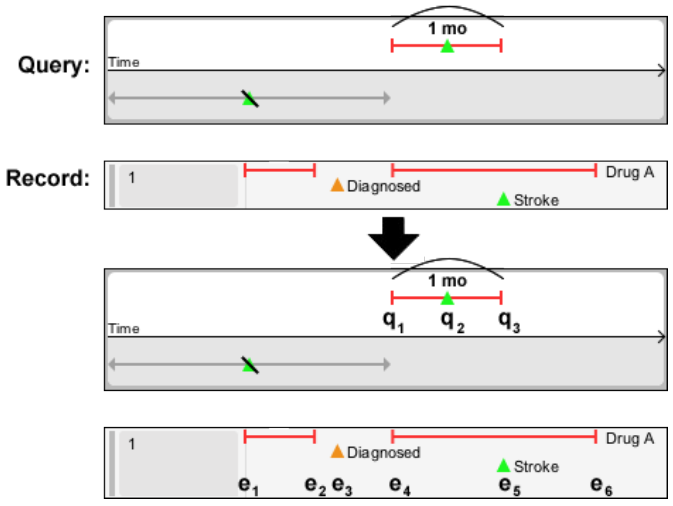

Fig. 15: The query and the event record get broken into sets $Q = \{q_1, q_2, q_3\}$ and $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ respectively.

For the sake of clarity in the example, event timestamps and categories will be represented as integers. To distinguish between interval start and end points, which have the same event category but cannot be matched to the same events, interval end points will be represented by appending a decimal

to the original category. The sets Q and E, and the match table are shown in tables II and III respectively.

$q_y = \{q_{y\_c}\}$	$e_x = \{e_{x\_t}, e_{x\_c}\}$
$q_1 = \{1\}$	$e_1 = \{1,1\}$
$q_2 = \{2\}$	$e_2 = \{3,1.5\}$
$q_3 = \{1.5\}$	$e_3 = \{4,3\}$
	$e_4 = \{6,1\}$
	$e_5 = \{7,2\}$
	$e_6 = \{9,1.5\}$

TABLE II: Sets Q (left) and E (right).

	q_1	q_2	q_3
e_1	$m_{1,1}$	$m_{1,2}$	$m_{1,3}$
e_2	$m_{2,1}$	$m_{2,2}$	$m_{1,1}$
e_3	$m_{3,1}$	$m_{3,2}$	$m_{3,3}$
e_4	$m_{4,1}$	$m_{4,2}$	$m_{4,3}$
e_5	$m_{5,1}$	$m_{5,2}$	$m_{5,3}$
e_6	$m_{6,1}$	$m_{6,2}$	$m_{4,1}$

TABLE III: Binary variables of the match table.

The IP is then formulated using the 9 constraints and objective function depicted in Figure 16. The one-to-one constraints are enforced by equations 21-21. Category constraints are enforced by equations 21-21, and ordering is maintained by equations 21 and 21 (note that the Stroke event and the prescription end point can occur in any order). The absence of a Stroke event prior to the prescription is handled by equation 21. Finally, the objective function is set by equation 21.

Hopefully it is clear that, intuitively, the given event record should match to this query based on the following event matches: $q_1 \rightarrow e_4$, $q_2 \rightarrow e_5$, $q_3 \rightarrow e_6$ (Figure 17), which corresponds to the match table shown in Table IV. These values can be plugged back into our 9 constraints to confirm that they all hold.

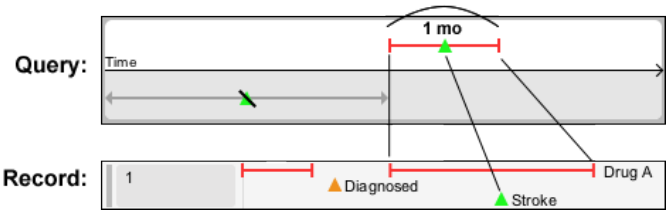

Fig. 17: The query and the event record are a match given $q_1 \rightarrow e_4$, $q_2 \rightarrow e_5$, $q_3 \rightarrow e_6$.

	q_1	q_2	q_3
e_1	$m_{1,1} = 0$	$m_{1,2} = 0$	$m_{1,3} = 0$
e_2	$m_{2,1} = 0$	$m_{2,2} = 0$	$m_{1,1} = 0$
e_3	$m_{3,1} = 0$	$m_{3,2} = 0$	$m_{3,3} = 0$
e_4	$m_{4,1} = 1$	$m_{4,2} = 0$	$m_{4,3} = 0$
e_5	$m_{5,1} = 0$	$m_{5,2} = 1$	$m_{5,3} = 0$
e_6	$m_{6,1} = 0$	$m_{6,2} = 0$	$m_{4,1} = 1$

TABLE IV: The final match table values.

VII. Experiments and Discussion

To evaluate the feasibility of IP query processing in practice, three representative sample queries of increasing complexity (Q1, Q2, and Q3 - shown in Figures 18, 19, and 20, respectively) were run against four different record sizes: 20 events, 40 events, 80 events, and 160 events. The queries were run using the open source (mixed-integer) linear programming system, `lp_solve` [25]. Each query was run against both a matching and a non-matching record at each record size. The run times are depicted in Figures 21 and 22.

Again, the goal of these experiments was to evaluate whether an IP processing approach would even be feasible in order to be considered as a potential back-end for the EventFlow query system. The experiments were meant to target aspects such as whether the query completed, completed accurately, and completed within a reasonable time frame. An exhaustive evaluation across larger sets of records to determine the myriad factors that can subtly affect run times will follow in a future work.

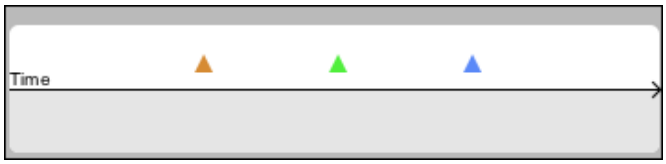

Fig. 18: Q1 - Test query for three sequential point events.

Fig. 19: Q2 - Test query incorporating both absences and repetition.

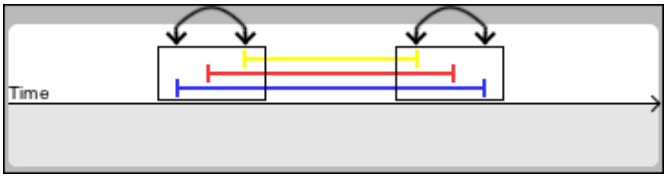

Fig. 20: Q3 - Test query for interval overlapping.

The three queries were chosen for their ability to push the limits of `lp_solve` in different ways as they scaled up to larger event records. Q1 is the simplest query in that it is the easiest to match. That is, an event record with any permutation of these three event types will likely be able to match this subsequence of events. However, as the record gets larger, the number of possible ways to match the query increases exponentially, requiring the system to choose the best match out of a large number of possible options.

Q2 was designed to stretch the number of constraints that comprise the IP. Due to the absence element in the query, two constraints must be added for each record event of that category. The number of absence constraint pairs increased from 7 to 63 as the record size increased from 20 events to 160 events.

Finally, Q3 was chosen to represent the hardness of the interval subgraph isomorphism problem. This query is not only the most complex, but is rarely addressed without the use of custom preprocessing and data structures.

$$m_{1,1}+m_{2,1}+m_{3,1}+m_{4,1}+m_{5,1}+m_{6,1}=1$$
$$m_{1,2}+m_{2,2}+m_{3,2}+m_{4,2}+m_{5,2}+m_{6,2}=1$$
$$m_{1,3}+m_{1,1}+m_{3,3}+m_{4,3}+m_{5,3}+m_{4,1}=1$$
$$m_{1,1}+1.5m_{2,1}+3m_{3,1}+m_{4,1}+2m_{5,1}+1.5m_{6,1}=1$$
$$m_{1,2}+1.5m_{2,2}+3m_{3,2}+m_{4,2}+2m_{5,2}+1.5m_{6,2}=2$$
$$m_{1,3}+1.5m_{1,1}+3m_{3,3}+m_{4,3}+2m_{5,3}+1.5m_{4,1}=1.5$$
$$m_{1,1}+3m_{2,1}+4m_{3,1}+6m_{4,1}+7m_{5,1}+9m_{6,1}<m_{1,2}+3m_{2,2}+4m_{3,2}+6m_{4,2}+7m_{5,2}+9m_{6,2}$$
$$m_{1,1}+3m_{2,1}+4m_{3,1}+6m_{4,1}+7m_{5,1}+9m_{6,1}+1<m_{1,3}+3m_{2,3}+4m_{3,3}+6m_{4,3}+7m_{5,3}+9m_{6,3}$$
$$10>m_{1,1}+3m_{2,1}+4m_{3,1}+6m_{4,1}+7m_{5,1}+9m_{6,1}$$
$$\text{MIN: } m_{1,3}+3m_{1,1}+4m_{3,3}+6m_{4,3}+7m_{5,3}+9m_{4,1}-m_{1,2}-3m_{2,2}-4m_{3,2}-6m_{4,2}-7m_{5,2}-9m_{6,2}$$

Fig. 16: The 9 constraints of the IP, followed by the objective function (Equation 21).

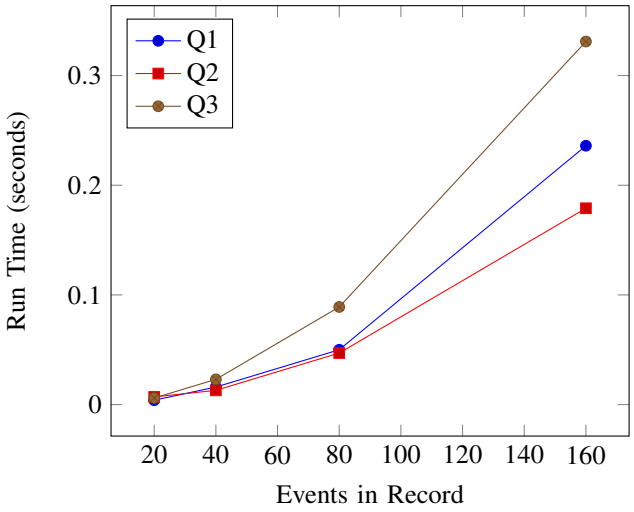

Fig. 21: Run times against matching records.

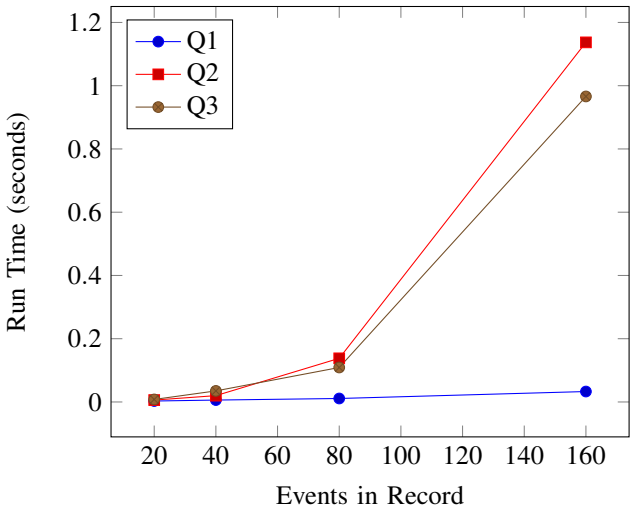

Fig. 22: Run times against non-matching records.

As would be expected, the run time of `lp_solve` scaled exponentially with all three queries as the record sizes increased. However, the testing revealed three interesting phenomenon:

1) Running the IP against matching records scaled better than running the IP against non-matching records. For Q3, the non-matching record was 3 times slower. For Q2, it was 6 times slower.
2) Q1 was the exception to this observation, indicating that the nature of the non-match may have a noticeable effect on the resulting run time.
3) The large number of constraints in Q2 had no noticeable effect on the run time. In fact, the run time for Q2 against a matching record was faster than both Q1 and Q3.

It was also observed, but not officially documented, that the objective function had a noticeable effect on the resulting run time. For consistency, the testing for all three queries employed an objective function that was dependent on the duration of some component of the matched event sequence. However, during initial testing, some of the objective functions used were dependent on the sequence location in the record (i.e. find the match closest to the end of the record). These queries appeared to have faster run times than those used in the actual testing. Our next step will be to more clearly determine the effect that both the objective function and non-matching records have on the overall run times.

Despite the exponential scaling of the IP run times, this processing strategy may still be feasible in the 20-80 event record range. It is also an appealing solution for complex queries that scale poorly across all processing strategies due to the IP's ability to integrate elements such as repetition and flexible ordering without increasing the complexity of the underlying instructions. Further testing is required to determine whether this method scales consistently to large, multi-record datasets.

VIII. Conclusion

In this paper, we demonstrated that best-match temporal pattern matching (TPM) queries can be formulated and solved as integer programs. This approach was motivated by the innate complexity of these queries, and offers tangible advantages over other approaches in that it does not require the events in the underlying record to be sorted, provides a natural ranking metric for the results, and can be both constructed and altered incrementally as the query requirements change.

Though solving integer programs is NP-Hard, we showed that certain types of temporal event queries are NP-Hard as well, and thus an IP processing strategy should not be ruled out. Though the speed of the IP processing scales exponentially as the event record sizes increase, the approach is still feasible at small record sizes. We plan next to gain a better understanding of the factors that affect the IP run times, and develop strategies for mitigating long run times on simpler queries.

ACKNOWLEDGMENT

We appreciate the help and feedback of Seth Powsner (Yale Medical School) and Bill Gasarch (University of Maryland, Computer Science). We also appreciate the partial support of the Oracle Corporation.

REFERENCES

[1] G. Navarro, "A guided tour to approximate string matching," in *ACM Computing Surveys (CSUR)*, 2001, pp. 31 – 88.

[2] R. T. Snodgrass, *Developing Time-Oriented Database Applications in SQL*. San Francisco: Morgan Kaufmann, 1999.

[3] ——, *The TSQL2 Temporal Query Language*. Kluwer Academic Pub, 1995, vol. 330.

[4] J. Dunn, S. Davey, A. Descour, and R. T. Snodgrass, "Sequenced subset operators: Definition and implementation," in *IEEE International Conference on Data Engineering (ICDE2002)*. IEEE, 2002, pp. 81–92.

[5] V. Kouramajian and M. Gertz, "A visual query editor for temporal databases," in *Proceedings of OOER'95*, 1995, pp. 388–399.

[6] S. Michael and R. Feldman, "Visual query and exploration system for temporal relational database," in *Advances in Data Mining. Theoretical Aspects and Applications*, ser. Lecture Notes in Computer Science, P. Perner, Ed. Springer Berlin / Heidelberg, 2007, vol. 4597, pp. 283–295.

[7] C. S. Jensen, J. Cliord, and S. K. G. et al., "The tsql benchmark," in *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, 1993, pp. QQ1–QQ28.

[8] R. T. Snodgrass, "The temporal query language tquel," in *ACM Transactions on Database Systems (TODS)*, vol. 12. ACM, 1987, pp. 247–298.

[9] P.-S. Kam and A. W.-C. Fu, "Discovering temporal patterns for interval-based events," in *DaWaK 2000 Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery*. Springer-Verlag, 2000, pp. 317–326.

[10] S.-Y. Wu and Y.-L. Chen, "Mining nonambiguous temporal patterns for interval-based events," in *IEEE Transactions on Knowledge and Data Engineering*, vol. 19. IEEE, 2007, pp. 742–758.

[11] D. Patel, W. Hsu, and M. L. Lee, "Mining relationships among interval-based events for classification," in *SIGMOD '08 Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 393–404.

[12] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh, "Querying and mining of time series data: experimental comparison of representations and distance measures," in *Proceedings of the VLDB Endowment*. VLDB Endowment, 2008, pp. 1542–1552.

[13] T. D. Wang, A. Deshpande, and B. Shneiderman, "A temporal pattern search algorithm for personal history event visualization," in *Discrete Mathematics*, 2012, p. 31643173.

[14] D. Patnaik, P. Butler, N. Ramakrishnan, L. Parida, B. J. Keller, and D. A. Hanauer, "Experiences with mining temporal event sequences from electronic medical records: initial successes and some challenges," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2011, pp. 360–368.

[15] J. Sun, F. Wang, J. Hu, and S. Edabollahi, "Supervised patient similarity measure of heterogeneous patient records," in *SIGKDD ACM Special Interest Group on Knowledge Discovery in Data*, vol. 14. ACM, 2012, pp. 16–24.

[16] ——, "Visual cluster analysis in support of clinical decision intelligence," in *AMIA Annual Symposium Proceedings*, 2011, p. 481490.

[17] H. Obweger, M. Suntinger, J. Schiefer, and G. Raidl, "Similarity searching in sequences of complex events," in *Proc. International Conf. on Research Challenges in Information Science (RCIS)*. IEEE, 2010, pp. 631–640.

[18] K. Wongsuphasawat, C. Plaisant, M. Taieb-Maimon, and B. Shneiderman, "Querying event sequences by exact match or similarity search: Design and empirical evaluation," in *Interacting with Computers*, vol. 24, 2012, p. 5568.

[19] M. Monroe, R. Lan, J. M. del Olmo, C. Plaisant, B. Shneiderman, and J. Millstein, "The challenges of specifying intervals and absences in temporal queries:a graphical language approach," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2013, pp. 2349–2358.

[20] R. Lan, H. Lee, A. Fong, M. Monroe, C. Plaisant, and B. Shneiderman, "Temporal search and replace: An interactive tool for the analysis of temporal event sequences," HCIL, University of Maryland, College Park, Maryland, Tech. Rep. HCIL-2013-TBD, 2013.

[21] I. S. Dániel Marx, "Cleaning interval graphs," in *Algorithmica*, vol. 65, 2013, pp. 275–316.

[22] P. Heggernes, D. Meister, and Y. Villanger, "Induced subgraph isomorphism on interval and proper interval graphs," in *Algorithms and Computation*, vol. 6507, 2010, pp. 399–409.

[23] S. Kijimaa, Y. Otachib, T. Saitohc, and T. Unod, "Subgraph isomorphism in graph classes," in *IEEE Transactions on Knowledge and Data Engineering*, vol. 312, 2012, pp. 799–812.

[24] J. F. Allen and G. Ferguson, "Actions and events in interval temporal logic," in *Journal of Logic and Computation*, vol. 4, 1994, pp. 531–579.

[25] M. Berkelaar, K. Eikland, and P. Notebaert, *lp_solve*, 5th ed., Open source (Mixed-Integer) Linear Programming system, August 2010.