Temporal Regular Path Queries: Syntax, Semantics, and Complexity

Marcelo Arenas
Universidad Católica & IMFD, Chile
marenas@ing.puc.cl

Pedro Bahamondes
Universidad Católica & IMFD, Chile
pibahamondes@uc.cl

Julia Stoyanovich
New York University, USA
stoyanovich@nyu.edu

ABSTRACT

In the last decade, substantial progress has been made towards standardizing the syntax of graph query languages, and towards understanding their semantics and complexity of evaluation. In this paper, we consider temporal property graphs (TPGs) and propose temporal regular path queries (TRPQ) that incorporate time into TPG navigation. Starting with design principles, we propose a natural syntactic extension of the MATCH clause of popular graph query languages. We then formally present the semantics of TRPQs, and study the complexity of their evaluation. We show that TRPQs can be evaluated in polynomial time if TPGs are time-stamped with time points. We also identify fragments of the TRPQ language that admit efficient evaluation over a more succinct interval-annotated representation. Our work on the syntax, and the positive complexity results, pave the way to implementations of TRPQs that are both usable and practical.

1 INTRODUCTION

The importance of networks in scientific and commercial domains is undeniable. Networks are represented by graphs, and we will use the terms *network* and *graph* interchangeably. Considerable research and engineering effort aims to develop effective and efficient graph representations and query languages. Property graphs have emerged as the de facto standard, and have been studied extensively, with efforts underway to unify the semantics of query languages for these graphs [3, 4].

Many interesting questions about graphs are related to their evolution rather than to their static state. Some areas where *temporal graphs* are being studied are social networks [28, 38, 39, 49], biological networks [6, 9, 54], and the Web [17, 45]. Consequently, several recent proposals have been made to extend representation and querying of property graphs with time. Examples of temporal graph query languages include [15, 21, 31, 36, 43]; we will discuss how these lines of work relate to our approach in Section 2.

Our focus in this paper is on incorporating time into path queries. More precisely, we (a) outline the design principles for a temporal extension of Regular Path Queries (RPQs) with time; (b) propose a natural syntactic extension of Cypher [26], a state of the art query language for conventional (non-temporal) property graphs, that supports temporal RPQs (TRPQs); (c) formally present the semantics of this language; and (d) study the complexity of evaluation of several variants of this language. We show that, by adhering to the design principles that draw on decades of work on graph databases and on temporal relational databases, we are able to achieve polynomial-time complexity of evaluation, paving the way to implementations that are both usable and practical.

1.1 Running example

As a preview of our proposed methods, consider Figure 1 that depicts a contact tracing network for a communicable disease with airborne transmission between people in enclosed locations, such as a cafe or a bus. In this network, different actors and their interactions are presented as a *temporal property graph* or TPG for short. (We will define temporal property graphs formally in Section 3).

As in conventional property graphs [4], nodes and edges in a TPG are labeled. The graph in Figure 1 contains two types of nodes, `Person` and `Bus`, and three types of edges: bi-directional edges `meets` and `cohabits` (lives together), and directed edge `rides`. Nodes and edges have optional properties that are associated with values. For example, node n_1 of type `Person` has properties `name` with value `'Ann'` and `risk` with value `'low'`. As another example, edge e_2 of type `meets` has property `loc` with value `'park'`.

The purpose of the graph in Figure 1 is to allow identification of individuals who may have been exposed to the disease. In particular, we are interested in identifying potentially infected individuals who are considered high risk, due to age or pre-existing conditions. These types of questions can be naturally phrased as *temporal regular path queries* (TRPQs) that interrogate reachability over time. We will give an example of a TRPQ momentarily.

To support TRPQs, all nodes and edges in a TPG are associated with *time intervals of validity* (or *intervals* for short) that represent consecutive time points during which no change occurred for a node or an edge, in terms of its existence or property values. For example, node n_1 (Ann) is associated with the interval `[1, 9]`, signifying that this node was present in the graph and took on the specified property values during nine consecutive time points. We will discuss in Section 5 that intervals are used as a representation device, to compactly denote a finite set of time points they contain. As another example, node n_2, a person named Bob, exists during the same interval as n_1, but undergoes a change in the value of the property `risk` at time 4, when it changes from `'low'` to `'high'`. We represent a change in the state of an entity (a node or an edge) with nested boxes inside an outer box that denotes the entity.

Now, consider an example of a TRPQ that extends the syntax of Cypher (in a way that will be made precise in Section 4), to retrieve the list of high-risk people (`x`) who met someone (`y`), who subsequently tested positive for an infectious disease:

```
MATCH (x:Person {risk = 'high'})-
   /FWD/:meets/FWD/NEXT*/-
   (y:Person {test = 'pos'})
ON contact_tracing
```

This contact tracing query produces the following *temporal binding table* when evaluated over the TPG in Figure 1:

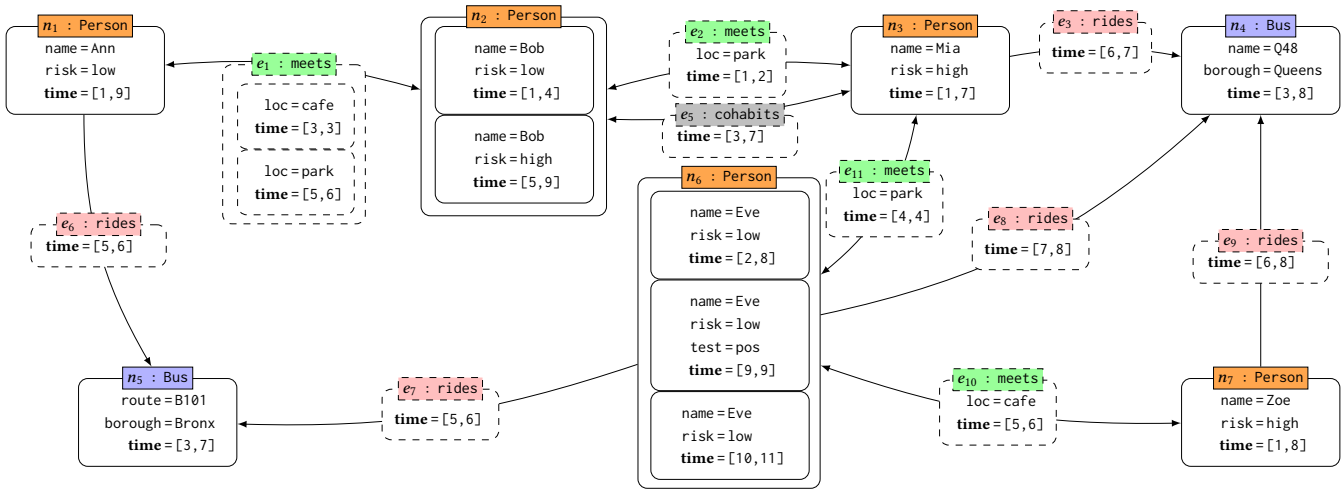

Figure 1: A TPG used for contact tracing. The graph contains two types of nodes, `Person` and `Bus`, and three types of edges: bi-directional edges `meets` and `cohabits`, and directed edge `rides`. Person nodes have properties `name`, `risk` (`'high'` or `'low'`) of complications, and `test` (`'pos'` or `'neg'`) of disease status. Eve (node n_6) tested positive for a communicable disease at time 9.

x	x_time	y	y_time
n_6	9	n_7	5
n_6	9	n_7	6
n_6	9	n_3	4

1.2 Summary of our approach

In the remainder of this paper, we formally develop the concepts that are necessary to evaluate this and other useful TRPQs over TPGs. We adopt a conceptual TPG model that naturally extends property graphs with time, and is both simple and sufficiently flexible to support the evolution of graph topology and of the properties of its nodes and edges. We use this model to evaluate TRPQs under *point-based semantics* [12], in which operators adhere to two principles: snapshot reducibility and extended snapshot reducibility, discussed in Section 2. Our conceptual TPG model admits two logical representations, both associating temporal information with graph objects but differing in the kind of time-stamping they use [44]. One associates objects with time points, while the other associates them with time intervals, for a more compact representation.

Design principles. We carefully designed our TRPQ language based on several principles, outlined below. By adhering to these principles, we achieved polynomial-time complexity of evaluation for TPGs that are time-stamped with time points. Further, we identified a significant fragment of the language that can be efficiently evaluated for interval time-stamped TPGs. These principles are:

Navigability: Include operators that refer to the dynamics of navigating through the TPG. Temporal navigation operators refer to movements on the graph over time, and spatial navigation operators refer to movements across locations in its topology.

Navigation orthonormality: Temporal and spatial navigation operators must be orthogonal in the query language, allowing non-simultaneous single-step time and space movement.

Node-edge symmetry: Nodes and edges should be treated as first-class citizens of the query language, allowing equivalent representation and operations.

Static testability: Testing must be independent of navigation.

Snapshot reducibility: When time is removed from a query, pairs of temporal objects satisfying the query should correspond to pairs of objects in a single snapshot, or temporal state, of the graph, and every pair satisfying the query in the snapshot of the TPG should correspond to a path satisfying it in the temporal graph.

Paper organization. In what follows, we first give some background on temporal graph models and path query languages in Section 2. We then formally define temporal graphs, in Section 3. We go on to propose a syntax for adding time to a practical graph query language in Section 4. Next, in Section 5, we give precise semantics of the language, and we show polynomial-time complexity of evaluation if TPGs are time-stamped with time points. In Section 6, we consider the case in which intervals are used to represent time compactly, and provide an analysis of the complexity of evaluation in this scenario. Most notably, we identify a significant fragment of the language that can be efficiently evaluated even for interval time-stamped TPGs. We conclude in Section 7.

2 BACKGROUND AND RELATED WORK

Substantial research has been undertaken in the area of *temporal relational databases* since the 1980s, producing a significant body of work, so much so that a large portion of the definitive Encyclopedia of Database Systems [41] is dedicated to temporal topics. This work includes representation of time [19, 30, 51], semantics of temporal models [11], temporal algebras [22], and access methods [48]. Results of some of this work are part of the SQL:2011 standard [35].

Temporal graph models. Temporal graph models differ in what temporal semantics they encode, what time representation they use (time point, interval, or implicitly with a sequence), what entities

they time-stamp (graphs, nodes, edges, or attribute-value assignments), and whether they represent evolution of topology only or also of the attributes. With a few exceptions, discussed next, the current de facto standard representation of temporal graphs is the *snapshot sequence*, where a state of a graph is associated with either a time point or an interval during which the graph was in that state [14, 24, 25, 32–34, 37, 46, 50, 52, 60]. This simple representation supports operations such as computing answers to subgraph or reachability queries within each snapshot. This processing adheres to the principle of *snapshot reducibility*, which states that applying a temporal operator to a database is equivalent to applying the non-temporal variant of the operator to each database state [12].

For example, the G* system by Labouseur et al. [36] stores a temporal graph as a snapshot sequence and provides two query languages, the procedural PGQL and the declarative DGQL. PGQL includes operators such as retrieving graph vertices and their edges at a given time point, along with non-graph operators like aggregation, union, projection, and join. Neither PGQL nor DGQL support temporal path queries, which are the focus of our work.

The fundamental disadvantage of using the snapshot sequence as the conceptual representation of a temporal graph is that it does not support operations that explicitly reference temporal information associated with nodes, edges, and their attributes. Semantics of operations that make explicit references to time are formalized in the temporal relational literature as the principle of *extended snapshot reducibility*, where timestamps are made available to operators by propagating time as data [12], allowing explicit references to time in predicates. Considering that, our goal in this work is to support temporal regular path queries, having access to temporal information during navigation is crucial.

In response to this important limitation of the snapshot sequence representation, proposals have been made to annotate graph nodes, edges, or attributes with time. Moffitt and Stoyanovich [43] proposed to model the evolution of a graph's topology, and of the attributes of its nodes and edges, by associating periods of validity —in the form of temporal intervals— with graph nodes, edges, and property values. They also developed a compositional temporal graph algebra that provides a temporal generalization of common graph operations including subgraph, node creation, union, and join, but does not include reachability or path constructs. In our work, we adopt a similar representation of temporal graphs, but focus on temporal regular path queries.

Paths in temporal graphs. Specific kinds of path queries over temporal graphs have been considered in the literature. Wu et al. [57–59] studied path query variants over temporal graphs, in which nodes are time-invariant and edges are associated with a starting time and an ending time. (Nodes and edges do not have type labels or attributes.) The authors introduced four types of "minimum temporal path" queries, including the earliest-arriving path and the fastest path, which can be seen as generalizations of the shortest path query for temporal graphs. They proposed algorithms and indexing methods to process minimum temporal path and temporal reachability queries efficiently.

Byun et al. [15] introduced ChronoGraph, a temporal graph traversal system in which edges are traversed in adherence with the "temporal constraint", time-forward. The authors show three use cases for their approach: temporal breadth-first search, temporal depth-first search, and temporal single-source shortest-path, instantiated over Apache Tinkerpop.

Johnson et al. [31] introduced Nepal, a query language that has SQL-like syntax and supports regular path queries over temporal multi-layer communication networks, represented by temporal graphs that associate a sequence of intervals of validity with each node and edge. The key novelty of this work is its support for time-travel path queries to retrieve past network states, which is practically important for their management.

Finally, Debrouvier et al. [21] introduced T-GQL, a query language for TPGs with Cypher-like syntax. T-GQL operates over graphs in which (a) nodes persists but their attributes (with values) can change over time, and so are associated with periods of validity; and (b) edges are associated with periods of validity but their attributes are time-invariant. This asymmetry in the handling of nodes and edges is due to the authors' commitment to a specific (lower-level) representation of such TPGs in a conventional property graph system. Specifically, they assume that Objects (representing nodes), Attributes, and Values are stored as conventional property graph nodes, whereas time intervals are stored as properties of these nodes. Temporal edges are, in turn, stored as conventional edges, with time interval as one of their properties. T-GQL supports three types of path queries over such graphs, syntactically specified with the help of named functions: (1) "Continuous path" queries retrieve paths valid during each time point—snapshot semantics. (2)"Pairwise continuous paths" require that the incoming and the outgoing edge for a node being traversed must exist during some overlapping time period. (3) "Consecutive paths" encode earliest arrival, latest departure, fastest, and shortest path queries.

In summary, our proposal differs from prior work in that we develop a general-purpose query language for temporal paths, which works over a simple conceptual definition of TPGs that is nonetheless general enough to represent different kinds of temporal and structural evolution of property graphs. Our language is syntactically simple: it directly, and minimally, extends the `MATCH` clause of popular query languages, and does not rely on custom functions.

3 A TEMPORAL GRAPH MODEL

In this section, we formalize the notion of temporal property graph, which extends the widely used notion of property graph [3, 4, 26] to include explicit access to time. In this way, we can model the evolution of the topology of a property graph, as well as the changes in node and edge properties.

From now on, we assume *Lab*, *Prop* and *Val* to be infinite sets of label names, property names and actual values, respectively. We define temporal property graphs over finite sets of time points. Time points can take on values that correspond to the units of time as appropriate for the application domain, and may represent seconds, weeks, or years. For the sake of presentation, we will represent the universe of time points by $\mathbb{N}$. More precisely, a temporal domain Ω is a finite set of consecutive natural numbers, that is, $\Omega = \{i \in \mathbb{N} \mid a \leq i \leq b\}$ for some $a, b \in \mathbb{N}$ such that $a \leq b$.

Definition 3.1. A temporal property graph (TPG) is a tuple $G = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$, where

- Ω is a temporal domain;

- N is a finite set of *nodes*, E is a finite set of *edges*, and $V \cap E = \emptyset$;
- $\rho : E \to (N \times N)$ is a function that maps an edge to its source and destination nodes;
- $\lambda : (N \cup E) \to Lab$ is a function that maps a node or an edge to its label;
- $\xi : (N \cup E) \times \Omega \to \{\text{true}, \text{false}\}$ is a function that maps a node or an edge, and a time point to a Boolean. Moreover, if $\xi(e, t) = \text{true}$ and $\rho(e) = (v_1, v_2)$, then $\xi(v_1, t) = \text{true}$ and $\xi(v_2, t) = \text{true}$.
- $\sigma : (N \cup E) \times Prop \times \Omega \to Val$ is a partial function that maps a node or an edge, a property name, and a time point to a value. Moreover, there exists a finite number of triples $(o, p, t) \in (N \cup E) \times Prop \times \Omega$ such that $\sigma(o, p, t)$ is defined, and if $\sigma(o, p, t)$ is defined, then $\xi(o, t) = \text{true}$. □

Observe that Ω in Definition 3.1 denotes the *temporal domain* of G, a finite set of linearly ordered time points starting from the time associated with the earliest *snapshot* of G, and ending with the time associated with its latest snapshot, where a snapshot of G refers to a conventional (non-temporal) property graph that represents the state of G at a given time point.

Function ρ in Definition 3.1 is used to provide the starting and ending nodes of an edge. In what follows, if $\rho(e) = (v_1, v_2)$, then we use notation $\text{src}(e) = v_1$ and $\text{tgt}(e) = v_2$. Moreover, function λ provides the label of a node or an edge, while function ξ indicates whether a node or an edge exists at a given time point in $\mathbb{N}$, which is represented by the value true. Finally, function σ indicates the value of a property for a node or an edge at a given time point in Ω.

Two extra conditions are imposed on TPGs to enforce that they conceptually correspond to sequences of valid conventional property graphs. In particular, an edge can only exist at a time when both of the nodes it connects exist, and that a property can only take on a value at a time when the corresponding object exists.

Moreover, observe that by imposing that $\sigma(o, p, t)$ is defined for a *finite* number of triples (o, p, t), we are ensuring that each node or edge can have values for a finite number of properties. Thus, we have that each TPG has a finite representation. Finally, Definition 3.1 assumes, for simplicity, but without loss of generality, that property values are drawn from the infinite set *Val*. That is, we do not distinguish between different data types. If a distinction between data types is necessary, then *Val* can be replaced by a domain of values of some k different data types, $Val_1, \ldots, Val_k$.

Recall our running example discussed in Section 1.1 and shown in Figure 1. This example illustrates Definition 3.1; it shows a TPG used for contact tracing for a communicable disease, with airborne transmission between people (represented by nodes with label `Person`) in enclosed locations (*e.g.* nodes with label `Bus`). This TPG has a temporal domain $\Omega = \{1, \ldots, 11\}$, although any set of consecutive natural numbers containing Ω can serve as the temporal domain of this TPG, for example the set $\{0, \ldots, 12\}$.

In the TPG in Figure 1, `Person` nodes have properties name, risk (`'low'` or `'high'`), and test (`'pos'` or `'neg'`). For example, Eve, represented by node n_6, is known to have tested positive for the disease at time 9. Note that each node and edge refers to a specific time-invariant real-life object or event. These real-life objects in turn correspond to a sequence of temporal objects, each with a set of properties. For instance, node n_2 corresponds to a sequence of 9 temporal objects, one for each time point 1 through 9. These are represented in the figure by two boxes inside the outer box for n_2, one for each interval during which no change occurred: $[1, 4]$ with name Bob, and low risk, and $[5, 9]$ with name Bob, and high risk. To simplify the figure, we do not show internal boxes for nodes or edges associated to a single time interval, such as n_1 and e_6.

From now on, given a TPG $G = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$, a pair (o, t), such that $o \in (N \cup E)$ and $t \in \Omega$, is called a *temporal object* in G.

4 ADDING TIME TO A PRACTICAL GRAPH QUERY LANGUAGE

The main goal of this paper is to introduce a simple yet general query language for temporal property graphs, where queries like the ones shown in the previous section can be easily expressed. This language must have a precise syntax and semantics, to allow an unambiguous evaluation and a formal study of the complexity of evaluating its different components. These tasks will be the focus of the following sections. In this section, we concentrate on the intuition of how it works, and how to incorporate it into current proposals for graph query languages. For this purpose, we provide the reader with a guided tour of the query language through the examples shown in the previous section.

The `MATCH` clause is a fundamental construct in popular graph query languages such as Cypher [26], PGQL [55], and G-Core [3]. By using graph patterns, the `MATCH` clause allows to bind variables with objects in a property graph, giving rise to *binding tables* that are subsequently processed by the other components of the query languages. As an important step towards the construction of a temporal graph query language, we show how the `MATCH` clause can be extended to bind variables with temporal objects in a TPG. In particular, we show how the syntax and semantics of the query language G-Core [3] can be extended to accommodate temporal graph patterns. As the syntax and semantics of G-Core are compatible with those of Cypher [26] and PGQL [55], these languages can accommodate such temporal graph patterns as well. Moreover, as these query languages play a fundamental role in the current standardization effort for a graph query language [5], our proposal is appropriate to provide a natural temporal extension for this standard.

In what follows, assume that `contact_tracing` is the temporal property graph shown in Figure 1. Then the following G-Core expression extracts the list of people from `contact_tracing`:

```
MATCH (x:Person) ON contact_tracing
```

The operator `ON` specifies that `contact_tracing` is the input graph, and `(x:Person)` indicates that `x` is a variable to be assigned nodes with label `Person` from the input graph. The evaluation of a `MATCH` clause in G-Core results in a table consisting of bindings that assign to each variable an object from the input graph: a node, an edge, a label or a property value. The following binding table is the result of evaluating the previous `MATCH` clause:

x
n_1
n_2
n_3
n_6
n_7

At this point, two observations should be made: (i) G-Core does not consider `contact_tracing` as a temporal property graph, so no explicit time is associated with the objects in a binding table; (ii) Cypher [26] and PGQL [55] produce the same bindings as G-Core when evaluating the previous `MATCH` clause. How should this clause be evaluated if `contact_tracing` is considered as a temporal property graph? The first issue to be taken into account is that variables in the `MATCH` clause are to be assigned temporal objects; for example, `(x:Person)` indicates that `x` is a variable to be assigned a temporal object (v, t), where v is a node with label `Person` that exists at time point t. This issue is addressed by adding an extra column for each variable to indicate the time point when that variable exists (table entries appear side-by-side to save vertical space):

x	x_time
n_1	1
...	
n_1	9

x	x_time
n_2	1
...	
n_7	8

Observe that the time point t for each value v of `x` is stored in the column `x_time`. Hence, the binding `x` $\mapsto n_1$, `x_time` $\mapsto 1$ is in the resulting table, since n_1 is a node with label `Person` that exists at time point 1 in the temporal property graph `contact_tracing`, and similarly for the other bindings in the table.

Having explained how bindings to temporal objects are represented, we can now give an idea of the main features of our query language. As in other popular graph query languages, we use curly brackets to indicate restrictions on property values. As our first example, consider the following `MATCH` clause:

```
MATCH (x:Person {risk = 'low'})
ON contact_tracing
```

The expression `{risk = 'low'}` is used to indicate that the value of property `risk` must be `'low'`. The following binding table is the result of evaluating the previous `MATCH` clause:

x	x_time
n_1	1
...	
n_1	9

x	x_time
n_2	1
...	
n_2	4

x	x_time
n_6	2
...	
n_6	9

Observe that the binding `x` $\mapsto n_2$, `x_time` $\mapsto 4$ is in this table, since n_2 is a node such that the label of n_2 is `Person`, n_2 exists at time point 4, and the value of property `risk` is `'low'` for n_2 at time point 4, and likewise for the other bindings in this table. As a second example, consider the following `MATCH` clause:

```
MATCH (x:Person {risk = 'low' AND time = '1'})
ON contact_tracing
```

In this case, we use the reserved word `time` to indicate that we are considering temporal objects at time point 1. The following is the result of evaluating this `MATCH` clause:

x	x_time
n_1	1
n_2	1

Other operators can be used to limit the time under consideration, for example, to considering temporal objects at time less than 10:

```
MATCH (x:Person {risk = 'low' AND time < '10'})
ON contact_tracing
```

Now, suppose that we want to retrieve the pairs of low- and high-risk people who have met, along with information about their meeting. The following `MATCH` clause can be used for this purpose:

```
MATCH (x:Person {risk = 'low'})-
   [z:meets]->(y:Person {risk = 'high'})
ON contact_tracing
```

The result of evaluating this `MATCH` clause is:

x	x_time	z	z_time	y	y_time
n_1	5	e_1	5	n_2	5
n_1	6	e_1	6	n_2	6
n_2	1	e_2	1	n_3	1
n_2	2	e_2	2	n_3	2

As in other popular graph query languages [3, 26, 55], an expression of the form `-[:meets]->` indicates the existence of an edge with label `meets`. We assign the variable `z` to the temporal object that represents that edge.

Importantly, an expression of the form `-[...]->` represents the spatial navigation operator that is conceptually evaluated over the snapshots (temporal states) of the graph. This is the reason why each binding in the resulting table has the same value in columns `x_time`, `z_time`, and `y_time` . For example, the binding `x` $\mapsto n_1$, `x_time` $\mapsto 5$, `z` $\mapsto e_1$, `z_time` $\mapsto 5$, `y` $\mapsto n_2$, `y_time` $\mapsto 5$ is in this table, since n_1 is a low-risk person at time point 5, n_2 is a high-risk person at time point 5, and there exists an edge e_1 with label `meets` between n_1 and n_2 at time point 5.

To ensure that our proposal is practically useful, a minimum requirement is that queries can be evaluated in polynomial time over TPGs. Hence, we have to choose very carefully how spatial navigation is combined with temporal navigation, and how we refer to time in the query language, as the complexity can quickly become intractable when navigation patterns are combined with functionalities for comparing property values [40]. In fact, this negative result holds even for some fixed queries, that is, when only graphs are considered as input of the query evaluation problem [56].

The basic temporal navigation operators in our language are `PREV` and `NEXT` that move by one unit of time into the past and into the future, respectively. Consider the following `MATCH` clause:

```
MATCH (x:Person {test = 'pos'})-
   /PREV/-(y:Person)
ON contact_tracing
```

Here, `x` and `y` are temporal objects that correspond to *the same* real-world object —a node of type `Person`. In this case, `x` has the value `'pos'` in the property `test`, meaning that `x` received a positive test result at some time point, and `y` denotes the same node at the time immediately before they received their positive test result.

This example illustrates the use of notation `-/.../-` to specify a pattern that a path connecting objects `x` and `y` must satisfy. In general, such a pattern is a regular expression that can include temporal and spatial operators (see formal definition in Section 5). In this example, assuming that the temporal object (o_1, t_2) corresponds to `(x:Person {test = 'pos'})`, and the temporal object (o_2, t_2) corresponds to `(y:Person)`, then the expression `-/PREV/-` indicates that (o_1, t_1) must be connected with (o_2, t_2) through a path conforming to `PREV`, that is, $t_2 = t_1 - 1$. Importantly, `-/PREV/-` is evaluated under

the restriction that no spatial navigation must have occurred, given the separation between temporal and spatial navigation that we are arguing for in this work. Hence, we conclude that $o_2 = o_1$. The following binding table is the result of evaluating the `MATCH` clause:

x	x_time	y	y_time
n_6	9	n_6	8

Temporal and spatial navigation can be combined to retrieve information about which bus person `x` was riding immediately before she received a positive test result:

```
MATCH (x:Person {test = 'pos'})-
   /PREV/-(y:Person)-[:rides]->(z:Bus)
ON contact_tracing
```

The result of evaluating this `MATCH` clause is:

x	x_time	y	y_time	z	z_time
n_6	9	n_6	8	n_4	8

Observe that the temporal operator `PREV` is used to move from (`x`, `x_time`) to (`y`, `y_time`), while the spatial operator`-[:rides]->` is used to move from (`y`, `y_time`) to (`z`, `z_time`). Hence, temporal and spatial navigation are carried out separately. Besides, observe that the intermediate variable `y` is not needed when retrieving the list of buses that person `x` was riding, we just included it to show the paths that are constructed when using the different operators in the `MATCH` clause. In fact, the following simplified `MATCH` clause

```
MATCH (x:Person {test = 'pos'})-
   /PREV/-()-[:rides]->(z:Bus)
ON contact_tracing
```

can be used to obtain the desired answer:

x	x_time	z	z_time
n_6	9	n_4	8

At this point the reader may be wondering why the language is asymmetric, and it includes different notation for temporal and spatial navigation. We have kept the notation `-[...]->` to be compatible with graph query languages used today [3, 26, 55], but an important feature of our proposal is the use of notation `-/.../-` to include regular expressions combining temporal and spatial operators. Hence, we include two basic spatial navigation operators, `BWD` ("backward") and `FWD` ("forward"), that are analogous to the temporal operators `PREV` and `NEXT`. Assume that an edge is given

$$(n, t) \xrightarrow{(e,t)} (n', t), \tag{1}$$

which, in the formal TPGs notation (see Definition 3.1), represents the fact that $\rho(e) = (n, n')$, $\xi(n, t) = \text{true}$, $\xi(e, t) = \text{true}$, and $\xi(n', t) = \text{true}$. Then, operator `FWD` moves forward from node n to edge e, or from edge e to node n', while keeping time t unchanged. That is, `FWD` operates in a TPG snapshot corresponding to time t. Similarly, operator `BWD` moves backwards from node n' to edge e, and from edge e to node n in a TPG snapshot corresponding to time t. Thus, we can rewrite the previous `MATCH` clause as follows:

```
MATCH (x:Person {test = 'pos'})-
   /PREV/FWD/:rides/FWD/-(z:Bus)
ON contact_tracing
```

The regular expression `PREV/FWD/:meets/FWD` uses the concatenation operator `/` to indicate that operator `PREV` has to be executed first followed by the expression `FWD/:rides/FWD`, which is executed in the same way. (The precise syntax and semantics of such expressions are presented in Section 5.) Observe that in our query language, the expression`-[:rides]->` is equivalent to `-/FWD/:rides/FWD/-`. This is because, given an edge of the form of (1), the first operator `FWD` is used to move from n to e, then `:rides` is used to check that the label of e is `rides`, and finally the last operator `FWD` is used to move from e to n', thus obtaining the same result as using the operator `-[:rides]->` in an edge of the form (1).

So far we only looked at expressions that navigate one step at a time, temporally or spatially. Our language also supports the Kleene star, indicating zero or more occurrences of an operator. For example, the following expression retrieves the list of buses person `x` rode at any time prior to receiving a positive test (including also at the time when `x` received the test):

```
MATCH (x:Person {test = 'pos'})-
   /PREV*/FWD/:rides/FWD/-(z:Bus)
ON contact_tracing
```

producing the following temporal bindings:

x	x_time	z	z_time
n_6	9	n_4	8
n_6	9	n_4	7
n_6	9	n_5	6
n_6	9	n_5	5

As another example, we can retrieve the high-risk people who met someone who subsequently tested positive for an infectious disease:

```
MATCH (x:Person {risk = 'high'})-
   /FWD/:meets/FWD/NEXT*/-({test = 'pos'})
ON contact_tracing
```

Recall that the temporal operator `NEXT` allows to move in time by one unit into the future. This query returns the following temporal bindings when evaluated over the graph in Figure 1:

x	x_time
n_3	4
n_7	5
n_7	6

Observe that the term `({test = 'pos'})` does not include a variable, as we are not storing the contacts who tested positive to avoid stigmatizing them, and only record those who are potentially at risk for complications.

Moreover, our query language allows to specify the number of times an operator is used. Thus, assuming that the time unit in `contact_tracing` is an hour, we can retrieve the list of high-risk people who met someone who tested positive for an infectious disease in two weeks prior to the meeting:

```
MATCH (x:Person {risk = 'high'})-
   /FWD/:meets/FWD/PREV[0,336]-({test = 'pos'})
ON contact_tracing
```

Assume now that we need to consider the following notion of close contact for an infectious disease. If person a rides a bus with person b, and b is tested positive for this disease at most two weeks after they ride together, then a is considered to have been in close contact with an infected person. The following `MATCH` clause is

used to retrieve the list of high-risk people who have been in close contact with an infected person:

```
MATCH (x:Person {risk = 'high'})-
   /FWD/:rides/FWD/:Bus/BWD/:rides/
   BWD/NEXT[0,336]/-({test = 'pos'})
ON contact_tracing
```

Observe that, as was the case for edge labels, node labels can be used inside an expression `-/.../-`, and so `-/:Bus/-` in the expression above is equivalent to `-(:Bus)-`. The following is the binding table resulting from the evaluation of this `MATCH` clause:

x	x_time
n_3	7
n_7	7
n_7	8

As the final example, assume that if person a meets with person b, and b is tested positive for an infections disease at most two weeks after their meeting, then a should also be considered to have been in close contact with an infected person. Then the previous `MATCH` clause can be extended to consider this additional case:

```
MATCH (x:Person {risk = 'high'})-
    /(FWD/:meets/FWD/NEXT[0,336]) + (FWD/:rides
      /FWD/:Bus/BWD/:rides/BWD/NEXT[0,336]
    )/-({test = 'pos'})
ON contact_tracing
```

This query produces the following bindings:

x	x_time	x	x_time
n_3	4	n_7	6
n_3	7	n_7	7
n_7	5	n_7	8

As usual in regular expressions, operator `+` is used to represent union. Thus, the regular expression in the previous `MATCH` clause indicates that the results of `FWD/:meets/FWD/NEXT[0,336]` should be put together with the results of `FWD/:rides/FWD/:Bus/BWD/:rides/BWD/NEXT[0,336]`. Observe that parentheses are used to have unambiguous expressions that can be parsed in a unique way. For example, the previous expression can be rewritten as follows to avoid using the temporal operator `NEXT[0,336]` twice. (Observe the required use of parentheses to get the desired effect.)

```
MATCH (x:Person {risk = 'high'})-
    /((FWD/:meets/FWD) + (
      FWD/:rides/FWD/:Bus/BWD/:rides/BWD
    ))/NEXT[0,336]/-({test = 'pos'})
ON contact_tracing
```

In this section, we illustrated the main features of our proposed language and showed how popular graph query languages [3, 26, 55] can be extended to include these features. We will define the syntax and the semantics of our language in the next section.

5 TEMPORAL REGULAR PATH QUERIES

Our proposed syntax for temporal regular path queries can be summarized as the following extension of the `MATCH` clause:

```
MATCH (x)-/path/-(y) ON graph
```

Here, `graph` is a TPG, and `path` is an expression that can contain temporal and spatial navigation operators, together with some other functionalities like testing the label of a node or an edge, and verifying the value of a property of a node or an edge.

In this section, we provide a formal syntax and semantics for the expression `path` described above, and study the complexity of evaluating such an expression. More precisely, in Section 5.1 we extend the widely used notion of regular path query [1, 4, 7, 16] to deal with temporal objects in TPGs. In particular, the semantics of such expressions is defined by following the semantics of widely used query languages such as XPath and regular path queries [1, 4, 7, 16, 18, 29, 42, 47]. The query language defined in Section 5.1 is called NavL[PC,NOI], which is a shorthand for Navigational Language (NavL) with path conditions (PC) and numerical occurrence indicators (NOI). Then, in Section 5.2, we show how NavL[PC,NOI] provides a formalization and an extension of the practical query language proposed in Section 4. Finally, in Section 5.3, we study the complexity of the query evaluation problem for NavL[PC,NOI].

5.1 Syntax and semantics of NavL[PC, NOI]

In this section, assume that $G = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$ is a TPG, and recall that labels, property names, and property values are drawn from the sets *Lab*, *Prop*, and *Val*, respectively. Then the expressions in NavL[PC,NOI], which are called temporal regular path queries (TRPQs), are defined by the following grammar:

$$\begin{aligned} \text{path} ::= \text{test} \mid \text{axis} \mid (\text{path}/\text{path}) \mid (\text{path} + \text{path}) \mid \\ \text{path}[n, m] \mid \text{path}[n, \_] \end{aligned} \tag{2}$$

where n and m are natural numbers such that $n \leq m$. Intuitively, test checks a condition on a given node or edge at a given time point, axis allows spatial or temporal navigation, $(\text{path}/\text{path})$ is used for the concatenation of two TRPQs, $(\text{path} + \text{path})$ allows for the disjunction of two TRPQs, $\text{path}[n, m]$ allows path to be repeated a number of times that is between n and m, whereas $\text{path}[n, \_]$ only imposes a lower bound of at least n repetitions of expression path. The Kleene star path^* can be expressed as $\text{path}[0, \_]$, and the expression $\text{path}[\_, n]$ is equivalent to $\text{path}[0, n]$.

Conditions on temporal objects are defined by the grammar:

$$\begin{aligned} \text{test} ::= \textbf{Node} \mid \textbf{Edge} \mid \ell \mid p \mapsto v \mid < k \mid \exists \mid \\ (?\text{path}) \mid (\text{test} \vee \text{test}) \mid (\text{test} \wedge \text{test}) \mid (\neg \text{test}) \end{aligned} \tag{3}$$

where $\ell \in Lab$, $p \in Prop$, $v \in Val$, and $k \in \mathbb{N}$. Intuitively, test is meant to be applied to a temporal object, (*i.e.*, a pair (o, t) with object o and time point t). **Node** and **Edge** are used to test whether the object is a node or an edge, respectively; the term ℓ checks whether the label of the object is ℓ; the term $p \mapsto v$ checks whether the value of property p is v for the object at the given time point; $\exists$ checks whether the object exists at the given time point; and $< k$ checks whether the current time point is less than k. Additionally, we allow test to be $(?\text{path})$, where path is an expression satisfying grammar (2), which means that there is a path starting on the tested temporal object that satisfies the TRPQ path. Finally, test can be a disjunction or a conjuction of a pair of test expressions, or a negation of a test expression.

Furthermore, the following grammar defines *navigation*:

$$\text{axis} ::= \textbf{F} \mid \textbf{B} \mid \textbf{N} \mid \textbf{P} \tag{4}$$

Operators **F**, **B** are used to move spatially in a TPG: **F** moves forward by following the direction of an edge, while **B** moves backward by following the reverse direction of an edge. Analogously, operators **N**, **P** are used to move temporally in a TPG: **N** moves to the next time point, while **P** moves to the previous time point.

We now define the semantics of NavL[PC,NOI]. More precisely, the evaluation of a TRPQ path in NavL[PC,NOI] over the TPG G is defined by the set of tuples (o, t, o', t') such that there exists a sequence of temporal objects starting in (o, t), ending in (o', t'), and conforming to path. Note that notation (o, t, o', t') is used to represent a pair of temporal objects, namely (o, t) and (o', t'). We use this simplified notation because it reduces the number of parenthesis and represents the notion of binding table used in Section 4. Moreover, an expression test defined according to grammar (3) is also considered as a navigation expression, but that allows to stay in the same temporal object if test is satisfied. In what follows, we formalize these ideas assuming that $\mathrm{PTO}(G)$ is the set of pairs of temporal objects in G, that is, $\mathrm{PTO}(G) = (N \cup E) \times \Omega \times (N \cup E) \times \Omega$.

Let test be an expression defined according to grammar (3). To define the semantics of such a condition, recall that $G = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$, and consider a temporal object (o, t) in G, that is, $o \in N \cup E$ and $t \in \Omega$. Then, the satisfaction of condition test by (o, t), denoted by $(o, t) \models \text{test}$, is recursively defined as follows (omitting the usual semantics for Boolean connectives):

- If test = **Node**, then $(o, t) \models \text{test}$ if $o \in N$;
- If test = **Edge**, then $(o, t) \models \text{test}$ if $o \in E$;
- If test = ℓ, with $\ell \in Lab$, then $(o, t) \models \text{test}$ if $\lambda(o) = \ell$;
- If test = $p \mapsto v$, with $p \in Prop$ and $v \in Val$, then $(o, t) \models \text{test}$ if $\sigma(o, p, t)$ is defined and $\sigma(o, p, t) = v$;
- If test = $\exists$, then $(o, t) \models \text{test}$ if $\xi(o, t) = \text{true}$;
- If test = $< k$, then $(o, t) \models \text{test}$ if $t < k$;
- If test = (?path) for an expression path conforming to grammar (2), then $(o, t) \models \text{test}$ if there exists a temporal object (o', t') in G such that (o, t, o', t') conforms to path, that is, $(o, t, o', t') \in [\![\text{path}]\!]_G$, as defined next.

Observe that, under this semantics, it holds that $(o, t) \models \neg\, p \mapsto v$ if either $\sigma(o, p, t)$ is undefined or $\sigma(o, p, t)$ is defined but $\sigma(o, p, t) \neq v$. The evaluation of an expression path in NavL[PC, NOI] over the TPG G, denoted by $[\![\text{path}]\!]_G$, is defined as follows. First, we show how tests and axes in grammar (2) are evaluated:

$$
\begin{aligned}
[\![\text{test}]\!]_G &= \{(o, t, o, t) \in \mathrm{PTO}(G) \mid (o, t) \models \text{test}\} \\
[\![\mathbf{F}]\!]_G &= \{(v, t, e, t) \in \mathrm{PTO}(G) \mid \mathrm{src}(e) = v\} \cup \\
&\quad\ \{(e, t, v, t) \in \mathrm{PTO}(G) \mid \mathrm{tgt}(e) = v\} \\
[\![\mathbf{B}]\!]_G &= \{(v, t, e, t) \in \mathrm{PTO}(G) \mid \mathrm{tgt}(e) = v\} \cup \\
&\quad\ \{(e, t, v, t) \in \mathrm{PTO}(G) \mid \mathrm{src}(e) = v\} \\
[\![\mathbf{N}]\!]_G &= \{(o, t_1, o, t_2) \in \mathrm{PTO}(G) \mid t_2 = t_1 + 1\} \\
[\![\mathbf{P}]\!]_G &= \{(o, t_1, o, t_2) \in \mathrm{PTO}(G) \mid t_2 = t_1 - 1\}
\end{aligned}
$$

Assume now that we have some expressions path, path_1 and path_2 in NavL[PC, NOI]. Then, we have that:

$$
\begin{aligned}
[\![(\text{path}_1/\text{path}_2)]\!]_G = \{&(o_1, t_1, o_2, t_2) \in \mathrm{PTO}(G) \mid \\
&\exists (o, t) : (o_1, t_1, o, t) \in [\![\text{path}_1]\!]_G \text{ and} \\
&(o, t, o_2, t_2) \in [\![\text{path}_2]\!]_G\},
\end{aligned}
$$

and $[\![(\text{path}_1 + \text{path}_2)]\!]_G = [\![\text{path}_1]\!]_G \cup [\![\text{path}_2]\!]_G$. Finally, define $\text{path}^0 = (\exists \vee \neg\exists)$, so that $[\![\text{path}^0]\!]_G = \{(o, t, o, t) \in \mathrm{PTO}(G)\}$, and recursively define $\text{path}^{n+1} = (\text{path}/\text{path}^n)$ for $n \geq 0$. Then:

$$
\begin{aligned}
[\![\text{path}[n, m]]\!]_G &= \bigcup_{k=n}^{m} [\![\text{path}^k]\!]_G \\
[\![\text{path}[n, \_]]\!]_G &= \bigcup_{k \geq n} [\![\text{path}^k]\!]_G
\end{aligned}
$$

5.2 On the relationship of NavL[PC,NOI] with the practical query language

In Section 5.1 we introduced a query language with a formal syntax and semantics, to be able to rigorously study its complexity of evaluation. We now show that this language provides a formalization and an extension of the practical query language of Section 4.

Temporal navigation operators `PREV` and `NEXT` in the practical query language correspond to the analogous operators **P** and **N** in NavL[PC,NOI], respectively, while spatial navigation operators `BWD` and `FWD` in the practical query language correspond to the operators **B** and **F** in NavL[PC,NOI], respectively. Next, consider the following `MATCH` clause over an arbitrary TPG:

```
MATCH (x:Person {test = 'pos'})-/PREV/-(y)
ON graph
```

Our task is to construct a query path in NavL[PC,NOI] such that the evaluation of this `MATCH` clause over `graph` is equivalent to the evaluation of path over this TPG. More precisely, path should be defined in such a way that `x` $\mapsto n$, `x_time` $\mapsto t$, `y` $\mapsto n'$, `y_time` $\mapsto t'$ is a binding in the table resulting from evaluating this `MATCH` clause over `graph` if, and only if, $(n, t, n', t') \in [\![\text{path}]\!]_{\texttt{graph}}$. The following temporal path expression satisfies this condition:

$$(\mathbf{Node} \wedge \text{Person} \wedge \text{test} \mapsto \text{pos})/\mathbf{P}/(\mathbf{Node} \wedge \exists)$$

Observe that ($\mathbf{Node} \wedge$Person$\wedge$test $\mapsto$ pos) is used to check whether the following conditions are satisfied for a temporal object (o, t): o is a node with label Person and with value pos in the property test at time point t. Notice that, by definition of TPGs, the fact that test $\mapsto$ pos holds at time t implies that node o exists at this time point. Hence, ($\mathbf{Node} \wedge$Person$\wedge$test $\mapsto$ pos) is used to represent the expression `(x:Person {test = 'pos'})`. Moreover, temporal navigation operator **P** is used to move from the temporal object (o, t) to a temporal object (o, t') such that $t' = t - 1$, so that it is used to represent the expression `-/PREV/-`. Finally, the condition ($\mathbf{Node} \wedge \exists$) is used to test that o is a node that exists at time t'. Observe that we explicitly need to mention the condition $\exists$, as expressions in NavL[PC,NOI] do not enforce the existence of temporal objects by default. The main reason to choose such a semantics is that there are many scenarios where moving through temporal objects that do not exists is useful, in particular when these temporal objects only exist at certain time points. For example, if a bus service is interrupted for some time, then the temporal path expression

$$(\text{Bus} \wedge \neg\exists)/(\mathbf{N}/\neg\exists)[0, \_]/(\text{Bus} \wedge \exists)$$

can be used to look for the next time the service is available. Here, $(\mathbf{N}/\neg\exists)[0, \_]$ moves through an arbitrary number of time points during which the service is unavailable, until the condition $\exists$ holds, representing a time when the service becomes available.

As a second example, consider the following MATCH clause:

```
MATCH (x:Person {test = 'pos'})-
   /PREV*/FWD/:rides/FWD/-(z:Bus)
ON graph
```

Based on the previous discussion, we can see that such a clause can be represented as the following temporal path expression:

$$(\mathbf{Node} \wedge \text{Person} \wedge \text{test} \mapsto \text{pos})/ (\mathbf{P}/\exists)[0,\_]/\mathbf{F}/(\text{rides} \wedge \exists)/\mathbf{F}/(\mathbf{Node} \wedge \text{Bus}),$$

where all temporal objects must exist, as required by the practical query language of Section 4. Observe that we have not explicitly included the existence condition on the last node with label Bus, as the existence of an edge at time point t implies, according to the definition of TPGs, the existence of its starting and ending nodes.

In the previous example, it could be the case that no information about a person is available at some time point. In that case, it is, again, only useful to check that the endpoints of a path exist:

$$(\mathbf{Node} \wedge \text{Person} \wedge \text{test} \mapsto \text{pos})/ \mathbf{P}[0,\_]/\mathbf{F}/(\text{rides} \wedge \exists)/\mathbf{F}/(\mathbf{Node} \wedge \text{Bus}),$$

that is, we start from a Person who has tested positive for some infectious disease at time point t, and then we look for the buses she rode at any time t' prior to receiving this positive test, without imposing the restriction that information about such a person must exists at each intermediate time point $t'' \in [t'+1, t-1]$. Once again, this showcases the flexibility provided by the use of condition $\exists$.

As an additional example, consider the following MATCH clause that uses many of the features of NavL[PC,NOI]:

```
MATCH (x:Person {risk = 'high'})-
    /((FWD/:meets/FWD) +
      (FWD/:rides/FWD/:Bus/BWD/:rides/BWD)
     )/NEXT[0,336]/-({test = 'pos'})
ON graph
```

This MATCH clause corresponds to the temporal path expression:

$$(\mathbf{Node} \wedge \text{Person} \wedge \text{risk} \mapsto \text{high})/(\mathbf{F}/(\text{meets} \wedge \exists)/\mathbf{F} + \mathbf{F}/(\text{rides} \wedge \exists)/\mathbf{F}/\text{Bus}/\mathbf{B}/(\text{rides} \wedge \exists)/\mathbf{B})/ (\mathbf{N}/\exists)[0,336]/(\mathbf{Node} \wedge \text{test} \mapsto \text{pos})$$

As our final example, consider this MATCH clause from Section 4:

```
MATCH (x:Person {risk = 'low' AND time < '10'})
ON contact_tracing
```

The use of a condition over the reserved word time is represented in NavL[PC,NOI] by the condition $< k$. For example, time < '10' is represented by the condition < 10, as a temporal object (o, t) satisfies < 10 if, and only if, $t < 10$. Hence, the previous MATCH clause is equivalent to the following query in NavL[PC,NOI]:

$$(\mathbf{Node} \wedge \text{Person} \wedge \text{risk} \mapsto \text{low} \wedge <10)$$

Abbreviations can be introduced for some of the operators described in this section, and some other common operators, to make notation of the formal language easier to use. For example, we could use condition $=k$, which is written in NavL[PC, NOI] as $<k+1 \wedge \neg(<k)$, and operator **NE** that moves by one unit into the future if the object that is reached exists. However, as such operators are expressible in NavL[PC,NOI], we prefer to use a minimal notation in this formal language to simplify the definition of its syntax and semantics, and the analysis of the complexity of the evaluation problem.

5.3 NavL[PC, NOI] can be evaluated in polynomial time over TPGs

In this section, we show that NavL[PC, NOI] can be evaluated in polynomial time, considering a computational model where accessing the distinct elements of a TPG takes time $O(1)$. More precisely, for a TPG $G = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$, it is assumed that the following operations can be performed in time $O(1)$: given $e \in E$ and $n_1, n_2 \in N$, check whether $\rho(e) = (n_1, n_2)$; given $e \in E$, compute $\text{src}(e)$ and $\text{tgt}(G)$; given $o \in (N \cup E)$ and $\ell \in Lab$, check whether $\lambda(o) = \ell$; given $o \in (N \cup E)$ and $t \in \Omega$, check whether $\xi(o, t) = \text{true}$; and given $o \in (N \cup E)$, $p \in Prop$ and $v \in Val$, check whether $\sigma(o, p, t) = v$. Moreover, we use notation $\|\text{path}\|$ for the length of NavL[PC, NOI]-expression path as an input string over an appropriate alphabet. Then, it is possible to prove the following.

Theorem 5.1. *There exists an algorithm that, given a temporal property graph G and a* NavL[PC, NOI]*-expression* path*, computes $[\![\text{path}]\!]_G$ in time $\tilde{O}(\|\text{path}\|^2 \cdot |\Omega|^2 \cdot (|N| + |E|)^2)$.*

In the rest of this section, we describe the polynomial-time algorithm in the statement of Theorem 5.1. Let $G = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$ be a TPG and path be an expression in NavL[PC,NOI]. Moreover, assume that $M = |\Omega| \cdot (|N| + |E|)$ is the number of distinct (existing or non-existing) temporal objects in G. The algorithm constructs a parsing tree of path, where each node is associated with an operator in NavL[PC,NOI], and then, by using a bottom-up approach, computes, for each node u, the set of tuples (o, t, o', t') that satisfy the operator labeling u. For example, a parsing tree of NavL[PC,NOI]-expression $(\mathbf{Node} \wedge \text{Person} \wedge \text{test} \mapsto \text{pos})/\mathbf{P}[5, 9]/(\mathbf{Node} \wedge \exists)$ is shown in Figure 2. The algorithm starts by evaluating the leaves: $[\![\mathbf{Node}]\!]_G$, $[\![\text{Person}]\!]_G$, $[\![\text{test} \mapsto \text{pos}]\!]_G$, $[\![\mathbf{P}]\!]_G$ and $[\![\exists]\!]_G$, according to the semantics defined in Section 5.1, and then it combines the resulting tables by using the operators $\wedge$, $[5, 9]$ and $/$ in the order specified by the parsing tree. For instance, in the right-hand side of the tree, once $[\![\mathbf{P}]\!]_G$, $[\![\mathbf{Node}]\!]_G$ and $[\![\exists]\!]_G$ have been computed, the algorithm continues by constructing $[\![\mathbf{Node} \wedge \exists]\!]_G$, followed by $[\![\mathbf{P}[5, 9]]\!]_G$ and then by $[\![\mathbf{P}[5, 9]/(\mathbf{Node} \wedge \exists)]\!]_G$. Notice that at any given moment, the result of at most $\|\text{path}\|$ nodes is stored in the form of a table, each one with as many pairs of temporal objects as there are available, *i.e.*, with at most M^2 tuples.

We now explain in detail the different components of the algorithm, paying particular attention to the expressions of the form $\text{path}_1[n, m]$ and $\text{path}_2[n, \_]$, as they are the most expensive to evaluate in NavL[PC,NOI]. Initially, for each leaf of the parsing tree of path, we have to process either a test **Node**, **Edge**, ℓ, $p \mapsto v$, $\exists$ or $< k$, or a navigation operator **N**, **P**, **F**, or **B**. Basic tests can be evaluated in time $O(M)$ just by considering each tuple of the form $(o, t, o, t) \in \text{PTO}(G)$ and checking in $O(1)$ whether (o, t) satisfies the test. As for navigation operators, each one of them can also be evaluated in time $O(M)$ (recall that the existence of nodes or edges at a given time point is not required in the language).

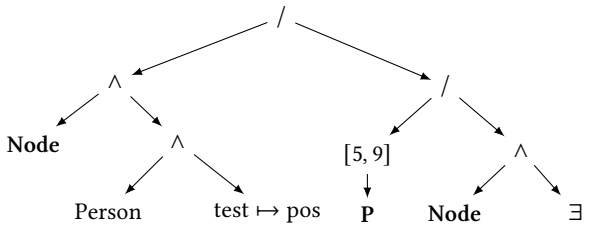

Figure 2: A parsing tree of NavL[PC,NOI]**-expression** path = (**Node** ∧ Person ∧ test ↦ pos)/**P**[5, 9]/(**Node** ∧ ∃)**.**

For example, $[\![\mathbf{P}]\!]_G$ can be constructed just by considering all objects $o \in V \cup E$ and then generating tuples $(o, t, o, t-1)$ such that $t \in \Omega$ and $t-1 \in \Omega$, while $[\![\mathbf{F}]\!]_G$ can be constructed by considering all edges $e \in E$, and then generating tuples $(\mathrm{src}(e), t, e, t)$ and $(e, t, \mathrm{tgt}(e), t)$ for each $t \in \Omega$.

For each internal node u of the parsing tree of path, we must consider one of the following two cases. Assume first that the label of u is either $\wedge$, $\vee$, $\neg$ or ?, so that u represents a more complex test expression $(\mathsf{test}_1 \wedge \mathsf{test}_2)$, $(\mathsf{test}_1 \vee \mathsf{test}_2)$, $(\neg\mathsf{test}_1)$ or $(?\mathsf{path}_1)$. If u represents test $(\mathsf{test}_1 \wedge \mathsf{test}_2)$, then the algorithm has already computed $T_1 = [\![\mathsf{test}_1]\!]_G$ and $T_2 = [\![\mathsf{test}_2]\!]_G$. Hence, to construct $[\![\mathsf{test}_1 \wedge \mathsf{test}_2]\!]_G$, the algorithm needs to compute the intersection of T_1 and T_2, which can be done in time $\tilde{O}(M^2)$ by sorting both tables (each of size at most M^2) and iterating with two pointers, one on each table, to see which elements occur in both. Recall that the notation $\tilde{O}(M^2)$ ignores the logarithmic factors, which in this case appear when sorting tables T_1 and T_2. The case where u represents either $(\mathsf{test}_1 \vee \mathsf{test}_2)$ or $(\neg\mathsf{test}_1)$ can be treated in a similar way. Finally, if u represents test $(?\mathsf{path}_1)$, for each tuple $(o, t, o', t') \in [\![\mathsf{path}_1]\!]_G$, we need to include the tuple (o, t, o, t) in the table for u, as $(o, t) \models (?\mathsf{path}_1)$ if and only if $(o, t, o', t') \in [\![\mathsf{path}_1]\!]_G$ for some temporal object (o', t') in G. This can be done in time $O(M^2)$ as $[\![\mathsf{path}_1]\!]_G$ contains at most M^2 tuples.

Assume now that the label of u is either /, +, $[n, m]$ or $[n, \_]$, so that u represents a more complex path expression $(\mathsf{path}_1/\mathsf{path}_2)$, $(\mathsf{path}_1 + \mathsf{path}_2)$, $\mathsf{path}_1[m, n]$ or $\mathsf{path}_1[m, \_]$. If u represents expression $(\mathsf{path}_1/\mathsf{path}_2)$, then the algorithm has already computed $T_1 = [\![\mathsf{path}_1]\!]_G$ and $T_2 = [\![\mathsf{path}_2]\!]_G$. Hence, to construct $[\![\mathsf{path}_1/\mathsf{path}_2]\!]_G$, the algorithm just need to sort T_1 by the third and fourth columns (the second pair of temporal objects), sort T_2 by the first and second column (the first pair of temporal objects), and then join T_1 with T_2 by looking at matching temporal objects on those columns. The overall time for this construction is $\tilde{O}(M^2)$, as it corresponds to a sort-merge join on two tables with at most M^2 tuples. If u represents the expression $(\mathsf{path}_1 + \mathsf{path}_2)$, the algorithm computes the union of T_1 and T_2. If u represents the expression $\mathsf{path}_1[n, m]$, then the algorithm proceeds as follows, assuming that $T_1 = [\![\mathsf{path}_1]\!]_G$ has already been computed. Given that $\mathsf{path}_1[n, m]$ is equivalent to the expression $\mathsf{path}_1[n, n]/\mathsf{path}_1[0, m-n]$, the procedure first runs Algorithm 1 ComputeRepetition(G, T_1, n) to compute $[\![\mathsf{path}_1[n, n]]\!]_G$ in a similar way to the exponentiation by squaring algorithm [20]. If $n = m$, then we are ready in time $\tilde{O}(\|\mathsf{path}_1\| \cdot M^2)$, since the most expensive operation is the sort-merge join, which is carried out in time $\tilde{O}(M^2)$ and at most

Algorithm 1: ComputeRepetition(G, T_1, n)

Input : A TPG G, a table T_1 such that $T_1 = [\![\mathsf{path}_1]\!]_G$ for some NavL[PC,NOI]-expression path_1, and $n \geq 0$
Output : $[\![\mathsf{path}_1[n, n]]\!]_G$
if $n = 0$ **then**
  return $\{(o, t, o, t) \in \mathrm{PTO}(G)\}$
else if $n = 1$ **then**
  return T_1
$n' \leftarrow \lfloor n/2 \rfloor$
$T_2 \leftarrow$ ComputeRepetition(G, T_1, n')
Compute $T_3 = \{(o_1, t_1, o_2, t_2) \mid \exists (o, t) : (o_1, t_1, o, t) \in T_2$ and $(o, t, o_2, t_2) \in T_2\}$ by doing a sort-merge join
if n *is even* **then**
  return T_3
Compute $T_4 = \{(o_1, t_1, o_2, t_2) \mid \exists (o, t) : (o_1, t_1, o, t) \in T_3$ and $(o, t, o_2, t_2) \in T_1\}$ by doing a sort-merge join
return T_4

Algorithm 2: ComputeIntervalRepetition(G, T_1, T_2, n)

Input : A TPG G, a table T_i such that $T_i = [\![\mathsf{path}_i]\!]_G$ for some NavL[PC,NOI]-expression path_i for $i = 1, 2$, and $n > 0$
Output : $[\![\mathsf{path}_1/\mathsf{path}_2[0, n]]\!]_G$
if $n = 1$ **then**
  return $T_1 \cup T_2$
$n' \leftarrow \lfloor n/2 \rfloor$
$T_3 \leftarrow$ ComputeIntervalRepetition(G, T_1, T_2, n')
Compute $T_4 = \{(o_1, t_1, o_2, t_2) \mid \exists (o, t) : (o_1, t_1, o, t) \in T_3$ and $(o, t, o_2, t_2) \in T_3\}$ by doing a sort-merge join
$T_5 \leftarrow T_3 \cup T_4$
if n *is even* **then**
  return T_5
Compute $T_6 = \{(o_1, t_1, o_2, t_2) \mid \exists (o, t) : (o_1, t_1, o, t) \in T_5$ and $(o, t, o_2, t_2) \in T_2\}$ by doing a sort-merge join
return $T_5 \cup T_6$

$O(\log(n))$ times, that is, $O(\|\mathsf{path}_1\|)$ times. Otherwise, Algorithm 2 ComputeIntervalRepetition$(G, T_1, T_2, m-n)$ is called to compute $[\![\mathsf{path}_1[n, n]/\mathsf{path}_1[0, m-n]]\!]_G$, where $T_2 = [\![\mathsf{path}_1[n, n]]\!]_G$ is the result of invoking ComputeRepetition(G, T_1, n). Here, $O(\log(m-n))$ sort-merge joins have to be carried out, which is again $O(\|\mathsf{path}_1\|)$, so this takes a total time of $\tilde{O}(\|\mathsf{path}_1\| \cdot M^2)$.

Finally, if u represents the expression $\mathsf{path}_1[n, \_]$, then the computation process is similar to the previous one, assuming that $T_1 = [\![\mathsf{path}_1]\!]_G$ has already been computed. As before, we act as if we have to compute the table for another, but equivalent, expression, $\mathsf{path}_1[n, n]/\mathsf{path}_1[0, M^2]$, which is done by first computing $T_2 =$ ComputeRepetition(G, T_1, n), and then invoking ComputeIntervalRepetition(G, T_1, T_2, M^2). This takes time $\tilde{O}(\|\mathsf{path}_1\| \cdot M^2)$ as the sort-merge join is carried out in time $\tilde{O}(M^2)$, and $O(\log(n) + \log(M^2))$ such joins need to be computed, that is $\tilde{O}(\|\mathsf{path}_1\|)$ such joins. Notice that $[\![\mathsf{path}_1[0, \_]]\!]_G = [\![\mathsf{path}_1[0, M^2]]\!]_G$ because: (a) $[\![\mathsf{path}_1[0, k]]\!]_G \subseteq [\![\mathsf{path}_1[0, k+1]]\!]_G$ for every $k \geq 0$; (b) $\left|[\![\mathsf{path}_1[0, k]]\!]_G\right| \leq M^2$ for every $k \geq 0$; and

(c) if $[\![\mathsf{path}_1[0,k]]\!]_G = [\![\mathsf{path}_1[0,k+1]]\!]_G$, then $[\![\mathsf{path}_1[0,k]]\!]_G = [\![\mathsf{path}_1[0,k']]\!]_G$ for every $k' > k$.

In summary, the table associated to each node of the parsing tree of path can be computed in time $\tilde{O}(\|\mathsf{path}\| \cdot M^2)$. Given that there are at most $O(\|\mathsf{path}\|)$ such nodes, the total computation time is $\tilde{O}(\|\mathsf{path}\|^2 \cdot M^2)$, that is, $\tilde{O}(\|\mathsf{path}\|^2 \cdot |\Omega|^2 \cdot (|N| + |E|)^2)$. This concludes the proof of Theorem 5.1.

6 QUERYING COALESCED TEMPORAL PROPERTY GRAPHS

In the previous section, we showed that NavL[PC, NOI] can be evaluated in polynomial time over TPGs in which time points are associated with nodes, edges, and properties. This is an important positive result. However, time-stamping objects with time points may be impractical in terms of space overhead. This motivates the development of interval-based representations, which are common for temporal models for both relations [23, 44] and graphs [15, 21, 43]. Point-based temporal semantics requires that representations be temporally coalesced, namely, that a pair of value-equivalent temporally adjacent tuples should be stored as a single tuple, and this property should be maintained through operations [13].

In this section, we formally define a succinct TPG representation that uses interval time-stamping, which we call coalesced temporal property graph (CTPG). We study the complexity of the evaluation problem for NavL[PC,NOI] queries over CTPGs. It is important to notice that this latter investigation is much more challenging as CTPGs can be exponentially more succinct than TPGs.

6.1 Coalesced temporal property graphs

An interval of $\mathbb{N}$ is a term of the form $[a, b]$ with $a, b \in \mathbb{N}$ and $a \leq b$, which is used as a concise representation of the set of natural numbers $\{i \in \mathbb{N} \mid a \leq i \leq b\}$ (that is, to specify this interval, we just need to mention its starting point a and its ending point b). Using Allen's interval algebra [2], given two intervals $[a_1, b_1]$ and $[a_2, b_2]$, we say that $[a_1, b_1]$ occurs during $[a_2, b_2]$ if $a_2 \leq a_1$ and $b_1 \leq b_2$, $[a_1, b_1]$ meets $[a_2, b_2]$ if $b_1 + 1 = b_2$, and $[a_1, b_1]$ is before $[a_2, b_2]$ if $b_1 + 1 < a_2$.

A finite family $\mathcal{F}$ of intervals is said to be *temporally coalesced* [10], or *coalesced* for short, if $\mathcal{F} = \{[a_1, b_1], \ldots, [a_n, b_n]\}$ and $[a_j, b_j]$ is before $[a_{j+1}, b_{j+1}]$ for every $j \in \{1, \ldots, n-1\}$. For example, $\mathcal{F}_1 = \{[1, 4], [6, 8]\}$ is coalesced, while $\mathcal{F}_2 = \{[1, 2], [3, 4], [6, 8]\}$ is not, because $[1, 2]$ meets $[3, 4]$. The set of all finite coalesced families of intervals is denoted by FC. Observe that $\emptyset \in \mathsf{FC}$. Moreover, given $\mathcal{F}_1, \mathcal{F}_2 \in \mathsf{FC}$, family $\mathcal{F}_1$ is said to be contained in family $\mathcal{F}_2$, denoted by $\mathcal{F}_1 \sqsubseteq \mathcal{F}_2$, if for every $[a_1, b_1] \in \mathcal{F}_1$, there exists $[a_2, b_2] \in \mathcal{F}_2$ such that $[a_1, b_1]$ occurs during $[a_2, b_2]$. Finally, given an interval Ω, we use $\mathsf{FC}(\Omega)$ to denote the set of all families $\mathcal{F} \in \mathsf{FC}$ such that for every $[a, b] \in \mathcal{F}$, it holds that $[a, b]$ occurs during Ω.

Given an interval $[a, b]$ and $v \in \mathit{Val}$, the pair $(v, [a, b])$ is a *valued* interval. A finite family $\mathcal{F}$ of valued intervals is said to be coalesced if $\mathcal{F} = \{(v_1, [a_1, b_1]), \ldots, (v_n, [a_n, b_n])\}$ and for every $j \in \{1, \ldots, n-1\}$, either $[a_j, b_j]$ is before $[a_{j+1}, b_{j+1}]$, or $[a_j, b_j]$ meets $[a_{j+1}, b_{j+1}]$ and $v_j \neq v_{j+1}$. For example, $\mathcal{F}_1 = \{(v, [1, 2]), (v, [5, 8])\}$ and $\mathcal{F}_2 = \{(v, [1, 2]), (w, [3, 4])\}$ are both coalesced (assuming that $v \neq w$). On the other hand, $\mathcal{F}_3 = \{(v, [1, 2]), (v, [3, 4])\}$ is not coalesced because $[1, 2]$ meets $[3, 4]$ and these intervals have the same value in $\mathcal{F}_3$. Moreover, the set of all finite coalesced families of valued intervals is denoted by vFC. Finally, given an interval Ω, we use $\mathsf{vFC}(\Omega)$ to denote the set of all families $\mathcal{F} \in \mathsf{vFC}$ such that for every $(v, [a, b]) \in \mathcal{F}$, it holds that $[a, b]$ occurs during Ω.

We finally have the necessary ingredients to introduce the notion of coalesced temporal property graph.

Definition 6.1. A coalesced temporal property graph (CTPG) is a tuple $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$, where N, E, ρ and λ are defined exactly as for the case of TPGs (see Definition 3.1). Moreover,

- Ω is an interval of $\mathbb{N}$;
- $\xi : (N \cup E) \rightarrow \mathsf{FC}(\Omega)$ is a function that maps a node or an edge to a finite coalesced family of intervals occurring during Ω;
- $\sigma : (N \cup E) \times \mathit{Prop} \rightarrow \mathsf{vFC}(\Omega)$ is a function that maps a node or an edge, and a property name to a finite coalesced family of valued intervals occurring during Ω.

In addition, C satisfies the following conditions:

- If $\rho(e) = (n_1, n_2)$, then $\xi(e) \sqsubseteq \xi(n_1)$ and $\xi(e) \sqsubseteq \xi(n_2)$.
- There exists a finite set of pairs $(o, p) \in (N \cup E) \times \mathit{Prop}$ such that $\sigma(o, p) \neq \emptyset$. Moreover, if $\sigma(o, p) = \{(v_1, [a_1, b_1]), \ldots, (v_n, [a_n, b_n])\}$, then $\{[a_1, b_1], \ldots, [a_n, b_n]\} \sqsubseteq \xi(o)$. □

In the definition of a CTPG, given a node or edge o, function ξ is used to indicate the time intervals where o exists, and function σ is used to indicate the values of a property p for o. More precisely, if $\sigma(o, p) = \{(v_1, [a_1, b_1]), \ldots, (v_n, [a_n, b_n])\}$, then the value of property p for o is v_j in every time point in the interval $[a_j, b_j]$ $(1 \leq j \leq n)$. Moreover, observe that two additional conditions are imposed on C, which enforce that a CTPG conceptually corresponds to a finite sequence of valid conventional property graphs. In particular, as was the case for TPGs, an edge can only exist at a time when both of the nodes it connects exist, and a property can only take on a value at a time when the corresponding node or edge exists. For instance, assume that $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$ is a CTPG corresponding to our running example in Figure 1. Then, we have that $\Omega = [1, 11]$, $\xi(n_2) = \{[1, 9]\}$, $\xi(n_3) = \{[1, 7]\}$ and $\xi(e_2) = \{[1, 2]\}$, so that $\xi(e_2) \sqsubseteq \xi(n_2)$ and $\xi(e_2) \sqsubseteq \xi(n_3)$. Moreover, for the property risk, we have that $\sigma(n_2, \mathsf{risk}) = \{(\mathsf{low}, [1, 4]), (\mathsf{high}, [5, 9])\}$.

We conclude this section by defining how a temporal regular path query is evaluated over a CTPG. Notice that there is a one-to-one correspondence between TPGs and CTPGs. In particular, a CTPG $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$ can be transformed into an equivalent TPG $G = (\Omega', N, E, \rho, \lambda, \xi', \sigma')$ as follows. Assuming that $\Omega = [m, n]$:

- $\Omega' = \{i \in \mathbb{N} \mid m \leq i \leq n\}$;
- for every $o \in (N \cup E)$ and $t \in \Omega'$: $\xi'(o, t) = \mathsf{true}$ if and only if there exists $[a, b] \in \xi(o)$ such that $a \leq t \leq b$;
- for every $o \in (N \cup E)$, $p \in \mathit{Prop}$, $t \in \Omega'$ and $v \in \mathit{Val}$: $\sigma'(o, p, t) = v$ if and only if there exists $(v, [a, b]) \in \sigma(o, p)$ such that $a \leq t \leq b$;

The canonical translation of a CTPG C, denoted by $\mathsf{can}(C)$, is defined as the TPG G obtained by using the previous rules. Then, the evaluation of an expression path in NavL[PC, NOI] over a CTPG C is simply defined by considering its canonical translation, and the definition of the semantics of NavL[PC,NOI] for TPGs:

$$[\![\mathsf{path}]\!]_C \quad = \quad [\![\mathsf{path}]\!]_{\mathsf{can}(C)}.$$

6.2 Evaluating NavL[PC,NOI] over coalesced temporal property graphs

In order to understand the complexity of querying a CTPG with an expression in the language NavL[PC, NOI], we introduce the following decision problem parameterized by a query language $\mathcal{L}$.

Problem:	TupleEval($\mathcal{L}$)
Input:	A CTPG C, an expression path in $\mathcal{L}$ and a pair (o, t), (o', t') of temporal objects in C
Output:	true if $(o, t, o', t') \in [\![\text{path}]\!]_C$, false otherwise.

By studying the complexity of TupleEval($\mathcal{L}$) for different fragments $\mathcal{L}$ of NavL[PC, NOI], we can understand how the use of the operators in NavL[PC, NOI] affects the complexity of the evaluation problem and, in particular, which operators are mode difficult to implement. We start our analysis by considering the decision problem associated to the full language.

Theorem 6.2. TupleEval(NavL[PC, NOI]) *is* Pspace*-complete.*

Proof of Theorem 6.2, and of all other results stated in this section, can be found in Appendix A.

Given that TupleEval(NavL[PC, NOI]) is intractable, in what follows we look for some natural restrictions of the language for which the evaluation problem is feasible.

6.2.1 Removing numerical occurrence indicators. We start by considering a restriction of our query language in which numerical occurrence indicators are not allowed. Formally, this means that grammar (2) is replaced by:

$$\text{path} ::= \text{test} \mid \text{axis} \mid (\text{path}/\text{path}) \mid (\text{path} + \text{path}) \quad (5)$$

The resulting language is called NavL[PC]. The following positive result shows that NavL[PC] can be evaluated efficiently over CTPGs, and so is a natural candidate to be implemented in practice:

Theorem 6.3. TupleEval(NavL[PC]) *can be solved in polynomial time.*

6.2.2 Removing path conditions. Consider a second restriction of our language in which there are no path conditions. Formally, this means that instead of grammar (3), we use:

$$\text{test} ::= \textbf{Node} \mid \textbf{Edge} \mid \ell \mid p \mapsto v \mid < k \mid \exists \mid (\text{test} \vee \text{test}) \mid (\text{test} \wedge \text{test}) \mid (\neg \text{test}) \quad (6)$$

The resulting language is called NavL[NOI], for which we get the following result:

Theorem 6.4. TupleEval(NavL[NOI]) *is* Σ_2^p*-hard, and it is in* Pspace.

A natural question is whether there is a restriction on NavL[NOI] that can reduce the complexity of the evaluation problem but is still expressive enough to represent some useful queries. At this point, a restriction used in the study of XPath comes to the rescue [42]. In what follows, we show that the complexity of the evaluation problem is lower if numerical occurrence indicators are only allowed in the axes. Consider a grammar for tests as in (6), where path conditions are not allowed, and a grammar for path expressions where numerical occurrence indicators are only used in the axes:

$$\text{path} ::= \text{test} \mid \text{axis} \mid \text{axis}[n, m] \mid \text{axis}[n, \_] \mid (\text{path}/\text{path}) \mid (\text{path} + \text{path}) \quad (7)$$

The resulting language is called NavL[ANOI], where ANOI refers to numerical occurrence indicators used only in the axes. The following result shows that such a restriction is effective, in the sense that the complexity of the query evaluation problem decreases:

Theorem 6.5. TupleEval(NavL[ANOI]) *is* NP*-complete.*

6.2.3 Allowing numerical occurrence indicators only in the axes. By the results in the previous sections, TupleEval(NavL[PC]) can be solved in polynomial time, and TupleEval(NavL[ANOI]) is NP-complete. A natural question then is whether the complexity remains the same if these functionalities are combined. Notice that TupleEval(NavL[PC, NOI]) is Pspace-complete, so a positive answer to this question means a significant decrease in the complexity of the query evaluation problem. Unfortunately, we show that the complexity of the entire language does not decrease by restricting numerical occurrence indicators to occur only in the axes.

Theorem 6.6. TupleEval(NavL[PC, ANOI]) *is* Pspace*-complete.*

The results of this section can be considered as a guide for future implementations of NavL[PC, NOI] over coalesced temporal property graphs. The main message is that the language including path conditions can be efficiently evaluated over such graphs. Besides, a reduction in the complexity for numerical occurrence indicators can be obtained by restricting them to occur only on the axes.

7 CONCLUSIONS

In the last decade, considerable research attention has been focused on property graphs. Substantial progress has been made towards standardizing the syntax of graph query languages, and towards thoroughly understanding their semantics and complexity of evaluation. Regular path queries (RPQs) are a basic building block of graph query languages. In this paper, we considered temporal property graphs (TPGs) that associate temporal information with graph nodes and edges to represent evolution of graph topology, and of node and edge properties, over time. We proposed temporal regular path queries (TRPQ) that incorporate time into TPG navigation.

Starting with design principles for a temporal extension of RPQs, we proposed a natural syntactic extension of the `MATCH` clause of popular query languages for conventional (non-temporal) graphs like Cypher, PGQL, and G-Core. We then formally presented the semantics of the TRPQ language, and studied its complexity of evaluation. We showed that TRPQs can be evaluated in polynomial time if TPGs are time-stamped with time points. We then considered a more succinct, and thus more practical, interval-annotated representation of TPGs, and identified fragments of the TRPQ language that admit efficient evaluation over this representation. Our work on the syntax, and the positive complexity results, pave the way to implementations of TRPQs that are both usable and practical.

We also showed that the problem of evaluating the full TRPQ language over interval-annotated TPGs is Pspace-complete. Even after reducing its expressive power, the problem remains hard in presence of numerical occurrence indicators. We leave open determining the exact complexity of the fragment of the language that admits numerical indicators.

REFERENCES

[1] Serge Abiteboul and Victor Vianu. 1997. Regular Path Queries with Constraints. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '97)*. Association for Computing Machinery, New York, NY, USA, 122–133. https://doi.org/10.1145/263661.263676

[2] James F Allen. 1983. Maintaining Knowledge about Temporal Intervals. *Commun. ACM* 26, 11 (1983), 832–843.

[3] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1421–1432. https://doi.org/10.1145/3183713.3190654

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40. https://doi.org/10.1145/3104031

[5] Association of ISO Graph Query Language Proponents. 2020. GQL Standard. https://www.gqlstandards.org.

[6] Sitaram Asur, Srinivasan Parthasarathy, and Duygu Ucar. 2009. An Event-Based Framework for Characterizing the Evolutionary Behavior of Interaction Graphs. *ACM Trans. Knowl. Discov. Data* 3, 4, Article 16 (Dec. 2009), 36 pages. https://doi.org/10.1145/1631162.1631164

[7] Pablo Barceló Baeza. 2013. Querying Graph Databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '13)*. Association for Computing Machinery, New York, NY, USA, 175–188. https://doi.org/10.1145/2463664.2465216

[8] Piotr Berman, Marek Karpinski, Lawrence L. Larmore, Wojciech Plandowski, and Wojciech Rytter. 1997. On the Complexity of Pattern Matching for Highly Compressed Two-Dimensional Texts. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM '97)*. Springer-Verlag, Berlin, Heidelberg, 40–51.

[9] Antje Beyer, Peter Thomason, Xinzhong Li, James Scott, and Jasmin Fisher. 2010. Mechanistic Insights into Metabolic Disturbance during Type-2 Diabetes and Obesity Using Qualitative Networks. *Transactions on Computational Systems Biology XII, Special Issue on Modeling Methodologies* 12 (2010), 146–162. https://doi.org/10.1007/978-3-642-11712-1_4

[10] Michael Böhlen. 2009. *Temporal Coalescing*. Springer US, Boston, MA, 2932–2936. https://doi.org/10.1007/978-0-387-39940-9_388

[11] Michael H Böhlen, Renato Busatto, and Christian S Jensen. 1998. Point Versus Interval-based Temporal Data Models. In *Proceedings of the 14th IEEE ICDE*. IEEE, Orlando, FL, 192–200. http://people.cs.aau.dk/{~}csj/Thesis/pdf/chapter7.pdf

[12] Michael H Böhlen, Christian S Jensen, and Richard T Snodgrass. 2000. Temporal Statement Modifiers. *ACM Transactions on Database Systems* 25, 4 (2000), 407–456.

[13] Michael H. Böhlen, Richard T. Snodgrass, and Michael D. Soo. 1996. Coalescing in Temporal Databases. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*. 180–191.

[14] Karsten M. Borgwardt, Hans-Peter Kriegel, and Peter Wackersreuther. 2006. Pattern Mining in Frequent Dynamic Subgraphs. In *Proceedings of the Sixth International Conference on Data Mining (ICDM '06)*. IEEE Computer Society, USA, 818–822. https://doi.org/10.1109/ICDM.2006.124

[15] Jaewook Byun, Sungpil Woo, and Daeyoung Kim. 2020. ChronoGraph: Enabling Temporal Graph Traversals for Efficient Information Diffusion Analysis over Time. *IEEE Trans. Knowl. Data Eng.* 32, 3 (2020), 424–437. https://doi.org/10.1109/TKDE.2019.2891565

[16] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. 2002. Rewriting of regular expressions and regular path queries. *J. Comput. System Sci.* 64, 3 (2002), 443–465.

[17] Jeffrey Chan, James Bailey, and Christopher Leckie. 2008. Discovering correlated spatio-temporal changes in evolving graphs. *Knowledge and Information Systems* 16, 1 (2008), 53–96. https://doi.org/10.1007/s10115-007-0117-z

[18] James Clark and Steve DeRose. W3C Recommendation 16 November 1999. XML Path Language (XPath) Version 1.0.

[19] James Clifford and Abdullah Uz Tansel. 1985. On an Algebra for Historical Relational Databases: Two Views. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data (SIGMOD '85)*. Association for Computing Machinery, New York, NY, USA, 247–265. https://doi.org/10.1145/318898.318922

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press.

[21] Ariel Debrouvier, Eliseo Parodi, Matías Perazzo, Valeria Soliani, and Alejandro Vaisman. 2021. A model and query language for temporal graph databases. *VLDB Journal* (2021). https://doi.org/10.1007/s00778-021-00675-4

[22] Anton Dignös, Michael H. Böhlen, and Johann Gamper. 2012. Temporal Alignment. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 433–444. https://doi.org/10.1145/2213836.2213886

[23] Anton Dignös, Michael H. Böhlen, and Johann Gamper. 2012. Temporal alignment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman (Eds.). ACM, 433–444. https://doi.org/10.1145/2213836.2213886

[24] Arash Fard, Amir Abdolrashidi, Lakshmish Ramaswamy, and John Miller. 2012. Towards Efficient Query Processing on Massive Time-Evolving Graphs. In *Proceedings of the 8th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*. 567–574. https://doi.org/10.4108/icst.collaboratecom.2012.250532

[25] Afonso Ferreira. 2004. Building a reference combinatorial model for MANETs. *IEEE Network* 18, 5 (2004), 24–29. https://doi.org/10.1109/MNET.2004.1337732

[26] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1433–1445. https://doi.org/10.1145/3183713.3190657

[27] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA.

[28] Michaela Goetz, Jure Leskovec, Mary McGlohon, and Christos Faloutsos. 2009. Modeling Blog Dynamics. In *Proceedings of the Third International Conference on Weblogs and Social Media, ICWSM 2009, San Jose, California, USA, May 17-20, 2009*, Eytan Adar, Matthew Hurst, Tim Finin, Natalie S. Glance, Nicolas Nicolov, and Belle L. Tseng (Eds.). The AAAI Press, San Jose, CA, 26–33. http://aaai.org/ocs/index.php/ICWSM/09/paper/view/152

[29] Georg Gottlob, Christoph Koch, and Reinhard Pichler. 2005. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.* 30, 2 (2005), 444–491.

[30] Christian S. Jensen, Michael D. Soo, and Richard T. Snodgrass. 1994. Unifying temporal data models via a conceptual model. *Information Systems* 19, 7 (1994), 513 – 547. https://doi.org/10.1016/0306-4379(94)90013-2

[31] Theodore Johnson, Yaron Kanza, Laks V. S. Lakshmanan, and Vladislav Shkapenyuk. 2016. Nepal: a path query language for communication networks. In *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics, NDA@SIGMOD 2016, San Francisco, California, USA, July 1, 2016*, Akhil Arora, Shourya Roy, and Sameep Mehta (Eds.). ACM, 6:1–6:8. https://doi.org/10.1145/2980523.2980530

[32] Andrey Kan, Jeffrey Chan, James Bailey, and Christopher Leckie. 2009. A Query Based Approach for Mining Evolving Graphs. In *Proceedings of the Eighth Australasian Data Mining Conference - Volume 101 (AusDM '09)*. Australian Computer Society, Inc., AUS, 139–150.

[33] Udayan Khurana and Amol Deshpande. 2013. Efficient Snapshot Retrieval over Historical Graph Data. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, USA, 997–1008. https://doi.org/10.1109/ICDE.2013.6544892

[34] Udayan Khurana and Amol Deshpande. 2016. Storing and Analyzing Historical Graph Data at Scale. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT'16*. Bordeaux, France, 65–76. arXiv:1509.08960 http://arxiv.org/abs/1509.08960

[35] Krishna G. Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL: 2011. *SIGMOD Record* 41, 3 (2012), 34–43. https://doi.org/10.1145/2380776.2380786

[36] Alan G. Labouseur, Jeremy Birnbaum, Paul W. Olsen, Sean R. Spillane, Jayadevan Vijayan, Jeong Hyon Hwang, and Wook Shin Han. 2014. The G* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases* 33, 4 (2014), 479–514. https://doi.org/10.1007/s10619-014-7140-3

[37] M. Lahiri and T.Y. Berger-Wolf. 2008. Mining Periodic Behavior in Dynamic Social Networks. In *2008 Eighth IEEE International Conference on Data Mining*. 373–382. https://doi.org/10.1109/ICDM.2008.104

[38] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. 2007. The Dynamics of Viral Marketing. *ACM Trans. Web* 1, 1 (May 2007), 5–es. https://doi.org/10.1145/1232722.1232727

[39] Jure Leskovec, Lars Backstrom, Ravi Kumar, and Andrew Tomkins. 2008. Microscopic Evolution of Social Networks. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '08)*. Association for Computing Machinery, New York, NY, USA, 462–470. https://doi.org/10.1145/1401890.1401948

[40] Leonid Libkin, Wim Martens, and Domagoj Vrgoc. 2016. Querying Graphs with Data. *J. ACM* 63, 2 (2016), 14:1–14:53.

[41] Ling Liu and M. Tamer Zsu. 2009. *Encyclopedia of Database Systems* (1st ed.). Springer Publishing Company, Incorporated, Boston, MA.

[42] Maarten Marx. 2005. Conditional XPath. *ACM Trans. Database Syst.* 30, 4 (2005), 929–959.

[43] Vera Zaychik Moffitt and Julia Stoyanovich. 2017. Temporal Graph Algebra. In *Proceedings of The 16th International Symposium on Database Programming Languages (DBPL '17)*. Association for Computing Machinery, New York, NY, USA, Article 10, 12 pages. https://doi.org/10.1145/3122831.3122838

[44] Angelo Montanari and Jan Chomicki. 2009. *Time Domain*. Springer US, Boston, MA, 3103–3107. https://doi.org/10.1007/978-0-387-39940-9_427

[45] Panagiotis Papadimitriou, Ali Dasdan, and Hector Garcia-Molina. 2010. Web graph similarity for anomaly detection. *J. Internet Services and Applications* 1, 1 (2010), 19–30. https://doi.org/10.1007/s13174-010-0003-x

[46] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. 2011. On Querying Historical Evolving Graph Sequences. *Proceedings of the VLDB Endowment* 4, 11 (2011), 726–737.

[47] Jonathan Robie, Michael Dyck, and Josh Spiegel. W3C Recommendation 21 March 2017. XML Path Language (XPath) 3.1.

[48] Betty Salzberg and Vassilis J. Tsotras. 1999. Comparison of access methods for time-evolving data. *Comput. Surveys* 31, 2 (jun 1999), 158–221. https://doi.org/10.1145/319806.319816

[49] Purnamrita Sarkar, Deepayan Chakrabarti, and Michael I. Jordan. 2012. Nonparametric Link Prediction in Dynamic Networks. In *Proceedings of the 29th International Coference on International Conference on Machine Learning (ICML'12)*. Omnipress, Madison, WI, USA, 1897–1904.

[50] Konstantinos Semertzidis, Evaggelia Pitoura, and Kostas Lillis. 2015. TimeReach: Historical Reachability Queries on Evolving Graphs. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*, Gustavo Alonso, Floris Geerts, Lucian Popa, Pablo Barceló, Jens Teubner, Martín Ugarte, Jan Van den Bussche, and Jan Paredaens (Eds.). OpenProceedings.org, Brussels, Belgium, 121–132. https://doi.org/10.5441/002/edbt.2015.12

[51] Richard Snodgrass and Ilsoo Ahn. 1985. A Taxonomy of Time Databases. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data (SIGMOD '85)*. ACM, New York, NY, USA, 236–246. https://doi.org/10.1145/318898.318921

[52] Kumar Sricharan and Kamalika Das. 2014. Localizing anomalous changes in time-evolving graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. Snowbird, Utah USA, 1347–1358. https://doi.org/10.1145/2588555.2612184

[53] L. J. Stockmeyer and A. R. Meyer. 1973. Word Problems Requiring Exponential Time(Preliminary Report). In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing (STOC '73)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/800125.804029

[54] Joshua M. Stuart, Eran Segal, Daphne Koller, and Stuart K. Kim. 2003. A gene-coexpression network for global discovery of conserved genetic modules. *Science* 5643, 302 (2003), 249––255.

[55] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A Property Graph Query Language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems (GRADES '16)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2960414.2960421

[56] Moshe Y. Vardi. 1982. The Complexity of Relational Query Languages (Extended Abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC '82)*. Association for Computing Machinery, New York, NY, USA, 137–146. https://doi.org/10.1145/800070.802186

[57] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path Problems in Temporal Graphs. *Proc. VLDB Endow.* 7, 9 (2014), 721–732. https://doi.org/10.14778/2732939.2732945

[58] Huanhuan Wu, James Cheng, Yiping Ke, Silu Huang, Yuzhen Huang, and Hejun Wu. 2016. Efficient Algorithms for Temporal Path Computation. *IEEE Trans. Knowl. Data Eng.* 28, 11 (2016), 2927–2942. https://doi.org/10.1109/TKDE.2016.2594065

[59] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. 2016. Reachability and time-based path queries in temporal graphs. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 145–156. https://doi.org/10.1109/ICDE.2016.7498236

[60] Lei Yang, Lei Qi, Yan-Ping Zhao, Bin Gao, and Tie-Yan Liu. 2007. Link analysis using time series of web graphs. In *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007*, Mário J. Silva, Alberto H. F. Laender, Ricardo A. Baeza-Yates, Deborah L. McGuinness, Bjørn Olstad, Øystein Haug Olsen, and André O. Falcão (Eds.). ACM, 1011–1014. https://doi.org/10.1145/1321440.1321598

A PROOFS

A.1 Proof of Theorem 6.2

Consider the following well-known decision problem called True Quantified Boolean Formula (TQBF), which is well known to be PSPACE-complete [53]:

Problem:	TQBF
Input:	A quantified Boolean formula $\psi = Q_1 x_1 \dots Q_n x_n \, \varphi(x_1, \dots, x_n)$ in prenex normal form where $\varphi(x_1, \dots, x_n)$ is a Boolean formula on variables $x_1, \dots, x_n$, and $Q_1, \dots, Q_n$ are quantifiers ($\forall$ or $\exists$).
Output:	true if ψ is valid, and false otherwise.

Without loss of generality, φ can be assumed to be in conjunctive normal form. We will show that TQBF can be reduced to our problem TUPLEEVAL(NavL[PC, NOI]) by proceeding in three steps. Let $\psi = Q_1 x_1 \dots Q_n x_n \, \varphi(x_1, \dots, x_n)$. First, we will show that a predicate $\text{bit}(i, t)$ (defined below) can be written in our language. Then, by using that predicate, we will show that φ can be encoded in our language. Finally, we will show that an expression representing the quantifiers of ψ can be added to the expression encoding φ, which yields the result.

We start with a QBF formula ψ as described above to build an input for the problem TUPLEEVAL(NavL[PC, NOI]). More precisely, the input CTPG will be $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$ where $\Omega = [0, 2^n - 1]$, $N = \{v\}$, $E = \varnothing$, ρ is an empty function, $\lambda(v) = l$, $\xi(v) = \{[0, 2^n - 1]\}$ and σ is an empty function. In other words, C is a CTPG consisting of only one node existing from time 0 to time $2^n - 1$, with no edges or properties. Moreover, we will build an expression r such that $(v, 0, v, 0) \in [\![r]\!]_C$ if and only if ψ is valid, which concludes the reduction. The steps to construct r are shown next.

Step 1: Expressing the predicate bit with an expression in NavL[PC, NOI]**.** Consider predicate $\text{bit}(i, t)$ that tests whether the i-th bit of time t (from right to left when written in its binary representation) is 1. For instance, $\text{bit}(1, 0)$ is false, and $\text{bit}(5, 30)$ is true, since the first bit of 0 is 0, whereas 30 is 11110 in binary, and its fifth bit is 1. Now, consider the following expression:

$$r_i \quad = \quad ?\left(\mathbf{P}\left[2^i, 2^i\right][0, \_] \; / \left(< 2^i \wedge \neg < 2^{i-1}\right)\right)$$

Notice that r_i is a test. Thus, for a pair of temporal objects (o_1, t_1, o_2, t_2) to satisfy r_i, (o_1, t_1) must be equal to (o_2, t_2). The expression to satisfy is a path test, so there must be some temporal object (o_3, t_3), such that $(o_1, t_1, o_3, t_3) \in [\![\mathbf{P}\left[2^i, 2^i\right][0, \_] \; / (< 2^i \wedge \neg < 2^{i-1})]\!]_C$. Since the right part is a test as well, we can split the expression into two parts. Firstly, we must have that $(o_1, t_1, o_3, t_3) \in [\![\mathbf{P}\left[2^i, 2^i\right][0, \_]]\!]_C$, which implies that $t_3 = t_1 - k * 2^i$ for some integer $k \geq 0$. Secondly, we must have that $(o_3, t_3) \models (< 2^i \wedge \neg < 2^{i-1})$, which implies that $2^{i-1} \leq t_3 < 2^i$. This means that by writing t_3 in its binary form, we get 1 as its i-th bit. Together, these two conditions imply that t_1 also has 1 as its i-th bit, when written in its binary form. In consequence, we get that

$$(o, t, o, t) \in [\![r_i]\!]_C \text{ if and only if } \text{bit}(i, t) \text{ is true}$$

Besides, we trivially get that:

$$(o, t, o, t) \in [\![\neg r_i]\!]_C \text{ if and only if } \text{bit}(i, t) \text{ is false}$$

Finally, notice that both r_i and $\neg r_i$ have linear length with respect to i, which will be important later.

Step 2: Expressing any CNF formula in NavL[PC, NOI]**.** Assume that

$$\varphi(x_1, \dots, x_n) \quad = \quad \bigwedge_{j=1}^{m} \bigvee_{k=1}^{m_j} l_{j,k}$$

where for every j and k, $l_{j,k}$ is a literal, *i.e.*, either a variable in $\{x_1, \dots, x_n\}$ or its negation. Then, for every $j \in \{1, \dots, m\}$ and $k \in \{1, \dots, m_j\}$, we define:

$$L_{j,k} \quad = \quad \begin{cases} r_i & \text{if } l_{j,k} = x_i \\ \neg r_i & \text{if } l_{j,k} = \neg x_i \end{cases}$$

We can use these expressions to build a regular expression that tests the satisfiability of our formula $\varphi(x_1, \dots, x_n)$ by any valuation $\sigma : \{x_1, \dots, x_n\} \to \{\text{false}, \text{true}\}$. To do this, we will use a time value $t \in [0, 2^n - 1]$ to represent a valuation σ_t, where $\sigma_t(x_i) = \text{true}$ if and only if the i-th bit of t is 1. We can do so by employing the expressions $L_{i,k}$ on tests along with conjunctions ($\wedge$) and disjunctions ($\vee$), which are also present in NavL[PC, NOI]:

$$r_{\varphi(x_1,\dots,x_n)} \quad = \quad \bigwedge_{j=1}^{m} \bigvee_{k=1}^{m_j} L_{j,k}$$

Here again, $r_{\varphi(x_1,\dots,x_n)}$ is a test, so if it is satisfied by (o_1, t_1, o_2, t_2), then $(o_1, t_1) = (o_2, t_2)$ Furthermore, we show that, because of how the expressions $L_{j,k}$ are defined, we have:

$$(o, t, o, t) \in [\![r_{\varphi(x_1,\dots,x_n)}]\!]_C \text{ if and only if } \sigma_t\left(\varphi(x_1, \dots, x_n)\right) = \text{true}$$

To show the previous assertion, first assume that $(o,t,o,t) \in [\![r_{\varphi(x_1,\ldots,x_n)}]\!]_C$. Because of how the expression is defined, for each $j \in \{1,\ldots,m\}$ we must have that $(o,t,o,t) \in [\![\bigvee_{k=1}^{m_j} L_{j,k}]\!]_C$. Hence, for any arbitrary $j \in \{1,\ldots,m\}$, we immediately get that there must exist $k \in \{1,\ldots,m_j\}$ such that $(o,t,o,t) \in [\![L_{j,k}]\!]_C$. If $l_{j,k} = x_i$, then we must also have that $(o,t,o,t) \in [\![r_i]\!]_C$, which, as we already showed, is equivalent to having that $\mathrm{bit}(i,t)$ is true, which in turn is equivalent to $\sigma_t(x_i) = \mathsf{true}$. Otherwise, if $l_{j,k} = \neg x_i$, then we must have that $(o,t,o,t) \in [\![\neg r_i]\!]_C$, which, as we also showed, is equivalent to having that $\mathrm{bit}(i,t)$ is false, which in turn is equivalent to $\sigma_t(x_i) = \mathsf{false}$. In both cases, we get that $\sigma_t(l_{j,k}) = \mathsf{true}$, which means that $\sigma_t\left(\bigvee_{k=1}^{m_j} l_{j,k}\right) = \mathsf{true}$. Since this holds for an arbitrary value j, it holds for the entrie conjunction. Hence, $\sigma_t\left(\varphi(x_1,\ldots,x_n)\right) = \mathsf{true}$.

Now, to show that the inverse is also true, assume that there is some $t \in \Omega$ such that $\sigma_t\left(\varphi(x_1,\ldots,x_n)\right) = \mathsf{true}$. By definition, this means that for every $j \in \{1,\ldots,m\}$, $\sigma_t\left(\bigvee_{k=1}^{m_j} l_{j,k}\right) = \mathsf{true}$. In turn, this means that for every j there is some $k \in \{1,\ldots,m_j\}$ such that $\sigma_t(l_{j,k}) = \mathsf{true}$. As we saw earlier, this condition is equivalent to having that $(o,t,o,t) \in [\![L_{j,k}]\!]_C$, hence, for every $j \in \{1,\ldots,m\}$, we also have that $(o,t,o,t) \in [\![\bigvee_{k=1}^{m_j} L_{j,k}]\!]_C$. Given that this is true for every j, by definition, it is also true for the conjunction of them. Hence, we get that $(o,t,o,t) \in [\![r_{\varphi(x_1,\ldots,x_n)}]\!]_C$. This concludes the proof in Step 2.

Observation: Notice that the previous results already implies NP-hardness and coNP-hardness for the problem. Since every valuation $\sigma : \{x_1,\ldots,x_n\} \to \{\mathsf{false},\mathsf{true}\}$ has a corresponding time point t such that $\sigma = \sigma_t$, and a Boolean CNF formula $\varphi(x_1,\ldots,x_n)$ on n variables $x_1,\ldots,x_n$ is satisfiable if and only if there exists a valuation σ such that $\sigma\left(\varphi(x_1,\ldots,x_n)\right) = \mathsf{true}$, it is also true that $\varphi(x_1,\ldots,x_n)$ is satisfiable if and only if $(v,0,v,0) \in [\![?(\mathbf{N}[0,\_] \,/\, r_{\varphi(x_1,\ldots,x_n)})]\!]_C$ (that is, advance in time to an arbitrary time point t and check the condition that implies that $\sigma_t\left(\varphi(x_1,\ldots,x_n)\right) = \mathsf{true}$ for the temporal object (v,t)). Similarly, $\varphi(x_1,\ldots,x_n)$ is a tautology if and only if $(v,0,v,0) \in [\![\neg?(\mathbf{N}[0,\_] \,/\, \neg r_{\varphi(x_1,\ldots,x_n)})]\!]_C$ (there is no path to a time point t such that $\sigma_t\left(\varphi(x_1,\ldots,x_n)\right) = \mathsf{false}$, hence there is no valuation that makes φ to be false). In what follows, we show PSPACE-hardness of the problem.

Step 3: Expressing satisfiability of quantified Boolean formulae with an expression in NavL[PC,NOI]**.** Consider a quantified Boolean formula in prenex normal form

$$Q_1 x_1 \ldots Q_n x_n \; \varphi(x_1,\ldots,x_n)$$

Since we already have a way to express $\varphi(x_1,\ldots,x_n)$, we only need to express the possible valuations (*i.e.*, time points) generated by the sequence of quantifiers $Q_1,\ldots,Q_n$.

First, assume Q_i is the existential quantifier ($\exists$). This means that we can either make x_i take valuation true or false to satisfy our formula. Considering time points, this is equivalent to have either 1 or 0 at the i-th bit of the time t at which we are standing. The intuition is that this can easily be expressed by starting at time 0, and then deciding whether to move into a future time point with the expression $(\mathbf{N}[2^{i-1},2^{i-1}] + \mathbf{N}[0,0])$ to set the i-th bit of the time point. Notice that if there are only expressions of the form $\mathbf{N}[2^{i-1},2^{i-1}]$, and they are only mentioned once (at least, as prefixes of our test expressions) for each i, they will not affect other bits of t. Hence, if s_{i+1} represents the part of the subformula $Q_{i+1}x_{i+1}\ldots Q_n x_n\; \varphi(x_1,\ldots,x_n)$, then the subformula $\exists x_i\; Q_{i+1}x_{i+1}\ldots Q_n x_n\; \varphi(x_1,\ldots,x_n)$ can be represented by first navigating through time, only affecting the first $i-1$ bits, and then testing that the reached temporal object satisfies the following test:

$$s_i \quad = \quad ?\left(\left(\mathbf{N}[2^{i-1},2^{i-1}] + \mathbf{N}[0,0]\right) \;/\; s_{i+1}\right).$$

In turn, if Q_i is the universal quantifier ($\forall$), we will employ the fact that $\forall x\, \psi(x)$ is equivalent to $\neg\exists x\, \neg\psi(x)$ for every formula $\psi(x)$ with free variable x. Hence, if s_{i+1} represents the part of the subformula $Q_{i+1}x_{i+1}\ldots Q_n x_n\; \varphi(x_1,\ldots,x_n)$, then the subformula $\forall x_i\; Q_{i+1}x_{i+1}\ldots Q_n x_n\; \varphi(x_1,\ldots,x_n)$ can be represented by first only navigating through time, only affecting the first $i-1$ bits, and then testing whether the reached temporal object satisfies the following test:

$$s_i \quad = \quad \neg\left(?\left(\left(\mathbf{N}[2^{i-1},2^{i-1}] + \mathbf{N}[0,0]\right) \;/\; (\neg s_{i+1})\right)\right).$$

Finally, define $s_{n+1} = r_{\varphi(x_1,\ldots,x_n)}$. We claim that, for $i \in \{n+1,n,\ldots,1\}$ (*i.e.*, 0 to n quantifiers), if $t < 2^{i-1}$, then:

$$(v,t,v,t) \in [\![s_i]\!]_C \text{ if and only if } Q_i x_i \ldots Q_n x_n\; \varphi(\sigma_t(x_1),\ldots,\sigma_t(x_{i-1}),x_i,\ldots,x_n) \text{ is valid.}$$

In particular, for n quantifiers, *i.e.*, when $i = 1$, this result gives us PSPACE-hardness for TUPLEEVAL(NavL[PC, NOI]), since we will have that ψ is true if and only if $(v,0,v,0) \in [\![s_1]\!]_C$, where C and s_1 can be constructed in polynomial time in the size of ψ. We will prove this claim by induction over the number of quantifiers preceding $\varphi(x_1,\ldots,x_n)$.

The base case consists of the formula with no quantified variables, *i.e.*, when $i = n+1$. We must show that if $t < 2^n$, then $(v,t,v,t) \in [\![r_{\varphi(x_1,\ldots,x_n)}]\!]_C$ if and only if $\varphi(\sigma(x_1),\ldots,\sigma(x_n))$ is true. Notice that $\varphi(\sigma(x_1),\ldots,\sigma(x_n))$ is true is equivalent to having that $\sigma_t\left(\varphi(x_1,\ldots,x_n)\right)$ is true. Hence, by step 2, this is equivalent to having that $(v,t,v,t) \in [\![r_{\varphi(x_1,\ldots,x_n)}]\!]_C$.

For the inductive case, assume that the claim holds for k quantifiers, for some k such that $0 \le k \le n$. This means that for $i = n-k+1$, if $t < 2^{i-1}$, then the following condition holds:

$$(v,t,v,t) \in [\![s_i]\!]_C \text{ if and only if } Q_i x_i \ldots Q_n x_n\; \varphi(\sigma_t(x_1),\ldots,\sigma_t(x_{i-1}),x_i,\ldots,x_n) \text{ is valid}$$

We then have to prove that the condition holds for $k+1$ quantifiers, *i.e.*, for $i-1 = n-k$. That is, if $t' < 2^{i-2}$, then we have to show that:

$$(v, t', v, t') \in [\![s_{i-1}]\!]_C \text{ if and only if } Q_{i-1}x_{i-1}\dots Q_n x_n\ \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), x_{i-1}, \dots, x_n) \text{ is valid} \tag{8}$$

Let $t' < 2^{i-2}$, and consider the following cases.

- If $Q_{i-1} = \exists$, recall that

$$s_{i-1} \quad = \quad ?\left(\left(\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0,0]\right)\ /\ s_i\right).$$

  Notice then that by definition of s_{i-1}, we have that $(v, t', v, t') \in [\![s_{i-1}]\!]_C$ if and only if there exists $t_1 \in \Omega$ such that $(v, t', v, t_1) \in [\![\left(\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0,0]\right)\ /\ s_i]\!]_C$. By definition of s_i, the previous condition holds if and only if $(v, t', v, t_1) \in [\![\left(\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0,0]\right)]\!]_C$ and $(v, t_1, v, t_1) \in [\![s_i]\!]_C$. Now, this means that $(v, t', v, t') \in [\![s_{i-1}]\!]_C$ if and only if there exists $t_1 \in \{t', t' + 2^{i-2}\}$ satisfying that $(v, t_1, v, t_1) \in [\![s_i]\!]_C$.
  To prove the direction ($\Rightarrow$) of (8) assume that $(v, t', v, t') \in [\![s_{i-1}]\!]_C$, which implies that there exists $t_1 \in \{t', t' + 2^{i-2}\}$ satisfying that $(v, t_1, v, t_1) \in [\![s_i]\!]_C$. Given that t' is an integer with $i-2$ bits, t_1 comes from either putting 1 or 0 as the $(i-1)$-th bit of t', which means that $t_1 < 2^{i-1}$ must hold. By induction hypothesis, this implies that $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t_1}(x_1), \dots, \sigma_{t_1}(x_{i-1}), x_i, \dots, x_n)$ is valid. Since t_1 and t' share the same first $i-2$ bits, we get that $\sigma_{t'}(x_j) = \sigma_{t_1}(x_j)$ for $j \in \{1, ..., i-2\}$. Therefore, we conclude that $\exists x_{i-1} Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), x_{i-1}, \dots, x_n)$ is valid.
  To prove the direction ($\Leftarrow$) of (8) suppose that $\exists x_{i-1} Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), x_{i-1}, \dots, x_n)$ is valid. Then we know that $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), b, x_i, \dots, x_n)$ is valid for some value $b \in \{\text{true}, \text{false}\}$. Define $\mathbb{1}_b$ as 1 if $b = \text{true}$, and as 0 otherwise. Notice then that by taking $t_1 = t' + 2^{i-2} \cdot \mathbb{1}_b$, we get that $\sigma_{t_1}(x_{i-1}) = b$. Moreover, $\sigma_{t_1}(x_j) = \sigma_{t'}(x_j)$ for every $j \in \{1, \dots, i-2\}$ since $t' < 2^{i-2}$ and t_1 shares all its first $i-2$ bits with t'. In consequence, this gives us that $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t_1}(x_1), \dots, \sigma_{t_1}(x_{i-1}), x_i, \dots, x_n)$ is valid. By induction, this means that $(v, t_1, v, t_1) \in [\![s_i]\!]_C$. Since $t_1 = t' + 2^{i-2} \cdot \mathbb{1}_b$, we have that either $t_1 = t'$ or $t_1 = t' + 2^{i-2}$. In any case, we get that $(v, t', v, t_1) \in [\![\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0,0]]\!]_C$, so we have that $(v, t, v, t_1) \in [\![(\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0,0])\ /\ s_i]\!]_C$. By definition of s_{i-1}, we conclude that $(v, t', v, t') \in [\![s_{i-1}]\!]_C$, which was to be shown.
- If $Q_{i-1} = \forall$, recall that

$$s_{i-1} \quad = \quad \neg?\left(\left(\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0,0]\right)\ /\ (\neg s_i)\right).$$

  Now, by definition of s_{i-1}, we have that $(v, t', v, t') \in [\![s_{i-1}]\!]_C$ if and only if

$$(v, t') \quad \not\models \quad ?\left(\left(\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0,0]\right)\ /\ (\neg s_i)\right).$$

  This, in turn, is equivalent to the fact that there is no time point $t_1 \in \Omega$ such that $(v, t', v, t_1) \in [\![\left(\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0,0]\right)\ /\ (\neg s_i)]\!]_C$. Hence, we know that $(v, t', v, t') \in [\![s_{i-1}]\!]_C$ if and only if, for every time point $t_1 \in \Omega$:

$$(v, t', v, t_1) \notin [\![\left(\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0,0]\right)\ /\ (\neg s_i)]\!]_C.$$

  This condition means that each t_1 satisfies $(v, t', v, t_1) \notin [\![\left(\mathbf{N}[2^{i-2}, 2^{i-2}] + \mathbf{N}[0,0]\right)]\!]_C$ or $(v, t_1) \not\models (\neg s_i)$. This, in turn, is equivalent to saying that if $t_1 \in \{t', t' + 2^{i-2}\}$ then $(v, t_1) \not\models (\neg s_i)$, *i.e.*, $(v, t_1) \models s_i$. As a consequence, $(v, t', v, t') \in [\![s_{i-1}]\!]_C$ if and only if $(v, t') \models s_i$ and $(v, t' + 2^{i-2}) \models s_i$, which is equivalent to having that $(v, t', v, t') \in [\![s_i]\!]_C$ and $(v, t' + 2^{i-2}, v, t' + 2^{i-2}) \in [\![s_i]\!]_C$.
  To prove the direction ($\Rightarrow$) of (8) suppose that $(v, t', v, t') \in [\![s_{i-1}]\!]_C$, so we also have that $(v, t', v, t') \in [\![s_i]\!]_C$ and $(v, t' + 2^{i-2}, v, t' + 2^{i-2}) \in [\![s_i]\!]_C$. Notice then that $t' < 2^{i-2}$, so t' and $t' + 2^{i-2}$ are both smaller than 2^{i-1}. By induction hypothesis, we get then that both quantified Boolean formulae $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-1}), x_i, \dots, x_n)$ and $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'+2^{i-2}}(x_1), \dots, \sigma_{t'+2^{i-1}}(x_{i-1}), x_i, \dots, x_n)$ are valid. Furthermore, since t' is smaller than 2^{i-2}, $\sigma_{t'}(x_{i-1}) = \text{false}$, and since $t' + 2^{i-2}$ only differs from t' in its $(i-1)$-th bit, which is 1, we get that $\sigma_{t'+2^{i-2}}(x_{i-1}) = \text{true}$ and, for every $j \in \{1, \dots, i-2\}$, it holds that $\sigma_{t'}(x_j) = \sigma_{t'+2^{i-2}}(x_j)$. Hence, both quantified Boolean formulae $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), \text{false}, x_i, \dots, x_n)$ and $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), \text{true}, x_i, \dots, x_n)$ are valid. Therefore, we conclude that the quantified Boolean formula $\forall x_{i-1} Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), x_{i-1}, x_i, \dots, x_n)$ is valid.
  To show the direction ($\Leftarrow$) of (8) suppose that $\forall x_{i-1} Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), x_{i-1}, x_i, \dots, x_n)$ is valid. Then we get that both quantified Boolean formulae $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), \text{false}, x_i, \dots, x_n)$ and $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-2}), \text{true}, x_i, \dots, x_n)$ are valid. As before, since t' is smaller than 2^{i-2}, $\sigma_{t'}(x_{i-1}) = \text{false}$, and since $t' + 2^{i-2}$ only differs from t' in its $(i-1)$-th bit, which is 1, we get that $\sigma_{t'+2^{i-2}}(x_{i-1}) = \text{true}$ and for every $j \in \{1, \dots, i-2\}$, it holds that $\sigma_{t'}(x_j) = \sigma_{t'+2^{i-2}}(x_j)$. This allows us to conclude that both $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'}(x_1), \dots, \sigma_{t'}(x_{i-1}), x_i, \dots, x_n)$ and $Q_i x_i \dots Q_n x_n \varphi(\sigma_{t'+2^{i-2}}(x_1), \dots, \sigma_{t'+2^{i-1}}(x_{i-1}), x_i, \dots, x_n)$ are valid. Finally, by induction hypothesis, this implies that $(v, t', v, t') \in [\![s_i]\!]_C$ and $(v, t' + 2^{i-2}, v, t' + 2^{i-2}) \in [\![s_i]\!]_C$, which, as shown before, holds if and only if $(v, t', v, t') \in [\![s_{i-1}]\!]_C$, which concludes the proof for this case.

As we mentioned, all this together implies that TupleEval(NavL[PC, NOI]) is Pspace-hard, since CTPG C, expression s_1 and tuple $(v, 0, v, 0)$ can be constructed in polynomial time in the size of ψ, and the problem of determining whether ψ is valid can be reduced to the problem of verifying whether $(v, 0, v, 0) \in [\![s_1]\!]_C$.

Thus, it only remains to show that TupleEval(NavL[PC, NOI]) is in Pspace. To do this, we will provide an algorithm in Pspace that, given a CTPG C, an expression r in NavL[PC,NOI], and a tuple $(o, t, o', t') \in \mathrm{PTO}(C)$, computes whether $(o, t, o', t') \in [\![r]\!]_C$.

Algorithm 3: TupleEvalSolve $(C, r, (o_1, t_1, o_2, t_2))$ (part I)

Input : A CTPG $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$, an expression r in NavL[PC, NOI] and a pair of temporal objects $(o_1, t_1, o_2, t_2) \in \mathrm{PTO}(C)$
Output: true if and only if $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$

```
if r is a test then
    if (o1, t1) ≠ (o2, t2) then
        return false
    if r = Node then
        return (o1 ∈ N)
    else if r = Edge then
        return (o1 ∈ E)
    else if r = ℓ for some ℓ ∈ Lab then
        return λ(o1) = ℓ
    else if r = p ↦ v for some p ∈ Prop and v ∈ Val then
        foreach valued interval (v', I) ∈ σ(o1, p) do
            if t1 ∈ I then
                return (v' = v)
    else if r = < k with k ∈ Ω then
        return (t1 < k)
    else if r = ∃ then
        foreach interval I ∈ ξ(o1) do
            if t1 ∈ I then
                return true
    else if r = (?r') then
        foreach (o', t') ∈ (N ∪ E) × Ω do
            if TupleEvalSolve(C, r', (o1, t1, o', t')) then
                return true
    else if r = (test1 ∨ test2) then
        return TupleEvalSolve(C, test1, (o1, t1, o1, t1)) or TupleEvalSolve(C, test2, (o1, t1, o1, t1))
    else if r = (test1 ∧ test2) then
        return TupleEvalSolve(C, test1, (o1, t1, o1, t1)) and TupleEvalSolve(C, test2, (o1, t1, o1, t1))
    else if r = (¬r') then
        return not TupleEvalSolve(C, r', (o1, t1, o1, t1))
else if r = N then
    return o1 = o2 and t2 = t1 + 1
else if r = P then
    return o1 = o2 and t2 = t1 − 1
else if r = F then
    return t1 = t2 and ((o1 ∈ E and o2 = tgt(o1)) or (o2 ∈ E and o1 = src(o2)))
else if r = B then
    return t1 = t2 and ((o1 ∈ E and o2 = src(o1)) or (o2 ∈ E and o1 = tgt(o2)))
else if r = (r1 + r2) then
    return TupleEvalSolve(C, r1, (o1, t1, o2, t2)) or TupleEvalSolve(C, r2, (o1, t1, o2, t2))
else if r = (r1 / r2) then
    foreach (o', t') ∈ (N ∪ E) × Ω do
        if TupleEvalSolve(C, r1, (o1, t1, o', t')) and TupleEvalSolve(C, r2, (o', t', o2, t2)) then
            return true
```

We will prove that this algorithm is in Pspace by showing that, for every CTPG $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$, expression r in NavL[PC, NOI] and tuple $(o_1, t_1, o_2, t_2) \in \mathrm{PTO}(C)$, it holds that $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$ if and only if TupleEvalSolve $(C, r, (o_1, t_1, o_2, t_2))$ returns true, all while using only polynomial space.

Notice firstly that the algorithm is recursive, and that the depth of the recursion is polynomial, since at every step on which the algorithm is called, the size of the path expression strictly decreases. There is one exception, that happens when r is of the form path$[n, _]$. Notice

Algorithm 3: TupleEvalSolve $(C, r, (o_1, t_1, o_2, t_2))$ (part II)

```
else if r = r'[n, m] then
    if m = n then
        if n = 0 then
            return (o1, t1) = (o2, t2)
        else if n = 1 then
            return TupleEvalSolve(C, r', (o1, t1, o2, t2))
        l ← ⌊n/2⌋
        if m is even then
            foreach (o', t') ∈ (N ∪ E) × Ω do
                if TupleEvalSolve(C, r'[l, l], (o1, t1, o', t')) and TupleEvalSolve(C, r'[l, l], (o', t', o2, t2)) then
                    return true
        else if m is odd then
            foreach (o', t') ∈ (N ∪ E) × Ω do
                foreach (o'', t'') ∈ (N ∪ E) × Ω do
                    if (
                        TupleEvalSolve(C, r'[l, l], (o1, t1, o', t')) and
                        TupleEvalSolve(C, r', (o', t', o'', t'')) and
                        TupleEvalSolve(C, r'[l, l], (o', t', o2, t2))
                    ) then
                        return true
    else if n = 0 then
        if m = 1 then
            return(o1, t1) = (o2, t2) or TupleEvalSolve(C, r', (o1, t1, o2, t2))
        l ← ⌊m/2⌋
        if m is even then
            foreach (o', t') ∈ (N ∪ E) × Ω do
                if TupleEvalSolve(C, r'[0, l], (o1, t1, o', t')) and TupleEvalSolve(C, r'[0, l], (o', t', o2, t2)) then
                    return true
        else if m is odd then
            foreach (o', t') ∈ (N ∪ E) × Ω do
                foreach (o'', t'') ∈ (N ∪ E) × Ω do
                    if (
                        TupleEvalSolve(C, r'[0, l], (o1, t1, o', t')) and
                        TupleEvalSolve(C, r'[0, 1], (o', t', o'', t'')) and
                        TupleEvalSolve(C, r'[0, l], (o', t', o2, t2))
                    ) then
                        return true
    else
        foreach (o', t') ∈ (N ∪ E) × Ω do
            if TupleEvalSolve(C, r'[n, n], (o1, t1, o', t')) and TupleEvalSolve(C, r'[0, m − n], (o', t', o2, t2)) then
                return true
else if r = r'[n, _] then
    m ← n + (|Ω| · |N ∪ E|)^2
    return TupleEvalSolve(C, r'[n, m], (o1, t1, o2, t2))
return false
```

that here this expression is treated as if it was path$[n, m]$, where $m = n + |\Omega| \cdot |N \cup E|$ (a term with polynomial size with respect to the input). Thus, the whole expression r can be thought as an equivalent expression r' where all terms of the form path$[n, _]$ are replaced with similar ones, in a manner that makes the whole input remain polynomial to the original. Although it might seem that path$[n, m]$ could reach an exponential number of recursive calls, notice that this expression is always parsed as two expressions, path$[n, n]$ and path$[0, m - n]$, and then each of those is solved in a way similar to that of exponentiation by squaring, which allows to always get rid of the numerical

occurence indicator $[n, m]$ after at most $O(\log m)$ recursive calls, a number that is polynomial in the size of the input. Hence the recursion tree has polynomial height.

Secondly, assume that conjonctions ($\wedge$) and disjunctions ($\vee$) are computed from left to right, *i.e.*, $A(x) \wedge A(y)$ first computes $A(x)$ and then computes $A(y)$. Hence, at the most, we will need to have in memory as many calls to the algorithm as the recursion tree height. This number is polynomial, and since every non-recursive step is either a non-deterministic guess or clearly in polynomial time in the size of the input, we get that the whole algorithm gives an answer in PSPACE.

Now, let $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$ be a coalesced TGraph, let r be an expression in NavL[PC, NOI] and let $(o_1, t_1, o_2, t_2) \in \text{PTO}(C)$ be a tuple concatenating two temporal objects. First suppose that $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$. We will show, by induction on the recursion level of the recursion tree, that there exists an execution of algorithm TUPLEEVALSOLVE $(C, r, (o_1, t_1, o_2, t_2))$ that returns true. Notice that the base case is given for those cases where there is no recursion.

The base test cases, *i.e.*, when r is equal to either **N**, **P**, **F**, **B**, **Node**, **Edge**, ℓ, $p \mapsto v$, $< k$, or $\exists$, are easily checked, since all the algorithm does is checking their definitions over the temporal object (o_1, t_1) after checking that it is equal to the temporal object (o_2, t_2). Notice that for property-value checking and existence checking, the default value is false, returned at the end of the algorithm.

The base navigation operators **N**, **P**, **F** and **B** are also easily checked by their definitions. For time navigation, we check that the objects are the same and that their associated times are consecutive, whereas for spatial navigation, we check that the times are equal, and that the respective objects are consecutive, by looking at the functions tgt and src, as defined by the operators.

As for the recursive cases, assume that the property holds up to recursion level n and we want to prove that it holds at recursion level $n - 1$ (one level higher in the recursion tree).

Firstly, a path expression matching any of the regular expressions (test $\wedge$ test), (test $\vee$ test) or ($\neg$test) can also be checked quite straightforwardly by definition, and since the flow is deterministic, we will omit further formal proofs.

Secondly, if the path expression r is of the form $(?r')$, then we know that $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$ if and only $(o_1, t_1) = (o_2, t_2)$ and there exists a temporal object $(o', t') \in \text{PTO}(C)$ such that $(o_1, t_1, o', t') \in [\![r']\!]_C$. In such case, the algorithm iterates one by one over the possible temporal objects to find one that satisfies the condition. If such temporal object exists, the call to TUPLEEVALSOLVE returns true, since we then know that the tuple (o_1, t_1, o', t') satisfies r' if and only if there exist an execution of the call that returns true. Conversely, if no such temporal object exists, all recursive calls to TUPLEEVALSOLVE will return false by induction hypothesis. In this case, the algorithm will finish the loop without returning and it will then reach the last line (in part II), returning false.

As for regular path expressions r of the form $(r_1 + r_2)$ where r_1 and r_2 are also regular path expressions, we know that by definition $[\![(r_1 + r_2)]\!]_C = [\![r_1]\!]_C \cup [\![r_2]\!]_C$. Hence, $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$ if and only if $(o_1, t_1, o_2, t_2) \in [\![r_1]\!]_C$ or $(o_1, t_1, o_2, t_2) \in [\![r_2]\!]_C$. By induction hypothesis, this means that $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$ if and only if either TUPLEEVALSOLVE$(C, r_1, (o_1, t_1, o_2, t_2))$ returns true or TUPLEEVALSOLVE$(C, r_2, (o_1, t_1, o_2, t_2))$ returns true. As the algorithm returns the disjunction of this two results, we get that $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$ if and only if TUPLEEVALSOLVE$(C, r, (o_1, t_1, o_2, t_2))$ returns true

For regular path expressions r of the form $(r_1 \,/\, r_2)$ where r_1 and r_2 are also TRPQs, we know that if $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$, then there must exist a temporal object $(o', t') \in \text{PTO}(C)$ such that $(o_1, t_1, o', t') \in [\![r_1]\!]_C$ and $(o', t', o_2, t_2) \in [\![r_2]\!]_C$. Thus, by iterating over all temporal object (o', t'), when we reach that exact temporal object, both TUPLEEVALSOLVE$(C, r_1, (o_1, t_1, o', t'))$ and TUPLEEVALSOLVE$(C, r_2, (o', t', o_2, t_2))$ will be true, so the algorithm will return true**and**true = true. On the other hand, if $(o_1, t_1, o_2, t_2) \notin [\![r]\!]_C$, then no matter what temporal object (o', t') is being iterated, we will either have that $(o_1, t_1, o', t') \notin [\![r_1]\!]_C$ or $(o', t', o_2, t_2) \notin [\![r_2]\!]_C$. By induction hypothesis, this means that, for every execution, either the first call will be false or the second will, so the condition that makes the algorithm return true will not be met. Hence, the last line is reached and the algorithm returns false.

For expressions r matching regular expressions with numerical occurrence indicators of the form $r'[n, m]$, recall that, by definition, $[\![r'[n, m]]\!]_C = \bigcup_{k=n}^{m} [\![r'^k]\!]_C$. This implies that $(o_1, t_1, o_2, t_2) \in [\![r'[n, m]]\!]_C$ if and only if there exists an integer $k \in [n, m]$ such that $(o_1, t_1, o_2, t_2) \in [\![r'^k]\!]_C$. We split this case into three cases.

(1) When $n = m$, then $(o_1, t_1, o_2, t_2) \in [\![r'[n, m]]\!]_C$ if and only if $(o_1, t_1, o_2, t_2) \in [\![r'^n]\!]_C$. Recall that the concatenation operator is associative, and that $[\![r'^n]\!]_C = [\![r' \,/\, r'^{n-1}]\!]_C = [\![r' \,/\, \ldots \,/\, r']\!]_C$ (n repetitions). Hence, if we define $l = \lfloor n/2 \rfloor$, then if n is even, $[\![r'^n]\!]_C = [\![(r'^l \,/\, r'^l)]\!]_C$, whereas if n is odd, $[\![r'^n]\!]_C = [\![(r'^l \,/\, r' \,/\, r'^l)]\!]_C$.
In the first case then, by the definition of concatenation, $(o_1, t_1, o_2, t_2) \in [\![(r'^l \,/\, r'^l)]\!]_C$ if and only if there exists a temporal object (o', t') such that $(o_1, t_1, o', t') \in [\![r'^l]\!]_C$ and $(o', t', o_2, t_2) \in [\![r'^l]\!]_C$. By induction hypothesis this is equivalent to having that there exists a temporal object (o', t') such that both TUPLEEVALSOLVE$(C, r'[l, l], (o_1, t_1, o', t'))$ and TUPLEEVALSOLVE$(C, r'[l, l], (o', t', o_2, t_2))$ return true, since $[\![r'^l]\!]_C = [\![r'[l, l]]\!]_C$. Since the algorithm iterates over all possible temporal objects to check this condition, if such temporal object exists, it will return true, if it does not, then it will reach the last line and return false.
The second case is similar, except that now we need two temporal objects as there are two concatenations, which means that $(o_1, t_1, o_2, t_2) \in [\![(r'^l \,/\, r' \,/\, r'^l)]\!]_C$ if and only if there exist two temporal objects (o', t') and (o'', t'') such that $(o_1, t_1, o', t') \in [\![r'^l]\!]_C$, $(o', t', o'', t'') \in [\![r']\!]_C$ and $(o'', t'', o_2, t_2) \in [\![r'^l]\!]_C$. By induction hypothesis, and recalling again that $[\![r'^l]\!]_C = [\![r'[l, l]]\!]_C$, that means that $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$ if and only if there exist two temporal objects (o', t') and (o'', t'') such that the calls TUPLEEVALSOLVE$(C, r'[l, l], (o_1, t_1, o', t'))$, TUPLEEVALSOLVE$(C, r', (o', t', o'', t''))$ and TUPLEEVALSOLVE$(C, r'[l, l], (o'', t'', o_2, t_2))$ return true. Again, since the algorithm iterates over all possible pairs of temporal objects (o', t') and (o'', t'') to check this condition, if such temporal objects exist, it will return true, if it does not, then it will reach the last line and return false.

Since this recursion must stop at some point, the base case $n = 1$ is included, in which case r is $r'[1,1]$, which is equivalent to r', since in such case we know that $(o_1, o_2, t_1, t_2) \in [\![r]\!]_C$ if and only if $(o_1, t_1, o_2, t_2) \in [\![r']\!]_C$. By hypothesis induction, this happens if and only if TupleEvalSolve$(C, r', (o_1, t_1, o_2, t_2))$ returns true, which is why the algorithm returns that result.
Finally, the recursion works for $n \geq 2$. The case when $n = 1$ is covered as a base case, so it only remains to look for the case when $n = 0$. For such case, recall that $r'^0 = (\exists \vee \neg\exists)$, *i.e.*, a test that is a tautology. Hence, $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$ if and only if $(o_1, t_1) = (o_2, t_2)$, which is what the algorithm tests.

(2) When $n = 0$, the base case will be slightly different. We can assume that $n \neq m$ since the case where $n = m$ was already covered. Hence, the base case only needs to consider the value $m = 1$. In this case, we have that $[\![r]\!]_C = [\![r'[0,1]]\!]_C = [\![r'^0]\!]_C \cup [\![r'^1]\!]_C$. As we already discussed, checking whether $(o_1, t_1, o_2, t_2) \in [\![r'^0]\!]_C$ comes down to checking whether $(o_1, t_1) = (o_2, t_2)$, and also $[\![r'^1]\!]_C = [\![r']\!]_C$. By induction, we know that $(o_1, t_1, o_2, t_2) \in [\![r']\!]_C$ if and only if TupleEvalSolve$(C, r', (o_1, t_1, o_2, t_2))$ returns true. Since the algorithm returns true if either of these conditions hold, this case is correctly covered.
As for the recursive case, it is very similar to the previous one. If we define again $l = \lfloor m/2 \rfloor$, we can notice that, if m is even, then $[\![r'[0,m]]\!]_C = [\![(r'[0,l] \,/\, r'[0,l])]\!]_C$, whereas if m is odd, then $[\![r'[0,m]]\!]_C = [\![(r'[0,l] \,/\, r' \,/\, r'[0,l])]\!]_C$.
To show the part when m is even, notice that $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$ if and only if $(o_1, t_1, o_2, t_2) \in [\![r'^i]\!]_C$ for some $i \in [0, m]$. For every $i \in [0, m]$ there exist two integers, $j_i := \lfloor i/2 \rfloor$ and $k_i := \lceil i/2 \rceil$, both in the interval $[0, l]$, that satisfy that $j_i + k_i = i$. Since the concatenation operator is associative, this means that $[\![r'_i]\!]_C = [\![(r'^{j_i} \,/\, r'^{k_i})]\!]_C$, which in turn implies that $[\![r]\!]_C = \bigcup_{i=0}^{m} [\![r'_i]\!]_C = \bigcup_{i=0}^{m} [\![(r'^{j_i} \,/\, r'^{k_i})]\!]_C$.
Then, by definition of the concatenation operator, $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^{m} [\![(r'^{j_i} \,/\, r'^{k_i})]\!]_C$ if and only if there exists a temporal object (o', t') such that $(o_1, t_1, o', t') \in [\![r'^{j_i}]\!]_C$ and $(o', t', o_2, t_2) \in [\![r'^{k_i}]\!]_C$.
Since both j_i and k_i are bounded by l, $[\![r'^{j_i}]\!]_C \subseteq [\![\bigcup_{i=0}^{l} r'^i]\!]_C$, and $[\![r'^{k_i}]\!]_C \subseteq [\![\bigcup_{i=0}^{l} r'^i]\!]_C$, so any tuple (o, t, o', t') in $[\![r'^{k_i}]\!]_C$ or $[\![r'^{j_i}]\!]_C$ will also be in $[\![\bigcup_{i=0}^{l} r'^i]\!]_C = [\![r'[0,l]]\!]_C$. Hence, $(o_1, t_1, o', t') \in [\![r'^{j_i}]\!]_C$ implies that $(o_1, t_1, o', t') \in [\![r'[0,l]]\!]_C$ and $(o', t', o_2, t_2) \in [\![r'^{k_i}]\!]_C$ implies that $(o', t', o_2, t_2) \in [\![r'[0,l]]\!]_C$. It can then be inferred that having that $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^{m} [\![(r'^{j_i} \,/\, r'^{k_i})]\!]_C$ can only hold if there exists a temporal object (o', t') satisfying that $(o_1, t_1, o', t') \in [\![r'^{\lfloor}0,l]]\!]_C$ and $(o', t', o_2, t_2) \in [\![r'[0,l]]\!]_C$. As a result, $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^{n} [\![(r'^{j_i} \,/\, r'^{k_i})]\!]_C$ implicates that $(o_1, t_1, o_2, t_2) \in [\![(r'[0,l] \,/\, r'[0,l])]\!]_C$. In consequence, we get that $\bigcup_{i=0}^{n} [\![(r'^{j_i} \,/\, r'^{k_i})]\!]_C \subseteq [\![(r'[0,l] \,/\, r'[0,l])]\!]_C$.
For the inverse inclusion, notice that if $(o_1, t_1, o_2, t_2) \in [\![(r'[0,l] \,/\, r'[0,l])]\!]_C$, then there must exist a temporal object (o', t') and two integers j and k in $[0, l]$ such that $(o_1, t_1, o', t') \in [\![r'^j]\!]_C$ and $(o', t', o_2, t_2) \in [\![r'^k]\!]_C$, which implies that $(o_1, t_1, o_2, t_2) \in [\![(r'^j \,/\, r'^k)]\!]_C$. Again, since concatenation is associative, $[\![(r'^i \,/\, r'^j)]\!]_C = [\![r'^{i+j}]\!]_C$, and because $i + j$ is at most n, $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^{n} [\![r'^i]\!]_C$, *i.e.*, $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$. As a result, we get that $[\![(r'[0,l] \,/\, r'[0,l])]\!]_C \subseteq \bigcup_{i=0}^{n} [\![(r'^{j_i} \,/\, r'^{k_i})]\!]_C$.
Combining these two inclusions with the equality $[\![r]\!]_C = \bigcup_{i=0}^{n} [\![(r'^{j_i} \,/\, r'^{k_i})]\!]_C$, we get that $[\![r]\!]_C = [\![(r'[0,l] \,/\, r'[0,l])]\!]_C$.
To show the part where n is odd, notice that $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$ if and only if there exists an integer $i \in [0, m]$ such that $(o_1, t_1, o_2, t_2) \in [\![r'^i]\!]_C$. Notice then that i can be written as the sum of three integers, $k_i := \lfloor i/2 \rfloor$ $(\in [0, l])$, $j_i =: \lfloor i/2 \rfloor$ $(\in [0, l])$ and $s_i := (i \mathit{mod} 2)$ $(\in [0, 1])$. Since the concatenation operator is associative, this means that $[\![r'_i]\!]_C = [\![(r'^{j_i} \,/\, r'^{s_i} \,/\, r'^{k_i})]\!]_C$, which in turn implies that $[\![r]\!]_C = \bigcup_{i=0}^{m} [\![r'_i]\!]_C = \bigcup_{i=0}^{m} [\![(r'^{j_i} \,/\, r'^{s_i} \,/\, r'^{k_i})]\!]_C$.
Then, by definition of the concatenation operator, $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^{m} [\![(r'^{j_i} \,/\, r'^{s_1} \,/\, r'^{k_i})]\!]_C$ if and only if there exist two temporal objects (o', t') and (o'', t'') such that $(o_1, t_1, o', t') \in [\![r'^{j_i}]\!]_C$, $(o', t', o'', t'') \in [\![r'^{s_i}]\!]_C$ and $(o'', t'', o_2, t_2) \in [\![r'^{k_i}]\!]_C$.
Since both j_i and k_i are bounded by l, $[\![r'^{j_i}]\!]_C \subseteq [\![\bigcup_{i=0}^{l} r'^i]\!]_C$, and $[\![r'^{k_i}]\!]_C \subseteq [\![\bigcup_{i=0}^{l} r'^i]\!]_C$, so any tuple (o, t, o', t') in $[\![r'^{k_i}]\!]_C$ or $[\![r'^{j_i}]\!]_C$ will also be in $[\![\bigcup_{i=0}^{l} r'^i]\!]_C = [\![r'[0,l]]\!]_C$. Similarly, any tuple in $[\![r'^{s_1}]\!]_C$ will also be in $[\![r'[0,1]]\!]_C$. Hence, $(o_1, t_1, o', t') \in [\![r'^{j_i}]\!]_C$ implies that $(o_1, t_1, o', t') \in [\![r'[0,l]]\!]_C$, $(o', t', o'', t'') \in [\![r'^{s_i}]\!]_C$ implies that $(o', t', o'', t'') \in [\![r'[0,1]]\!]_C$ and $(o'', t'', o_2, t_2) \in [\![r^{k_i}]\!]_C$ implies that $(o'', t'', o_2, t_2) \in [\![r[0,l]]\!]_C$. It can then be inferred that having that $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^{m} [\![(r'^{j_i} \,/\, r'^{s_i} \,/\, r'^{k_i})]\!]_C$ can only hold if there exist two temporal objects (o', t') and (o'', t'') satisfying that $(o_1, t_1, o', t') \in [\![r'^{\lfloor}0,l]]\!]_C$, $(o', t', o'', t'') \in [\![r'[0,1]]\!]_C$ and $(o'', t'', o_2, t_2) \in [\![r[0,l]]\!]_C$. As a result, $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^{n} [\![(r'^{j_i} \,/\, r'^{k_i})]\!]_C$ implicates that $(o_1, t_1, o_2, t_2) \in [\![(r'[0,l] \,/\, r'[0,1] \,/\, r'[0,l])]\!]_C$. In consequence, we get that $\bigcup_{i=0}^{n} [\![(r'^{j_i} \,/\, r'^{k_i})]\!]_C \subseteq [\![(r'[0,l] \,/\, r'[0,1] \,/\, r'[0,l])]\!]_C$.
For the inverse inclusion, notice that if $(o_1, t_1, o_2, t_2) \in [\![(r'[0,l] \,/\, r'[0,1] \,/\, r'[0,l])]\!]_C$, then there must exist two temporal objects (o', t') and (o'', t''), and three integers $j \in [0, l]$, $s \in [0, 1]$ and $k \in [0, l]$ such that $(o_1, t_1, o', t') \in [\![r'^j]\!]_C$, $(o'', t'') \in [\![r'^s]\!]_C$ and $(o'', t'', o_2, t_2) \in [\![r'^k]\!]_C$, which implies that $(o_1, t_1, o_2, t_2) \in [\![(r'^j \,/\, r'^s \,/\, r'^k)]\!]_C$. Again, since concatenation is associative, $[\![(r'^i \,/\, r'^s \,/\, r'^j)]\!]_C = [\![r'^{i+s+j}]\!]_C$, and because $i + s + j$ is at most n, $(o_1, t_1, o_2, t_2) \in \bigcup_{i=0}^{n} [\![r'^i]\!]_C$, *i.e.*, $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$. As a result, we get that $[\![(r'[0,l] \,/\, r'[0,1] \,/\, r'[0,l])]\!]_C \subseteq \bigcup_{i=0}^{n} [\![(r'^{j_i} \,/\, r'^{k_i})]\!]_C$.
As before, combining these two inclusions with the equality $[\![r]\!]_C = \bigcup_{i=0}^{n} [\![(r'^{j_i} \,/\, r'^{l_i} \,/\, r'^{k_i})]\!]_C$, we get that $[\![r]\!]_C = [\![(r'[0,l] \,/\, r'[0,1] \,/\, r'[0,l])]\!]_C$. With these two results in mind then, *i.e.*, that $[\![r'[0,m]]\!]_C = [\![(r'[0,l] \,/\, r'[0,l])]\!]_C$ when m is even, whereas if m is odd, then $[\![r'[0,m]]\!]_C = [\![(r'[0,l] \,/\, r' \,/\, r'[0,l])]\!]_C$, we know that $(o_1, t_1, o_2, t_2) \in [\![r'[0,m]]\!]_C$ if and only if (i) m is even and there exists a temporal object (o', t') such that $(o_1, t_1, o', t') \in [\![r'[0,l]]\!]_C$ and $(o', t', o_2, t_2) \in [\![r'[0,l]]\!]_C$, or (ii) m is

odd and there exist two temporal objects (o', t') and (o'', t'') such that $(o_1, t_1, o', t') \in [\![r'[0, l]]\!]_C$, $(o', t', o'', t'') \in [\![r'[0, 1]]\!]_C$ and $(o'', t'', o_2, t_2) \in [\![r'[0, l]]\!]_C$.
By induction, (i) holds if and only if there exists a temporal object (o', t') such that both TupleEvalSolve$(C, r'[0, l], (o_1, t_1, o', t'))$ and TupleEvalSolve$(C, r'[0, l], (o', t', o_2, t_2))$ return true. Since the algorithm iterates over all temporal objects (o', t') for this case, and then checks that both those conditions are met to return true, it will return true if (i) holds, and it will reach the last line and return false if no such pair existed.
Also by induction, (ii) holds if and only if there exist two temporal objects (o', t') and (o'', t'') such that the three calls TupleEvalSolve$(C, r'[0, l], (o_1, t_1, o', t'))$, TupleEvalSolve$(C, r'[0, l], (o', t', o'', t''))$ and TupleEvalSolve$(C, r'[0, l], (o'', t'', o_2, t_2))$ return true. Here again, since the algorithm iterates over all pairs of temporal objects (o', t') and (o'', t'') and sees if these three conditions are met to return true, it will return true if (ii) holds, and it will reach the last line and return false otherwise.
Hence, when $n = 0$, the algorithm also returns true if and only if $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$

(3) When $m \neq n$ and $n \neq 0$, then $(o_1, t_1, o_2, t_2)[\![r'[n, m]]\!]_C$ if and only if there exists $i \in [n, m]$ such that $(o_1, t_1, o_2, t_2)[\![r'^i]\!]_C$. In this case, $i = n+d$ for some $d \in [0, m-n]$, and since the concatenation operator is associative, $(o_1, t_1, o_2, t_2)[\![r'^i]\!]_C$ if and only if there exists (o', t') such that $(o_1, t_1, o', t') \in [\![r'^n]\!]_C$ and $(o', t', o_2, t_2) \in [\![r'^d]\!]_C$. By induction, and since $[\![r'^n]\!]_C = [\![r'[n, n]]\!]_C$, $(o_1, t_1, o', t') \in [\![r'^n]\!]_C$ if and only if TupleEvalSolve$(C, r'[n, n], (o_1, t_1, o', t'))$. Similarly, $(o', t', o_2, t_2) \in [\![r'^d]\!]_C$ for some $d \in [0, m - n]$ if and only if $(o', t', o_2, t_2) \in [\![r'[0, m - n]]\!]_C$, which by induction holds if and only if TupleEvalSolve$(C, r'[0, m - n], (o', t', o_2, t_2))$.
Together, this means that $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$ if and only if there exists a temporal object (o', t') such that TupleEvalSolve$(C, r'[n, n], (o_1, t_1, o', t')$ and TupleEvalSolve$(C, r'[0, m - n], (o', t', o_2, t_2))$. Since the algorithm iterates over all temporal objects (o', t') and checks if these conditions are met to return true, it will return true if $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$ and it will reach the last line and return false otherwise.

Finally, for expressions r matching regular expressions with numerical occurrence indicators of the form $r'[n, \_]$, recall that we showed on section 5.3 that $[\![r'[n, \_]]\!]_C = [\![r'[n, n + (|\Omega| + |V \cup E|)^2]]\!]_C$, where the expression $m := n + (|\Omega| + |V \cup E|)^2$ is polynomial in the size of the original input.

By induction, $(o_1, t_1, o_2, t_2) \in [\![r'[n, m]]\!]_C$ if and only if TupleEvalSolve$(C, r'[n, m], (o_1, t_1, o_2, t_2))$ returns true, which is what the algorithm returns for this case.

Altogether, we proved that TupleEvalSolve is in Pspace and that TupleEvalSolve$(C, r, (o_1, t_1, o_2, t_2))$ returns true if and only if $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$. This completes the proof needed to show that TupleEval(NavL[PC, NOI]) is in Pspace. This, along, with the previous result, implies Pspace-completeness, so the theorem follows.

A.2 Proof of Theorem 6.3

First of all notice that basic tests such as **Node**, **Edge**, ℓ, $p \mapsto v$, $< k$ and $\exists$ can be checked in $O(1)$, so in absence of numerical occurrence indicators, checking a test is equivalent to checking the satisfaction of a Boolean formula on a given valuation, which can be done efficiently in time $O(n)$, where n is the size of the formula. We will assume in the following then that we have a linear-time function CheckTestNoPC$(C, (o, t), \text{test})$ that takes as input a CTPG C, a temporal object (o, t) in C and a test expression test in NavL[PC], and returns true if $(o, t) \models$ test in C, and false otherwise.

To show that TupleEval(NavL[PC]) is in PTIME, we present a polynomial-time procedure in Algorithm 4, called TupleEvalSolveOnlyPC, that, given a CTPG C, a tuple representing a pair of temporal objects (o_1, t_1, o_2, t_2) and an expression r in NavL[PC], checks whether $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$. In what follows, we show that Algorithm 4 works in polynomial time.

Notice first that we do not directly return the result. Instead, we first look at a hashing table that stores previously stored results, and only if this was not previously computed, we compute the result for the input, and store it in the table before returning the value. We employ this to avoid an exponential number of calls when recursively calling the algorithm. This is possible since, in absence of numerical occurrence indicators, navigation is done at most one step at a time. Thus, if (o_2, t_2) is a temporal object reached from (o_1, t_1) using an expression r, then $|t_1 - t_2|$ is at most the number of symbols **N** and **P** occurring in r. Hence, if $\|r\|$ is the length of expression r, then there are at most $O(\|r\| \cdot |N \cup E|)$ temporal objects that we will need to consider for this call, which means, at most $O(\|r\|^2 \cdot |N \cup E|^2)$ tuples representing pairs of temporal objects. Hence, given that there are at most $\|r\|$ sub-expressions of r that can be reached in the tree decomposition of r, we need to store at most $O(\|r\|^3 \cdot |N \cup E|^2)$ different results for TupleEvalSolveOnlyPC.

The rest of the algorithm is quite straightforward, and it considers the case when the result has not been precomputed. If r is a temporal navigation operator, then by the definitions of **N** and **P**, a single temporal object can be reached, $(o_1, t_1 + 1)$ or $(o_1, t_1 - 1)$, respectively, so we check that (o_2, t_2) is equal to that respective temporal object. This can be easily done in $O(1)$. If r is a spatial navigation operator, then we have to look at the mapping from edges to source and destination nodes, ρ, to determine the set of objects that can be reached. If o is an edge, and we move forward, we look for its destination node, if we move backward, for its source node. If o is a node, and we move forward, we are looking for the edges that have o as their source, and if we move backward, then we look for those who have o as their destination. The whole process can be done in time $O(\|\rho\|)$, which is $O(\|C\|)$. When r is testing a condition, we just call the previously mentioned algorithm CheckTestNoPC to check whether $(o, t) \models r$ in C. This last base case can be done in time $O(\|r\|)$.

Algorithm 4: TupleEvalSolveOnlyPC$(C, (o_1, t_1, o_2, t_2), r)$

Input : A CTPG $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$, an expression r in NavL[PC] and a pair of temporal objects $(o_1, t_1, o_2, t_2) \in \text{PTO}(C)$
Output : true if $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$, and false otherwise
Initialization : A global variable H initially empty, storing a hash table with the currently computed values for TupleEvalSolveOnlyPC.

if (o_1, t_1, o_2, t_2, r) *is a key in hash table* H **then**
  return $H[(o_1, t_1, o_2, t_2, r)]$
if $r =$ **N then**
  $A \leftarrow (o_1 = o_2$ **and** $t_2 = t_1 + 1)$
else if $r =$ **P then**
  $A \leftarrow (o_1 = o_2$ **and** $t_2 = t_1 - 1)$
else if $r =$ **F then**
  $A \leftarrow (t_1 = t_2$ **and** $((o_1 \in E$ **and** $o_2 = \text{tgt}(o_1))$ **or** $(o_2 \in E$ **and** $o_1 = \text{src}(o_2))))$
else if $r =$ **B then**
  $A \leftarrow (t_1 = t_2$ **and** $((o_1 \in E$ **and** $o_2 = \text{src}(o_1))$ **or** $(o_2 \in E$ **and** $o_1 = \text{tgt}(o_2))))$
else if r *is a test* **then**
  $A \leftarrow ((o_1, t_1) = (o_2, t_2)$ **and** CheckTestNoPC$(C, (o_1, t_1), r))$
else if $r = (r_1 + r_2)$ **then**
  $A \leftarrow$ TupleEvalSolveOnlyPC$(C, (o_1, t_1, o_2, t_2), r_1)$
  if ***not***A **then**
    $A \leftarrow$ TupleEvalSolveOnlyPC$(C, (o_1, t_1, o_2, t_2), r_2)$
else if $r = (r_1 \ / \ r_2)$ **then**
  $A \leftarrow$ false
  $l_1 \leftarrow \|r_1\|$
  $l_2 \leftarrow \|r_2\|$
  foreach $(o, t) \in (N \cup E) \times \{t \in \Omega \mid (|t - t_1| \leq l_1 \ \wedge \ |t - t_2| \leq l_2)\}$ **do**
    if *TupleEvalSolveOnlyPC*$(C, (o_1, t_1, o, t), r_1)$ **then**
      if *TupleEvalSolveOnlyPC*$(C, (o, t, o_2, t_2), r_2)$ **then**
        $A \leftarrow$ true
        break
Store the value A for TupleEvalSolveOnlyPC$(C, (o_1, t_1, o_2, t_2), r)$ in the hash table H with key (o_1, t_1, o_2, t_2, r)
return A

When r is of the form $(r_1 + r_2)$, where r_1 and r_2 are expressions in NavL[PC], we have that $(o_1, t_1, o_2, t_2) \in [\![(r_1 + r_2)]\!]_C$ if and only if $(o_1, t_1, o_2, t_2) \in [\![r_1]\!]_C$ or $(o_1, t_1, o_2, t_2) \in [\![r_2]\!]_C$, so if suffices that the call to TupleEvalSolveOnlyPC with any of inputs r_1 or r_2 returns true, and this can be easily checked by the algorithm. The last case is when r is of the form $(r_1 \ / \ r_2)$ where r_1 and r_2 are expressions in NavL[PC], where we have that $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$ if there exists a temporal object (o, t) such that $(o_1, t_1, o, t) \in [\![r_1]\!]_C$ and $(o, t, o_2, t_2) \in [\![r_2]\!]_C$. We know that in absence of numerical occurrence indicators, t will be at distance at most $\|r_1\|$ from t_1 and at most $\|r_2\|$ from t_2, since one can move only as many times as there are **N** and **P** symbols in the respective formulas, so this gives us a polynomial-size set from which we can extract candidates to satisfy this condition.

Finally, notice that at every call to TupleEvalSolveOnlyPC$(C, (o_1, t_1, o_2, t_2), r)$, we either already have computed the value for the key (o_1, t_1, o_2, t_2, r), in which case we can give an answer immediately, or we are computing a new value to store in the hash table H, which has size bounded by $O(\|r\|^3 \cdot |N \cup E|^2)$. In any case, since the most expensive step performs at most $O(|N \cup E| \cdot \|r\|)$ recursive calls, we will be getting an answer in time $O(\|r\|^4 \cdot |N \cup E|^3)$, which is polynomial in the size of the input.

A.3 Proof of Theorem 6.4

Consider the following decision problem called Generalized Subset Sum (G-SUBSET-SUM) which is known to be Σ_2^p-complete [8]:

Problem:	G-SUBSET-SUM
Input:	Natural numbers vectors u and w of dimensions $\dim(u)$ and $\dim(w)$, respectively, and a positive integer $S \in \mathbb{N}$
Output:	true if there exists $x \in \{0,1\}^{\dim(u)}$ such that, for all $y \in \{0,1\}^{\dim(w)}$, it holds that $x \cdot u + y \cdot w \neq S$, and false otherwise.

In this problem, $a \cdot b$ represents the inner product between vectors a and b. Given vectors $u = (u_1, \ldots, u_n) \in \mathbb{N}^n$ and $w = (w_1, \ldots, w_m) \in \mathbb{N}^m$, and the integer $S \in \mathbb{N}$, the goal is to provide a polynomial-time algorithm that returns a CTPG C, a tuple (o_1, t_1, o_2, t_2), and an expression r in NavL[NOI] such that $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$ if and only if $\exists x \in \{0,1\}^n \; \forall y \in \{0,1\}^m \; x \cdot u + y \cdot w \neq S$.

Let $M = 2 \cdot \left(\sum_{i=1}^{n} u_i + \sum_{j=1}^{m} w_j\right)$, which can be easily computed in polynomial time from u and w. Then C will be the CTPG $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$ where $\Omega = [0, 2 \cdot M]$, $N = \{v\}$, $E = \emptyset$, ρ is an empty function, $\lambda(v) = l$, $\xi(v) = \{[0, 2 \cdot M]\}$ and σ is an empty function. In other words, C is a CTPG consisting of only one node existing from time 0 to time $2 \cdot M$, with no edges or properties. The tuple (o_1, t_1, o_2, t_2) in our reduction will be given by $(v, M, v, 2 \cdot M)$. As for the expression r, it will be defined recursively as follows. First define an expression for each component u_i of u that will represent whether $u_i \cdot x_i$ will be chosen to be u_i or 0:

$$r_{u_i} \quad = \quad \mathbf{N}[u_i, u_i][0, 1].$$

The idea is that the time t of the temporal object that is being reached will store the sum given by $x \cdot u + y \cdot w$, plus M to avoid having negative numbers on the time dimension when testing that the result is different from S (this will be explained in more detail later). Define then an expression for u, representing the sum accumulated by the $\exists x \in \{0,1\}^n$ part of the problem:

$$r_u \quad = \quad r_{u_1} \,/\, \ldots \,/\, r_{u_n}.$$

We will now use a recursive construction to represent the sum accumulated by the $\forall y \in \{0,1\}^m$ part of the problem. First define condition $r_{t \neq S+M}$, that represents that the accumulated sum is not S:

$$r_{t \neq S+M} \quad = \quad (< S + M \vee \neg < S + M + 1)$$

By taking $r_0 := r_{t \neq S+M}$, now recursively define r_{j+1} from r_j, for $j \in \{0, \ldots, m-1\}$, as follows:

$$r_{j+1} \quad = \quad \left(\mathbf{N}[w_{j+1}, w_{j+1}] \,/\, r_j \,/\, \mathbf{P}[2 \cdot w_{j+1}, 2 \cdot w_{j+1}]\right)[2, 2] \;\; / \, \mathbf{N}[2 \cdot w_{j+1}, 2 \cdot w_{j+1}]$$

The formula $r_w = r_m$ will allow to iterate over all the accumulated sums implied by the $\forall y \in \{0,1\}^m$ part of the problem. Finally, r is defined as follows:

$$r \quad = \quad r_u \,/\, r_w \,/\, \mathbf{N}[0, \_] \,/\, (\neg < 2 \cdot M)$$

We now prove that $(v, M, v, 2 \cdot M) \in [\![r]\!]_C$ if and only if $\exists x \in \{0,1\}^n \; \forall y \in \{0,1\}^m \; x \cdot u + y \cdot w \neq S$.

Assume that both t and t' are in Ω. By induction on the definition of numerical occurrence indicators, it is easy to see that $(v, t, v, t') \in [\![\mathbf{N}[k, k]]\!]_C$ if and only if $t' = t + k$. Hence, by definition of r_{u_i}, we have that $(v, t, v, t') \in [\![r_{u_i}]\!]_C$ if and only if $t' = t$ (0 occurrences) or $t' = t + u_i$ (1 occurrence), or what is equivalent, if there exists $x_i \in \{0, 1\}$ such that $t' = t + x_i \cdot u_i$. In fact, it can be proved by induction that $(v, t, v, t') \in [\![r_u]\!]_C$ if and only if $\exists x \in \{0,1\}^n$ such that $t' = t + x \cdot u$. We will demonstrate something stronger, which is that $(v, t, v, t') \in [\![r_{u_1} \,/\, \ldots \,/ r_{u_i}]\!]_C$ if and only if there exists $(x_1, \ldots, x_i) \in \{0,1\}^i$ such that $t' = t + \sum_{k=1}^{i} x_i \cdot u_i$.

Our base case will be checking the property for r_{u_1}, which, by the same exact reasoning as above, satisfies that $(v, t, v, t') \in [\![r_{u_1}]\!]_C$ if and only if there exists $x_1 \in \{0, 1\}$ such that $t' = t + x_1 \cdot u_1$. Suppose now that $(v, t, v, t') \in [\![r_{u_1} \,/\, \ldots \,/ r_{u_i}]\!]_C$ if and only if there exists $(x_1, \ldots, x_i) \in \{0,1\}^i$ such that $t' = t + \sum_{k=1}^{i} x_i \cdot u_i$. We also know by the previous reasoning that $(v, t', v, t'') \in [\![r_{u_{i+1}}]\!]_C$ if and only if there exists $x_{i+1} \in \{0, 1\}$ such that $t'' = t' + x_{i+1} \cdot u_{i+1}$. Hence, by definition of (r_1/r_2), we get that $(v, t, v, t'') \in [\![r_{u_1} \,/\, \ldots \,/ r_{u_{i+1}}]\!]_C$ if and only if there exists (v, t') such that there exists $(x_1, \ldots, x_i) \in \{0,1\}^i$ such that $t' = t + \sum_{k=1}^{i} x_i \cdot u_i$ and there exists $x_{i+1} \in \{0, 1\}$ such that $t'' = t' + x_{i+1} \cdot u_{i+1}$, *i.e.*, if and only if there exists $(x_1, \ldots, x_{i+1}) \in \{0,1\}^{i+1}$ such that $t'' = t + \sum_{k=1}^{i+1} x_i \cdot u_i$. This yields the result.

Moreover, by definition of $[\![(r_1/r_2)]\!]_C$, we get that $(v, M, v, 2 \cdot M) \in [\![r]\!]_C$ if and only if there exists $x \in \{0,1\}^n$ such that:

$$(v, M + x \cdot u, v, 2 \cdot M) \in [\![r_v \,/\, \mathbf{N}[0, \_] \,/\, (\neg < 2 \cdot M)]\!]_C$$

Notice also that the right part of this formula is built in the following way: $(\neg < 2 \cdot M)$ is a test that is only satisfied by $(v, 2 \cdot M, v, 2 \cdot M)$, whereas $\mathbf{N}[0, \_]$ is satisfied by any tuple (v, t, v, t') such that $t \leq t'$. Hence, $(v, t, v', t') \in [\![\mathbf{N}[0, \_] \,/\, (\neg < 2 \cdot M)]\!]_C$ if and only if t is any time point in Ω and $t' = 2 \cdot M$. Thus, all we need to prove now is that there exists some time point t such that $(v, M + x \cdot u, v, t) \in [\![r_v]\!]_C$ if and only if $\forall y \in \{0,1\}^m \; x \cdot u + y \cdot w \neq S$.

First, we show by induction that if $(v, t, v, t') \in [\![r_j]\!]_C$, then $t = t'$. The base case is trivial, since r_0 is a test. For the inductive case, assume that if $(v, t, v, t') \in [\![r_j]\!]_C$, then $t' = t$. For conciseness, define $a := w_{j+1}$. In the case of j+1, we know that if $(v, t_1, v, t_2) \in [\![\mathbf{N}[a, a] \,/\, r_j \,/\, \mathbf{P}[2a, 2a]]\!]_C$, then there exist time points t_3 and t_4 such that $(v, t_1, v, t_3) \in [\![\mathbf{N}[a, a]]\!]_C$, $(v, t_3, v, t_4) \in [\![r_j]\!]_C$ and $(v, t_4, v, t_2) \in [\![\mathbf{P}[2a, 2a]]\!]_C$. Notice then that these conditions only hold respectively if $t_3 = t_1 + a$, by definition of operator $\mathbf{N}$ and numerical occurrence indicators, $t_3 = t_4$ by induction hypothesis and $t_2 = t_4 - 2a$. Thus, $t_2 = t4 - 2a = t_3 - 2a = t_1 - a$. It is clear that if $(v, t_1, v, t_5) \in [\![(\mathbf{N}[a, a] \,/\, r_k \,/\, \mathbf{P}[2a, 2a])[2, 2]]\!]_C$, then $t_5 = t_1 - 2a$. Finally, if $(v, t_5, v, t') \in [\![\mathbf{N}[2a, 2a]]\!]_C$, then $t' = t_5 + 2a$, so by definition of $[\![(r_1/r_2)]\!]_C$, we conclude that if $(v, t, v, t') \in [\![r_{j+1}]\!]_C$, then $t' = t$.

Given the conclusion in the previous paragraph, all we need to prove now is that $(v, M + x \cdot u, v, M + x \cdot u) \in [\![r_v]\!]_C$ if and only if $\forall y \in \{0,1\}^m \; x \cdot u + y \cdot w \neq S$. In fact, we will prove by induction a stronger condition:

$$\text{for every } k \in \{0, \ldots, m\}, \text{it holds that } \forall y \in \{0,1\}^k \; t + \sum_{j=1}^{k} y_j \cdot w_j \neq S \text{ if and only if } (v, M + t, v, M + t) \in [\![r_k]\!]_C$$

The case when $t = x \cdot u$ and $k = m$ yields Theorem 6.4 as a result. In the base case $k = 0$, we need to prove that $t \neq S$ if and only if $(v, M + t, v, M + t) \in [\![r_0]\!]_C$. Recall that $r_0 = r_{t \neq S+M}$. It can be easily checked that $(v, t) \models r_0$ if and only if $t \neq S + M$. Also, r_0 is a test, so $(v, t, v, t') \in [\![r_0]\!]_C$ if and only if $t = t'$ and $t \neq S + M$. Hence, we have that $(v, M + t, v, M + t) \in [\![r_0]\!]_C$ if and only if $M + t = M + t$ (which is trivially satisfied) and $M + t \neq M + S$, *i.e.*, if and only if $t \neq S$. For the inductive case, assume that for $k \in \{0, \ldots, m\}$, it holds that:

$$\forall y \in \{0,1\}^k \; t + \sum_{j=1}^{k} y_j \cdot w_j \neq S \text{ if and only if } (v, M + t, v, M + t) \in [\![r_k]\!]_C.$$

Then we have to prove that:

$$\forall y \in \{0,1\}^{k+1} \; t + \sum_{j=1}^{k+1} y_j \cdot w_j \neq S \text{ if and only if } (v, M + t, v, M + t) \in [\![r_{k+1}]\!]_C$$

To prove this, define again $a := w_{k+1}$ for conciseness. Recall that $r_{k+1} = ((\mathbf{N}[a, a] \,/\, r_k \,/\, \mathbf{P}[2a, 2a])[2, 2]/\mathbf{N}[2a, 2a])$. Notice then that $(v, M + t, v, t') \in [\![\mathbf{N}[a, a] \,/\, r_k \,/\, \mathbf{P}[2a, 2a]]\!]_C$ if and only if there exist time points t_1 and t_2 such that $(v, M + t, v, t_1) \in [\![\mathbf{N}[a, a]]\!]_C$, $(v, t_1, v, t_2) \in [\![r_k]\!]_C$ and $(v, t_2, v, t') \in [\![\mathbf{P}[2a, 2a]]\!]_C$. The first condition is equivalent to having that $t_1 = M + t + a$. The second condition implies that $t_1 = t_2$, given what we proved in the previous paragraphs. Hence, given that $(v, M + t + a, v, M + t + a) \in [\![r_k]\!]_C$, we conclude by induction hypothesis that $\forall y \in \{0,1\}^k \; t + a + \sum_{j=1}^{k} y_j \cdot w_j \neq S$. The third condition is equivalent to having that $t' = t_2 - 2a$. Altogether, this means that $(v, M + t, v, t') \in [\![\mathbf{N}[a, a] \,/\, r_k \,/\, \mathbf{P}[2a, 2a]]\!]_C$ if and only if $t' = M + t - a$ and $\forall y \in \{0,1\}^k$, it holds that $t + a + \sum_{j=1}^{k} y_j \cdot w_j \neq S$. Now, this means that $(v, M + t, v, t') \in [\![(\mathbf{N}[a, a] \,/\, r_k \,/\, \mathbf{P}[2a, 2a])[2, 2]]\!]_C$ if there exists a time point t'' such that $t'' = M + t - a$, $t' = M + (t - a) - a = M + t - 2a$, $\forall y \in \{0,1\}^k$, it holds that $t + a + \sum_{j=1}^{k} y_j \cdot w_j \neq S$, and $\forall y \in \{0,1\}^k$, it holds that $(t - a) + a + \sum_{j=1}^{k} y_j \cdot w_j \neq S$. Therefore, $(v, M + t, v, t') \in [\![(\mathbf{N}[a, a] \,/\, r_k \,/\, \mathbf{P}[2a, 2a])[2, 2]]\!]_C$ if and only if $t' = M + t - 2a$ and for every $y \in \{0,1\}^{k+1}$, it holds that $t + \sum_{j=1}^{k+1} y_j \cdot w_j \neq S$. Given that for $t_1, t_2 \in \Omega$, it holds that $(v, t_1, v, t_2) \in [\![\mathbf{N}[2a, 2a]]\!]_C$ if and only if $t_2 = t_1 + 2a$, we conclude that $(v, M + t, v, t') \in [\![(\mathbf{N}[a, a] \,/\, r_k \,/\, \mathbf{P}[2a, 2a])[2, 2] \,/\, \mathbf{N}[2a, 2a]]\!]_C$ if and only if $t' = M + t$ and $\forall y \in \{0,1\}^{k+1} \; t + \sum_{j=1}^{k+1} y_j \cdot w_j \neq S$. Finally, this gives us the result we are trying to prove, that is, $(v, M + t, v, M + t) \in [\![(\mathbf{N}[a, a] \,/\, r_k \,/\, \mathbf{P}[2a, 2a])[2, 2] \,/\, \mathbf{N}[2a, 2a]]\!]_C$ if and only if $\forall y \in \{0,1\}^{k+1} \; t + \sum_{j=1}^{k+1} y_j \cdot w_j \neq S$.

To conclude the proof of the theorem, notice that r can be constructed in polynomial time with respect to the sizes of u, v and S, so the entire reduction can be computed in polynomial time.

A.4 Proof of Theorem 6.5

To show NP-hardness, consider the following decision problem called Subset Sum (SUBSET-SUM), which is known to be NP-complete [27]:

Problem:	SUBSET-SUM
Input:	A finite set of integers $A \subsetneq \mathbb{N}$, and a positive integer $S \in \mathbb{N}$
Output:	true if there exists a subset $A' \subseteq A$ of A such that $\sum_{a \in A'} a = S$.

Given a set $A \subsetneq \mathbb{N}$, and an integer $S \in \mathbb{N}$, the goal is to provide a polynomial-time algorithm that returns a CTPG C, a tuple (o_1, t_1, o_2, t_2), and an expression r in NavL[ANOI] such that $(o_1, t_1, o_2, t_2) \in [\![r]\!]_C$ if and only if there exists $A' \subseteq A$ such that $\sum_{a \in A'} a = S$. More specifically, C will be the CTPG $(\Omega, N, E, \rho, \lambda, \xi, \sigma)$ where $\Omega = [0, S]$, $N = \{v\}$, $E = \emptyset$, ρ is an empty function, $\lambda(v) = l$, $\xi(v) = \{[0, S]\}$ and σ is an empty function. In other words, C is a CTPG consisting of only one node existing from time 0 to time S, with no edges or properties. The tuple (o_1, t_1, o_2, t_2) in our reduction will be given by $(v, 0, v, S)$. Moreover, assuming that $A = \{a_1, \ldots, a_n\}$, expression r is defined as follows:

$$r \quad = \quad (\mathbf{N}[a_1, a_1] + \mathbf{N}[0, 0]) \;/\; \cdots \;/\; (\mathbf{N}[a_n, a_n] + \mathbf{N}[0, 0])$$

Notice that CTPG C, expression r in NavL[ANOI] and tuple $(v, 0, v, S)$ can be computed in polynomial time in the sizes of A and S. Besides, it is straightforward to prove that $(v, 0, v, S) \in [\![r]\!]_C$ if and only if there exists $A' \subseteq A$ such that $\sum_{a \in A'} a = S$. This concludes the of NP-hardness of TupleEval(NavL[ANOI]).

To show that this problem is NP-complete, it only remains to show that the problem is also in NP. We present a nondeterministic algorithm that works in polynomial time, TupleEvalSolve_ANOI, that, given a CTPG C, an expression r in NavL[ANOI] and a pair of temporal objects (o_1, t_1, o_2, t_2), has a run that returns true if and only if $(o, t, o', t') \in [\![r]\!]_C$. This procedure is presented in Algorithm 5.

TupleEvalSolve_ANOI is very similar to TupleEvalSolve, so we will not discuss in detail what it does. Instead, we give an intuition of what the differences are that allow to return the right answer in non-deterministic polynomial time, instead of polynomial space. First, notice that if r is a test, then the algorithm works by solving basic tests efficiently, and then conjunctions, disjunctions and negations of tests are solved just by using directly the definition of these Boolean connectives. Hence, unlike what happens in the presence of path conditions, where we can have nested expressions with existential conditions and negations of existential conditions, sub-expressions for tests are efficiently solved by TupleEvalSolve_ANOI. Second, notice that for spatial navigation, we write the problem in terms of the reachability problem for graphs in a number of steps in a set $\{n, \ldots, m\}$. This problem can be efficiently solved by using exponentiation by squaring on the adjacency matrix. Besides, notice that for spatial navigation expressions in NavL[ANOI], we need to consider as many new objects as there are in $V \cup E$ since the time is fixed, which is why expressions $\mathbf{F}[n, \_]$ and $\mathbf{B}[n, \_]$ are equivalent to $\mathbf{F}[n, m]$ and $\mathbf{B}[n, m]$, respectively,

Algorithm 5: TupleEvalSolve_ANOI($C, (o_1, t_1, o_2, t_2), r$) (part I)

Input : A CTPG $C = (\Omega, N, E, \rho, \lambda, \xi, \sigma)$, an expression r in NavL[ANOI] and a pair of temporal objects (o_1, t_1, o_2, t_2)
Output : true if $(o, t, o', t') \in [\![r]\!]_C$

```
1  if r is a test then
2      if (o1, t1) ≠ (o2, t2) then
3          return false
4      else if r = Node then
5          return (o1 ∈ N)
6      else if r = Edge then
7          return (o1 ∈ E)
8      else if r = ℓ for some ℓ ∈ Lab then
9          return (λ(o1) = ℓ)
10     else if r = p ↦ v for some p ∈ Prop and v ∈ Val then
11         foreach valued interval (v', I) ∈ σ(o1, p) do
12             if t1 ∈ I then
13                 return v' = v
14         return false
15     else if r = < k with k ∈ Ω then
16         return (t1 < k)
17     else if r = ∃ then
18         foreach interval I ∈ ξ(o1) do
19             if t1 ∈ I then
20                 return true
21         return false
22     else if r = (test1 ∨ test2) then
23         return TupleEvalSolve (C, (o1, t1, o1, t1), test1) or TupleEvalSolve (C, (o1, t1, o1, t1), test2)
24     else if r = (test1 ∧ test2) then
25         return TupleEvalSolve (C, (o1, t1, o1, t1), test1) and TupleEvalSolve (C, (o1, t1, o1, t1), test2)
26     else if r = (¬r') then
27         return not TupleEvalSolve (C, (o1, t1, o1, t1), r')
28 else if r = N then
29     return (o1 = o2 and t2 = t1 + 1)
30 else if r = P then
31     return (o1 = o2 and t2 = t1 − 1)
32 else if r = F then
33     return (t1 = t2 and ((o1 ∈ E and o2 = tgt (o1)) or (o2 ∈ E and o1 = src (o2))))
34 else if r = B then
35     return (t1 = t2 and ((o1 ∈ E and o2 = src(o1)) or (o2 ∈ E and o1 = tgt(o2))))
```

with $m = n + |N \cup E| = n + |N| + |E|$. Finally, polynomial time executions are ensured by the non-deterministic guess for (o', t') in Line 71, and the fact that the depth of the recursion tree is linear with respect to the size of the input expression r. In particular, we do a single non-deterministic guess in Line 71, instead of an exponential number of attempts (with respect to the size of the representation of Ω) that would be necessary to find the right pair (o', t') in a deterministic algorithm.

A.5 Proof of Theorem 6.6

Notice that every expression in NavL[PC, ANOI] is also an expression in NavL[PC, NOI], so Pspace-membership follows immediately from Theorem 6.2. Hence, we only need to prove Pspace-hardness for NavL[PC, ANOI].

To show this, we replace test expressions r_i in the proof in Section A.1 by an expression in NavL[PC, ANOI] that will be denoted by q_i. Expression q_i is defined in such a way that, for every time t, it holds that $(v, t, v, t) \in [\![q_i]\!]_C$ if and only if $(v, t, v, t) \in [\![r_i]\!]_C$, *i.e.*, if and only if $\mathsf{bit}(i, t)$ is true, where $\mathsf{bit}(i, t)$ holds if the i-th bit of time t (from right to left when written in its binary representation) is 1. More precisely, expression q_i is defined as follow:

$$q_i \quad = \quad ?\left(\left((\mathbf{P}[0,0] + \mathbf{P}[2^n, 2^n]) \;/\; \ldots \;/\; (\mathbf{P}[0,0] + \mathbf{P}[2^i, 2^i])\right) \;/\; \left(< 2^i \wedge \neg < 2^{i-1}\right)\right)$$

Algorithm 5: TupleEvalSolve_ANOI$(C, (o_1, t_1, o_2, t_2), r)$ (part II)

else if $r = \mathbf{N}[n, m]$ **then**
  return $(o_1 = o_2$ **and** $n \leq (t_2 - t_1) \leq m)$
else if $r = \mathbf{P}[n, m]$ **then**
  return $(o_1 = o_2$ **and** $n \leq (t_1 - t_2) \leq m)$
else if $r = \mathbf{F}[n, m]$ **then**
  if $t_1 \neq t_2$ **then**
    return false
  else
    Let $G = (N', E')$ be the graph where:
      - $N' = (N \cup E)$
      - $E' = \{(v, e) \in N \times E \mid \mathsf{src}(e) = v\} \cup \{(e, v) \in E \times N \mid \mathsf{tgt}(e) = v\}$
    return o_2 is reachable from o_1 in k steps in G, where $k \in \{n, \ldots, m\}$
else if $r = \mathbf{B}[n, m]$ **then**
  if $t_1 \neq t_2$ **then**
    return false
  else
    Let $G = (N', E')$ be the graph where:
      - $N' = (N \cup E)$
      - $E' = \{(v, e) \in N \times E \mid \mathsf{tgt}(e) = v\} \cup \{(e, v) \in E \times N \mid \mathsf{src}(e) = v\}$
    return o_2 is reachable from o_1 in k steps in G, where $k \in \{n, \ldots, m\}$
else if $r = \mathbf{N}[n, \_]$ **then**
  return $(o_1 = o_2$ **and** $n \leq (t_2 - t_1))$
else if $r = \mathbf{P}[n, \_]$ **then**
  return $(o_1 = o_2$ **and** $n \leq (t_1 - t_2))$
else if $r = \mathbf{F}[n, \_]$ **then**
  $m \leftarrow n + |N| + |E|$
  return TupleEvalSolve_ANOI$(C, (o_1, t_1, o_2, t_2), \mathbf{F}[n, m])$
else if $r = \mathbf{B}[n, \_]$ **then**
  $m \leftarrow n + |N| + |E|$
  return TupleEvalSolve_ANOI$(C, (o_1, t_1, o_2, t_2), \mathbf{B}[n, m])$
else if $r = (r_1 + r_2)$ **then**
  Guess $i \in \{1, 2\}$
  return TupleEvalSolve_ANOI$(C, (o_1, t_1, o_2, t_2), r_i)$
else if $r = (r_1 \;/\; r_2)$ **then**
  Guess $(o', t') \in (N \cup E) \times \Omega$
  return TupleEvalSolve_ANOI$(C, (o_1, t_1, o', t'), r_1)$ **and** TupleEvalSolve_ANOI$(C, (o', t', o_2, t_2), r_2)$

Notice that the length of the representation 2^k is k, so the whole expression q_i has length $O(n^2)$, which is polynomial with respect to the size of ψ. Also, notice that as before, we only need a polynomial number of these expressions for the reduction, and no further nesting of numerical occurrence indicators is required for the proof. Hence, we only need to prove that $(v, t, v, t) \in [\![q_i]\!]_C$ if and only if $\mathrm{bit}(i, t)$ is true. Recall that for this reduction, C is a CTPG consisting of only one node v, existing from time 0 to time $2^n - 1$, with no edges or properties, so any temporal object considered will be of the form (v, t).

First, notice that q_i is a path test, so $(v, t, v, t) \in [\![q_i]\!]_C$ if and only if there exists a time point t' such that

$$(v, t, v, t') \quad \in \quad [\![\left((\mathbf{P}[0,0] + \mathbf{P}[2^n, 2^n]) \;/\; \ldots \;/\; (\mathbf{P}[0,0] + \mathbf{P}[2^i, 2^i]) \right) \;/\; \left(< 2^i \wedge \neg < 2^{i-1} \right)]\!]_C$$

As in **Step 1** of Section A.1, since the last part of the expression is a test, this is equivalent to the existence of a time point t' such that $(v, t') \models \left(< 2^i \wedge \neg < 2^{i-1} \right)$, *i.e.*, the i-th bit of t' is 1 and

$$(v, t, v, t') \quad \in \quad [\![(\mathbf{P}[0,0] + \mathbf{P}[2^n, 2^n]) \;/\; \ldots \;/\; (\mathbf{P}[0,0] + \mathbf{P}[2^i, 2^i])]\!]_C \tag{9}$$

We now prove that $(v, t, v, t) \in [\![q_i]\!]_C$ if and only if $\mathrm{bit}(i, t)$ is true. To show direction ($\Leftarrow$), suppose that $\mathrm{bit}(i, t)$ is true. Notice then that $(v, t_1, v, t_2) \in [\![(\mathbf{P}[0,0] + \mathbf{P}[2^k, 2^k])]\!]_C$, if and only if $t_2 = t_1$ or $t_2 = t_1 - 2^k$. In particular, if the $(k+1)$-th bit of t_1 is 1, then $t_2 = t_1 - 2^k$ has a binary representation that is equal to that of t_1 except on the $(k+1)$-th bit, and $(v, t_1, v, t_2) \in [\![(\mathbf{P}[0,0] + \mathbf{P}[2^k, 2^k])]\!]_C$. Similarly, if the $(k+1)$-th

bit of t_1 is 0, then $t_2 = t_1$ has a binary representation that is equal to that of t_1, and also $(v, t_1, v, t_2) \in [\![(\mathbf{P}[0,0] + \mathbf{P}[2^k, 2^k])]\!]_C$. As in Section A.1, given $b \in \{\text{true}, \text{false}\}$, let $\mathbb{1}_b$ be 1 if $b = \text{true}$, and be 0 otherwise. Moreover, define the sequence of time points $t_{n+1}, \ldots, t_i$ such that $t_{n+1} = t$ and $t_k = t_{k+1} - \mathbb{1}_{\text{bit}(k+1,t)} \cdot 2^k$ for $k \in \{i, \ldots, n\}$. Then for every $k \in \{i, \ldots, n\}$, it holds that $(v, t_k, v, t_{k+1}) \in [\![(\mathbf{P}[0,0] + \mathbf{P}[2^k, 2^k])]\!]_C$, and in particular, $t' = t - \sum_{k=i}^{n} \mathbb{1}_{\text{bit}(k+1,t)} \cdot 2^k$ satisfies (9). Therefore, if the i-th bit of t is 1, then the i-th bit of t' will be 1 as well. Hence, given $\text{bit}(i, t)$ is true, we conclude that $(v, t, v, t) \in [\![q_i]\!]_C$, since for $t' = t - \sum_{k=i}^{n} \mathbb{1}_{\text{bit}(k+1,t)} \cdot 2^k$, equation (9) holds and $(v, t') \models (< 2^i \wedge \neg < 2^{i-1})$.

To show direction ($\Rightarrow$), suppose that $(v, t, v, t) \in [\![q_i]\!]_C$. Then there exists a time point t' such that (9) holds, which only holds if there exists a sequence of time points $t_{n+1}, \ldots, t_i$ where $t_{n+1} = t$ and either $t_k = t_{k+1}$ or $t_k = t_{k+1} - 2^k$ for $k \in \{i, \ldots, n\}$, and $t_i = t'$. Notice that for such values for k, 2^k is a multiple of 2^i, so $t' = t + d \cdot 2^i$ for some integer d. We conclude that the i-th bit of t is equal to 1 if and only if the i-th bit of t' is equal to 1. Moreover, $(v, t') \models (< 2^i \wedge \neg < 2^{i-1})$, so the i-th bit of t' is indeed equal to 1, so $\text{bit}(i, t)$ must be equal to true.

From the previous paragraphs, we conclude that $(v, t, v, t) \in [\![q_i]\!]_C$ if and only if $\text{bit}(i, t)$ is true. Hence, by replacing r_i with q_i in the proof of Theorem 6.2, we deduce that NavL[PC, ANOI] is also PSPACE-hard, which was to be shown.