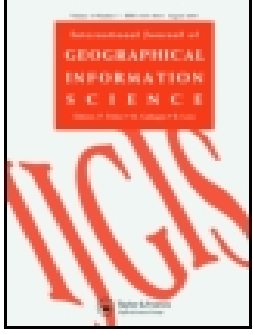

International Journal of Geographical Information Science

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/tgis20

Stop-and-move sequence expressions over semantic trajectories

Yenier Torres Izquierdo , Grettel Monteagudo García , Marco A. Casanova , Luiz André P. Paes Leme , Christos Sardianos , Konstantinos Tserpes , Iraklis Varlamis & Lívia C. Ruback Rodrigues

To cite this article: Yenier Torres Izquierdo , Grettel Monteagudo García , Marco A. Casanova , Luiz André P. Paes Leme , Christos Sardianos , Konstantinos Tserpes , Iraklis Varlamis & Lívia C. Ruback Rodrigues (2020): Stop-and-move sequence expressions over semantic trajectories, International Journal of Geographical Information Science, DOI: 10.1080/13658816.2020.1793157

To link to this article: https://doi.org/10.1080/13658816.2020.1793157

© 2020 The Author(s). Published by Informa UK Limited, trading as Taylor & Francis Group.

Published online: 20 Jul 2020.

Submit your article to this journal

Article views: 77

View related articles

View Crossmark data

Full Terms & Conditions of access and use can be found at
https://www.tandfonline.com/action/journalInformation?journalCode=tgis20

INTERNATIONAL JOURNAL OF GEOGRAPHICAL INFORMATION SCIENCE
https://doi.org/10.1080/13658816.2020.1793157

RESEARCH ARTICLE

OPEN ACCESS

Stop-and-move sequence expressions over semantic trajectories

Yenier Torres Izquierdo [a], Grettel Monteagudo García [a], Marco A. Casanova [a], Luiz André P. Paes Leme [b], Christos Sardianos [c], Konstantinos Tserpes [c], Iraklis Varlamis [c] and Lívia C. Ruback Rodrigues [d]

[a]Department of Informatics, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil; [b]Institute of Informatics, Federal Fluminense University, Niteroi, Brazil; [c]Department of Informatics and Telematics, Harokopio University of Athens, Athens, Greece; [d]Department of Computing, Federal Rural University of Rio de Janeiro, Seropédica, Brazil

ABSTRACT

Stop-and-move semantic trajectories are segmented trajectories where the stops and moves are semantically enriched with additional data. A query language for semantic trajectory datasets has to include selectors for stops or moves based on their enrichments and sequence expressions that define how to match the results of selectors with the sequence the semantic trajectory defines. This article addresses the problem of searching semantic trajectories, using stop-and-move sequence expressions. The article first proposes a formal framework to define semantic trajectories and introduces stop-and-move sequence expressions, with well-defined syntax and semantics, which act as an expressive query language for semantic trajectories. Then, it describes a concrete semantic trajectory model in RDF, defines SPARQL stop-and-move sequence expressions and discusses strategies to compile such expressions into SPARQL queries. Lastly, the article specifies user-friendly keyword search expressions over semantic trajectories based on the use of keywords to specify stop-and-move queries, and the adoption of terms with predefined semantics to compose sequence expressions. It then shows how to compile such keyword search expressions into SPARQL queries. Finally, it provides a proof-of-concept experiment over a semantic trajectory dataset constructed with user-generated content from Flickr, combined with Wikipedia data.

ARTICLE HISTORY
Received 27 February 2020
Accepted 4 July 2020

KEYWORDS
Semantic trajectories search; stop-and-move sequences; RDF; SPARQL

1. Introduction

In the recent years, massive amounts of tracking data have been generated for the benefit of applications that address human or animal mobility, traffic management, etc. Research works in the field (Renso *et al*. 2013b, Bogorny *et al*. 2014) begin with a *raw trajectory* that consists of spatio-temporal positions extracted from a raw movement track. Then they partition its points into homogeneous segments, that have common properties, such as *stops*, where the speed of the object is lower than a certain threshold, and *moves*, where

CONTACT Marco A. Casanova casanova@inf.puc-rio.br

© 2020 The Author(s). Published by Informa UK Limited, trading as Taylor & Francis Group.
This is an Open Access article distributed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits non-commercial re-use, distribution, and reproduction in any medium, provided the original work is properly cited, and is not altered, transformed, or built upon in any way.

the speed is greater than such threshold (Spaccapietra *et al*. 2008). The subsequent *semantic enrichment step* enriches the *segmented trajectory* with additional information (e.g. traffic, weather, etc.) retrieved from external repositories (Parent *et al*. 2013). The final output is a *semantic trajectory* (Renso *et al*. 2013b).

Given a semantic trajectory dataset, one immediate question arises: *how to query the dataset?* The query language must include selectors for stops and moves based on their enrichments and must define how to match the selectors with the sequence of stops and moves contained in the semantic trajectory. For example, a query on semantic trajectories of tourists in the city of Pisa can search for 'walking trajectories that begin with a stop at the *Leaning Tower*, then at *Campo Santo* and, later on, stop at a museum'. The intended interpretation of 'then' is that the first two stops are consecutive, but 'later on' indicates that there might be several stops between *Campo Santo* and the museum, as long as all moves are by walking (as transportation means).

In this work, we focus on *stop-and-move semantic trajectories* of humans (Renso *et al*. 2013a). The trajectory segments are rich in information (e.g. stops contain information for Points-Of-Interest – POIs and moves contain information on the transportation means, duration and distance traveled) and may contain information about the moving object. For querying the resulting semantic trajectory dataset, we employ stop-and-move sequence expressions. Although we employ the familiar concepts of stops and moves (Alvares *et al*. 2007a, Spaccapietra *et al*. 2008, Baglioni *et al*. 2008), we support querying more than the stops' properties. The proposed expressions allow us to search for semantic trajectories using the stops' and moves' properties and the intercalation of stops and moves.

The article proposes a formal framework for semantic trajectories, which is based on Description Logic and formally introduces the syntax and semantics of stop-and-move sequence expressions. It also provides a semantic trajectory model in RDF (Resource Description Framework) and defines user-friendly keyword expressions that search for individual stops and moves or their sequences. Finally, it explains how such stop-and-move sequence expressions can be mapped to queries in the SPARQL Protocol and RDF Query Language (or SPARQL for simplicity), taking advantage of the concrete framework.

The main contributions of this research, therefore, are threefold:

- It provides a formal representation of semantic trajectories, which for the first time allows sequence queries that combine stops and moves;
- It implements a query mechanism for semantic trajectories, which considers in tandem stop-and-move semantics and the semantics of their sequence in the semantic trajectory;
- It provides an end-to-end explanation of how keyword search expressions over semantic trajectories are mapped to SPARQL.

The article also includes a proof-of-concept experiment to validate the proposed solution. The experiment adopts a triplified version of the TripBuilder dataset, a semantic trajectory dataset constructed from user-generated content obtained from Flickr, combined with data from Wikipedia. The experiment illustrates how to compile a set of keyword search expressions into SPARQL.

The remainder of this article is organized as follows. Section 2 summarizes related work. Section 3 introduces the use-case trajectory dataset and sample informal queries

that help illustrate the discussion in the next sections. Section 4 provides a formal model for semantic trajectories and defines stop-and-move sequence expressions. Section 5 defines an RDF model for semantic trajectories and for stop-and-move sequence expressions. Section 6 introduces keyword search expressions over semantic trajectories and discusses their translation to SPARQL. Section 7 describes the proof-of-concept experiment. Finally, Section 8 contains the conclusions and suggests directions for future research.

2. Related work

The abundance of positioning data and the applications that use them quickly raised the interest in adding semantics to trajectories. The *Baquara* framework (Fileto *et al*. 2013) was a pioneering approach in the domain of semantically enriched trajectories, that supported a limited amount of Linked Open Data sources. The Athena ontology (Renso *et al*. 2013a) was a step towards an OWL-based[1] reasoning process for semantic trajectories. This allowed meaningful pattern interpretations of human behavior by combining inductive and deductive reasoning.

In order to take advantage of semantic trajectories, which are represented in an abstract model, it is necessary to develop the appropriate query mechanisms (e.g. in SPARQL). However, the simplicity of keyword-based queries makes them more preferable among non-expert users and raises the need for a SPARQL transcription mechanism (Oliveira *et al*. 2015, Bast *et al*. 2016). Examples of graph-based approaches, which directly explore the RDF dataset for generating and ranking candidate SPARQL queries, are: i) mapping techniques, such as (Zhou *et al*. 2007) and (Rihany *et al*. 2018), which use Wordnet to map keywords to elementary SPARQL query building blocks Zheng *et al*. (2016), ii) ranking techniques (e.g. (Ghanbarpour and Naderi 2019)) that search for the top-k answers of the keyword-based query and iii) graph summarization algorithms of Tran *et al*. (2009), Le *et al*. (2014), Lin *et al*. (2018) and Wen *et al*. (2018). Another schema-based approach is QUICK (Zenz *et al*. 2009), which employs user feedback on the selection of intermediate queries that will be finally executed. Finally, the compositional approach of Han *et al*. (2017) uses the keywords to first obtain elementary query graph building blocks and then applies a bipartite graph matching-based best-first search method to assemble the final query.

Baquara[2], by Fileto *et al*. (2015), introduced the concepts of movement segments (e.g. stops and moves), events and movement objects and allowed SPARQL queries over them. The datAcron ontology (Santipantakis *et al*. 2017) conceptualised trajectories as temporal sequences of segments, which relate to a behaviour, event, etc. Every expression results in an iterative execution of parametrized SPARQL queries that collects all the relevant trajectory information. Spaccapietra *et al*. (2008) defined concepts for *Stops, Moves*, trajectory *Begin* and *End* but ignored stop ordering. The Geo-Ontology (Hu *et al*. 2013) considered the ordering of points in the trajectory (e.g. concepts *hasNext, hasSuccessor*, etc.) but did not support SPARQL and intercalated stop-and-move sequence queries.

When keyword query expressions target semantic trajectories, it is important to allow trajectory or sub-trajectory matching, either exact or approximate. It is also important to support operations that capture the semantic properties of a trajectory (Alvares *et al*. 2007b, Yan *et al*. 2008, Parent *et al*. 2013), such as stops (or moves) contained in it with or

without order (Furtado *et al*. 2016, Petry *et al*. 2019). For this purpose, we aim at retrieving trajectories of interest using approximate criteria and semantic matching operators of increased flexibility compared to exact matching. The semantics of the trajectory refer to various aspects of stops and moves, and the scenario we employ assumes human trajectories in an urban environment.

In this work, we propose an algorithm for converting a keyword-based query into SPARQL, which is composed by: i) restriction clauses that represent keyword matches and ii) join clauses that connect the restriction clauses. The keyword-based query conversion is schema-based since it exploits the RDF schema to compile the final SPARQL query. The generation of the join clauses builds upon the idea of candidate networks (Hristidis and Papakonstantinou 2002), following the successful paradigms of (Oliveira *et al*. 2015, Bergamaschi *et al*. 2016). The answer to the SPARQL query is a subgraph of the RDF graph, which contains literal nodes that match the keywords, and paths that connect the literal nodes.

3. A keyword search over semantic trajectories use-case

This section briefly describes the *TripBuilder* (Brilhante *et al*. 2014) semantic trajectory dataset, an example dataset used in this paper, and provides a set of sample queries that follow our proposed notation for stop-and-move sequence expressions.

TripBuilder contains tourist trajectories from different Italian cities. The trajectories have been constructed by clustering users' Flickr photos in the spatial dimension and relating them to points-of-interest (POIs). POI semantics (e.g. POI category) have been extracted from Wikipedia. For example, for the city of Pisa, the dataset contains 3,430 trajectories by 1,825 distinct users, from which only 389 trajectories (approximately 11%) have a length between 4 and 20. Table 1 illustrates two trajectories, both of which have six POIs. This research uses a triplified version of the original TripBuilder dataset (https://doi.org/10.6084/m9.figshare.11559090). The triplification follows the RDF schema described in Section 5.

The sample queries defined over the TripBuilder RDF dataset can be expressed using:

- a *symbolic notation*, similar to that of regular expressions, that defines sequences of stop-and-move queries
- a *reserved terms-based notation*, which combines query terms and reserved terms that define the properties and interrelations of stops and moves.

Table 2 summarises the symbols and reserved terms of the proposed notation, with the list of symbols in the first column, their equivalent terms in the second column, and

Table 1. Two illustrative trajectories, based on the TripBuilder dataset.

Trajectory 1	Trajectory 2
1: 'Porta_Nuova_(Pisa)',	1: 'Torre_del_Leone',
2: 'Museo_delle_sinopie',	2: 'Torre_pendente_di_Pisa',
3: 'Cappella_Dal_Pozzo',	3: 'Camposanto_monumentale',
4: 'Museo_delle_sinopie',	4: 'Torre_del_Leone',
5: 'Chiesa_di_San_Giorgio_ai_Tedeschi',	5: 'Torre_pendente_di_Pisa',
6: 'Museo_delle_sinopie'	6: 'Camposanto_monumentale'

Table 2. Alphabet for keyword queries over semantic trajectories.

Symbol	Reserved Term	Description
Stop	`'any stop'`	the set of stops
Move	`'any move'`	the set of moves
Begin	`Begin`	the set of all beginning stops of trajectories
End	`End`	the set of all end stops of trajectories
$E \sqcup F$	`or`	the union of the results of queries *E* and *F*
$E \sqcap F$	`and`	the intersection of the results of queries *E* and *F*
E^{+}	`'at least once'`	repeat query *E* at least once
E^{*}	`'zero or more times'`	repeat query *E* zero or more times
$E \mid F$	`or`	execute query *E* or query *F* (but not both)
$E; F$	`'and then'`	execute query *E* and then query *F*
$E?$	`optionally`	execute query *E* at most once
$< M >$	`'by . . . to'`	move from one stop to the next, where *M* is a query on moves

Note: *E* and *F* are queries that define a set of stops or a set of moves, depending on the context.

a description of their meanings in the last column. In Table 3, we demonstrate the two notations using a set of example queries for the city of Pisa. Queries are first expressed in natural language, and then written using the symbolic and the reserved terms-based notation. Finally, Table 4 shows sample query terms used in the TripBuilder RDF dataset, their free-text equivalents and their meaning.

Queries 1 to 1 ignore moves and focus only on the semantics of stops and their sequences. Queries 1 to 1 seek for semantic trajectories that combine stops and moves in a specific sequence. In 1 'Stop*' indicates that the trajectory may have zero or more stops of any kind between the stop that satisfies 'Museidipisa' and the end of the trajectory. In 1 'Move' indicates that the trajectory may have any kind of move between the stop that satisfies 'Chiesedipisa' and the end of the trajectory.

We assume that a query *Q* is evaluated against some segment of a trajectory τ. The segment is required neither to start at the beginning of τ nor to terminate at the end of τ. If *Q* must be evaluated against the complete trajectory, then the user must resort to the reserved symbols *Begin* and *End*, as in Examples Q5, Q7 and Q10. Section 4.2.2 provides a formal definition of how queries are evaluated against trajectories. Section 5 shows how to write the example queries in SPARQL.

4. A formal framework for querying semantic trajectories

4.1. A formal framework for semantic trajectories

The formal framework for semantic trajectories provides the concepts needed to define the syntax and semantics of the query expressions and to define an RDF model and a SPARQL implementation of such expressions. We present its main concepts in two groups: i) the *core model* that suffices to formalize semantic trajectories in Description Logic and ii) the *extended model* that includes concepts for writing query expressions.

Core model: The core model has three classes, *Trajectory*, *Stop* and *Move*, and a set of other classes collectively called *enrichment classes*. The individuals in *Trajectory* are called *trajectory individuals*, those in *Stop* are called *stop individuals*, those in *Move* are called *move individuals*, and those in the enrichment class are called *enrichment individuals*.

Table 3. A set of sample queries.

Qid	Free text query	Symbolic notation	Reserved terms-based notation
Q1	Trajectories that stop at a museum and then at a chapel	`Museidipisa; Cappelledipisa`	Museidipisa 'and then' Cappelledipisa
Q2	Trajectories that stop at a tower, then stop at a chapel or church, and then at a museum	`Torridipisa;(Cappelledipisa | Chiesedipisa); Museidipisa`	Torridipisa 'and then' (Cappelledipisa or Chiesedipisa) 'and then' Museidipisa
Q3	Trajectories that stop at least once in a tower, and then at a museum	Torridipisa^{+}; Museidipisa	Torridipisa 'at least once' 'and then' Museidipisa
Q4	Trajectories that stop at the Lion Tower and then at the Leaning Tower, or stop at the Leaning Tower and then at the Lion Tower	`Torre_del_Leone; Torre_pendente_di_pisa) | (Torre_pendente_di_pisa; Torre_del_Leone)`	(Torre_del_Leone 'and then' Torre_pendente_di_pisa) or (Torre_pendente_di_pisa 'and then' Torre_del_Leone)
Q5	Trajectories that begin at a museum and then end at a chapel	(Begin ⊓ Museidipisa); (Cappelledipisa ⊓ End)	(Begin and Museidipisa) 'and then' (Cappelledipisa and End)
Q6	Trajectories that stop at a museum and, later on, end at a chapel or a church optionally	Museidipisa; Stop*; (Cappelledipisa | Chiesedipisa)? ⊓ End	Museidipisa 'and then' 'any stop zero or more times' 'and then' (Cappelledipisa or Chiesedipisa) optionally and End
Q7	Trajectories that begin at a chapel, stop at zero or more chapels, and end at a chapel	Begin ⊓ Cappelledipisa; Cappelledipisa*; Cappelledipisa ⊓ End	Begin and Cappelledipisa 'and then' Cappelledipisa 'zero or more times' 'and then' Cappelledipisa and End
Q8	Trajectories that stop at a museum and then take a bus to a chapel	Museidipisa <Bus> Cappelledipisa	Museidipisa 'by Bus to' Cappelledipisa
Q9	Trajectories that begin at a chapel or a church, always move by bus between stops, and end at the Leaning Tower	(Begin ⊓ (Cappelledipisa | Chiesedipisa)) <Bus^{+}> (Torre_pendente_di_pisa ⊓ End)	Begin and (Cappelledipisa or Chiesedipisa) 'by Bus at least once to' Torre_pendente_di_pisa and End
Q10	Trajectories that begin at a tower, then walk to take a bus to a church, and then using any transportation means end at a palace	Begin ⊓ Torridipisa <Walk;Bus> Chiesedipisa <Move> Palazzidipisa ⊓ End	Begin and Torridipisa 'by walk and then Bus to' Chiesedipisa 'by any move to' Palazzidipisa and End

Table 4. Terms used on the TripBuilder dataset and the sample queries.

TripBuilder term	Reserved Terms	Description
Museidipisa	Musei Pisa	the set of museums located in the city of Pisa
Cappelledipisa	`Cappele Pisa`	the set of chapels located in the city of Pisa
Chiesedipisa	`Chiese Pisa`	the set of churches located in the city of Pisa
Torredipisa	`Torre Pisa`	the set of towers located in the city of Pisa
transportation	`transportation`	indicates the transportation means of moves
Torre_pendente_di_pisa	`Torre pendent Pisa`	the Leaning Tower located in the city of Pisa
Torre_del_Leone	`Torre del Leone Pisa`	the Lion Tower located in the city of Pisa
Walk	`Walk`	the transportation means is 'by walking'
Taxi	`Taxi`	the transportation means is 'by taxi'
Bus	`Bus`	the transportation means is 'by Bus'
Subway	`Subway`	the transportation means is 'by subway'

Whenever possible, the term '*individual*' is omitted. The core model also has four binary relationships, *enrichedBy*, *begins*, *from* and *to*.

Figure 1 schematically depicts a trajectory individual *t* in the core (complete lines) and the extended model (dashed lines). A formalization of the core model reduces to capturing the following assumptions:

[A1.] *Trajectory*, *Stop* and *Move* are disjoint;

[A2.] *Trajectory*, *Stop* and *Move* are disjoint from the enrichment classes;

[A3.] *enrichedBy* relates a stop or move to one or more enrichments;

[A4.] *begins* relates a trajectory to a single stop, called the *begin stop* of the trajectory, and a begin stop is related to a single trajectory by *begins*;

[A5.] *from* relates a move to a single stop, and a stop is related to a single move by *from*;

[A6.] *to* relates a move to a single stop, and a stop is related to a single move by *to*;

[A7.] *from* is defined for a move m_j iff *to* is also defined for m_j;

[A8.] *to* does not map a move to the begin stop of a trajectory.

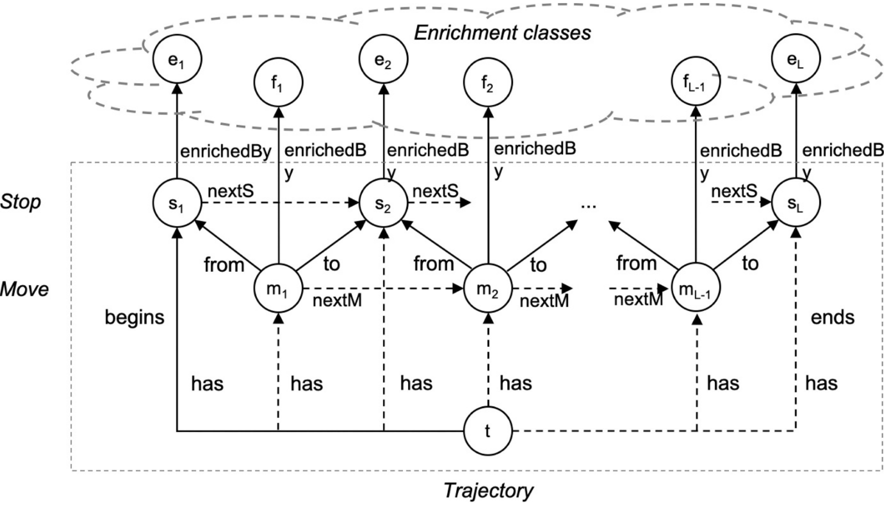

Figure 1. Schematic trajectory in the extended model.

For example, a pair in '*enrichedBy* relates stop s_1 to *torre_di_pisa*', which corresponds to the Leaning Tower of Pisa, and another pair relates move m_j to *bus*", which corresponds to the respective transportation mean used in move m_j.

Using assumptions A4-A8, for a trajectory t that begins with s_1, we define a unique sequence of stops $\sigma = (s_1, \ldots, s_L)$, the *stop sequence* of t, and traverse from s_1 to the other stops, using *from* and *to* relationships. Likewise, we can define a unique sequence of moves, $\mu = (m_1, \ldots, m_{L-1})$, called the *move sequence* of t, by traversing from stop s_j to stop s_{j+1} moving by move m_j, using the *from* and *to* relationships, for each $j \in [1, L-1]$. Following assumption A3, the *semantic trajectory* Σ induced by trajectory t will be a pair $\Sigma = ((\sigma, \mu), (\theta, \phi))$, where θ is a sequence of sets of stop enrichments, $\theta = (e_1, \ldots, e_L)$, and ϕ is a sequence of sets of move enrichments, $\phi = (f_1, \ldots, f_{L-1})$. Consequently, the following properties hold:

[P1.] A stop or move belongs to at most one trajectory;

[P2.] A stop or move is not repeated in σ or μ.

Extended model: The extended model adds two more classes, *Begin* and *End*, and four binary relationships, *has*, *nextS*, *nextM* and *ends*, as follows:

[D1.] *Begin* is the set of begin stops of the trajectories;

[D2.] *End* is the set of the last stops (*end stops*) in the stop sequences of the trajectories;

[D3.] *nextS* relates each pair of consecutive stops of the stop sequence of each trajectory, that is, *nextS* is the composition of the inverse of *from* with *to*;

[D4.] *nextM* relates each pair of consecutive moves of the move sequence of each trajectory, that is, *nextM* is the composition of *to* with the inverse of *from*;

[D5.] *has* relates each trajectory to each stop of the stop sequence of the trajectory, and to each move of the move sequence of the trajectory;

[D6.] *ends* relates each trajectory to its end stop.

Additional classes and relationships of the extended model do not increase the expressiveness of the model, from the formal point of view, but they facilitate writing query expressions over semantic trajectories, as well as their SPARQL counterparts.

Description Logic: Before providing the formalization of the proposed framework, we briefly summarize the core Description Logic (DL) (Baader *et al*. 2003) concepts.

The *atomic concepts* and *atomic roles* of a DL alphabet $\mathcal{A}$ capture the classes and properties of the domain of discourse; the *universal concept T* and the *bottom concept* $\bot$ are atomic concepts of $\mathcal{A}$, and the *identity relation* $\mathbb{I}$ is an atomic role of $\mathcal{A}$. The individuals of the domain are denoted using a set of *constants* in $\mathcal{A}$.

We use $c_1, c_2, \ldots$ to denote the atomic concepts of $\mathcal{A}$, $r_1, r_2, \ldots$ to denote the atomic roles of $\mathcal{A}$, and $a_1, a_2, \ldots$ to denote the constants of $\mathcal{A}$. We can then denote *concept expressions* $C_1, C_2, \ldots$ and *role expressions* $R_1, R_2, \ldots$ over $\mathcal{A}$. The atomic concepts and roles are the simplest expressions.

Based on the above, it is possible to define for concept expressions C_i and C_j: i) the *negation* '$\neg C_i$', ii) *full existential quantifications* of the form '$\exists r_j.C_i$', iii) the *union* '$C_i \sqcup C_j$', iv) the *intersection* '$C_i \sqcap C_j$' and v) the *product* '$C_i \times C_j$'. Similarly, for role expressions R_i, R_j, we can define: i) the *inverse* 'R_j^-', ii) the *transitive closure* 'R_i^+', iii) the *intersection* '$R_i \sqcap R_j$' and iv) the *composition* '$R_i \circ R_j$'. Finally we can define *axioms* '$C_i \sqsubseteq C_j$', '$C_i \equiv C_j$',

'$R_i \sqsubseteq R_j$', or '$R_i \equiv R_j$' and *assertions* of the form '$C_i(a_k)$' or '$R_j(a_k, a_l)$' for concept and role expressions.

The classes and binary relationships of the core and the extended models are accommodated by considering that alphabet $\mathcal{A}$ has five special atomic concepts, *Trajectory*, *Stop*, *Move*, *Begin*, and *End*, and eight special atomic roles, *enrichedBy*, *has*, *nextS*, *nextM*, *from*, *to*, *begins* and *ends*. The special symbols are called the *trajectory symbols* of $\mathcal{A}$, and the other symbols the *enrichment symbols* of $\mathcal{A}$. The axioms of the core model, which correspond to assumptions (A1-A8), and the definitions of the extended model, which correspond to definitions (D1-D6), have been omitted due to space restrictions and can be available upon request.

Trajectories as sequences: Let $\mathcal{A}$ be an alphabet, I an interpretation for $\mathcal{A}$ and $t \in Trajectory^I$. The *stop-and-move sequences* over I induced by t is the pair $\tau = (\sigma, \mu)$ such that:

- $\sigma = (s_1, \ldots, s_L)$ is the sequence of stops of I such that $(t, s_1) \in begins^I$, $(t, s_L) \in ends^I$ and $(s_i, s_{i+1}) \in nextS^I$, for each $i \in [1, L-1]$
- $\mu = (m_1, \ldots, m_{L-1})$ is the sequence of moves of I such that $(m_j, s_j) \in from^I$, for each $j \in [1, L-1]$

We say that L is the *length* of τ. Note that an empty trajectory is allowed, as well as a trajectory with just one stop, in which case μ is the empty sequence.

The *enrichment sets sequences* induced by t or simply *the enrichments of t*, are defined by the pair $\varepsilon = (\theta, \phi)$, where $\theta = (e_1, \ldots, e_L)$ is the sequence such that e_i is the set of pairs in $enrichedBy^I$ whose first element is s_i, called the *enrichments* of s_i in I, for $i = 1, \ldots, L$ and $\phi = (f_1, \ldots, f_{L-1})$ is the sequence such that f_j is the set of pairs in $enrichedBy^I$ whose first element is m_j, called the *enrichments* of m_j in I, for $j = 1, \ldots, L-1$.

Finally, the *semantic trajectory* over I induced by t is a pair $\Sigma = (\tau, \varepsilon)$ such that τ is the *stop-and-move sequences* over I induced by t and ε is the enrichment sets sequences induced by I. Note that ε is entirely determined by τ and the interpretation that I assigns to *enrichedBy*.

4.2. Query expressions over semantic trajectories

This section defines a query language for semantic trajectory datasets that includes: (1) stop-and-move queries that select a stop or a move based on its enrichments; and (2) sequence expressions that define how to match the stop-and-move queries with the sequence of actions (i.e. stops or moves) defined in the semantic trajectory. It first treats stop-and-move expressions as separated sequences, which is convenient from the formal point of view. Then, it introduces expressions that intercalate stop-and-move queries.

In what follows, let $\mathcal{A}$ be a DL alphabet and I be an interpretation for $\mathcal{A}$, satisfying the assumptions and formalization of Section 4.1.

4.2.1. Enrichment, stop and move queries

An *enrichment query* is simply a concept expression C_i over the enrichment symbols of $\mathcal{A}$. A *stop query* over $\mathcal{A}$ is either one of the atomic concepts *Stop*, *Begin*, *End*, or a concept expression of the form

$$Stop \sqcap \exists enrichedBy.C_i \tag{1}$$

where C_i is an enrichment query. A stop query then defines the set of stops that have at least one enrichment that satisfies C_i. Indeed, the interpretation of $Stop^I$ is the set of stops of I, and the interpretation of $\exists enrichedBy.C_i$ in I is the set of individuals that $enrichedBy^I$ maps to some individual in C_i^I. Therefore, the interpretation of $Stop \sqcap \exists enrichedBy.C_i$ is the set of stops with an enrichment in C_i^I.

We recursively expand the set of stop queries to include *stop concept expressions* of the forms '$Q_i \sqcap Q_j$' and '$Q_i \sqcup Q_j$', where Q_i and Q_j are stop queries or stop concept expressions.

Likewise, a *move query* over $\mathcal{A}$ is either the atomic concept *Move* or a concept expression of the forms

$$Move \sqcap \exists enrichedBy.C_i \tag{2}$$

where C_i is an enrichment query. Again, we recursively expand the set of move queries to include *move concept expressions* defined as above.

The semantics of the enrichment query, as well as those of the stop-and-move queries, need not be explicitly defined, since they result from the standard semantics of DL concept expressions, briefly summarized in Section 4.1.

4.2.2. Stop-and-move sequence expressions

A *stop sequence expression* is a regular expression of stop queries, a *move sequence expression* is a regular expression of move queries, and a *stop/move sequence expression* is a pair (S_i, M_j), where S_i is a stop sequence expression and M_j is a move sequence expression.

More precisely, the set of *stop sequence expressions* over $\mathcal{A}$ is recursively defined as:

(1) The empty sequence λ is a stop sequence expression over $\mathcal{A}$.
(2) Any stop query of over $\mathcal{A}$ is a stop sequence expression over $\mathcal{A}$.
(3) If S_i is a stop sequence expression over $\mathcal{A}$, then (S_i), $S_i?$, S_i^+, and S_i^* are also stop sequence expressions over $\mathcal{A}$.
(4) If S_i and S_j are stop sequence expressions over $\mathcal{A}$, then $(S_i|S_j)$ and $(S_i;S_j)$ are also stop sequence expressions over $\mathcal{A}$.

Parentheses may be omitted if no ambiguity arises. The set of *move sequence expressions* over $\mathcal{A}$ is likewise defined, using move queries as a basis.

The definition of the semantics of stop (or move) sequence expressions is simplified if we treat such expressions as specifying a set of sequences of the stop (or move) queries.

Recall that *Stop* is also a stop query that returns the set of all stops. Let λ again denote the empty sequence. Let $s_i; s_j$ denote the concatenation of two sequences s_i and s_j, with $s_i; s_j = s_i$, if $s_j = \lambda$, and $s_i; s_j = s_j$, if $s_i = \lambda$. Let s^n denote the n-fold concatenation $s; \ldots; s$ of s with itself, with $s^0 = \lambda$.

The *expansion* of a stop (or move) sequence expression S_i, denoted $expand(S_i)$, is a set of sequences of stop (or move) queries defined as follows:

(1) If S_i is the empty sequence λ, then $expand(S_i) = \{\lambda\}$

(2) If S_i is a stop (or move) query Q_i, then $expand(S_i) = \{Q_i\}$

(3) If S_i is an expression of the form (S_j), then $expand(S_i) = expand(S_j)$

(4) If S_i is an expression of the form $S_j?$, then $expand(S_i) = \{\lambda\} \cup expand(S_j)$

(5) If S_i is an expression of the form $(S_j|S_k)$, then $expand(S_i) = expand(S_j) \cup expand(S_k)$

(6) If S_i is an expression of the form $(S_j; S_k)$, then $expand(S_i) = \{s_j; s_k s_j \in expand(S_j) \ \wedge s_k \in expand(S_k)\}$

(7) If S_i is an expression of the form S_j^{+}, then $expand(S_i) = \bigcup_{m>0} expand(S_j^{m})$

(8) If S_i is an expression of the form S_j^{*}, then $expand(S_i) = \{\lambda\} \cup expand(S_j^{+})$

We immediately derive that:

(9) The expressions '$S_j?$' and '$(\lambda|S_j)$' are semantically equivalent, since: $expand(S_j?) = \{\lambda\} \cup expand(S_j) = expand(\lambda) \cup expand(S_j) = expand((\lambda|S_j))$ from (4), (1), and (5).

(10) The expressions 'S_j^{*}' and '$(\lambda|S_j^{+})$' are semantically equivalent, since: $expand(S_j^{*}) = \{\lambda\} \cup expand(S_j^{+}) = expand(\lambda) \cup expand(S_j^{+}) = expand((\lambda|S_j^{+}))$ from (8), (1), and (5).

Let $s = (s_1, \ldots, s_k)$ be a sequence with k elements, and i, j be two positive integers. Then, $segment(s, i, j) = (s_i, \ldots, s_j)$ is the segment of s starting at the ith element and ending at the jth element of s, if $1 \leq i \leq j \leq k$, $segment(s, i, j) = segment(s, i, k)$, if $1 \leq i \leq k < j$, and $segment(s, i, j) = \lambda$, if $k < i$ or $j < i$.

We extend the notion of segment to a trajectory $\tau = (\sigma, \mu)$, with length L, so that $segment(\tau, i, j) = (segment(\sigma, i, j), segment(\mu, i, j-1))$, for any two positive integers i and j. We say that a trajectory τ' is a *segment* of τ iff there are positive integers i and j such that $\tau' = segment(\tau, i, j)$.

Let I be an interpretation for $\mathcal{A}$ and $\tau = (\sigma, \mu)$ be a trajectory over I, with length L, where $\sigma = (s_1, \ldots, s_L)$ is a sequence of stops of I and $\mu = (m_1, \ldots, m_{L-1})$ is a sequence of moves of I. Let S_i be a stop (or move) sequence expression.

There are at least two options for the semantics of stop (or move) sequence expressions, depending on how S_i is evaluated against τ:

- *Strong semantics*, denoted $\tau \models_s S_i$, when S_i is evaluated from the beginning to the end of τ.
- *Weak semantics*, denoted $\tau \models_w S_i$, when S_i is evaluated against a segment of τ, which is required neither to start at the beginning of τ nor to terminate at the end of τ.

This research adopts the weak version as the default semantics. However, note that one can force an expression S_i to be evaluated from the beginning to the end of the trajectories by using *Begin* and *End* at the beginning and at the end of S_i.

Strong satisfiability is defined as follows. If τ is a non-empty trajectory, then τ *strongly satisfies* S_i iff

- $\lambda \in expand(S_i)$ (the empty sequence λ is in $expand(S_i)$), or
- There is a non-empty sequence $Q_1; \ldots; Q_k$ in $expand(S_i)$ such that, for each $i \in [1, k]$, $s_i \in Q_i^{I}$ (or $m_i \in Q_i^{I}$, for move sequence expressions), and $k = L$, where L is the length of the trajectory; in this case, we say that k is the *effective length* of S_i induced by its evaluation in τ.

If τ is the empty trajectory, then τ *strongly satisfy* S_i iff $\lambda \in expand(S_i)$.

We say that τ *strongly satisfies* a stop/move sequence expression (S_i, M_j) iff τ strongly satisfies S_i, τ strongly satisfies M_j, and $k = l + 1$, where k is the effective length of S_i induced by its evaluation in τ and l is the effective length of M_j induced by its evaluation in τ.

We say that τ *weakly satisfies* S_i iff there is a segment τ' of τ such that $\tau' \models_s S_i$.

We conclude this section with examples that illustrate the differences between strong and weak satisfiability. Under the notion of weak satisfiability, the queries in Table 3, with their intended interpretation, are expressions over the alphabet A_{Pisa}, where

- The terms Museidipisa, Cappelledipisa, Torridipisa and Chiesedipisa are atomic concepts of A_{Pisa}.
- The terms Torre_pendente_di_pisa and Torre_del_Leone are constants of A_{Pisa}.

Furthermore, the queries 1 to 1 in Table 3 must be rewritten as follows:

- Each constant a is replaced by the concept expression $\{a\}$.
- Each stop query E_i is replaced by the concept expression $Stop \sqcap \exists enrichedBy.E_i$, as in the example below, to conform with Eq. 1.
- Likewise, each move query E_i is replaced by the concept expression $Move \sqcap \exists enrichedBy.E_i$, to conform with Eq. 2.

For example, Queries 1 and 1 are respective formally rewritten as:

Q1: `Stop ⊓ ∃enrichedBy.Museidipisa; Stop ⊓ ∃enrichedBy.Cappelledipisa`

Q5: `Begin ⊓ ∃enrichedBy.Museidipisa; End ⊓ ∃enrichedBy.Cappelledipisa`

where *Begin* and *End* are specializations of *Stop*.

4.2.3. Intercalated stop-and-move sequence expressions

The definition of a stop/move sequence expression as a pair (S_i, M_j), where S_i is a stop sequence expression and M_j is a move sequence expression, is attractive from a formal point of view, but it may hide some complexities. Indeed, if a semantic trajectory τ satisfies (S_i, M_j), then the effective length of M_j must be one less than the effective length of S_i, by definition, to be able to intercalate the two sequences. But this requirement cannot be verified by a syntactical inspection of S_i and M_j and is introduced only in the semantic notion of satisfiability.

For example, consider a stop/move sequence expression (F_1, G_1), where:

$$F_1 = p; Stop; r; s \text{ and } G_1 = v^+; w.$$

Since G_1 uses the '+' operator, it denotes sequences of move queries of arbitrary lengths, but only the sequence '$v; v; w$', which has a length equal to 3, could be properly intercalated with F_1, which has a length equal to 4.

We therefore define an *intercalated stop-and-move sequence expression* as an expression N_k of the form: $N_k = S_0 < M_1 > S_1 < M_2 > S_2 \ldots S_{n-1} < M_{n-1} > S_n$, where S_i is a stop sequence expression and M_j is a move sequence expression, for $i \in [0, n]$ and $j \in [1, n-1]$.

To define the semantics of intercalated stop-and-move sequence expression, we proceed as in Section 4.2.2, attaining only to weak satisfiability. The *expansion* of N_k, denoted $expand(N_k)$, is defined as for stop sequence expressions, but respecting the intercalation of stop-and-move expressions.

Let $E \in expand(N_k)$. Assume, without loss of generality, that E is of the form: $E = P_0 < Q_1 > P_1 < Q_2 > P_2 \ldots P_{n-1} < Q_n > P_n$, where P_i is a sequence of stop queries and Q_j is a sequence of move queries, for $i \in [0, n]$ and $j \in [1, n]$.

Let $Stop^0 = \lambda$ and $Stop^n = Stop^{n-1}; Stop$, for $n \geq 1$, and likewise for $Move^n$. The *stop projection* of E is the sequence of stop queries F and the *move projection* of E is the sequence of move queries G such that:

$$F = P_0; Stop^{m_1}; P_1; Stop^{m_2}; P_2 \ldots P_{n-1}; Stop^{m_n}; P_n$$

$$G = Move^{s_0}; Q_1; Move^{s_1}; Q_2; Move^{s_2} \ldots Move^{s_{n-1}}; Q_n; Move^{s_n}$$

where m_j is the length of Q_j and s_i is the length of P_i, for $i \in [0, n]$ and $j \in [1, n]$.

An example of an intercalated stop-and-move sequence expression would be:

$$N_1 = (p \,|\, q) < v^* > r^+ < v^*; w > s$$

The following sequences pertain to $expand(N_1)$:

$$E_1 = p < v; v > r < w > s \text{ and } E_2 = q < v > r; r < v; w > s$$

The stop projection of E_1 is F_1 and the move projection of E_1 is G_1, where:

$$F_1 = p; Stop^1; r; Stop^0; s = p; Stop; r; s$$

$$G_1 = Move^0; v; Move^0; v; Move^0; w = v; v; w$$

The equalities follow if we observe that $Stop^0 = Move^0 = \lambda$. Note that if we intercalate F_1 and G_1 we obtain E_1 again:

$$p < v > Stop < v > r < w > s = p < v; v > r < w > s = E_1$$

Finally, let τ be a trajectory. If τ is the empty trajectory, then τ *does not weakly satisfy* N_k. If τ is a non-empty trajectory then τ *weakly satisfies* N_k iff there is $E \in expand(N_k)$ such that τ weakly satisfies (F, G), where F is the stop projection of E and where G is the move projection of E.

5. An RDF framework for querying semantic trajectories

Based on the formal framework of Section 4, this section introduces a concrete RDF framework for querying semantic trajectories. It first defines an RDF model for representing semantic trajectories. Then, it introduces the SPARQL stop (or move) queries and SPARQL stop (or move) sequence expressions. Next, it discusses how to compile SPARQL stop sequence expressions into equivalent SPARQL queries. Finally, it shows how to process SPARQL intercalated stop-and-move sequence expressions.

5.1. SPARQL query expressions over semantic trajectories

The model is expressed as an RDF schema with classes (Trajectory, Stop, Move, Begin, End), properties (enrichedBy, from, to, nextS, nextM, has, begins, ends) and declarations (see Figure 1). The enrichment classes and properties are not part of the proposed RDF model for semantic trajectories, as highlighted in Section 4.1. We assume that they are defined in a knowledge base.

A *SPARQL enrichment query* is a SPARQL select query over the enrichments knowledge base whose TARGET clause has a single variable and, thus, returns a set of IRIs that identify enrichments. Figure 2 illustrates two SPARQL enrichment queries. The property function <http://jena.apache.org/text#query>, in the query of Figure 2(a), combines SPARQL and full-text search via Lucene in Apache Jena. SPARQL enrichment queries such as these may, in fact, be automatically generated from keyword queries, as discussed in Section 6. We stress that a SPARQL enrichment query is not restricted to queries of the forms shown in Figure 2, but they can be any SPARQL query over the enrichments knowledge base, whose TARGET clause has a single variable.

Table 5 provides a summary of stop expressions and the respective SPARQL queries. The definition of SPARQL move queries is an exact parallel and is omitted.

As in Section 4.2.2, a *SPARQL stop sequence expression* is a regular expression of SPARQL stop queries, a *SPARQL move sequence expression* is a regular expression of SPARQL move queries, and a *SPARQL stop/move sequence expression* is a pair (S_i, M_j), where S_i is a SPARQL stop sequence expression and M_j is a SPARQL move sequence expression. The notion of *SPARQL intercalated stop-and-move sequence expressions* is defined as in Section 4.2.3. Finally, we introduce the notion of a *restricted SPARQL stop*

```
1 select ?poi1
2 where {
3   ?poi1 text:query "torre
4   pendente di pisa".
5 }
```

(a) SPARQL query that returns the IRI of the Leaning Tower in Pisa.

```
1 select ?poi2
2 where {
3   ?s1 rdfs:label ?lb1.
4   ?s2 rdfs:label "pisa".
5   filter regex(?lbl,"museo")
6   ?poi2 :category ?s1.
7   ?poi2 :locatedIn ?s2.
8 }
```

(b) SPARQL query that returns the IRIs of the museums in Pisa.

Figure 2. Examples of two SPARQL enrichment queries.

Table 5. A summary of expressions and their SPARQL equivalent.

Expressions	SPARQL	Notation
Stop, *Begin*, *End*	**select** ?v **where** {?v rdf:type C}	C $\in \{Stop, Begin, End\}$
$Stop \sqcap \exists enrichedBy.C_i$	**select** ?v **where** {?v rdf:type:Stop;:enrichedBy ?p. {E[?p]}}	E[?p] is enrichment of ?p
Intersection $Q_i \sqcap Q_j$	**select** ?v **where** {{Q1[?v]}. {Q2[?v]}}	Q_1, Q_2 are stop queries
Union $Q_i \sqcup Q_j$	**select** ?v **where** {{Q1[?v]} **UNION** {Q2[?v]}}	Q1[?v], Q2[?v] are stop queries of ?v

sequence expression, defined exactly as a SPARQL stop sequence expression, except that expressions of the forms S_i^* and S_i^+ are allowed only when S_i is a SPARQL stop query (and not recursively a SPARQL stop sequence expression). The same holds for $<M_i^*>$ and $<M_i^+>$. Likewise, a *restricted SPARQL intercalated stop-and-move sequence expression* allows expressions of the forms S_i^* and S_i^+ only when S_i is a SPARQL stop query, and expressions of the forms $<M_i^*>$ and $<M_i^+>$ only when M_i is a SPARQL move query.

5.2. Processing of SPARQL stop sequence expressions

This section discusses how to compile restricted SPARQL stop sequence expressions to SPARQL queries. The processing of SPARQL move sequence expressions is entirely similar. The compilation process recursively parses a restricted SPARQL stop sequence expression *Expr* and replaces each sub-expression of *Expr* by a SPARQL graph pattern that depends on the syntax of the sub-expression. The result is a SPARQL graph pattern, which is further post-processed to eliminate redundant triple patterns. The final SPARQL graph pattern is used to construct the WHERE clause of the SPARQL query **Q** that corresponds to *Expr*. The target clause of **Q** is a list of three variables, ?t, ?begin and ?end. When executed, **Q** binds ?t to a trajectory τ, and ?begin and ?end to stops s_B and s_E of τ, such that the segment of τ from s_B to s_E strongly satisfies *Expr*, and τ weakly satisfies *Expr*. Section 6 contains an example of the compilation process.

The compilation process uses templates, which are expressions of the form

`Template(`*Expr*`;?t,?begin,?end)`

where *Expr* is a restricted SPARQL stop sequence expression, and ?t, ?begin and ?end are SPARQL variables. When called, the template expands to a schematic graph pattern *G*, in the same way, that a macro expands in traditional programming languages. The schematic graph pattern *G* may contain calls to other templates, and non-standard SPARQL commands, such as if-then-else's, as in Template 1. The expansion process replaces the formal parameters by the concrete parameter values passed in the call, renames the other variables used in *G* to avoid conflicts, and eliminates non-standard SPARQL commands. In the resulting SPARQL graph pattern:

- If *Expr* is the empty stop sequence expression λ, then *G* is the empty group pattern '{}' that matches any graph (including the empty graph) with one solution that does not bind any variables. This is justified since the empty stop sequence expression matches any trajectory.
- If *Expr* is not the empty stop sequence expression, *G* binds ?t to a trajectory τ, and ?begin and ?end to a stops s_B and s_E of τ such that the sequence of consecutive stops of τ from s_B to s_E satisfies *Expr*.

Recall that a stop SPARQL query Q has a single variable ?v in the TARGET clause and let Q[?u] denote Q with ?v replaced by ?u. Two of the most common query templates are presented in the following:

```
(1) Template ('S1;S2';? t,? begin,? end)
   1 Template ('S1';? t,? begin,? endS1).
   2 if bound (? endS1)
   3    then {   ? endS1    :nextS     ? beginS2.
   4             Template ('S2'; ? t , ? beginS2 , ? endS2)   }
   5    else {   Template ('S2' ; ? t , ? begin   , ? endS2)   }
   6 bind ( IF ( bound( ? endS2 ) , ? endS2 , ? endS1 )  as ? end)
```

Note: This template uses a non-standard SPARQL if-then-else construct.

First note that the template call for S1 may evaluate to λ, leaving ?endS1 unbound, and the template calls for S2 may also evaluate to λ, leaving ?endS2 unbound. There are therefore four cases to consider:

Case 1: The template calls for S1 and S2 do not evaluate to λ. Then, they bind ?t to a trajectory τ and ?begin, ?endS1, ?beginS2 and ?end to stops s_B, s_{E1}, s_{B2} and s_E of τ, respectively, such that: (1) the sequence of consecutive stops in τ from s_B to s_{E1} satisfies S1; (2) the sequence of consecutive stops in τ from s_{B2} to s_E satisfies S2; (3) s_{E1} and s_{B2} are consecutive stops of τ (by Line 3). Line 6 then makes the binding of ?end equal to that of ?endS2.

Case 2: The template call for S1 does not evaluate to λ and the template call for S2 evaluates to λ. Then, the template call for S1 binds ?t to a trajectory τ and ?begin and ?endS1 to stops s_B and s_{E1} such that the sequence of consecutive stops in τ from s_B to s_{E1} satisfies S1. The template call for S2 does not bind ?endS2. Line 6 then makes the binding of ?end equal to that of ?endS1.

Case 3: The template call for S1 evaluates to λ and the template call for S2 does not evaluate to λ. The template call for S1 does not bind ?endS1. Then, by Line 2, the template call for S2 in Line 5 binds ?t to a trajectory τ and ?begin and ?endS2 to stops s_B and s_{E2} of τ such that the sequence of consecutive stops in τ from s_B to s_{E2} satisfies S2. Line 6 then makes the binding of ?end equal to that of ?endS2.

Case 4: The template calls for S1 and S2 both evaluate to λ. Then, both ?endS1 and ?endS2 are unbound and Line 6 leaves ?end unbound.

```
(2) Template('Q+';?t,?begin,?end)
   1 ? t : has   ? begin.
   2 ? begin  : nextS*   ?end.
   3 {  Q[ ? begin ]  }.
   4 {  Q[?end ]  }.
   5 filter   not   exists {
   6    ? begin   : nextS*    ?stopM.
   7    ? stopM   :nextS*    ?end.
   8    filter    not    exists    {Q[?stopM]   }   }
```

This graph pattern binds variable ?t to a trajectory τ and variables ?begin and ?end to stops s_B of s_E of τ such that all consecutive stops from s_B to s_E in τ satisfy Q, including s_B and s_E.

Notes:

(a) The actual implementation of the template places Lines 3 and 4 before Lines 1 and 2, for efficiency reasons.

(b) This template applies only when Q is a SPARQL stop query, and not a SPARQL stop sequence expression, in view of the use of the SPARQL path expression ':nextS*'.

(c) Lines 5 to 7 explore the fact that $x(Q)$ is equivalent to $\neg\exists x\neg(Q)$. Thus, the sentence '*for any stop s between s_1 and s_2, s satisfies Q*' is equivalent to '*there is no stop s between s_1 and s_2 such that s does not satisfy Q.*'

The final SPARQL graph pattern is subjected to a simplification process. However, a detailed discussion of the simplification process is outside the scope of this article. Section 6 contains an example that illustrates how to compile a restricted SPARQL stop sequence expression to an equivalent SPARQL query, and how to simplify the query.

5.3. Processing of SPARQL intercalated stop-and-move sequence expressions

A restricted SPARQL intercalated stop-and-move sequence expression allows expressions of the forms S_i^* and S_i^+ only when S_i is a SPARQL stop query, and expressions of the forms $<M_i^*>$ and $<M_i^+>$ only when M_i is a SPARQL move query. Let S1 and S2 be restricted SPARQL stop sequence expressions as those presented in Section 5.2. Let M, M1 and M2 be SPARQL move queries. Recall that M has a single variable in the TARGET clause, and let M[?u] denote M with this single variable replaced by ?u (and likewise for M1 and M2). The templates that follow are not an exhaustive list, but illustrate the extension process.

(3) `Template('S1< M1;M2> S2',?t,?begin,?end)`

1 Template('S1' , ? t , ? begin , ? endS1).
2 Template('S2' , ? t , ? beginS2 , ? end) .
3 ?move1 : **from** ? endS1 ; : to ? stop2 .
4 ?move2 : **from** ? stop2 ; : to ? beginS2 .
5 { M1[? move1] } .
6 { M2[?move2] }

The recursive template calls in Lines 1 and 2 bind variable ?t to a trajectory τ and variables ?begin, ?endS1, ?beginS2 and ?end to stops s_B, s_{E1}, s_{B2} and s_E of τ, respectively, and Lines 3 and 4 bind variables ?move1 and ?move2 to moves m_1 and m_2 of τ, respectively, such that: (1) the sequence of consecutive stops in τ from s_B to s_{E1} satisfies S1; (2) the sequence of consecutive stops in τ from s_{B2} to s_E satisfies S2; (3) m_1 is from s_{E1} to a stop s_2 and m_2 from s_2 to s_{B2} (by Lines 3 and 4); (4) m_1 satisfies M1 and m_2 satisfies M2 (by Lines 5 and 6).

Note: This templates assume that the recursive template calls do not evaluate to λ. A more elaborated template would include tests for unbound variables, as in Template 1.

(4) `Template('S1 < M+> S2';?t,?begin,?end)`

1 Template('S1' ; ? t , ? begin , ? endS1) .
2 Template('S2' ; ? t , ? beginS2 , ? end) .
3 ?moveB : **from** endS1 .
4 ?moveE : to beginS2 .
5 ?moveB : nextM* ?moveE .
6 { M[? moveB] } .
7 { M[? moveE] } .
8 **filter** not exists {

```
 9   ?moveB :nextM*  ?moveM .
10   ?moveM :nextM*  ?moveE .
11   filter  not  exists  { M[?moveM] } }
```

The recursive template calls in Lines 1 and 2 bind variable ?t to a trajectory τ and variables ?begin, ?endS1, ?beginS2 and ?end to stops s_B, s_{E1}, s_{B2} and s_E of τ, respectively, and Lines 3 and 4 bind variables ?moveB and ?moveE to moves m_B and m_E of τ, respectively, such that: (1) the sequence of consecutive stops in τ from s_B to s_{E1} satisfies S1; (2) the sequence of consecutive stops in τ from s_{B2} to s_E satisfies S2; (3) m_B is from s_{E1} and m_E is to s_{B2} (by Lines 3 and 4); (4) all moves in the sequence of moves of τ from m_B to m_E satisfy M (by Lines 5–11).

6. Keyword query expressions over semantic trajectories in RDF

This section introduces keyword query expressions over semantic trajectories in RDF as a user-friendly alternative to SPARQL intercalated stop-and-move sequence expressions. It also discusses, with the help of an example, how to process such keyword query expressions, based on the results of Section 5.

As a brief motivation, we observe that keyword search proved to be a popular paradigm under the users' perspective. From the origins of information retrieval, keyword search was used to retrieve items (usually documents) that are somehow relevant to the search keywords. These techniques evolved to relational and RDF database search, where entities and their attributes are considered to be the documents that must be retrieved. Advanced graph-search algorithms allow us to retrieve relevant entities from a database and also show how these entities relate to each other. This makes such algorithms attractive for searching large collections of semantic trajectory data. However, despite their advantages, graph-based queries are still not convenient for expressing restrictions on the sequence of stops and moves. The keyword query expressions introduced in this section overcome this difficulty.

The definitions that follow are similar to those in Sections 4.2 and 5.1. A *keyword stop query* is a finite set $K = \{k_1, \ldots, k_n\}$ of literals, called *keywords*, that defines a set of stops based on their enrichments. These queries would then be applied to the RDF knowledge base to select a set R of enrichments, which are then used to select the set of stops that are related to the enrichments in R by the *enrichedBy* property. A *keyword move query* is likewise defined.

Schema-based algorithms (García *et al*. 2017, Izquierdo *et al*. 2018) can then be used to translate keyword stop (or move) queries into SPARQL queries that retrieve resources by their name, such as 'Torre Pendente di Pisa', or by their attributes, such as 'Musei di Pisa'. In the first case, the query would retrieve a single POI id that corresponds to the Leaning Tower of Pisa (see Figure 2(a)), while in the second case it would return a list of ids that correspond to the museums in Pisa (see Figure 2(b)). One can relax the scope of the query by informing only some keywords, such as 'Torre di Pisa', in which case the keyword query could return the ids of the 'Torre Pendente di Pisa' and 'Ristorante La Torre Pisa'. A detailed discussion of the process of translating keyword queries to SPARQL queries is outside of the scope of this article.

A *keyword intercalated stop-and-move sequence expression* is a regular expression based on keyword stop-and-move queries, as in Section 4.2. The regular expression symbols may be replaced by the reserved terms listed in Table 2.

The processing of a keyword intercalated stop-and-move sequence expression *N* to SPARQL has two basic steps:

(1) Translate *N* into a SPARQL intercalated stop-and-move sequence expression *S*:

(a) If necessary, replace the reserved terms “Stop”, “Move”, “Begin” and “End” by SPARQL queries, as discussed in Section 5.1.
(b) Also, if necessary, replace the reserved terms that denote regular expressions by equivalent symbols, using Table 2.
(c) Translate each keyword stop (or move) query into a SPARQL query.

(2) Process *S* as discussed in Section 5.

We illustrate the processing of 1 (see Table 3), in the Jena ARQ SPARQL engine:

Step 1: Translate the keyword queries ‘Cappelledipisa’, ‘Chiesedipisa’ and ‘Torre_pendente_di_pisa’ to SPARQL enrichment queries, as shown in Figure 3, to retrieve the resources associated with these POIs. Note that variables ?v1, ?v2 and ?v3 bind the IRIs of POIs resources associated with ‘Cappelledipisa’, ‘Chiesedipisa’ and ‘Torre_pendente_di_pisa’, respectively.

Step 2: Process the resulting SPARQL intercalated stop-and-move sequence expression *S*. The process recognizes that *S* satisfies Template 5.3: `Template('S1<M+>S2'; ?t, ?begin, ?end)` where:

- `S1` is the SPARQL query corresponding to ‘`Begin ⊓ S3';`
- `S3` is the SPARQL query corresponding to ‘`(Cappelledipisa|Chiesedipisa)'`
- `S2` is the SPARQL query corresponding to ‘`S4 ⊓ End`’
- `S4` is the SPARQL query corresponding to `Torre_pendente_di_pisa`
- `M` is the SPARQL query corresponding to `<Bus>`

and combines the different templates to the final template for *S*

Step 3: The compilation process ends by setting the TARGET clause (as a SELECT form) with variable ?t that binds the queried trajectories, and applying some simplifications (indicated after the query). The final synthesized SPARQL query is:

```
select ?v1
where {
 ?v1 text:query
        "(Cappelledipisa)"
}
```

```
select ?v2
where {
 ?v2 text:query
        "(Chiesedipisa)"
}
```

```
select ?v3
where {
 ?v3 text:query "(Torre_pendente_di_pisa)"
}
```

Figure 3. The SPARQL enrichment queries of the keyword queries in *S*.

```
1 select   ? t ,  ? begin ,  ? end
2 where  {
3   ? t    : begins  ? begin .
4   {
5     { ? begin    : enrichedBy  ?v1  .
6     ?v1  text    : query   'Cappelledipisa'}
7     UNION
8     { ? begin  : enrichedBy   ?v2  .
9     ?v2  text  : query   'Chiesedipisa' }
10 }
11 ? t  : ends   ?end  .
12 { ? end  : enrichedBy  ?v3  .
13      ?v3  text : query  'Torre_pendente_di_pisa' }
14    ?moveE  : to   ? end ;  : enrichedBy   ? transpE  .
15    filter     (?transpE =   : Bus)
16    ?moveB   : from  ?begin ;  : nextM*   ?moveE;  :enrichedBy   ?transpB .
17    filter  (?transpB = : Bus)
18    filter   not   exists   {
19      ?moveB  :nextM*   ?moveM   .
20      ?moveM  :nextM*   ?moveE   .
21      filter   not   exists   { ? moveM  : enrichedBy   ? transpM   filter   ( ?
      transpM   =   : Bus ) }
22   }
23 }
```

7. A proof-of-concept experiment

For the construction of the *TripBuilder RDF dataset*, we wrote a Java program that parses the TripBuilder city data files, extracts data for POIs, stops and trajectories and triplifies the resulting data in RDF. We also included:

(1) The property:length, added to the instances of the:Trajectory class, indicates the length of a trajectory.
(2) The property:move_number, added to the instances of the:Move class, indicates the sequential position of a move in a trajectory.
(3) The class:Transportation whose resources were automatically generated, labeled with a value in the set {"Walk", "Taxi", "Bus", "Subway"}, and randomly linked to resources of the:Move class using the property:enrichedBy. This new class helps exploit the capabilities of the translation algorithm, given that the original TripBuilder dataset does not contain information about moves.

The resulting RDF dataset contains a total of 1,617,582 triples, which break down to 5 rdfs: Class declarations, 255,018 class instances (47% of them corresponding to stops, 30% to moves and 21.5% to trajectories) and 1,973 indexed property values.

We put the RDF dataset on a Jena ARQ SPARQL server (running on a quad-core processor Intel(R) Core(TM) i7-5820 K CPU@3.30 GHz, 64 GB of RAM and SSD 1TB, with

GNU/Linux Ubuntu 16.04.6 LTS oS). The string property values, including rdfs:label values, were indexed using Lucene (hosted in the same server) thus allowing both SPARQL queries and full-text search.

The experiments evaluated our approach on the keyword query expressions of Table 3, measuring: (1) the templates that the SPARQL stop-and-move sequence expressions satisfy; and (2) the average execution time of 10 repetitions of each synthesized query.

Given that queries Q1 to Q7 are stop sequence expressions, and Q8 to Q10 are examples of intercalated stop-and-move sequence expressions the results can be summarized as:

- Q1 and Q2 are quite simple and the equivalent SPARQL queries have a small runtime.
- The runtime of the SPARQL query for Q3 is considerably higher due to the nested filter not exists group patterns and the SPARQL Property Path operator "*" inside the stop patterns.
- Q7 contains a stop sequence expression similar to Q3, and its SPARQL query has a runtime comparable to that of the SPARQL query for Q3 (about 4 seconds).
- However, the SPARQL query for Q9 has a small runtime (similar to Q2). The runtime of its SPARQL query for Q10 is about 3 seconds.

To conclude, we observe that the complexity of the SPARQL query synthesized in each case naturally reflects the complexity of the keyword query expressions. As expected, queries with triple patterns with the property path operator '*' inside nested 'filter not exists' group patterns (Q3, Q7), or with complex graph patterns (Q10) had a high runtime. However, unexpectedly, even with a complex graph pattern, Q9 had a small runtime. Queries with less complex graph patterns (Q1, Q2, Q4, Q5, Q8), or with just UNION or OPTIONAL patterns (Q6) had acceptable runtime. Overall, all queries were executed within the specified timeout of 1 minute. The average runtime of the test suite queries, about 1.5 s, was reasonable. Hence, the experiment suggests that the proposed approach, based on keyword query expressions and RDF, is feasible.

8. Conclusions and future work

This article addressed the question of retrieving semantic trajectories with a query language that includes: (1) stop-and-move queries that select sets of stops or moves based on their enrichments; and (2) sequence expressions that define how to match the stop-and-move queries with the sequence of actions defined in the semantic trajectory.

The article first introduced a formal model for semantic trajectories and defined stop-and-move sequence expressions, with well-defined syntax and semantics. Then, it moved to a concrete semantic trajectory model in RDF and described how to process SPARQL stop-and-move sequence expressions, using state-of-the-art, efficient SPARQL query processors. Using keywords to capture stop-and-move queries, and terms with predefined semantics to define sequence expressions it offers a user-friendly query notation that can be compiled into SPARQL queries. Finally, the article described a proof-of-concept experiment using the TripBuilder dataset, a semantic trajectory dataset constructed from user-generated content obtained from Flickr, combined with data from Wikipedia.

In more detail, Section 4.1 formalized the semantic trajectory model adopted, within the tradition of Description Logic. The adoption of Description Logic is justified since it is a well-known formalism, the notions of atomic concept and atomic role suffice to model semantic trajectories, and it has been used before to formalize trajectory models. Section 4.2 covered the syntax and semantics of stop-and-move sequence expressions – the central concept of the article – which, for the first time, explore the richness of intercalated sequences of stops and moves, and not just stop sequences or just move sequences. Since a sequence expression E imposes a complex restriction on a semantic trajectory τ, this step was required to clarity how E evaluates over τ. This article adopted the weak semantics when E is evaluated against a segment of τ, which is required neither to start at the beginning of τ nor to terminate at the end of τ. The formal semantic trajectory model led to a concrete model in RDF and to concrete SPARQL queries, which opened the way to implementations of the concepts on top of standard RDF stores. Section 5 introduced SPARQL stop-and-move sequence expressions, a concrete realization of abstract sequence expressions, and showed how to compile them into standard SPARQL queries. The compilation process is recursive and non-trivial since SPARQL stop-and-move sequence expressions are constructed by composing atomic SPARQL queries and may involve complex operators, such as '*'. However, SPARQL has a sophisticated syntax, out of the reach of the typical user. The third step of the article was then to hide such complex syntax under a user-friendly, keyword-based notation. Section 6 introduced the class of keyword query expressions over semantic trajectories in RDF and showed how to compile such expressions into SPARQL stop-and-move sequence expressions, which are then compiled into SPARQL queries. Section 7, the last step, summarized a proof-of-concept experiment, within the space limitations of the article, that showed that the SPARQL queries compilated from stop-and-move sequence expressions have adequate performance. Therefore, the proof-of-concept suggested that stop-and-move sequence expressions, when compiled to SPARQL, are indeed a feasible language to search semantic trajectories and not just an abstract query framework.

As future work, we plan to run large-scale experiments with real-world semantic trajectory datasets, backed up by a robust implementation of the keyword stop-and-move sequence expression SPARQL compiler. A target application would be related to investigations of cargo vessel incidents. The stop-and-move sequence expressions introduced in this article would help, for example, to locate vessel trajectories that match disallowed movement patterns, such as 'trajectories of oil tankers that sailed from any oil rig port in country *A*, sailed through a high-risk region (e.g. a piracy prone area) and arrived at a port *P* in another country *B*'. Such trajectories are sometimes followed by captains at the risk of not been covered by insurance companies in the case of an attack. It could also be used to query for different types of trajectories, such as navigation patterns of interest in an e-shop: 'customers who visited the product page of a tv-set *A* (an analogy of stop), then followed a link (a type of move) to another tv-set *B* (another stop of the same type) and later on their session searched (i.e. a different type of "move" within a website) for a console *C*'. We also plan to adapt the proposed approach to other types of sequences, such as music playlists. Finally, we will invest in extending the approach to a question-and-answer scenario.

Note

1. OWL stands for Web Ontology Language.

Disclosure statement

No potential conflict of interest was reported by the author(s).

Notes on contributors

Yenier Torres Izquierdo is currently studying towards a D.Sc. degree in Informatics at the Department of Informatics of the Pontifical Catholic University of Rio de Janeiro – PUC-Rio, in the database area, with expected completion in 2021. He holds an M.Sc. in Informatics from PUC-Rio (2017), in the area of the database, and had a scholarship "Student Grade 10" in 2016 and 2019 from FAPERJ for his academic results during the M.Sc. and D.Sc. studies. He obtained a B.Sc. in Computer Science from the University of Havana (2012). He is currently a software developer at the Tecgraf Institute – PUC-Rio. His research topics are Process Mining, Data Mining, Databases, Compilation, DSL, Programming Languages, and Artificial Intelligence.

Grettel Monteagudo García obtained a D.Sc. in Informatics from the Pontifical Catholic University of Rio de Janeiro – PUC-Rio (2020), in the area of the database, an M.Sc. in Informatics from PUC-Rio (2016), also in the area of the database, and a B.Sc. in Computer Science from the University of Havana (2012), in the database management systems area. She had a scholarship "Student Grade 10" in 2018 and 2015 from FAPERJ for her academic results during the D.Sc and M.Sc. studies, respectively. She won the HackPUC 2014, hackathon organized by PUC-Rio. She is currently a researcher and software developer at the Tecgraf Institute – PUC-Rio. She worked at the Department of Information Systems at the José A. Echeverría Polytechnic Institute (CUJAE) in Havana, Cuba, in 2013, as a trainee professor and researcher. Her research interests include ontological models, Semantic Web technologies, keyword search, and graph algorithms.

Marco A. Casanova is a Full Professor at the Department of Informatics and Coordinator of the Central Planning and Evaluation Office of the Pontifical Catholic University of Rio de Janeiro – PUC-Rio. He graduated in Electronic Engineering at the Military Institute of Engineering (1974), obtained an M.Sc. in Informatics from PUC-Rio (1976), and an M.Sc. (1977) and a Ph.D. (1979) in Applied Mathematics from Harvard University. He was Graduate Program Coordinator (2005-2007) and Director (2007-2011) of the Department of Informatics of PUC-Rio. His research interests concentrate on database conceptual modeling and the construction of database management systems. In July 2012, he received the Scientific Merit Award from the Brazilian Computer Society.Luiz André Portes Paes Leme obtained a B.Sc. in Electrical Engineering from the State of Rio de Janeiro University (1989), and an M.Sc. in Informatics (2006) and a D.Sc. in Informatics (2009), both from the Pontifical Catholic University of Rio de Janeiro. He conducted postdoctoral research at the Consiglio Nazionale delle Ricerche (2015). Currently, he is Associate Professor at the Federal Fluminense University. His research interests include data integration, information retrieval, and mobile data analysis.

Luiz André P. Paes Leme obtained a B.Sc. in Electrical Engineering from the State of Rio de Janeiro University (1989), and an M.Sc. in Informatics (2006) and a D.Sc. in Informatics (2009), both from the Pontifical Catholic University of Rio de Janeiro. He conducted postdoctoral research at the Consiglio Nazionale delle Ricerche (2015). Currently, he is Associate Professor at the Federal Fluminense University. His research interests include data integration, information retrieval, and mobile data analysis

Christos Sardianos is currently a Ph.D. candidate and research assistant at the Department of Informatics & Telematics at Harokopio University of Athens. He also holds an MSc in the area of

Web Engineering and a BSc in Electronics Engineering. The research area of his Ph.D. is "Knowledge Mining from Large Scale Social Networks", under the supervision of Associate Professor Iraklis Varlamis. His main research topics of interest include Recommender Systems, Data Mining, Machine Learning, and Social Network Analysis. He has participated in numerous EU, international and national funded projects including Fortissimo (FP7), EM3, TEACHING, Palo Analytics, ICT4Growth, DemocracIT and MASTER (H2020).

Konstantinos Tserpes is an Assistant Professor at the Department of Informatics and Telematics of the Harokopio University of Athens. He holds a Ph.D. in the area of Distributed Systems from the school of Electrical and Computer Engineering of the National Technical University of Athens (2008). His research interests revolve around distributed systems, software and service engineering, Big Data analytics, and social systems. He has been involved in several EU and National funded projects leading research for solving issues related to scalability, interoperability, fault tolerance, and extensibility in application domains such as multimedia, e-governance, post-production, finance, e-health, and others. He has served as the scientific or general coordinator in several ICT projects such as +Spaces, SocIoS, Consensus, Fortissimo (FP7), and BASMATI (H2020) and the principal investigator for the projects ACCORDION, TEACHING, COLLABS, MASTER and SmartShip (H2020).

Iraklis Varlamis is an Associated Professor of Data Management at the Department of Informatics and Telematics of the Harokopio University of Athens. He holds a Ph.D. in Informatics from the Athens University of Economics and Business, Greece, and an MSc in Information Systems Engineering from UMIST, UK. He has been involved as a technical coordinator in a number of EU funded projects concerning knowledge management, data mining, machine learning, and expert systems. He has also coordinated several national R&D projects concerning data management and personalized delivery of information. He has authored more than 130 articles concerning text and graph mining and intelligent applications in social networks and the web and received more than 2200 citations. He has worked as a scientific coordinator or principal investigator in several collaborative research projects funded by National, EU (e.g. H2020 TEACHING, SustAGE, MASTER, FP7-Fortissimo) and International Funds (e.g. Qatar Funding-EM3).

Lívia C. Ruback Rodrigues obtained a D.Sc. in Informatics (2017) and an M.Sc. in Informatics (2013), both from the Pontifical Catholic University of Rio de Janeiro – PUC-Rio, and a B.Sc. in Computer Science from the Federal University of Juiz de Fora (2009). She currently is an Assistant Professor at the Department of Computing of the Federal Rural University of Rio de Janeiro and a member of the AI Inclusive Project. Her research interests include Linked Data, Semantic Web, Social Media, and Social Computing.

Data and codes availability statement

The original TripBuilder dataset was not created by the authors of this article, but the data and codes that support the findings of this study are available at https://doi.org/10.6084/m9.figshare.11559090.

Funding

This work was partly funded by grants [CAPES/88881.134081/2016-01, CNPq/302303/2017-0 and FAPERJ/E-26-202.818/2017]. This work was supported by the project "MASTER". MASTER project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement [No 777695]. The work reflects only the author's view and the EU Agency is not responsible for any use that may be made of the information it contains.

ORCID

Yenier Torres Izquierdo http://orcid.org/0000-0003-0971-8572
Grettel Monteagudo García http://orcid.org/0000-0001-9713-300X
Marco A. Casanova http://orcid.org/0000-0003-0765-9636
Luiz André P. Paes Leme http://orcid.org/0000-0001-6014-7256
Christos Sardianos http://orcid.org/0000-0001-7262-7310
Konstantinos Tserpes http://orcid.org/0000-0001-5183-1443
Iraklis Varlamis http://orcid.org/0000-0002-0876-8167
Lívia C. Ruback Rodrigues http://orcid.org/0000-0001-5000-2280

References

Alvares, L.O., *et al*. 2007a. Dynamic modeling of trajectory patterns using data mining and reverse engineering. *In*: *26th international conference on conceptual modeling, poster session*. Auckland, New Zealand: ACM, vol. 83, 149–154.

Alvares, L.O., *et al*. 2007b. A model for enriching trajectories with semantic geographical information. *In*: *15th annual ACM international symposium on advances in geographic information systems*. Auckland, New Zealand: ACM, 22.

Baader, F., *et al*. eds., 2003. *The description logic handbook: theory, implementation, and applications*. USA: Cambridge University Press.

Baglioni, M., *et al*. 2008. An ontology-based approach for the semantic modelling and reasoning on trajectories. *In*: *International conference on conceptual modeling*. *LNCS*. Berlin, Heidelberg: Springer, vol. 5232, 344–353.

Bast, H., Buchhold, B., and Haussmann, E., 2016. Semantic search on text and knowledge bases. *Foundations and Trends® in Information Retrieval*, 10 (1), 119–271. doi:10.1561/1500000032

Bergamaschi, S., *et al*. 2016. Combining user and database perspective for solving keyword queries over relational databases. *Information Systems*, 55, 1–19. doi:10.1016/j.is.2015.07.005.

Bogorny, V., *et al*. 2014. Constant – a conceptual data model for semantic trajectories of moving objects. *Transactions in GIS*, 18 (1), 66–88. doi:10.1111/tgis.12011

Brilhante, I., *et al*., 2014. TripBuilder: a tool for recommending sightseeing tours. *In*: *36th European Conf. on Information Retrieval (ECIR'14)*. Springer, Cham, 771–774.

Fileto, R., *et al*. 2013. Baquara: A holistic ontological framework for movement analysis using linked data. *In*: *International conference on conceptual modeling*. *LNCS*. Springer, Berlin, Heidelberg, vol. 8217, 342–355.

Fileto, R., *et al*. 2015. The baquara2 knowledge-based framework for semantic enrichment and analysis of movement data. *Data & Knowledge Engineering*, 98, 104–122. doi:10.1016/j.datak.2015.07.010.

Furtado, A.S., *et al*. 2016. Multidimensional similarity measuring for semantic trajectories. *Transactions in GIS*, 20 (2), 280–298. doi:10.1111/tgis.12156

García, G.M., *et al*. 2017. RDF keyword-based query technology meets a real-world dataset. *In*: *20th International Conference on Extending Database Technology*, Venice, Italy. OpenProceedings.

Ghanbarpour, A. and Naderi, H., 2019. A model-based keyword search approach for detecting top-k effective answers. *The Computer Journal*, 62 (3), 377–393. doi:10.1093/comjnl/bxy056

Han, S., *et al*. 2017. Keyword search on RDF graphs - A query graph assembly approach. *In*: *ACM Conference on Information and Knowledge, CIKM 2017*, Singapore, Singapore. ACM Press, vol. Part F1318, 227–236.

Hristidis, V. and Papakonstantinou, Y., 2002. Discover: keyword search in relational databases. *In*: *28th International Conference on Very Large Databases (VLDB'02)*. Hong Kong, China: VLDB Endowment, 670–681. Available from: https://linkinghub.elsevier.com/retrieve/pii/B9781558608696500652

Hu, Y., *et al*. 2013. A geo-ontology design pattern for semantic trajectories. *In*: *International Conference on Spatial Information Theory*. *LNCS*. Cham: Springer, vol. 8116. 438–456.

Izquierdo, Y.T., *et al*. 2018. QUIOW: A keyword-based query processing tool for RDF datasets and relational databases. *In*: *30th International Conference on Database and Expert Systems Applications (DEXA'18*), Regensburg, Germany. Springer, vol. 11030, 259–269.

Le, W., *et al*. 2014. Scalable keyword search on large RDF data. *IEEE Transactions on Knowledge and Data Engineering*, 26 (11), 2774–2788. doi:10.1109/TKDE.2014.2302294

Lin, X.Q., Ma, Z.M., and Yan, L., 2018. RDF keyword search using a type-based summary. *Journal of Information Science and Engineering*, 34 (2), 489–504.

Oliveira, P.D., Silva, A.D., and Moura, E.D., 2015. Ranking Candidate Networks of relations to improve keyword search over relational databases. *In*: *31st IEEE International Conference on Data Engineering (ICDE'15*), Seoul, S. Korea. IEEE, 399–410.

Parent, C., *et al*. 2013. Semantic trajectories modeling and analysis. *ACM Computing Surveys (CSUR)*, 45 (4), 42. doi:10.1145/2501654.2501656

Petry, L.M., *et al*. 2019. Towards semantic-aware multiple-aspect trajectory similarity measuring. *Transactions in GIS*, 23 (5), 960–975. doi:10.1111/tgis.12542

Renso, C., *et al*. 2013a. How you move reveals who you are: understanding human behavior by analyzing trajectory data. *Knowledge and Information Systems*, 37 (2), 331–362. doi:10.1007/s10115-012-0511-z

Renso, C., Spaccapietra, S., and Zimányi, E., 2013b. *Mobility data*. Cambridge: Cambridge University Press.

Rihany, M., Kedad, Z., and Lopes, S., 2018. Keyword search over RDF graphs using wordnet. *In*: *1st International Conference on Big Data and Cyber-Security Intelligence (BDCSIntell'18*), Hadath, Lebanon. CEUR-WS, vol. 2343, 75–82.

Santipantakis, G.M., *et al*. 2017. Specification of semantic trajectories supporting data transformations for analytics: the datacron ontology. *In*: *13th International Conference on Semantic Systems*. Amsterdam, Netherlands: ACM, 17–24.

Spaccapietra, S., *et al*. 2008. A conceptual view on trajectories. *Data & Knowledge Engineering*, 65 (1), 126–146. doi:10.1016/j.datak.2007.10.008

Tran, T., *et al*., 2009. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. *In*: *25th International Conference on Data Engineering, ICDE 2009*, Shanghai, China. IEEE, 405–416.

Wen, Y., Jin, Y., and Yuan, X., 2018. KAT: keywords-to-SPARQL translation over RDF graphs. *In*: *23rd International Conference on Database Systems for Advanced Applications (DASFAA'18*), Gold Coast, Australia. Springer, vol. 10827, 802–810.

Yan, Z., *et al*. 2008. Trajectory ontologies and queries. *Transactions in GIS*, 12, 75–91. doi:10.1111/j.1467-9671.2008.01137.x.

Zenz, G., *et al*. 2009. From keywords to semantic queries-Incremental query construction on the semantic web. *Journal of Web Semantics*, 7 (3), 166–176. doi:10.1016/j.websem.2009.07.005

Zheng, W., *et al*. 2016. Semantic SPARQL similarity search over RDF knowledge graphs. *42nd International Conference on Very Large Databases (VLDB'16)*, 9 (11), 840–851.

Zhou, Q., *et al*. 2007. SPARK: adapting keyword query to semantic search. *In*: *6th International Semantic Web Conference (ISWC'07*), Busan, Korea. Springer, vol. 4825, 694–707.