Ontology-Based Support for Graph Algorithms in Online Exploration Workflows

Thomas Hornung[1] and Wolfgang May[2]

[1] Institut für Informatik, Universität Freiburg,
hornungt@informatik.uni-freiburg.de
[2] Institut für Informatik, Universität Göttingen,
may@informatik.uni-goettingen.de

Abstract. Application domains often include notions that are inherently based on graph structures. In this paper, we propose a comprehensive generic ontology for graphs. It focuses on those aspects of graphs that are useful for workflows that require exploration of relevant parts of potentially large graphs by online algorithms. The goal of the ontology is to include as much information as possible supporting the graph exploration process declaratively into the specification of the graph. This allows to separate the (also declarative) specification of the actual exploration process from the maintenance of the graph itself. For concrete applications, the graph specification is given in RDF using this ontology. From the specification, an appropriate instantiation of the abstract datatype is automatically derived which is then used in information workflows.

1 Introduction

A recurring motive when designing informational workflows is the computation of (parts of) transitive closures in graphs. Graph algorithms in general are a traditional research topic; they usually assume a given graph and the focus is on employing additional suitable data structures for efficient algorithms. In the context of the Web, *online algorithms* [1, 4] became more relevant: there, the graph is neither known nor materialized a priori to run algorithms on it, but is *explored* only at runtime, using one or more Web data sources. Often, even the graph data itself is dynamic which does not allow for materialization or caching. These characteristics require completely different algorithms where the exploration and expansion strategy for the graph itself is the central issue. Most algorithms basically follow a *best-first-search* like A^* [15], *breadth-first-search*, or *depth-first-search* for exploration.

In this work, we present *CGDT (Configurable Graph DataType)* that provides an ontology and an API for *configurable* graphs. The design of CGDT combines generic graph behavior (insertion of edges etc.) with application-specific configurability. CGDT allows to encode the maintenance of the stored graph data inside the graph by (i) assigning properties to vertices, edges, and paths, and (ii) specifying how paths are obtained from existing edges and paths during the exploration process.

This allows to separate the (also declarative) specification of the actual exploration process and the basic acquisition of data from the Web from the maintenance of the graph itself. We show how to embed CGDT in the declarative specification of informational workflows that are specified in RelCCS [6], a process specification language that extends CCS [10] to relational data flow. RelCCS itself is based on the MARS framework [8, 5], an open framework that provides interoperability between nearly arbitrary languages that support relational dataflow. For the actual acquisition of data from the Web that takes place on-demand, MARS enables to embed query languages in RelCCS processes. In general, Web nodes that implement CGDT can act as Web-wide services for storing, maintaining and querying graph structures not only within MARS-based approaches. The prototype implementation of CGDT uses a relational database for storing the actual contents of the graphs.

Structure of the Paper. In the next section, we describe a concrete use case and analyze the general requirements and concepts for the CGDT ontology. Section 3 introduces the schema part of the ontology. Section 4 adds generic notions to specify how the graph develops during evaluation of an online algorithm. Section 5 discusses the implementation. Section 6 describes the exploration process of the chosen use case. Section 7 gives an overview of related work, and Section 8 concludes.

2 Application Scenario and General Considerations

Consider the problem to find either the cheapest or shortest (in terms of total time spent travelling) route to a given location (e.g., for a conference travel) or a combination of both. Human, manual search usually employs some kind of intuitive strategy. Roughly, the strategy is to start with considering a known set of airports near the hometown and to try to cover as much distance as possible by plane (assuming the distance is above a certain threshold), and then bridge the remaining distance by train or bus; if this fails, do backtracking. This shows that, although human problem solving usually considers one possibility (=tuple) at a time, in this case it is inherently based on a set-oriented model.

With the means of the presented approach, such tasks can be formulated as data workflows. The backtracking is here replaced by a search strategy, where the search space is explored stepwise and pruned based on intermediate results. While for train connections, sources usually are able to return transitive connections, flight portals only return transitive connections over the flights of the same airline. Thus, here an actual graph exploration is required. An important aspect that is typical for this use case (and many other ones) is that the search is subject to additional constraints, like arrival and departure times and required time for changing.

The expected answer is the *set* of k best alternatives (wrt. a weighted function of price and duration), where each solution contains the actual connection data (flight and train numbers, departure/arrival times). Furthermore, it should in

general be possible to extend the process specification in such a way that the best available one is actually booked automatically.

Pitfalls. Experiences with conference travels showed that real travel agencies are often challenged with finding the potential nearest airports to less standard destinations (e.g., St. Malo/France as for ICLP 2004), and are rather weak in finding non-direct flight connections using different airlines (e.g. Lufthansa + AirFrance) or via unexpected intermediate airports (via London Stansted to reach Dinard/France), or surprising connections (fly to Jersey Island and take the ferry to St. Malo) – actually, ferries are often contained in the railway portals, so we do not consider these separately. The latter example shows also that it would not be advantageous to try to save time by predefining the set of destination airports by the user, but to use a fully algorithmic search that is not biased in any way.

Comparison to Classical Graph Algorithms. On first sight, the problem looks like an application for classical "shortest path" graph algorithms like Prim [11, 2] or Kruskal [7, 2]. A more detailed analysis shows that even under some optimistic assumptions, this would not be an appropriate solution:

- Dynamics: the complete graph is not available, and is continuously changing: the availability of flights and their prices changes every moment.
- Constraints: the paths are further constrained by the requirement that the departure time must be after the arrival time at intermediate airports.
- Completeness: the airline connections' graph (which is, neglecting the availability issue, of a size that could efficiently be processed by breadth-first or A^* search), is not sufficient. Additionally, the connections between airports and the final destination must be considered. Thus, finding the solution in the graph depends on further information since below a certain remaining distance the process continues outside the main graph.

Generalization. The above considerations show that in such cases, a large search space has to be explored, and application-specific properties of the paths, like price and duration, have to be maintained *incrementally.* The stepwise exploration corresponds to *inductive* characterizations of these properties that are in fact common to the idea of properties of paths in a graph. The CGDT ontology provides generic notions to specify how this information is combined from the actual input (i.e., information about edges obtained from Web sources). The relevant features of the actual domain ontology are mapped and expressed in terms of the generic ontology.
CGDT supports the following generic functionality:

- materializing the relevant graph fragment (including the inductively defined properties) based on the explored edges,
- creating paths according to specified criteria,
- accessing those recently added vertices that serve for continuing the exploration,

– finally querying the result graph.

The design of the domain-specific workflow can then be separated into three issues:

– describe the domain-specific characteristics of the graph in terms of the CGDT ontology. This consists of the basic schema of the graph, the *constructive specification* how to extend the graph with relevant newly obtained knowledge, and *constraints* when newly obtained knowledge is relevant to expand the graph;
– actual acquisition of the data from the Web (including Deep Web sources). This means to identify appropriate data sources and to encode the access to them. Potentially, for each step also two or more sources must be accessed – for instance one to identify potential edges (in our example: which airports can be reached from a given one), and the second to query for the actual existence of the edges (in our example: actual availability and departure/arrival times of that connection for a given date);
– fill in a common breadth-first-search or A^*-search workflow pattern as a RelCCS process with case splits and Web queries.

After configuring the graph once in course of the initialization, the process will only submit edges to the graph, and query it for the vertices where the exploration should be continued. The compilation of the information in the graph itself, and the choice of the vertices for the next step is done automatically by the graph.

3 An Ontology for Graphs in Online Algorithms

The basic notions of any graph ontology are vertices, edges, and paths. In the following, we consider directed, labeled graphs of the form $G := (V, E, P)$, where V is the set of vertices, $E \subseteq V \times V$ is the set of directed edges between these vertices, P is a set of paths. While in the usual notion of graphs, the set of paths is defined as the transitive closure of edges (i.e., the set of paths is $\{(v_1, \ldots, v_n) \mid (v_1, v_2), \ldots, (v_{n-1}, v_n) \in E\}$), the set P of relevant paths in a configurable graph is a certain *subset* of all existing paths in the graph that satisfy additional constraints. Nevertheless, each path $p \in P$ is a path in the traditional sense which consists of multiple connected edges. A path p that ends in a vertex x can be extended by an edge (x, y), denoted by $p \circ (x, y)$. The set P will contain paths that are obtained by such extension steps according to configurable criteria.

3.1 Properties

A central feature of CGDT is that vertices, edges and paths can be adorned with sets VP, EP, and PP of (typed) properties. Each vertex property is associated with a literal type, taken from the XML Schema datatypes [18].

The properties can optionally be specified in terms of view definitions over other properties, or by external queries. For instance, given a vertex with its

airport code, the timezone can be obtained by a suitable Web query. The distance of a flight from A to B is the geographical distance between A and B's coordinates, and the price of a path is the sum of the prices of its edges.

For properties of vertices and edges where no definition is given, the value must be given when adding the edge to the graph. Often, vertices are added only with their key (when found by exploring edges), and their additional properties must be obtained by external queries that are automatically executed when the vertex is inserted. As paths are not inserted manually, but only by extending an existing path with an edge, all path properties must be derived properties. Here, often an inductive definition over the length of the paths is used.

3.2 Signature and Operations

The operations of CGDT are divided into a *Data Definition Language (DDL)* where the properties and the constraints are *defined*, and a *Data Manipulation Language (DML)* that provides *generic* operations for updating and querying the graph which are used during the actual process of exploration.

3.2.1 The DDL

While in SQL and related languages, the DDL has an own syntax, the DDL of CGDT is actually the *ontology language* RDF [12] that *declaratively* specifies which properties exist, together with their definitions, and with the constraints how to expand the graph.

In contrast to SQL, where the main notion of the schema is the *table*, the CGDT is based on three subschemas, i.e., a VertexSchema, an EdgeSchema, and a PathSchema. Each of them defines some properties (i.e., VertexProperties, EdgeProperties, and PathProperties) and optionally some constraints (to be discussed in Section 4) that guide the exploration process. Each of the subschemas can be regarded (and stored) as a table (as will be illustrated later wrt. Figure 2). The notions of the generic graph ontology itself (i.e., the DDL notions) are depicted in UML in Figure 1; an excerpt of the RDF Schema [13] definition is given in Turtle notation [17] in Appendix A.1.

The three subschemas contain some mandatory, built-in properties:

- vertex schema: id serves as key,
- edge schema: id (key, internally generated and used), from and to, referring to vertices x and y for an edge (x, y). Note that from and to are not key to allow different edges between the same vertices (e.g., several flights at the same day),
- path schema: id (key, internally generated and used), from, to, front and last, referring to a, y, p and (x, y) (the latter two referring to ids of an edge and a path, respectively) for the path $p \circ (x, y)$ where a is the first vertex of p.

A concrete application-specific CGDT specification then defines

- the names and datatypes of the additional application-specific properties of each subschema,
- the definitions of the derived properties,

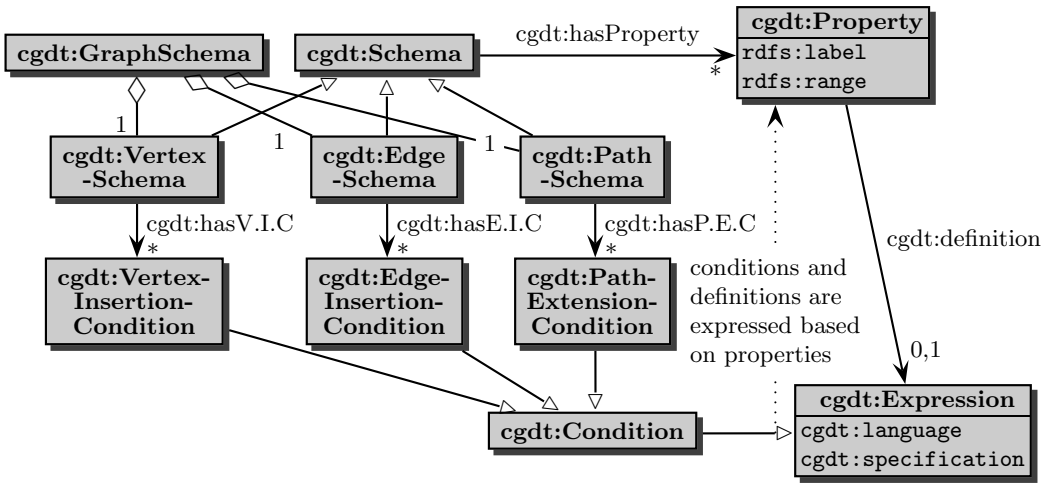


Fig. 1. Basic Notions of the CGDT Ontology

– conditions to configure the exploration process (that will be discussed in Section 4).

Derived Properties. Since derived properties are *views* over other properties or Web queries, they can be expressed by query languages. For accessing the Web, external queries can be embedded using the language management of the MARS framework. Throughout this paper, we use pseudocode expressions. Properties of paths are often defined inductively. For these, the specification of the base case (which is an edge, and thus builds upon the edge's properties) and of the inductive step (potentially using the path and the extending edge) have to be given. Instead of giving an inductive definition, path properties can also be specified to be SumProperties, CountProperties, or {Min|Max}Properties, which are defined as the aggregation of the values of a specified edge property.

Example 1 *In our running example, the concrete instantiation of CGDT is tailored to the travel application scenario and rooted shortest path search.*

The vertices (which are the train stations and airports) have two properties, i.e., the id *(which is e.g. the airport code) and the timezone. The timezone is defined by a Web query (against a wrapped Web source)*

timezone = getTimezone(<http://www.theairdb.com>, code) .

Edges, which are the direct connections, e.g., FRA-CDG (Frankfurt to Paris Charles de Gaulle), have domain-specific properties code *(the flight number),* dept*,* arr *(departure and arrival time wrt. the local timezone) and* price*. The* duration *is a derived property:*

duration = arr - dept + from.timezone - to.timezone.

The properties of the paths, from*,* to*,* dept*,* arr*,* price *and* duration *are defined inductively. For the base case where a path is just a single edge, they have the same values as for the edge. For paths of length* > 1*, they are defined as follows:*

from = front.from (built-in), dept = front.dept,
to = last.to (built-in), arr = last.arr,
price = front.price + last.price or equivalently as a SumProperty
= sum[e:edge](e.price) (= sum of prices of all edges of the path)
duration= front.duration + last.duration + last.dept - front.arr
which equals last.arr - front.dept + from.timezone - to.timezone.

The RDF specifications of the actual schemas of the CGDT for the traveling example are given in Appendixes A.2 and A.3 (where we use the actual CGDT XPath-based syntax for expressions).

The Constructor. The constructor *gid* ← getGraph(*rdf-spec*) initializes a new CGDT instance with a given specification *rdf-spec* (which is an RDF specification of the desired instance as given in Appendixes A.2 – A.4) and returns a unique graph id.

3.2.2 The DML

The DML is also independent from the actual application domain (similar to e.g. SQL as DML for relational databases). The modifiers allow to add items to the graph:

- addVertex(*id*, *vertex-property-name-value-pairs*) adds a new vertex *id* with the given vertex property values,
- addEdge(*from*, *to*, *edge-property-name-value-pairs*) adds a new edge (*from, to*) with the given edge property values (and adds the target vertex if not yet present).

In the pseudocode, we use a slot-based notation, e.g. addEdge("FRA", "CDG", [dept ← "10:30", arr ← "11:50", code ← "LH123", price ← 185.00]).
The accessors include the following:

- *var* ← getNewVerticesBFS() supports breadth-first-exploration and binds *var* to the ids of each of the new vertices that have been added since the previous call of getNewVerticesBFS(),
- *var* ← getNextVertexAStar() binds *var* to the id of the next vertex that must be extended according to A^* best-first-search (and a given valuation function, cf. Sec. 4.4),
- $(v_1, \ldots, v_n)$ ← getResultPaths(v_1 ← *attr*$_1$, …, v_n ← *attr*$_n$) returns a binding for variables $(v_1, \ldots, v_n)$ to the corresponding attributes of each path that is considered as a result. In Section 4.4 we will discuss how the intended result paths are specified in the ontology.

4 Configurability of the Exploration Process

Although breadth-first-search, best-first-search and depth-first-search proceed different in the large, the *configuration* of the exploration process can be specified by the same notions. In this paper, we exemplify it for the use in *breadth-first-search*, which illustrates the set-oriented features best by doing the expansion in parallel.

4.1 Breadth-First Search

The underlying principle of breadth-first-search is simple and makes the strategy well-suited for graph exploration in online algorithms: Starting with a set of one or more known vertices (e.g., airports), consider all edges from these vertices to any other (known or yet unknown) vertex. These edges are added to the graph, and (i) can be used to extend existing paths, and (ii) result in newly known vertices that can be used in the next step.

The configuration of the behavior of the graph consists of conditions that specify the following:

1. when a new edge is found, add it to the graph or discard it (e.g., when certain airlines or intermediate airports should be excluded),
2. when a new edge is inserted: under which conditions can it be used to extend an existing path p (e.g., its departure time must obviously be later than the arrival time of p),
3. under which conditions should a vertex be considered for the next exploration step?

By this, CGDT separates the acquisition of edges (that must be programmed explicitly in the processs) from the actual handling of their contributions to the graph (that is configured into the graph).

4.2 Insertion Conditions

For vertices and edges, conditions can be stated that need to be satisfied for insertion of the item into the graph. *Vertex insertion conditions* are *only* concerned with properties of the vertex itself (e.g., the exclusion of flights via London Heathrow (LHR) because of luggage handling problems can be expressed by the condition $\mathsf{id} \neq$ "LHR"). *Edge insertion conditions* are *only* concerned with properties of the edge itself (e.g. $\mathsf{duration} <$ "10:00"), its start and end vertices, and with general properties of a graph (e.g., forbid to make the graph cyclic). An edge is also not inserted if one of its vertices does not satisfy the vertex insertion conditions. The RDF specification of the conditions for the traveling use case is given in Appendix A.4.

4.3 Path Extension Conditions

Path Extension Conditions allow to state application-specific constraints when a new edge (x, y) can be *used* for extending a path p that ends in x to $p \circ (x, y)$. They are formulated in terms of the properties of the edge and of the path.

Example 2 *In our example, for a path* $((s, \ldots, x), [\mathsf{arr} = t_1])$ *and a new edge* $(x, y, [\mathsf{dept} = t_2])$*, the new path* $((s, \ldots, x, y), [\ldots])$ *is only added if* t_2–t_1> *"01:00".*

Consider an invocation of $\mathsf{addEdge}(x, y, [...])$ *(i.e. a direct connection). If the destination airport* y *is not yet contained in the graph, it is added as a vertex (automatically retrieving its* timezone *property from the Web). The connection itself is added as an edge with its properties, and for all paths* $p = (s, \ldots, x)$*, the path* $p' = (s, \ldots, x, y)$ *is a candidate for insertion. If the new edge's departure*

is more than one hour later than p's arrival, p' is actually inserted with the appropriately computed property values.

If for such newly added paths, edges (y, z) are already stored, the respective extended paths $(s, \ldots, x, y, z)$ are also candidates for insertion, and so on. Note that edges (like in the example $(x, y, [\mathsf{dept} = t_3])$ with $t_3 < t_1$) that cannot yet be used for extending an (already known) path can possibly be used later for extending other paths that reach y with an earlier arrival time than p. For that, path extension conditions are usually stricter than edge insertion conditions. Vertices should only be considered as "new" to be extended in the next step if they became actually newly reachable by a path.

4.4 Specification of Desired Result Paths and Termination

The above conditions control *how* the internal information of the CGDT instance is extended when adding edges. Additionally, it must be specified, when the process ends, and preferably already during the process, only new vertices that are promising to continue the search should be selected for the next step.

The *Result Specification* is expressed via a filtering condition which paths can qualify as intended results (in the example, those that end in the final destination), and optionally a valuation function on paths (that can be seen as a cost measure and that must be strictly monotonic wrt. path extension) and an integer k, how many results should be finally returned.

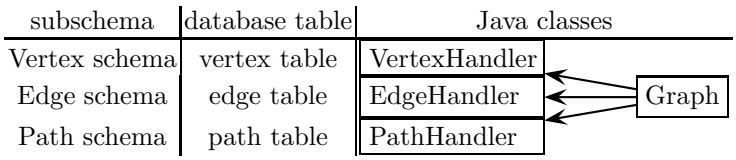
Example 3 *In our example, the valuation function is* duration (in hours) + price / 100 *(means, for 100€ saved, one accepts one more hour to travel). The filter condition is* to = finalDestination*, and* $k = 5$.

When breadth-first-search is applied, paths that are "above" (i.e. more expensive) the limit of the best k results so far are not further extended, and vertices that are only reachable by such paths are not expanded. This prunes the search space as soon as k paths have been found that satisfy the filter condition, and guarantees termination. In case of A^* search, the valuation function is used to choose the next vertex to be extended.

5 Implementation

CGDT is implemened as a Web Service that consists of two components: (i) an underlying database for storing the actual graphs, and (ii) the CGDT engine that translates graph specifications into database tables and that processes the abstract datatype's DML operations according to the conditions stated in the configuration.

The engine is mainly based on a *Graph* class. Each graph is represented by an instance of this class. Since each graph potentially has a different schema, each graph is stored separately. For each subschema, there is a database table that holds the data, and a *Vertex/Edge/PathHandler* instance on the Java level that manages the access operations on the table.



Processing of the DDL: Graph Specifications. The *Graph* constructor is invoked with the graph specification that contains the three subschemata and the constraints as introduced in Section 3.2.1. A *Graph* instance is created with the appropriate handlers that store the definition of the derived attributes, the insertion conditions etc., and the tables are created with the attributes of the respective subschema.

Processing of the DML. DML operations are submitted by the Web Service to the respective *Graph* instance and further delegated to appropriate methods of the Handler instances.

Upon insertion of a new item, first its derived attributes are computed, and then, if it satisfies the respective insertion condition, it is inserted in the respective database table. Upon insertion of an edge, additionally, the target vertex is added to the vertex table if not yet present, and the paths that are candidates for extension are selected (including evaluation of the path extension condition) and the extensions are inserted into the path table.

The accessors getNewVerticesBFS() and getNextVertexAStar() (implemented in the *VertexHandler*) select the ids of the appropriate items from the database and return them (additional internal columns for supporting the chosen search variant are maintained in the vertex table). Access to properties of certain vertices or edges via their id is also supported (including path expressions like *path*.front.to.timezone).

The accessor getResultPaths(*slot-selection-spec*) (implemented in the *PathHandler*) selects the paths from the path table that satisfy the ResultSpecification and returns the specified set of variable bindings.

6 The Exploration Process

Instances of CGDT are used in processes that invoke the modifiers and accessors and usually follow an A^* search or breadth-first-search pattern. Since the CGDT itself takes care of the maintenance of the graph data, the design of the process is mainly concerned with acquisition of data (i.e., edges) from the Web and controlling case-splits. For formalization of the process, we use –as already mentioned in the introduction– RelCCS [6]. RelCCS provides the common constructs for sequences, alternatives, concurrent branches, recursion, etc. For atomic constituents of processes, in our case, these are queries (e.g., Web queries in XPath/XQuery, Web Services calls, Deep Web queries), appropriate query languages can be embedded using the language management of the MARS framework [5].

6.1 Application Scenario: Travel Planning

We illustrate the approach with the above setting on travel planning. We abstract from the concrete data sources, which are for this example actually wrapped Deep Web sources:

1. `http://www.bahn.de` for (not only German) railways,
2. `http://www.theairdb.com` that provides information about airports and the existing (direct) connections between them, e.g., `http://www.theairdb.com/airport/FRA.html`. The source only provides for each airport X which other airports are reachable by direct flights, and by which airlines, but not the actual schedule. Updates to this data are infrequent.
3. `http://www.travenjoy.com` can be queried for available flights between airports X and Y at a given date, and reports departure, arrival time, and price. This data is continuously changing (availability, price contingents); thus an evaluation at runtime is inevitable.

Thus, the whole connection graph is not accessible like a database. Instead, the relevant fragment of the connection graph is built stepwise during the workflow, using a CGDT instance. Edges (i.e., connections) and their properties are obtained from Web queries, and added to the graph.

The workflow is started for given *start*, *dest*, and *date*, and proceeds as follows, implementing a breadth-first-search for shortest paths: If the overall distance is less than 400km, only train connections are searched for. Otherwise, train connections (for less than 1000km) and flights are investigated. For the latter, the 15 nearest airports to the starting point are selected by a query against a static geographic database. For these, the train connections are queried and added to the graph (with departure time, arrival time, and price).

With this situation, the iterative exploration starts: the remaining distance is bridged by connecting flights and, if necessary, a final train connection. For all vertices x (airports) that are relevant for the next exploration step (i.e., became reachable by a possible connection in the previous step), the following is done:

- if the remaining distance from x to the final destination is below 500km, train connections from x to *dest* are searched, and also added as edges to the graph.
- if the remaining distance from x to the final destination is more than 200km, all airports y connected by directed flights (obtained from a query against source (2)) are selected. For each such connected pair (x, y), source (3) is queried for *available* flights. Each connection is added as an edge to the graph. The graph will extend *only* paths with such an edge that satisfy the *path extension condition* that the connection leaves at least one hour later than arrival at x. Note that if y is already in the graph and outgoing edges from y are stored, further path extension is immediately applied for these.

With this, new vertices are added to the graph, and the iteration continues. The vertex relevance condition guarantees that the process terminates when k connections to the destination have been found, and all "open" paths are already more expensive than the best k known paths.

This guarantees that e.g., better paths over 4 steps are considered even if a more expensive connection has been found over 3 steps.

6.2 The RelCCS Process

The above strategy is implemented as a RelCCS process. Amongst the whole RelCCS functionality, that we regard mainly as a black box here, we will use the sequence operator ccs:Seq(_,...,_), the non-exclusive ccs:Union of branches, and the ccs:Topk operator, i.e., ccs:TopK(k, *variable*, *datatype*, *order*) is the relational operation that removes all tuples except those with the k lowest (*order* = *"ascending"*) values of *variable*. Recursion is expressed by giving a ccs:ProcessDefinition and invoking it by ccs:UseDefinition. While the RelCCS syntax in MARS is defined as an XML markup, we use an intuitive pseudocode notation.

The execution model uses relational dataflow, i.e., is based on *sets* of tuples of variable bindings. The process uses variables $start$ and $dest$ (destination), $date$ (which are initialized when calling the process), ap (relevant near airports), $dist$ (distance to airport), i, j (intermediate places), rd (remaining distance), dt and at (departure and arrival time), pr (price), and gid (graph id). We use the prefixes ccs and cgdt to indicate the respective languages. The iteration is encoded into a recursive process definition runGraph with local variables i, j, rd, dt, at, pr.

```
# process input: (start, dest, date)
ccs:Seq(ccs:Query(rd ← distance(start, dest)),
  ccs:Query(gid ← cgdt:newGraph(rdf-spec)),   # the config spec discussed above
  ccs:Union(
    ccs:Seq(ccs:Test(rd < 1000),  # consider to go by train
      ccs:Query((dt, at, pr) ← getTrainConnection(start, dest, date)),
      ccs:Action(cgdt:addEdge(gid, start, dest, [dept ← dt, arr ← at, price ← pr]))),
    ccs:Seq(ccs:Test(rd ≥ 400),   # consider also to use flights
      ccs:Query(ap ← getAirports()),  # all known airports
      ccs:Query(dist ← distance(start, ap)),  # compute spherical distance
      ccs:TopK(15,dist,xsd:decimal,ascending), # consider 15 nearest airports
      ccs:Query((dt, at, pr) ← getTrainConnection(start, ap, date)),
      ccs:Action(cgdt:addEdge(gid, start, ap, [dept ← dt, arr ← at, price ← pr])),
      ccs:Projection(gid, start, dest, date), ccs:Distinct,
      ccs:UseDefinition(runGraph[ ]))),
    cgdt:getResultPaths(gid, [pathId ← pid, price ← price])),

  ccs:ProcessDefinition(runGraph[local:i, j, rd, dt, at, pr]) :=
    # global vars gid, dest, date are known
    ccs:Seq(
      ccs:Query(i ← cgdt:getNewVerticesBFS(gid)), # consider all newly reached places
      ccs:Query(rd ← distance(i, dest)),
      ccs:Union(
        ccs:Seq(ccs:Test(i = dest)),  # no recursive call in this case → return
        ccs:Seq(ccs:Test(rd < 500 ∧ i ≠ dest )  # reach destination by train
          ccs:Query((dt, at, pr) ← getTrainConnection(i, dest, date)),
```

```
        ccs:Action(cgdt:addEdge(gid, i, dest, [dept ← dt, arr ← at, price ← pr])),
        ccs:Projection(gid, start, dest, date), ccs:Distinct,
        ccs:UseDefinition(runGraph[ ])),
      ccs:Seq(ccs:Test(rd ≥ 200), # try to get even nearer by flight
        ccs:Query(j ← getConnectedAirports(i)),   // www.theairdb.com
        ccs:Query((dt, at, pr) ← getFlights(i, j, date)),  // www.travenjoy.com
        ccs:Action(cgdt:addEdge(gid, i, j, [dept ← dt, arr ← at, price ← pr]))
        ccs:Projection(gid, start, dest, date), ccs:Distinct,
        ccs:UseDefinition(runGraph[ ])))
# postcondition: connections to the destination, either by train or train+flight+,
# or train+flight+train    (for simplicity, we do not deal here with the the case
# that a place is not reachable by train).
```

The workflow proceeds stepwise, set-oriented:

Example 4 *Consider to start the workflow with the single tuple*

{*start*/"Göttingen", *dest*/"St.Malo", *date*/"1.9.2009"} .

The query for the remaining distance extends the tuple to

{*start*/"Göttingen", *dest*/"St.Malo", *date*/"1.9.2009", *rd*/813} .

It starts then two branches, one for a train-only travel (since $rd < 1000$*), and one that includes consideration of flight connections (since* $rd > 400$*); the results of both will be collected at the end. We follow the second one: It will first evaluate a query for all known airports, which results in a set of (hundreds of) tuples, where each is extended in the subsequent step with distance between start and the respective airport, i.e.,*

{{*start*/"Gö.", *dest*/"St.Malo", *date*/"...", *rd*/813, *ap*/"FRA", *dist*/250},
{*start*/"Gö.", *dest*/"St.Malo", *date*/"...", *rd*/813, *ap*/"HAJ", *dist*/100},
{*start*/"Gö.", *dest*/"St.Malo", *date*/"...", *rd*/813, *ap*/"MLH", *dist*/550},
:
{*start*/"Gö.", *dest*/"St.Malo", *date*/"...", *rd*/813, *ap*/"JFK", *dist*/6230}, ...}

The following TopK *step keeps the 15 nearest ones, amongst them FRA (Frankfurt), HAJ (Hanover) and MLH (Basel-Mulhouse), but e.g., not JFK. For these, the next step looks up (multiple) train connections for each of them, binding the variables* dt, at, *and* pr, *and for each tuple, the connection is put into the graph (the destinations are also added to the vertex table, including the lookup of the timezones), and the tuples are projected back down to* $start, dest, date$, *and duplicates removed (so only the single tuple* {*start*/"Gö.", *dest*/"St.Malo", *date*/"1.9.2009"} *remains). Then, the process definition for* runGraph *is invoked. In its first step, the new vertices, which are the 15 nearest airports, are retrieved from the graph and bound to the variable* i *(intermediate):*

{{*start*/"Gö.", *dest*/"St.Malo", *date*/"1.9.2009", *i*/"FRA"},
{*start*/"Gö.", *dest*/"St.Malo", *date*/"1.9.2009", *i*/"HAJ"}, ... } (15 tuples)

The subsequent iterations extend the graph in parallel (i.e., set-oriented for all tuples) breadth-first search until connections to dest are found. This process is strongly based on the configured functionality of the graph service. Recall that

when a reached airport is less than 500km away from the destination (these are e.g. the airports RNS (Rennes, 60km), DNR (Dinard, 10km), and JER (Jersey, 60km), CDG (Paris CDG, 320km)), train (including ferry) connections to St.Malo are investigated.

Figure 2 illustrates some of the vertices, edges, and paths stored in the graph and their evolution. The sample illustrates that an edge can be used to extend multiple existing paths (CDG[14:40]-StM[19:20] by train, CDG[16:40]-RNS[17:50] by plane) [*]. Amongst the first solutions found (after 3 steps) there is the fastest one, but not the cheapest one. [***] illustrates that when RNS is expanded for the first time, the trains (RNS[15:30]-StM[16:20]) and (RNS[19:30]-StM[20:20]) are immediately appended to all paths that end in RNS. So, these 4-step paths are already created in the 3rd round. To guarantee completeness, paths all around the world are explored, until they are more expensive than the 5 best results.

Surprisingly, the fastest connection is via the quite distant airport MLH (Basel-Mulhouse) that features a direct connection to Rennes by the regional French "Airlinair" (which motivates to have the initial top-k consideration of "near" airports quite large). The cheapest (and still reasonable fast) is by Ryanair via London-Stansted (STN) to the rather unknown Dinard (DNR, that has only connections to UK). The straightforward connections via Hanover (HAJ) or Frankfurt and Paris CDG are expensive and time-consuming.

For the given use case, breadth-first-search investigates more edges than A^* would do, but for longer travels, first results are returned earlier (A^* would tend to first combine short, cheap steps without really bridging distance). For changing to A^*, an appropriate valuation function must be added to the CGDT configuration, but the process definition does not need any changes except for the `getNextVertexAStar()` *call; it must be continued until k paths are found.*

7 Related Work

The notions of *online algorithms* [1] in general and *dynamic graph algorithms* [4] cover a broad spectrum of aspects. This includes scenarios where the *current* situation is completely known, but *changes*, as well as situations where the underlying situation is actually static, but is not completely known and is processed *incrementally*, like dynamic search algorithms. CGDT is tailored to the special, but still very common case where exploration is dynamic, but monotonic: vertices and edges once added to the graph will remain unchanged forever. The underlying graph is also dynamic, but every run is based on a (non-transactional) snapshot that is *explored* dynamically.

Online algorithms over unknown graphs are investigated by many authors under different aspects (total exploration [3], search etc.). For path search, breadth-first-search and best-first-search by A^* (see e.g. [15] for an overview) are the most prominent ones.

Also, research on composition of Web Services like [14, 9] is a related area, but in general deals with a higher level of abstraction where the concrete modeling and algorithmic handling of the data is not described. Such approaches can be

VERT_GRAPH42

id	timezone
Gö	+1
HAJ	+1
FRA	+1
MLH	+1
CDG	+1
JER	0
:	:

EDGES_GRAPH42

id	from	to	code	dept	arr	price	duration
e_1	Gö	HAJ	ICE631	7:50	8:50	49.00	01:00
e_2	Gö	FRA	ICE777	7:30	9:30	89.00	02:00
e_{12}	Gö	MLH	ICE777,Bus	6:30	11:20	129.00	04:50
e_{13}	HAJ	CDG	LH123	10:00	11:20	229.00	01:20
e_{15}	HAJ	JER	RY002	13:50	14:30	536.00	01:40
e_{43}	FRA	CDG	LH456	11:50	13:00	239.00	01:10
:	:	:	:	:	:	:	:

PATHS_GRAPH42

id	front	last	(comment)	from	to	dept	arr	price
p_1	null	e_1	Gö→HAJ	Gö	HAJ	7:50	8:50	49.00
p_3	null	e_2	Gö→FRA	Gö	FRA	7:30	9:30	89.00
p_{12}	null	e_{12}	Gö→MLH	Gö	MLH	6:30	11:20	129.00
p_{26}	p_1	e_{13}	Gö→HAJ→CDG	Gö	CDG	7:50	13:30	278.00
p_{28}	p_1	e_{15}	Gö→HAJ→JER	Gö	JER	7:50	14:30	585.00
p_{33}	p_1	e_{20}	Gö→HAJ→STN	Gö	STN	7:50	11:50	129.00
p_{48}	p_3	e_{43}	Gö→FRA→CDG	Gö	CDG	7:30	13:00	328.00
p_{53}	p_3	e_{48}	Gö→FRA→JFK	Gö	JFK	7:30	17:00	628.00
p_{63}	p_{12}	e_{61}	Gö→MLH→RNS	Gö	RNS	7:30	14:10	568.00
p_{84}	p_{26}	e_{74}	Gö→HAJ→CDG→StM	Gö	StM	7:50	19:20	375.00 [*]
p_{85}	p_{48}	e_{74}	Gö→FRA→CDG→StM	Gö	StM	7:30	19:20	425.00 [*]
p_{86}	p_{26}	e_{75}	Gö→HAJ→CDG→RNS	Gö	RNS	7:50	17:50	519.00 [*]
p_{87}	p_{48}	e_{75}	Gö→FRA→CDG→RNS	Gö	RNS	7:30	17:50	569.00 [*]
p_{93}	p_{28}	e_{84}	Gö→HAJ→JER→StM	Gö	StM	7:50	17:30	628.00 [**]
p_{98}	p_{33}	e_{91}	Gö→HAJ→STN→DNR	Gö	DNR	7:50	15:30	208.00
p_{100}	p_{53}	e_{48}	Gö→FRA→JFK→ALB	Gö	ALB	7:30	19:30	817.00
p_{101}	p_{63}	e_{97}	Gö→MLH→RNS→StM	Gö	StM	7:30	16:20	628.00 [***]
p_{103}	p_{86}	e_{98}	Gö→HAJ→CDG→RNS→StM	Gö	RNS	7:50	20:20	579.00 [***]
p_{104}	p_{87}	e_{98}	Gö→FRA→CDG→RNS→StM	Gö	RNS	7:30	20:20	629.00 [***]
p_{116}	p_{98}	e_{104}	Gö→HAJ→STN→DNR→StM	Gö	DNR	7:50	17:00	218.00
:	:	:	:	:	:	:	:	:

Fig. 2. Sample contents of the CGDT vertex, edge and path tables

complemented with the use of CGDT, since it declaratively covers the data-oriented aspects, as shown in the sample process in Section 6.

Most works on graph schemas have a different goal, namely to describe a *graph-based data model* in the sense of semistructured data like RDF on the schema level by the labels of its vertices and edges. In these languages the graph is not part of the domain and it is not used for applying graph algorithms, but the domain is modeled as a graph which is updated and queried. Some works in the area of graph transformations, e.g. PROGRES [16] allow –like CGDT– to assign (optionally derived) attributes not only to vertices, but also to edges and paths. Paths are seen as derived edges that are declared in a rule-based way.

8 Conclusion

We presented an ontology for a configurable graph datatype CGDT that supports explorative online algorithms using Web information sources. CGDT allows to declaratively specify and encapsulate the handling of the collected graph data, and to separate it from the data acquisition and process control.

A prototype of the implementation has been completed. An online prototype for MARS and RelCCS, with further documentation and preliminary fragments of the above process can be found at http://www.semwebtech.org/mars/frontend/ → run CCS Process.

References

1. S. Albers. Online Algorithms: A Survey. *Math. Prog.*, 97(1-2):3–26, 2003.
2. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*.
3. X. Deng and C. H. Papadimitriou. Exploring an Unknown Graph. In *FOCS*, pages 355-361, 1990.
4. D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.
5. O. Fritzen, W. May, and F. Schenk. Markup and Component Interoperability for Active Rules. In *Web Reasoning and Rule Systems (RR)*, Springer LNCS 5341, pages 197–204, 2008.
6. T. Hornung, W. May, and G. Lausen. Process algebra-based query workflows. In *CAiSE*, Springer LNCS 5565, pp. 440-454, 2009.
7. J. Kruskal. On the shortest spanning subtree and the traveling salesman problem. *Proceedings of the American Mathematical Society*, (7):48–50, 1956.
8. W. May, J. J. Alferes, and R. Amador. Active rules in the Semantic Web: Dealing with language heterogeneity. In *RuleML*, Springer LNCS 3791, pages 30–44, 2005.
9. S. A. McIlraith and T. C. Son. Adapting GOLOG for composition of Semantic Web Services. In *KR*, pages 482–496. Morgan Kaufmann, 2002.
10. R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, pages 267–310, 1983.
11. R. C. Prim. Shortest connection networks and some generalisations. *Bell System Technical Journal*, (36):1389–1401, 1957.
12. Resource Description Framework (RDF). `http://www.w3.org/RDF`, 2000.
13. Resource Description Framework (RDF) Schema specification. `http://www.w3.org/TR/rdf-schema/`, 2000.
14. D. Roman and M. Kifer. Reasoning about the Behavior of Semantic Web Services with Concurrent Transaction Logic. In *VLDB*, pages 627–638. 2007.
15. S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach, 2003.
16. A. Schurr, A. J. Winter, and A. Zündorf. The Progres approach: language and environment. In *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*. World Scientific, 1999.
17. Turtle - Terse RDF Triple Language. `http://www.dajobe.org/2004/01/turtle/`.
18. XML Schema part 2: Datatypes. `http://www.w3.org/TR/xmlschema-2`, 1999.

A Appendix

A.1 RDF Ontology of the Generic Graph

```
@prefix cgdt: <http://www.semwebtech.org/languages/2008/cgdt#>.
# prefixes rdf, rdfs as usual

cgdt:VertexSchema rdfs:subClassOf cgdt:Schema.
cgdt:EdgeSchema rdfs:subClassOf cgdt:Schema.
cgdt:PathSchema rdfs:subClassOf cgdt:Schema.

cgdt:Property rdfs:subClassOf rdf:Property.
cgdt:hasProperty rdfs:domain cgdt:Schema; rdfs:range rdf:Property.
cgdt:InductiveProperty rdfs:subClassOf cgdt:Property.

cgdt:InsertionCondition rdfs:subClassOf cgdt:Condition.
cgdt:VertexInsertionCondition rdfs:subClassOf cgdt:InsertionCondition.
cgdt:EdgeInsertionCondition rdfs:subClassOf cgdt:InsertionCondition.
cgdt:PathExtensionCondition rdfs:subClassOf cgdt:Condition.

cgdt:has{{Edge|Path}Insertion|PathExtension}Condition # shortened syntax
    rdfs:domain cgdt:Schema;
    rdfs:range cgdt:{{Edge|Path}Insertion|PathExtension}Condition.
```

A.2 Vertex and Edge Schemas for the Travel Graph

(we omit `cgdt:language <http://www.w3.org/XPath>` for the specifications due to reasons of space)

```
@prefix cgdt: <http://www.semwebtech.org/languages/2008/cgdt#>.
@prefix tg: <foo:bla/thetravelgraph>. # arbitrary: only used locally
# @prefixes rdf, rdfs, xsd as usual

tg:travelgraph a cgdt:Graph; cgdt:hasVertexSchema tg:vs;
    cgdt:hasEdgeSchema tg:es; cgdt:hasPathSchema tg:ps.

tg:vs a cgdt:VertexSchema; cgdt:hasProperty tg:id, tg:timezone.
    cgdt:hasInsertionCondition tg:vertexInsCond1. # see Appendix A.4

tg:id a cgdt:Property; rdfs:label "id"; rdfs:range xsd:string.
tg:timezone a cgdt:Property; rdfs:label "timezone"; rdfs:range xsd:int;
    cgdt:definition [ — an external query that is omitted here — ].

tg:es a cgdt:EdgeSchema;
    cgdt:hasProperty cgdt:from, cgdt:to, # common: properties from and to
      tg:edgeCode, tg:edgeDept, tg:edgeArr, tg:edgePrice, tg:edgeDuration;
    cgdt:hasInsertionCondition tg:edgeInsCond1. # see Appendix A.4

cgdt:from a cgdt:Property; rdfs:label "from"; rdfs:range xsd:string.
cgdt:to a cgdt:Property; rdfs:label "from"; rdfs:range xsd:string.
tg:edgeCode a cgdt:Property; rdfs:label "code"; rdfs:range xsd:string.
tg:edgeDept a cgdt:Property; rdfs:label "dept"; rdfs:range xsd:dateTime.
tg:edgeArr a cgdt:Property; rdfs:label "arr"; rdfs:range xsd:dateTime.
tg:edgePrice a cgdt:Property; rdfs:label "price"; rdfs:range xsd:double.
tg:edgeDuration a cgdt:Property; rdfs:label "duration"; rdfs:range xsd:double;
    cgdt:definition [ cgdt:specification "tg:edgeArr - tg:edgeDept +
                                    cgdt:to/tg:timezone - cgdt:from/tg:timezone" ].
```

A.3 Path Schema for the Travel Graph

(we omit `cgdt:language <http://www.w3.org/XPath>` for the specifications due to reasons of space)

```
tg:ps a cgdt:PathSchema;
    cgdt:hasProperty cgdt:from, cgdt:to,
tg:pathDept, tg:pathArr, tg:pathPrice;
    cgdt:hasPathExtensionCondition tg:pathExtCond1.     # see Appendix A.4
tg:pathDept a cgdt:InductiveProperty;
    rdfs:label "dept"; rdfs:range xsd:dateTime;
    cgdt:baseCase [ cgdt:specification "tg:edgeDept" ];
    cgdt:inductiveCase [ cgdt:specification "cgdt:front/tg:pathDept" ].
tg:pathArr a cgdt:InductiveProperty;
    rdfs:label "arr"; rdfs:range xsd:dateTime;
    cgdt:baseCase [ cgdt:specification "tg:edgeArr" ];
    cgdt:inductiveCase [ cgdt:specification "cgdt:last/tg:edgeArr" ]].
tg:pathPrice a cgdt:SumProperty; rdfs:label "price";
    rdfs:range xsd:double;
    cgdt:baseProperty tg:edgePrice. # sum over tg:edgePrice of all edges
tg:pathDuration a cgdt:InductiveProperty;
    rdfs:label "duration"; rdfs:range xsd:double;
    cgdt:baseCase [ cgdt:specification "tg:edgeDuration"]
    cgdt:inductiveCase [ cgdt:specification
      "fn:add-dayTimeDurations(cgdt:front/tg:pathDuration,
        cgdt:last/tg:edgeDuration,
        fn:subtract-dayTimeDurations(cgdt:last/tg:edgeDept, cgdt:front/tg:pathArr))"].
```

A.4 RDF Specification of the Conditions for the Travel Graph

(for simplicity, we consider the concrete destination as fixed here; CGDT provides a mechanism for declaring global variables that can be set with the initialization)

```
tg:vertexInsCond1 a cgdt:VertexPropertyCondition;
    cgdt:specification "not(cgdt:id = 'LHR')".
tg:edgeInsCond1 a cgdt:EdgePropertyCondition;
    cgdt:specification "tg:duration < '10:00'".
tg:pathExtCond1 a cgdt:PathExtensionCondition;
    cgdt:specification "cgdt:front/tg:arr + '01:00' < cgdt:last/tg:dept".
tg:travelgraph cgdt:hasResultSpecification
    [ cgdt:filterCondition "to = 'St.Malo'"; cgdt:numberOfResults 5;
      cgdt:hasValuationFunction  "fn:hours-from-duration(duration) + price div 100". ]
```