Temporal Pattern Matching

Campbell, Wu

June 10, 2016

Todo list

1 Problem Definition

In this section, we provide a set of formal definitions of temporal graph and temporal graph query. Then, we will discussion different semantics for interpreting the graph pattern matching problem on temporal graphs.

1.1 Temporal Dimension

Given two time intervals $T_1, T_2 \in \mathbb{N}^2$, we define the following predicates and computation:

- $\pi_1(T_1) = t_1^s$, and $\pi_2(T_1) = t_1^f$;
- $T_1 = T_2 \Leftrightarrow t_1^s = t_2^s \wedge t_1^f = t_2^f$;
- $T_1 \subseteq T_2 \Leftrightarrow t_1^s \geq t_2^s \wedge t_1^f \leq t_2^f$;
- $T_1 \cap T_2 = (max(t_1^s, t_2^s), min(t_1^f, t_2^f))$;
- $T_1 \cup T_2 = (min(t_1^s, t_2^s), max(t_1^f, t_2^f))$;

1.2 Temporal Graph

A temporal graph is a node and edge-labeled graph that is annotated with time intervals. Formally,

Definition 1.1. *A **temporal graph** is a node and edge labeled $G = (V_G, E_G, L_G)$ where V_G is the set of vertices, and $E_G \subset V_G^2 \times \mathbb{N}^2$ is the set of edges, and $L_G : V_G \cup E_G \to \mathcal{L}$ is a label function that maps each node and each edge to a label in domain $\mathcal{L}$.*

We call the time interval (t^s, t^f) associated with each edge e the **active period** *of e.*

We provide a few helper functions to obtain the end nodes of each edge and the set the outgoing and incoming edges of a node:

- given a node $u \in V_G$, $out(u) = \{e \in E_G \mid e = (u, v) \text{ for some } v \in V_G\}$; $in(u) = \{e \in E_G \mid e = (v, u) \text{ for some } v \in V_G\}$.
- given an edge $e = (u, v) \in E_G$, $\pi_1(e) = u$ and $\pi_2(e) = v$.

$\mathrm{T} : E \to N^2$ is a function retrieve the active period for an edge $e \in E_G$, $\mathrm{T}(e) = (t_e^s, t_e^f)$.

We provide two helper function T_s and T_f that return the start and finish time of the active period of an edge: $\mathrm{T}_s(e) = \pi_1(T(e))$ and $\mathrm{T}_f(e) = \pi_2(T_(e))$.

Remarks:

1. G is not necessarily connected.
2. There maybe more than one edge between a pair of nodes, bearing different active period (t^s, t^f).
3. For each pair of edges e_1 and e_2 between the same pair of nodes (u, v) that have the same edge label, we can assume that the active period of the edges do not overlap, e.g., $\mathrm{T}(e_1) \cap \mathrm{T}(e_2) = \emptyset$, since if they do overlap, we can combine the two edges to form one whose active period is $\mathrm{T}(e_1) \cup \mathrm{T}(e_2)$.
4. There can be many simplified versions of the node and edge labeling. For instance, only nodes are labeled, but edges are not, hinting that all edges are labeled the same.
5. In this definition, we only associate edges with timestamps. We can assume that nodes are always active.
6. In this definition, G is a directed graph. To make it undirected, we can
   - define $E \subseteq P^2(V) \times \mathbb{N}^2$, where $P^2(V)$ is the powerset of size two over the naturals.
   - require that $(u, v, t^s, t^f) \in E_G \to (v, u, t^s, t^f) \in E_G$

Definition 1.2. *Given a graph $G = (V_G, E_G, L_G)$ and a set of edges $S_e \subseteq E_G$, we say that $\pi_{S_e}(G) = (V_{S_e}, S_e, L_G)$, where $V_{S_e} = \bigcup_{e \in S_e} \{\pi_1(e)\} \cup \{\pi_2(e)\}$, is the edge-induced graph of S_e on G.*

1.3 Temporal Graph Pattern

We define graph patterns in a way similar to how graph patterns are defined in sub-graph isomorphism problems, but allowing users to provide additional constraints on time.

Definition 1.3. *A* **temporal graph query** $q =< G_q, T_q >$ *consists of a graph pattern* G_q *and a time interval* T_q.

$T_q \in (\mathbb{N} \cup \{?\})^2$ *is called the* **global temporal constraint** *of* q.

The graph pattern is a connected graph $G_q = (V_q, E_q, L_q)$, *where* V_q *is the set of nodes and* $E_q \in V_q^2 \times (\mathbb{N} \cup \{?\})^2$ *is the set of edges, and* $L_q : V_q \cup E_q \rightarrow \mathcal{L} \cup \{?\}$ *is a label function that maps each node/edge to a label in* $\mathcal{L}$ *or ?.*

Associated with each edge in E_q, *user can also provide a temporal constraint in the form of a time interval, again, both the start and finish time can be a constant or ?. We call* $T(e_q)$ *for each* $e_q \in E_q$ *the* **local temporal constraints**.

We overload the helper functions introduced earlier to apply to graph pattern and time intervals that serve as temporal constraints. In the computations and operations associated with time intervals, ? is interpreted as $-\infty$ when used as start time, and ∞ when used as finish time.

Remark: the definition above can be incorporated easily into SPARQL. We can investigate the details when we settle on the definition.

1.4 Temporal Graph Pattern Matching

Definition 1.4. *A* **match** *of a graph pattern* G_q *in* G *is a total mapping* $h : \{e_q : e_q \in E_q\} \rightarrow \{e_G : e_G \in E_G\}$ *such that:*

- *for each edge* $e_q \in E_q$, *the edge label predicate associated with* e_q *is satisfied by* $h(e_q) \in E_G$.
- *for each node* $v_q \in V_q$, *the mapping of the outgoing and incoming edges of* v_q *share the same end node* $v_G \in V_G$ *and the node label predicate associated with* v_q *is satisfied by* v_G. *Formally, for any two edges* $e_1, e_2 \in E_q$, *if* $\pi_i(e_1) = \pi_j(e_2)$, *where* i, j *can be 0 or 1, the following must hold:* $\pi_i(h(e_1)) = \pi_j(h(e_2))$.

Please note that the definition of matching is the same as pattern matching defined for conjunctive queries on graphs. The new problem is how we can take the temporal constraints into consideration, which will be defined next.

Note that we allow users to provide temporal constraints in the form of a time window for the whole pattern and for each edge, but we also provide the flexibility for them not to provide any specific temporal constraints via the "?" option. Hence, users' temporal specification can be very strict, or very relaxed, or anywhere in between. Here are some scenarios:

- most strict: user specifies explicit global and local temporal constraints, and for each constraint specified, the start time is the same as the end time.

- most relaxed: user specifies temporal constraints with all ?'s, which means infinity.
- anywhere in between: including the cases in which some temporal constraints contains ?.
- conflicted: the intersection of the global temporal constraint and at least one of the local temporal constraint is empty. **remark:** we can easily identify conflict cases and return empty results without query evaluation.

We first define a few semantics that explicitly address user-specified temporal constraints:

Definition 1.5. *Given a graph G, a temporal graph query $q =< G_q, T_q >$, we say that a graph pattern matching h explicitly satisfies the temporal constraint of q if*

- *under the* EXACT *semantics, h satisfies that:*
  - *for all $e_q \in E_q$, $T(h(e_q)) = T(e_q)$ and $T(h(e_q)) \subseteq T_q$.*
- *under the* CONTAIN *semantics, h satisfies that:*
  - *for all $e_q \in E_q$, $T(h(e_q)) \subseteq T(e_q)$ and $T(h(e_q)) \subseteq T_q$.*
- *under the* CONTAINED *semantics, h must satisfy that:*
  - *for all $e_q \in E_q$, $T(h(e_q)) \supseteq T(e_q)$ and $T(h(e_q)) \supseteq T_q$.*
- *under the* INTERSECT *semantics, h must satisfy that:*
  - *for all $e_q \in E_q$, $T(h(e_q)) \cap T(e_q) \neq \emptyset$ and $T(h(e_q)) \cap T_q \neq \emptyset$.*

We will define $\mathcal{C}_{exp} = \{\text{EXACT}, \text{CONTAIN}, \text{CONTAINED}, \text{INTERSECT}\}$ to be the set of differentiating identifiers for the explicit semantics.

Finally, we will define a curried function $\varepsilon_{\text{LOC}} : \mathcal{C}_{exp} \to P(T^2 \times T^2) \to \mathbb{2}$ which will take an explicit temporal condition return a function that takes a set of pairs of time intervals and determine whether each pair of time intervals satisfies the given temporal semantic. For example $\varepsilon_{\text{LOC}}(\text{EXACT})(\{((1,4),(1,4)),((2,4),(2,4))\})$ returns True*, and $\varepsilon_{\text{LOC}}(\text{EXACT}) : P(T^2 \times T^2) \to \mathbb{2}$ is a function that determines whether a set of pairs of time intervals obeys the* EXACT *semantics.*

Remark: We need to think more carefully about how global temporal constraint T_q is interpreted in these semantics. Best way is to come up with some example queries.

The ***explicit temporal semantics*** defined above address only the issue of interpreting temporal constraints specified by users. Matching returned under all these semantics will include subgraphs that are not temporally traversable.

We next define a few ***implicit temporal semantics*** as remedy.

Definition 1.6. *Given a temporal graph G, we say that the graph is* **concurrent** *(denoted* CONCUR*) if* $\bigcap_{e \in E_G} T(e) \neq \emptyset$.

Definition 1.7. *Given a temporal graph G, we say that the graph is* ***weakly consecutive*** *(denoted* WCONSEC*) if for every $e_1 = (w, u, t_1^s, t_1^f), e_2 = (u, v, t_2^s, t_2^f) \in E_G$, $t_2^f \geq t_1^s$, where we say it is* **strongly consecutive** *(denoted* SCONSEC *) if for any $e_1 = (w, u), e_2 = (u, v) \in E_G$, $T(e_1) \cap T(e_2) \neq \emptyset$.*

Lemma 1.1. *If a temporal graph G is strongly consecutive, it must be weakly consecutive.*

We use $\mathcal{C}_{imp}$ to represent the set of implicit temporal constraints. $\mathcal{C}_{imp} = \{\text{CONCUR}, \text{SCONSEC}, \text{WCONSEC}\}$.

Now, we can define ***implicit temporal semantics*** to demand that resultant matching sub-graph be **concurrent**, **weakly consecutive**, or **strongly consecutive**.

Conjectures:

1. If the graph G is of star shape, e.g., there exist a center node u such that for any other nodes v there exists an edge between u and v, and if the in-degree of $u > 0$, then, G is concurrent iff G is consecutive.
2. If the graph G is a clique, then, G is concurrent iff G is consecutive.
3. If there exists a circle that traverse through all nodes in G, then, there exist an edge on this circle whose active period dominates those of all other edges on the circle.

Remarks:

- Whether these conjectures are true or not may depend on whether the graph is directed or undirected.

  If a graph is connected, it is a star shape, but with all edges going out, e.g., if there are no pairs of edges that have tail-to-head connection, then is it consecutive? apparently it satisfy our definition.

- Either we need to find some work that has proved them, or to prove them, which should not be hard.
- These conjectures, if proved true, can be used to speed up query evaluation process. It should be easy to identify star shape, clique and circle in query patterns, which are usually very small. This will allow us to strength the temporal constraints based on the properties described in the conjectures to enhance filtering.

We now define a few functions that apply the implicit temporal semantics.

Definition 1.8. *Given a temporal graph G, the function τ_b is a curried Boolean function $\tau_b : \mathcal{C}_{imp} \to P(E) \to \mathbf{2}$ that takes an implicit temporal constraint c and returns a function that takes a set of edges $S_e \subseteq E_G$ as input and returns true if the edge-induced graph $\pi_{S_e}(G)$ satisfies constraint c, and false otherwise. In other words for a given temporal condition $c \in \mathcal{C}_{imp}$, $\tau_b(c) : P(E) \to \mathbf{2}$. For simplicity we will use $\tau_b(c)(S_e)$ and $\tau_b(c, S_e)$ for a given set $S_e \subseteq E$ interdependently.*

Conjectures:

1. Given a temporal graph G, a set of edges $S_e \subseteq E_G$ and a temporal constraint c, $\tau_B(c)(S_e) \Rightarrow \forall S \subseteq S_e(\tau_B(c, S))$

2 Edge Isomorphism via Co-incidence Interval Graph

In this section we want to enumerate the basics of the interval graph method by expanding on the matches h we defined above in Definition 1.4.

We introduce the concept of the ***co-incidence interval graph*, a derived static graph representing the temporal and incidence relationships between adjacent edges of a temporal graph. This interval graph is a lossless encoding (shown to be bijective) of the full temporal graph G, which allows us to run existing algorithms to find appropriate patterns.**

Definition 2.1. *The co-incidence interval graph, $\mathcal{I}_G^c$ of a graph $G = (V_G, E_G, L_G)$ under implicit temporal constraint $c \in \mathcal{C}_{imp}$, is a graph $\mathcal{I}_G^c = (E, \mathcal{E})$, where its node set is E, the edge set of graph G, and it is edge set $\mathcal{E} \subseteq E^2$ is the set of "meta-edges" between elements in E. For any pair $e_1, e_e \in E$, $(e_1, e_2) \in \mathcal{E}$ if*

- *e_1 and e_2 share an endpoint, e.g., $\{\pi_1(e_1), \pi_2(e_1) \cap \pi_1(e_2), \pi_2(e_2) \neq \emptyset$, and*
- *$\tau_b(c, \{e_1, e_2\}) = True$*

The generic nature of this definition of the co-incidence interval graph accommodate any implicit temporal semantics defined earlier.

example

Let I_c be the function that computes this co-incidence interval graph, given an implicit temporal constraint c, e.g., $I_c(G) = \mathcal{I}_G^c$.

Lemma 2.1. *Given an implicit temporal constraint c, the function I_c is a bijection over unlabelled*[1] *graphs G with $\delta(G) \geq 1$.*

[1] This is a trivial extension that only serves to complicate the proof. The lemma indeed holds for labelled graphs as well.

Proof. Surjectivity follows the definition of function I_c, so, we will only show Injectivity.

Injectivity. All of the information necessary to restore the temporal graph G is stored in the edge-set E, since there is no vertex that is not an endpoint of an edge. Thus, if $I_c(G) = I_c(H)$, then $V(I_c(G)) = E(G) = E(H) = V(I_c(H))$, hence $G = H$. □

Remarks:

- Note that we didn't use the edge set of $\mathcal{I}_G^c$ at all in the proof above. This is because all of the structural information needed to describe the graph G is stored in the edge set (labels are handled by an external map L).
- Note that Lemma 2.1 only holds for graphs with $\delta(G) \geq 1$. This is because if there is a vertex that has degree zero, there is no edge that knows about it. This could be solved if you wanted to keep track of these vertices in $\mathcal{I}_G^c$. (Its also very unlikely for interesting large graphs for singletons to be of any use or importance. They will only be returned in trivial queries such as the empty graph or singletons).

Corollary 2.1. *Given temporal graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ such that $\delta(G) \geq 1$ and $\delta(H) \geq 1$), an implicit constraint c, and an isomorphism $f_c : V \to V$, then $I_c^{-1} \circ f_c \circ I_c : E \to E$ is a graph isomorphism if f_c uniquely maps labels.*

Proof. Since I_c losslessly encodes edges as vertices, f_c is really bijection between edge sets, that preserves the temporal coincidence relationship between edges. For arbitrary $e, e' \in E_G$ and $f, f' \in E_H$ such that $f_c(e) = f$ and $f_c(e') = f'$ show that if and only if e and e' are coincident in G, f and f' are coincident in H.

For vertex $e \in V(\mathcal{I}_G^c)$, we consider it to be a vertex with labels corresponding to its declaration in G. Hence the label of e is in $(V^2 \times T^2)$. So we let $L_{\mathcal{I}_G^c} = (u, v)$. Then, since f_c uniquely maps labels, $f_c(\mathcal{I}_G^C)$ will have preserved all adjacencies that violate the condition c (as well as those that do not). □

Algorithm 1 outlines the construction of a coincidence interval graph. Its complexity is $O(|E|d_{\max}(G))$ in the edge-relational representation of the graph).

Algorithm 2 depicts the procedure for restoring the original temporal graph given a co-incidence interval graph.

3 Implementation Structure

Since Ullman's original search-space pruning algorithm [10] published in 1976 there has been an influx of new algorithms attempting to find tighter subspaces and improved search orders, as well as storing partial results in graph indexes to allow for faster access. The most recent, and fastest algorithms have been

Algorithm 1: MakeCoincidenceInterval(G, c), equivalently $I_C(G)$

Input: A temporal graph $G = (V_G, E_G, L_G)$, a implicit temporal constraint c

Output: The coincidence interval graph $\mathcal{I}_G^c$

$\mathcal{E} = \emptyset$;
foreach *edge pair* $e, f \in E_G$ **do**
  if $\tau_B(c)(\{T(e), T(f)\})$ **then**
    Add meta-edge (e, f) to $\mathcal{E}$;
  end
end
return $(E_G, \mathcal{E})$;

Algorithm 2: UnmakeCoincidenceInterval(I_G^c, c), equivalently $I_c^{-1}(\mathcal{I}_G^c)$

Input: The co-incidince interval graph $\mathcal{I}_G^c = (E, \mathcal{E})$

Output: Temproal graph G

$E_G = E$;
$V_G = \emptyset$;
foreach $e \in E_G$ **do**
  $V = V \cup \{\pi_1(e), \pi_2(e)\}$;
end
return (V_G, E_G);

in the last 8 years. Notably, these are QuickSI [9], GraphQL [4], TurboIso [3], BoostIso [7], and DualIso [8].

A 2012 comparison of existing algorithms [5] concluded that QuickSI, and GraphQL were the fastest from among other algorithms including GADDI, SPath [12], and VF2 [1]. It created a common framework for all of the algorithms, that allowed for a more comprehensive understanding of the way in which these graphs are being queried. It is essentially broken up into four steps. FilterCandidates, which performs a label search on a given edge. Once this has been performed for all vertices, the recursive subroutine SubgraphSearch is called. Within this routine, there is the function NextQueryEdge, which determines the search order of the query graph, IsJoinable, which determines whether the proposed match is actually viable, UpdateState, which updates the mapping with the joinable pair, then the recursive call, and finally, RestoreState, which removes the pair from the mapping. This is explicitly stated in Algorithm 3. Note that a key difference between this algorithm and the one presented in [5] is that here we have an algorithm that finds an edge-mapping as opposed to a vertex-mapping, and so the structure is slightly different.

Algorithm 3: GenericQueryProc(Q, G)

Input: A query graph Q, A data graph G
Output: All subgraph ismorphisms of Q in G

Initialize the Mapping M to $\emptyset$;
foreach $e \in E(Q)$ **do**
  $\Phi(e)$:= FilterCandidates($G, Q, e, \cdots$);
  if $\Phi(e) = \emptyset$ **then**
    return ;
  end
end
SubgraphSearch($Q, G, M, \Phi, \cdots$);
Subroutine SubgraphSearch($Q, G, M, \Phi, \cdots$)
  if $|M| = |E(Q)|$ **then**
    Report M;
  else
    e := NextQueryEdge ($\cdots$);
    $\Phi'(e)$:= RefineCandidates ($M, u, \Phi(e), \cdots$);
    foreach $f \in \Phi'(e)$ *that is not yet matched* **do**
      if IsJoinable($Q, G, e, f, \cdots$) **then**
        UpdateState($M, e, f, \cdots$);
        SubgraphSearch($Q, G, M, \cdots$);
        RestoreState($M, e, f, \cdots$);
      end
    end
  end

In the next section we will detail how to go about developing this framework for existing graphs. Specifically, how we can use temporal information to further restrict the search space for existing algorithmic paradigms.

3.1 Temporal Postcondition

Similar to the way in which [6] developed several naive versions of the VF2 [1] algorithm to include the basics of the WCONSEC temporal semantics. We will consider a similar algorithm to the *To-Ti*, algorithm where the topographical information is considered before the temporal information. Here, we simply filter the results of any implementation of GENERICQUERYPROC with $\tau_b \circ T \circ E$. We get this naive algorithm in Algorithm 4.

Algorithm 4: TOPTIMEQUERY($< Q, T_q >, G, c$)

Input: A temporal query graph Q, a time range T_q, a data graph G, and a temporal condition c

Output: The set of patterns obeying the temporal semantic c matching Q in G

```
1  R := GenericQueryProc(Q, G);
2  foreach g ∈ R do
3      if c ≠ ∅ and T_q ≠ ∅ then
4          if not τ_b(c)({T(e)|e ∈ E(G)} ∪ {T_q}) then
5              R.remove g;
6              next;
7          end
8      else
9          if c ≠ ∅ and not τ_b(c)({T(e)|e ∈ E(G)} then
10             R.remove g;
11             next;
12         end
13         if T_q ≠ ∅ and not ∀e ∈ E(g).τ_B(c){T(e), T_q} then
14             R.remove g;
15             next;
16         end
17     end
18     foreach e' ∈ V(g) do
19         Let e ∈ V(Q) be the vertex mapped to e';
20         if T(e) ∩ T(e') = ∅ then
21             R.remove g;
22             break;
23         end
24     end
25 end
26 return R;
```

This algorithm simply filters out those result graphs that violate the temporal semantics and/or the time window T_q. When both c and T_q are given, the potential result graph must take T_q into consideration when checking if the potential result graph obeys c.

3.2 Preprocessing on Query Graph

Oftentimes a query graph will have contradictory or superfluous temporal information, for example, the local interval of a given edge may not intersect at all with the temporal condition or the local intervals given will preclude the explicit semantics, in which case, we can reject the query instantly with a null result.

Another interesting situation is when, for a given query $< Q, T_q, \text{INTERSECT}, \text{CO}$ and some edge $e \in E(G)$ with $T(e) \neq T(e) \cap T_q \neq \emptyset$. Then, if we enforce the INTERSECT semantics, we do not need to consider, for the edge e, the part of the interval $T(e) - T_q$, so we can update $T(e)$ to be $T(e) \cap T_q$.

change earlier definition to reflect this definition

Further, we need to consider the way that the CONCUR semantics will propogate to the neighbors of e. For this case, we need to enforce the INTERSECT semantics for every edge in Q, meaning that we can rewrite each $T(e)$ to be $T_q \cap \bigcap_{e' \in E} T(e')$. Of course for the EXACT semantics this loses pruning power, since we can just test perform an index search on the edges of the data-graph to find all time-restricted candidates. Similar fo We can perform a similar tightening for every combination in $\mathcal{C}_{imp} \times \mathcal{C}_{exp}$.

$\mathcal{C}_{exp}$	$\mathcal{C}_{imp}$	Tightening rules
CONTAIN	CONCUR	rewrite windows to intersection of all edges if $\tau_f(\text{CONCUR})(pred(e)) \cap \tau_f(\text{CONCUR})(succ(e)) \neq \emptyset$ enforce INTERSECT with
	SCONSEC	$\tau_f(\text{CONCUR})(pred(e)) \cap \tau_f(\text{CONCUR})(succ(e))$ else enforce CONTAINED with $T(e) - \tau_f(\text{CONCUR})(pred(e)) - \tau_f(\text{CONCUR})(succ(e))$
	WCONSEC	
CONTAINED	CONCUR	??
	SCONSEC	??
	WCONSEC	??
INTERSECT	CONCUR	??
	SCONSEC	??
	WCONSEC	??

We can also identify substructures of the query graph for which the semantic conditions are equivalent. What? that's possible? tell me more... Consider the following example of a cycle. Its a well-known fact that interval graphs that contain a cycle are chordal, so for 2-cycles all three semantics are equivalent, and for 3-cycles the CONCUR and SCONSEC semantics are equivalent.

We can also note that for a star in which the in- and out-degrees of the central vertex are at least one, the CONCUR and SCONSEC semantics are equivalent.

So, given a specific query, we can decompose it maximually into 2-cycles, and 3-cycles (i.e. if a 3-cycle contains a 2-cycle add the 3-cycle instead of the 2-cycle) and enforce the temporal constraints enforced by the tight semantics. Note that these are not the only structures for which the semantics are the same, but they were the easiest to detect.

Proposition 3.1. *3-cycles are the largest structures that allow us to reduce* SCONSEC *semantics to* CONCUR.

Proof. fill in here

. □

So, we will detect such two and three cycles using an $O(|V_Q||E_Q|)$ method, since we are assuming small query graphs with no more than 20 or 30 edges, we can store this in memory. The detection algorithm is defined in 5. It only makes sense to run this algorithm for WCONSEC and SCONSEC since we are attempting to leverage the selectivity of the CONCUR semantics.

Algorithm 5: DETECTSUBSTRUCTS

Input: A query graph Q and an implicit semantic m
Output: A set of structures reducing to CONCUR

```
Let the set of cycles C := ∅;
foreach e = (u, v) ∈ E(Q) do
    foreach e' = (v, u) ∈ E(Q) do
        add {e, e'} to T;
    end
    if m ≠ WCONSEC then
        foreach w ∈ V(Q) do
            foreach pair of edges e' = (v, w), e'' = (w, u) do
                add (e, e', e'') to T;
            end
        end
    end
end
```

Then, for each of these substructures S_e, we will build a hypernode, given $m \in \mathcal{C}_{imp}$, and $x \in \mathcal{C}_{exp}$, $N^m_{S_e}$ that reduces to CONCUR semantics. This hypernode will then have an active window itself.

We will then define a hypernode profile, which is an extended version of the temporal profile (see section 4). For a given node n, it is a tuple of the lexographically ordered labels (p_s), $\tau_f(\text{CONCUR})(S_e)$, and $\tau_f(m)(S_e)$. When we traverse the tree to try and map these vertices, we will need to prune results such that the internal edges obey the INTERSECT explicit semantics with $\tau_f(\text{CONCUR})(S_e)$,

and the incoming and outgoing edges obey the m implicit semantics with respect to $\tau_f(m)(S_e)$. Algorithm 6 describes the hypernode matching process.

Algorithm 6: HypernodeMatching

Input: A query hypernode n and its profile (p_s, p_{in}, p_{out}), a data hypernode n' and its profile $(p'_s, p'_{in}, p'_{out})$, the set of candidate sets Φ

Output: The updated candidates sets Φ

```
1 if p_s ≠ p'_s or p_in ∩ p'_in = ∅) or |E(n)| ≠ |E(q)| then
2 |   foreach e ∈ E(n) do
3 |   |   remove E(n') from Φ(e);
4 |   end
5 end
```

Once we have constructed such hypernodes, we can re-tighten the constraints on the graph as per the rules defined above.

Further, when traversing the graph, we will enforce, for f an incoming or outgoing edge to a matched hypernode in the data graph, $\tau(m)\{T(e), p_{out}\}$. And of course if we consider an edge that has is between a node and a hyper node, we will not match the two.

Remark: This is all a bit haphazard and messy. The rewrite rules are a bit awkward, and theres not an obvious way to extend the reduction to CONCUR semantics. Ill look at it again in a few days.

Revisit Tightening rules

3.3 Simple Modification of FilterCandidates

The method presented in section 3.1 is very simplistic and will result in a lot of unnecessary computation of branches of the search tree that might've been pruned earlier. In this section we begin to propose some simple modifications to existing algorithms that will combine to create a first algorithm for temporal pattern matching. The extension discussed in Section 3.6 is the one described in the temporal extension of VF2 [6].

The purpose of the function FILTERCANDIDATES is to provide a label-based index-boosted search of potential candidates for the input edge e. Some algorithms **??** will only perform a label search. while others will perform some signature based pruning **??**, and still others perform transformations on the query and data graphs **??**. We can extend the basic label search to include some basic temporal information. If for an edge e in the query graph and a potential candidate f in the data graph, if there is some temporal information $T(e)$, then we will enforce $\tau_b(c)(\{T(e), T(f)\})$. In the GraphQL algorithm, a function that discounts candidates based on signatures is composed with the standard label-index. We discuss our version of this algorithm in Section 4. The advantage of doing this here, is that we don't need to consider the specific temporal information of the edges in the query graph in the inner loops of SUBGRAPHSEARCH since we already know that they are matched appropriately.

3.4 Simple temporal extension of RefineCandidates

figure out what to do here

3.5 Temporal modification of NextQueryEdge

Here, we want to minimize the search space as early a possible and maintain the smallest number of search options as long as possible. To pick a start vertex, we will simply pick the edge $e \in E_Q$ that has the smallest $|\phi(e)|$ As in many existing algorithms, [1,4,11] we will limit our search to edges that are coincident to already mapped edges, but have not been paired. Of course we will also limit to those edges whose addition does not violate $\tau_b(m)$ Then we need to specify an optimal order among these. We will select the edge whose addition minimizes the result of $\tau_f(m)$, run on those edges matched so far, and the new edges to be added.

Remarks We can also consider other filtering methods. Most of them can just be composed with our existing ordering, probably after so that ties will be broken by the following conditions.

- Greedily minimize size of intermediate results [4]
- Take a statistical profile of all edges in the query graph. Use the frequency as a weight. Obtain a minimum spanning tree (of the coincidence graph) and search in order of insertion into min spanning tree. [9]

3.6 Simple Modification of IsJoinable

Now we want to similarly consider a more general form of the extension to VF2 [1] done in [6], in which the authors somewhat informally presented a $Ti\&To$ algorithm in which they considered the the temporal information as they considered the Topographical information by extending the semantic function built into VF2. This is the IsJoinable subroutine introduced in Algorithm 3 following the convention established in [5]. To mirror this simple extension, we will enforce the condition τ_B for every new edge introduced, and reject the edge if one of them fails. Essentially, if a query vertex u' is adjacent to u and has already been matched, then it ensures that there is a corresponding edge in the data graph (with matching label if necessary). In [1], they maintain the dates of previously accessed nodes to assure that the current node maintains the Wconsec condition. However, since we are finding a mapping between edges (which contains a mapping between nodes), our mapping contains all of the edges that have been used, so we already have the relevant information.

This algorithm is presented in Algorithm 7. First, we will need some notation. Let M_Q be the domain-so-far, and let M_G be the image-so-far. I.e. $M_Q :=$ `map fst`M and $M_G :=$ `map snd`M. The algorithm relies on the invariant that the mapping-so-far is contemporary with respect to some condition c.

Algorithm 7: IsJoinable(Q, T_q, G, e, f, M, c)

Input: A query graph Q, T_q a time interval, a data graph G, $e \in V(Q)$, $f \in G(Q)$, c a contemporaneity condition, and $M \in P(E(Q) \times G(Q))$ the mapping so far

Output: A boolean representing whether we can safely add the pair $e \mapsto f$ to M

```
1  Let e:=(u, u') and f:=(v, v');
2  foreach e' ∈ (Pred(e) ∪ Succ(e)) ∩ M_Q do
3      if ∃f' ∈ (Pred(e) ∪ Succ(e)) ∩ M_G.(e ↦ f') ∈ M then
4          if c ≠ ∅ and T_q ≠ ∅ then
5              if not τ(c)({T_q} ∪ {T(f'')|f'' ∈ M_G ∪ {f}}) then
6                  return False;
7              end
8          else
9              if c ≠ ∅ and not τ(c)({T(f'')|f'' ∈ M_G ∪ {f}}) then
10                 return False;
11             end
12             if T_q ≠ ∅ and not τ(c)({T(f), T_q}) then
13                 return False;
14             end
15         end
16     else
17         return False;
18     end
19     return True;
20 end
```

This is obviously going to significantly reduce the search space from the naive approach presented in section 3.1. This will prune very early the execution large sections of the tree that will not be searchable since they rely on a non-consecutive or non-intersecting edge.

4 Neighborhood Encoding

This section focuses on encoding methods that we can use to locally prune the search space. The algorithm that will be developed here is an additional subroutine that can be used in FilterCandidates. In a similar vein to the way that GADDI [11], QuickSI [9], and GraphQL [4] have developed subgraph signatures that allow for the search space to be pruned, we will extend, specifically, the notion of neighborhood subgraph profiles (GraphQL) to include temporal information. First, lets present the notion of Neighborhood Subgraphs [4].

Definition 4.1. *(Neigborhood Subgraph) Given a static graph G_s, vertex v and radius r, the r-neighborhood subgraph of vertex v, denoted $N(G, v, r)$, consists of all vertices within distance r from v and all edges between the vertices. Note that $N(G, v, 0) = (v, \emptyset)$.*

This is a fairly basic notion that is easily extensible to consider temporal information. The first requirement that will help us to prune the search space, is to force these subgraphs to be contemporary under some condition $c \in \mathcal{C}$ So it is possible that a given static r-neighborhood subgraph is contemporary under the given c, but that there are certain subgraphs of the static r-neighborhood subgraph that are. So the temporal extension must return a set of subgraphs, not just a single subgraph. Since we are mapping edges instead of vertices, we will define the temporal neighborhood subgraphs to be parameterized on edges instead of on vertices.

Definition 4.2. *(Temporal Neighborhood Subgraphs) Given a temporal graph G, an edge $e = (u, v)$, a radius r, and a condition c, the temporal r-neighborhood subgraph set $N_c(G, e, r)$ of a vertex e consists of the vertices and edges that are temporally reachable under c in r steps from v and u. Note that $\bigcup N_c(G, e, r) = N(G, v, r) \cup N(G, u, r)$.*

However, representing and using these temporal r-neighborhood subgraphs is expensive, so we will condense these into a much more lightweight *profile* [4]. For static r-neighborhood subgraphs, we simply lexographically order the vertices on the edges within the set. Then, given a similar ordering of the edges for the query graph Q, we can prune the search space when the generated r-neighborhood subgraph of the query graph is not a subsequence of the data graph profile. A significant improvement to this original profile models [**?**] includes a sequence of preceeding edge labels in each element of the sequence.

For temporal r-neighborhood subgraphs, we have more information that we can use to prune the search space. The *temporal profile* of an r-neighborhood

subgraph will be a tuple $p = (p_s, p_t)$ of the static profile and some temporal information. This temporal information will be an interval constructed from the intervals on the edges of graph. We will henceforth refer to p_S as the semantic profile, and p_t as the profile interval.

Definition 4.3. *Define τ_f to be a curried folding function $\tau_f : \mathcal{C} \to P(T^2) \to T^2$, that summarizes a set of time intervals in a single time interval dependent on the temporal semantics. The cases are defined below:*

$$\tau_f(c)(S) = \begin{cases} \bigcap S, & \text{if } c = \text{CONCUR}, \\ \bigcup S, & \text{otherwise} \end{cases}$$

This definition makes sense, since any temporal query that obeys the concurrent implicit temporal semantics will need to match the intersection of all of the activity windows of the edges, where if the query obeys the weak or strong consecutive semantics, we will approximate the behavior by taking the union of all the edges, i.e. the minimum start time and the maximum end time. This will allow us to prune grossly errant results.

Definition 4.4. *The temporal profile p of a graph $G = (V, E)$, given some contemporaneity condition $c \in \mathcal{C}$, is defined to be a tuple $(p_s, p_t)_c$ where*

- p_s `:= toSortedList` $\{L(v)|v \in V\}$, *and*
- p_t `:= foldr1` $\tau_f(c)$ $\{T(e)|e \in E\}$, *where* `foldr1` *accumulates $\tau_f(c)$ accross the given set, assuming the set is non-empty. If the set is empty then let p_t be (∞, ∞).*

Given two temporal profiles $p = (p_s, p_t)$ and $p' = (p'_s, p'_t)$, p is said to contain p' with respect to a condition c (denoted $p' \subseteq_c p$) if p'_s is a subseqence of p_s, and $c.p(p_t, p'_t)$.

Once we have this information, we will compare the temporal r-neighborhood profiles of the query graph and the data graph. The semantic condition will be that the query semantic profile must be a subsequence of the data semantic profile, where the interval condition will just be that the intersection must be nonempty. If there is no query interval specified, then we will consider the interval to be (∞, ∞). This local refinement is defined in Algorithm 8.

This LocalRefinement procedure is a part of the FilterCandidates described in Section 3.3.

5 Temporal Index

There are many efficient indexing techniques. But the temporal dimension allows us an additional method by which we can filter our data. In fact it is a very strong filter that can prune a search space from tens of millions of edges to on average about 10000 per arbitrary time slice. So how do we go about constructing a temporal index? We start with the observation that each edge is a tuple

Algorithm 8: LocalRefinement(Q,G,Φ, r, c)

Input: A query graph Q, data graph G, current candidate sets Φ, a radius r

Output: A refined set of Φ

```
foreach e ∈ E(Q) do
    Calculate the temporal profile p of N_c(Q, e, r);
    foreach f ∈ Φ(e) do
        Calculate the temporal profile p' of N_c(G, v, r);
        if p ⊈ p' then
            remove f from Φ(e);
        end
    end
end
```

in $V^2 \times \mathrm{T}^2$. Similar to how adjacency matrices model the edge-relationships in the 2 dimensional space V^2, we can create the 2-dimensional space T^2, where each time window represents a region in this space. Then the obvious solution is to create a spatial index. For any given edge $e = (u, v, t_s, t_e)$, we will create a 2-dimensional box $b(e)$ bounded by the system

$$b((u, v, t_s, t_e)) = \begin{cases} x \leq t_e \\ x \geq t_s \\ y \leq t_e \\ y \geq t_S \end{cases}$$

Notice that $T(e) \cap T(f) \neq \emptyset$ for e, f edges, if and only if $b(e)$ and $b(f)$ overlap. We can then use an RTree [2] to index this space. In this form, this tool is only useful for when we consider $c = \texttt{concur}, \texttt{Sconsec}$. More robustly we can create a specific index for when $c = \texttt{Wconsec}$, in which case we will generate the 2 dimensional space

$$\begin{cases} x \leq t_e \\ y \leq t_e \end{cases}$$

Thus, like everything else in this paper, our indexing methodology is dependent on the contemporaneity condition $c \in \mathcal{C}$.

Also, note that this index can be used for much more than just representing the edges in the graphs, it can also be used for indexing subgraphs based on the output of $\tau_f(c)$. Then, for any edge that we are curious about adding, we can use the index to figure out which edges statisfy the temporal condition.

6 Experiments

Here are the possible experiments to run:

- Existing algorithms on co-incidence interval graph.
- Naive Post-condition
- Modified GenericQueryProc using temporal extensions
- Modified GenericQueryProc using temporal extensions and index

References

[1] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[2] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[3] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. $\text{Turbo}_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 337–348. ACM, 2013.

[4] Huahai He and Ambuj K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 405–418. ACM, 2008.

[5] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.

[6] Ursula Redmond and Pádraig Cunningham. Subgraph isomorphism in temporal networks. *arXiv preprint arXiv:1605.02174*, 2016.

[7] Xuguang Ren and Junhu Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *PVLDB*, 8(5):617–628, 2015.

[8] Matthew Saltz, Ayushi Jain, Abishek Kothari, Arash Fard, John A. Miller, and Lakshmih Ramaswamy. Dualiso: An algorithm for subgraph pattern matching on very large labeled graphs. *IEEE International Congress on Big Data*, pages 498–505, 2014.

[9] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

[10] Julian R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[11] Shijie Zhang, Shirong Li, and Jiong Yang. GADDI: distance index based subgraph matching in biological networks. In Martin L. Kersten, Boris Novikov, Jens Teubner, Vladimir Polutin, and Stefan Manegold, editors, *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, volume 360 of *ACM International Conference Proceeding Series*, pages 192–203. ACM, 2009.

[12] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *PVLDB*, 3(1):340–351, 2010.