Temporal Pattern Matching

Campbell, Wu

June 8, 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.

A temporal graph is a graph that is annotated with time (of domain $\mathbb{Z}$). Formally,

Definition 1.1. *A **temporal graph** is a node and edge labeled $G = (V_G, E_G)$ where V_G is the set of vertices, and $E_G \subset V^2 \times \mathbb{Z}^2$ is the set of edges. A label function L_G maps each node and each edge to its label.*

Associated with each edge e is a time interval (ts_e, tf_e), which we call the **active period** *of the edge.*

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 $\pi_2(e) = v$.

$\mathrm{T} : E \to \mathbb{Z}^2$ is a function that for any $e \in E_G$, $\mathrm{T}(e) = (ts_e, tf_e)$.

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

Given two time intervals $T_1 = (ts_1, tf_1)$ and $T_2 = (ts_2, tf_2)$, we define the following predicates and computation:

- $T_1 = T_2 \Leftrightarrow ts_1 = ts_2 \wedge tf_1 = tf_2$;

- $T_1 \subseteq T_2 \Leftrightarrow ts_1 \geq ts_2 \wedge tf_1 \leq tf_2$;
- $T_1 \cap T_2 = (max(ts_1, ts_2), min(tf_1, tf_2))$;

A time interval $(ts, tf) = \emptyset$, if $tf < ts$.

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 the union of the two.
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 $E \subseteq P^2(V) \times \mathbb{Z}^2$, where $P^2(V)$ is the powerset of size two over the naturals.
   - require that $(u, v, t_s, t_f) \in E_G \rightarrow (v, u, t_s, t_f) \in E_G$

We define graph patterns in a way similar to how graph patterns are defined in SPARQL-like queries, but allowing users to provide additional constraints on time.

Definition 1.2. *A* **temporal graph query** $q =< G_q, T_q >$ *consists of a graph pattern* G_q *and a time interval* T_q.

Both the start and finish time of T_q *can be a constant or ?.*

The graph pattern is a connected graph $G_q = (V_q, E_q)$*, where* V_q *is the set of nodes and* E_q *is the set of edges. A label function* L_q *maps each node/edge to its label, which can also be ?. 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_q *the* **global temporal constraint** *of* q *and* $T(e_q)$ *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.

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

Definition 1.3. *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 SPARQL-like graph queries. 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 1.4. *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* **intersection** *semantics, h must satisfy that:*
  - *for all $e_q \in E_q$, $T(h(e_q)) \cap T(e_q) \neq \emptyset$ and $T(h(e_q)) \cap T_q \neq \emptyset$.*

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

Definition 1.6. *Given a temporal graph G, we say that the graph is* ***weakly consecutive*** *if for every $e = (w, u, t_s, t_f), e' = (u, v, t'_s, t'_f) \in E_G$, $t'_f \geq t_s$, where we say it is* **strongly consecutive** *if for any $e = (w, u), e' = (u, v) \in E_G$, $T(e) \cap T(e') \neq \emptyset$.*

Hence, 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, then, G is concurrent iff G is consecutive.
2. If the graph G is a clique, then, G is concurrent iff G is consecutive.2
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.
- 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.

2 Edge Isomorphism

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 1.3. Here we introduce the concept of the co-incidence interval graph, a derived static graph representing the temporal and incidence relationships between th e 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. However, what kind of temporal information it captures is determined by an *temporal condition* c, which determines which version of implicit temporal semantics is being used.

Definition 2.1. *A temporal condition c is an element of the condition set $\mathcal{C} = \{\text{CONCUR}, \text{SCONSEC}, \text{WCONSEC}\}$,that determine the temporal logic to be applied. This is essentially an* `enum` *type that will allow the user to switch between the different implicit semantics as defined in Definitions 1.5 and 1.6.*

Definition 2.2. *The function $\tau_B : \mathcal{C} \times P(T^2) \to \mathbb{2}$ determines whether a set of of edges adhere to the given temporal condition.*

Definition 2.3. *The co-incidence interval graph, $\mathcal{I}_G^c$ of a graph $G = (V, E)$ under temporal condition c, is a tuple $\mathcal{I}_G^c = (E, \mathcal{E})$, where E is the edge set of the graph G (and the vertex set of I_G^c, and $\mathcal{E} \subseteq E^2$ is the set of "meta-edges" between the edges of G (vertices of I_G^c). There is an edge between $e, f \in E$ exactly when*

- *e and e' share an endpoint, and*
- *$\tau_B(c, \{e, e'\})$ returns* True

Let the function that computes this graph under the condition c be I_c.

The generic nature of this definition of the coincidence interval graph allows for the user to define *how* the contemporaneity of the query can be defined. Simply, this can be extended to define the **implicit temporal semantics** of the resultant query.

example

Lemma 2.1. *Given a contemporaneity condition c, the function I_c is a bijection over graphs with $\delta(G) \geq 1$.*

Proof. Since surjectivity is evident from Definition 2.3i we will only show Injectivity.

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

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).

The construction of this graph (as defined in Algorithm 1) is a fairly straightforward algorithm (and in fact is $O(|E|d_{\max}(G))$ in the edge-relational representation of the graph). A quick corollary of Lemma 2.1 is that given a condition c, if there exists some isomorphism $f_c : \mathcal{I}_G^c \to \mathcal{I}_H^c$, then $I_c^{-1} \circ f_c \circ I_c : G \to H$ is also an isomorphism.

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

Input: A temporal graph $G = (V, E)$, a contemporaneity condition c
Output: The interval coincidence graph $\mathcal{I}_G^c$

```
Initialize 𝓔 to ∅;
foreach edge pair e = (w, u), f = (u, v) ∈ E do
    if τ_B(c)({T(e), T(f)}) then
        Add meta-edge (e, f) to 𝓔;
    end
end
return (E, 𝓔);
```

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

Input: The coincidince interval graph $\mathcal{I}_G^c$, c
Output: The original graph G

```
Initialize V and E to ∅;
foreach edge (u, v) ∈ V(I_G^c) do
    Add u and v to V;
    Add (u, v) to E;
end
return (V, E);
```

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 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.

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.

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 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

Algorithm 3: GenericQueryProc(Q, G)

```
   Input: A query graph Q, A data graph G
   Output: All subgraph ismorphisms of Q in G
 1 Initialize the Mapping M to ∅;
 2 foreach e ∈ E(Q) do
 3 |   Φ(e) := FilterCandidates(G, Q, e, ···);
 4 |   if Φ(e) = ∅ then
 5 |   |   return ;
 6 |   end
 7 end
 8 SubgraphSearch(Q, G, M, Φ, ···);
 1 Subroutine SubgraphSearch(Q, G, M, Φ, ···)
 2 |   if |M| = |E(Q)| then
 3 |   |   Report M;
 4 |   else
 5 |   |   e := NextQueryEdge (···);
 6 |   |   Φ′(e) := RefineCandidates (M, u, Φ(e), ···);
 7 |   |   foreach f ∈ Φ′(e) that is not yet matched do
 8 |   |   |   if IsJoinable(Q, G, e, f, ···) then
 9 |   |   |   |   UpdateState(M, e, f, ···);
10 |   |   |   |   SubgraphSearch(Q, G, M, ···);
11 |   |   |   |   RestoreState(M, e, f, ···);
12 |   |   |   end
13 |   |   end
14 |   end
```

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;
```

to existing algorithms that will combine to create a first algorithm for temporal pattern matching. The extension discussed in Section 3.5 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. Most algorithmswill only perform a label search. We can extend this 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.

cite which

3.3 Simple temporal extension of RefineCandidates

figure out what to do here

3.4 Temporal modification of NextQueryEdge

figure out what to do here

3.5 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 5. 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 5: 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 6.

This LocalRefinement procedure is a part of the FilterCandidates described in Section 3.2.

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 6: 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.