An Efficient, Large-scale, Non-lattice-detection Algorithm for Exhaustive Structural Auditing of Biomedical Ontologies

Guo-Qiang Zhang[a,b,c,*], Guangming Xing[d], Licong Cui[a,b]

[a]*Department of Computer Science, University of Kentucky, Lexington, KY, USA*
[b]*Institute for Biomedical Informatics, University of Kentucky, Lexington, KY, USA*
[c]*Department of Internal Medicine, University of Kentucky, Lexington, KY, USA*
[d]*Department of Computer Science, Western Kentucky University, Bowling Green, KY, USA*

Abstract

One of the basic challenges in developing structural methods for systematic audition on the quality of biomedical ontologies is the computational cost usually involved in exhaustive sub-graph analysis. We introduce ANT-LCA, a new algorithm for computing all non-trivial lowest common ancestors (LCA) of each pair of concepts in the hierarchical graph induced by an ontology. The computation of LCA is a fundamental step for non-lattice approach for ontology quality assurance. Distinct from existing approaches, ANT-LCA only computes LCAs for non-trivial pairs, those having at least one common ancestor. To skip all trivial pairs that may be of no practical interest, ANT-LCA employs a simple but innovative algorithmic strategy combining topological order and dynamic programming to keep track of non-trivial pairs. We provide correctness proofs and demonstrate a substantial reduction in computational time for two largest biomedical ontologies: SNOMED CT and Gene Ontology (GO). ANT-LCA achieved an average computation time of 30 and 3 seconds per version for SNOMED CT and GO, respectively, about 2 orders of magnitude faster than the best known approaches. Our algorithm overcomes a fundamental computational barrier in subgraph based structural analysis of large ontological systems. It enables the implementation of a new breed of structural auditing methods that not only identifies potential problematic areas, but also automatically suggests changes to fix the issues. Such structural auditing methods can lead to more effective tools supporting ontology quality assurance work.

*Corresponding author. Email address: gq.zhang@uky.edu (GQ Zhang). Address: University of Kentucky 230 Multidisciplinary Science Building, 725 Rose Street Lexington, KY 40536-0082

Keywords:
Biomedical ontology, partial order, graph-theoretic algorithm, SNOMED CT, lattice vs non-lattice, quality assurance

1. Introduction

In graph-theoretic representation of ontologies in biomedicine such as SNOMED CT [1], ontological concepts correspond to graph nodes, and is-a relations correspond to edges of the graph. When rendering the is-a relations as a graph, the Hasse diagram convention orients more general concepts above (or higher than) more specific concepts.

One of the desirable properties of the resulting graph structure is that the subsumption relationship (is-a hierarchy) should form a lattice ([2]). There are in general two types of lattice-based approaches to ontology quality assurance. One involves the direct application of Formal Concept Analysis (FCA [3]), mostly for auditing semantic completeness or missing concepts [4]. The second involves the extraction of lattice-violating fragments [5, 6], or *non-lattice fragments*, which represent violations of the FCA principle that systematic engineering approaches for constructing concept hierarchies always result in order structures that are lattices in the sense of lattice theory [3]. This non-lattice approach for ontology quality assurance involves the extraction of graph substructures (i.e. sub-orders) that violate the *lattice property*, which states that any two concept nodes have at most one minimal shared (common) ancestor and at most one maximal shared descendant.

As illustrated recently in [7], the use of the non-lattice approach for improving the quality of an ontology consists of the following general steps:

1. Identify node-pairs that violate the lattice property (i.e. non-lattice pairs) and extract the associated non-lattice fragments;
2. Detect ontological defects such as miss-aligned is-a relations or missing concepts in the extracted non-lattice fragments, often leveraging additional or external information;
3. Formulate and generate change suggestions automatically and present the suggestions in a usable format;
4. Perform reviews of the suggested changes and accept or reject such suggestions by a qualified ontology engineer or ontology editor, and incorporate the accepted changes

into the next release.

The non-lattice approach is unique in that while most ontology quality assurance techniques [8] merely identify potential errors, this approach can not only identify previously undiscovered errors confirmed by domain experts, but also suggest appropriate remediation (i.e., "auto-suggestion") [7, 9]. For example, Figure 1 (top), extracted from the September 2017 release of SNOMED CT (US edition), contains a substructure (1A) of is-a relations on the left, involving 5 concepts. This is a non-lattice fragment, because the concept nodes labeled 1 and 2 have two maximal shared descendants: concept nodes labeled 4 and 5. With a combination of structural and lexical information represented in this fragment, one can infer that "Epithelioid hemangioendothelioma of lung" is-a "Malignant tumor of lung parenchyma." Remarkably, adding such a missing edge (in red color) also makes the resulting subgraph (1B) conforming to the lattice property: concept nodes labeled 1 and 2 now have a unique maximal shared descendant: concept nodes labeled 4 (since concept 5 is no longer "maximal"). Similarly, the lower part of Figure 1 shows a non-lattice fragment (2A) in the Gene Ontology (GO) on the left, and the corrected structure (2B) on the right.

Both the FCA- and the non-lattice-based approaches incur computational costs that sometimes make exhaustive analyses prohibitive. For example, in Jiang and Chute's work [4], only 10% of SNOMED CT sub-hierarchies were sampled in order to assess semantic completeness. Three months of sequential computation ([5]) or three hours of 25-node parallel processing ([6]) were required to detect non-lattice pairs for each version of SNOMED CT. The detection of non-lattice pairs is a fundamental step for non-lattice-based approach for ontology quality assurance. The non-lattice pairs serve as seeds for systematic generation of non-lattice fragments, but including all nodes in-between the seed nodes and the maximal shared descendants. Therefore, more efficient algorithms for detection of non-lattice pairs is highly desirable.

This paper introduces ANT-LCA, a new algorithm for computing all non-trivial lowest common ancestors (LCA) of each pair of concepts in the graph induced by an ontological system. Here the lowest common ancestors in the context of a graph are exactly the maximal shared descendants in the context of an ontology. In the remainder of the paper, we discuss algorithms in graph-theoretic and order-theoretic terms. But whenever working with specific

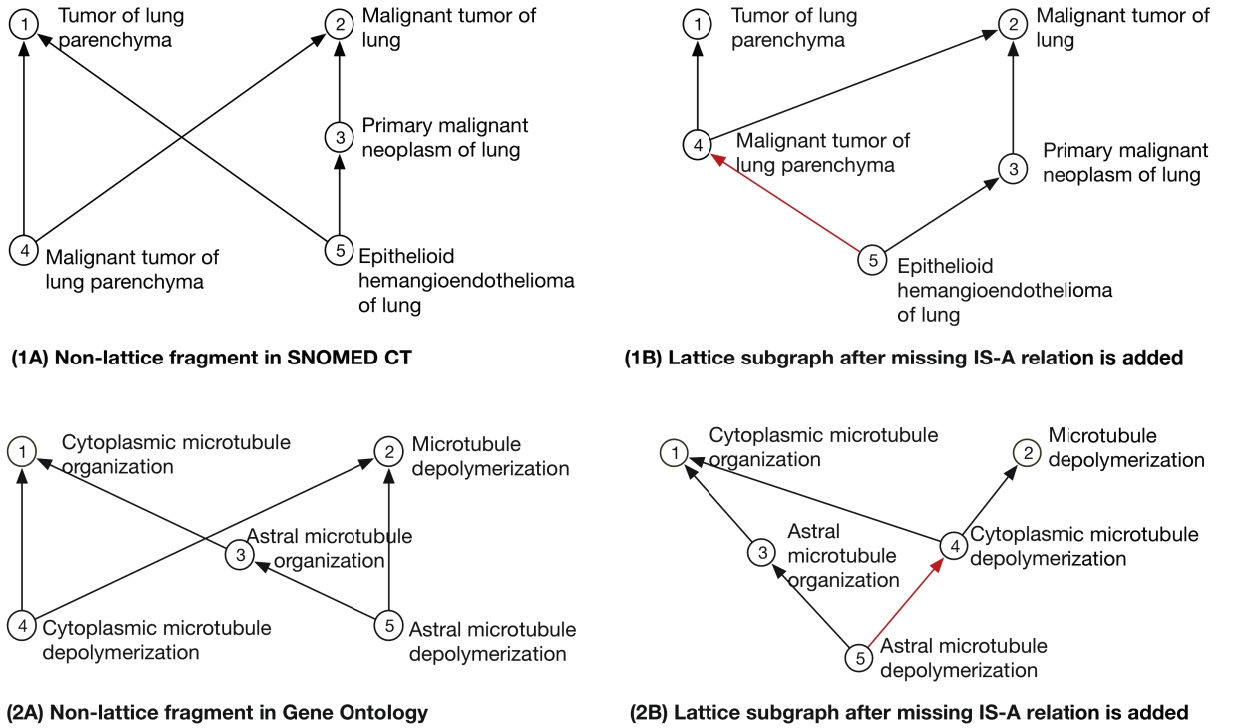

Figure 1: An example (1A) of non-lattice fragment of size 5 in SNOMED CT, as well as the resulting lattice subgraph (2B) after a missing IS-A relation is added (red link). Similarly, (2A) is a non-lattice fragment of size 5 in GO and (2B) is the correction.

ontological examples, we switch back to maximal shared descendants. Distinct from existing approaches, ANT-LCA only computes LCAs for non-trivial pairs, those having at least one common ancestor. To skip all trivial pairs that may be of no practical interest, ANT-LCA employs a simple but innovative algorithmic strategy combining topological order and dynamic programming [10] to keep track of non-trivial pairs.

We provide correctness proofs and demonstrate about 2-orders of magnitude reduction, compared with the best parallel algorithms known to date, in computational time for two of the largest biomedical ontologies: SNOMED CT and Gene Ontology (GO). ANT-LCA achieved an average computation time of 30 and 3 seconds per version for SNOMED CT and GO, respectively, confirming our complexity analysis with a time-bound involving *pairability-degree* (i.e. the constant in big-O analysis of time-complexity) as a quadratic factor. ANT-LCA overcomes a fundamental computational barrier in subgraph analysis of ontological structures. It enables the implementation of a new breed of structural auditing methods

that can not only identifies potential problematic areas, but also automatically suggests specific changes that are needed to fix the quality issues.

2. Background

2.1. LCA on directed acyclic graphs

In a directed acyclic graph (DAG), a common ancestor (CA) of a pair of nodes u, v is a node w that is a shared ancestor of u, v. A lowest CA is a node w such that no other shared ancestor is closer (nearer) to u, v than w. A pair of nodes u, v is trivial if they do not have a shared ancestor, or one of them is the ancestor of the other. Conversely, non-trivial pairs are those having at least one lowest common ancestor other than the nodes already in the pair. Given a subset of nodes X in a DAG, we denote the set of lowest common ancestors of X as $\mathsf{lca}(X)$, and common ancestors of X as $\mathsf{ca}(X)$, respectively. When X is a two-element set $\{a, b\}$ with two or more lowest common ancestors, it is called a *non-lattice pair*.

A pair of nodes (x, y) is called *pairable* if $\mathsf{lca}\{x, y\} \neq \emptyset$, $\mathsf{lca}\{x, y\} \neq \{x\}$, as well as $\mathsf{lca}\{x, y\} \neq \{y\}$. Intuitively, x, y is pairable if they share at least one non-trivial common ancestor. In this case we also say that x is *pairable with* y, and (x, y) a *non-trivial* pair. We use notation $x \downarrow y$ to indicate that x is pairable with y. A *trivial pair* is a pair (x, y) that is not pairable. In fact, (x, y) is trivial if and only if $\mathsf{lca}\{x, y\} \subseteq \{x, y\}$, i.e., $\mathsf{lca}\{x, y\} = \emptyset$, $\mathsf{lca}\{x, y\} = \{x\}$, or $\mathsf{lca}\{x, y\} = \{y\}$. We write $\pi(u) = \{v \mid u \downarrow v\}$ for the set of all nodes v that are pairable with u.

2.2. Non-lattice approach

The non-lattice approach [5] provides a mathematically grounded, error-agnostic method for exhaustive structural auditing of large and complex biomedical ontologies such as SNOMED CT. This approach focuses on the graph structure induced by the subsumption relation (is-a) in an ontology. It extracts *non-lattice pairs*, those with two or more lowest common ancestors, violating the lattice property.

Any non-lattice pair generates an *induced non-lattice fragment*, consisting of concepts in-between any lowest common ancestor and any member of the non-lattice pair, as well as all the relations between these concepts. Such induced non-lattice fragments represent

important areas of focus for ontological auditing ([5, 7]), because they are inconsistent with the ontology design principle that the subsumption relationship (is-a hierarchy) should form a lattice ([2, 5]). Non-lattice fragments are also in conflict with the Fundamental Theorem of Formal Concept Analysis ([3]), which states that concept hierarchies derived from the duality of intension and extension always have their order structure being a (complete) lattice.

In fact, non-lattice fragments are often indicative of missing hierarchical relations or concepts. As a demonstration of the practical utility of the non-lattice-based approach, Cui et al. [7] identified four lexical patterns among non-lattice subgraphs in SNOMED CT. Each lexical pattern is associated with a potential specific type of error. Applying the structural-lexical method to SNOMED CT (September 2015 U.S. edition), 6,801 non-lattice subgraphs matched these lexical patterns, of which 2,046 were amenable to visual inspection. Evaluation of a random sample of 100 small subgraphs resulted in 59 confirmed errors by domain experts. Abeysinghe et al. [9] further applied the four patterns to audit National Cancer Institute (NCI) Thesaurus (version 16.12d) and introduced two new lexical patterns to uncover potential errors and suggest remediations. A total of 8,143 non-lattice subgraphs were identified in NCI Thesaurus, among which 809 matched the six lexical patterns. Domain experts evaluated a random sample of 50 small subgraphs and verified that 33 of them contained errors and made correct suggestions. Such hybrid structural-lexical methods are innovative and proved effective not only in detecting errors, but also in suggesting remediation for these errors.

2.3. The computational challenge

Exhaustive generation of non-lattice fragments for large ontological graphs such as SNOMED CT, with over 300,000 concepts and 450,000 is-a relations, is computationally expensive if not prohibitive, using an exhaustive sequential approach. For example, in [5], 34 million pairs of SNOMED CT concepts were examined and 518,000 non-lattice pairs were identified using SPARQL queries over an RDF representation of the ontology. The time involved for such an exhaustive approach, 3 months using standard desktop machines, is inadequate for quality assurance applications.

In more recent work, a general MapReduce pipeline called MaPLE for Lattice-based

Evaluation [6] has been introduced for detecting non-lattice pairs. Using a Cloudera Hadoop cluster, MaPLE detected all non-lattice pairs in SNOMED CT, with an average total compute time of about 3 hours per version.

Our ANT-LCA algorithm provides a dramatic further reduction in computational time using sequential computation by a strategy that skips trivial pairs altogether, without even checking them.

3. Methods

3.1. The ANT-LCA algorithm

We present ANT-LCA in three components: initialization, pairability computation, and finding of shared ancestors imbedded into pairability computation. We treat pairability computation separately to highlight ANT-LCA's core algorithmic insight without dealing with irrelevant overhead.

3.1.1. Initialization

The initialization phase for ANT-LCA takes a DAG (V, E) as input and uses a modified version of topological sort [10] to obtain a topological order (index) for each node in V. This step takes linear time in $|V|$.

After initialization, we have two order relations on V: $\sqsubseteq$ and $\leq$. Here $\sqsubseteq$ (and the strict version $\sqsubset$) stands for the partial order determined by the input DAG (V, E) [10] (i.e., $v_1 \sqsubset v_2$ means there is an edge from v_1 to v_2). $\leq$ represents the usual arithmetic order on the topological index. By the property of topological sort, we have $u \sqsubset v$ implies $u < v$ for any $u, v \in V$.

3.1.2. Computing Pairability

The core algorithmic idea of ANT-LCA is captured by the computation of the pairable function $p_i(u)$, intended to compute the function $\pi(u)$, where $p_i(u)$ is the set of nodes pairable with u computed up to step i, and $\pi(u)$ is the set of all nodes pairable with u. Algorithm 1 initializes $p_i(u)$ by fixing proper values for $p_0(u)$ for each $u \in V$. Algorithm 2 updates $p_i(u)$ as i gets incremented, in order to capture all nodes pairable with u at the completion of the algorithm.

Input: (V, E) in topological order.

Output: Initialization of pairable elements for each node.

```
1 for i ∈ V do
2 |   for u ∈ i.to do
3 |   |   p_0(u) += i.to − {u};
4 |   end
5 end
```

Algorithm 1: Initialization phase for generating pairable sets. Here (V, E) is the input graph with V the set of nodes, and E the set of edges. $p_0(u)$ is the set of nodes pairable with u computed up to step 0 – the initialization step.

In Algorithm 1, $i.to$ consists of all t such that $(i, t) \in E$. For each i, Algorithm 1 updates each u such that $(i, u) \in E$ by appending distinct members in $i.to$, such as v (Figure 2, left) that are not comparable with u, into $p_0(u)$. Strictly speaking, for Algorithm 1 to be correct for arbitrary graphs, line 3 should be modified as $p_0(u) := i.to - \{x \mid u \sqsubseteq x\}$. This is, however, not necessary if nodes in $i.to$ are not comparable with each other, as is the case when the input graph has no "redundant" edges (when the is-a relation in an ontology is minimally represented without edges that are derivable from transitive closure).

Figure 2 (right) contains the Hasse diagram of an example DAG in topological order. The initialization results for $p_0(u)$ obtained by Algorithm 1 are displayed beside each node.

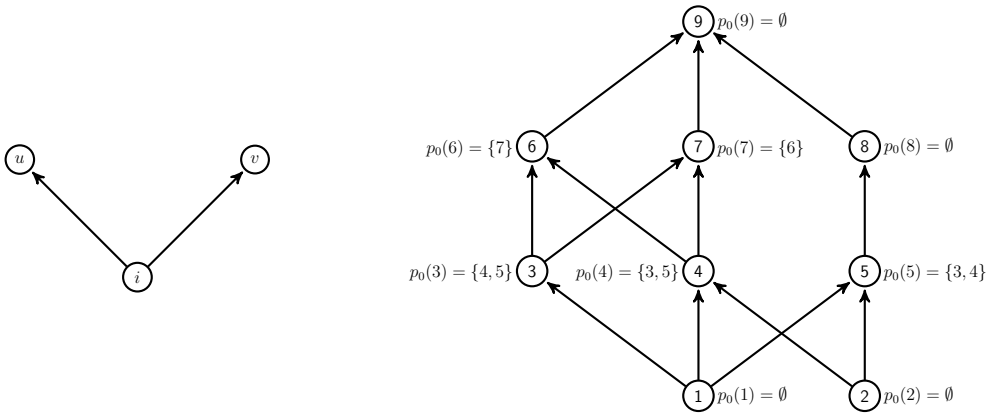

Figure 2: Left: the iterative pattern for each edge $(i, u) \in E$. Right: initializing the pairable function for a graph consisting of topologically ordered nodes 1 to 9.

Algorithm 2 updates each i's upper neighbor u (lines 1 and 2) by adding those nodes that are pairable with i but not comparable with u (line 3). All nodes v that are pairable with i also gets updated by adjoining the upper neighbors of i to its set of pairable nodes (line 5). For abbreviation, the notation of relative set union ($\cup_x$) is used in Algorithm 2. For subsets A, B of V and $x \in V$, we write $A \cup_x B$ for $A \cup B$ while making sure that nodes in the resulting set are not comparable to x, i.e.,

$$A \cup_x B := (A - \{a \in A \mid x \sqsubseteq a \text{ or } a \sqsubseteq x\}) \cup (B - \{b \in B \mid x \sqsubseteq b \text{ or } b \sqsubseteq x\}).$$

In practice, one can take advantage of fast computation of transitive closure [11] and efficient disjoint union [12] for computing $A \cup_x B$.

Input: (V, E) in topological order.

Output: The set of pairable nodes for each node.

```
1 for i ∈ V do
2 |   for u ∈ i.to do
3 |   |   p_i(u) := p_{i-1}(u) ∪_u p_{i-1}(i)
4 |   end
5 |   for v ∈ p_{i-1}(i) do
6 |   |   p_i(v) := p_{i-1}(v) ∪_v (i.to)
7 |   end
8 end
```

Algorithm 2: Main steps for generating pairable sets. Here (V, E) is the input graph with V the set of nodes, and E the set of edges. $p_i(u)$ is the set of nodes pairable with u computed up to step i (> 0).

3.1.3. Computing common ancestors of non-trivial pairs

Algorithm 3 combines the computation of pairable nodes with the computation of (a subset of) their common ancestors $q_i(u, v)$, which contains their lowest common ancestors. The main ingredients of Algorithm 3 is the addition of steps in lines 5, 13 and 14 which iteratively update common ancestors for pairable nodes (see section 3.1.4 for the intermediate results of step-by-step run of Algorithm 3 on the example in Figure 2). Note that Algorithm 3

does not guarantee that all common ancestors of u, v will eventually be included in $q_i(u, v)$, but it does include all lowest common ancestors of u, v (see Theorem 4 in section 3.2). Therefore, an additional straightforward step is needed to extract the lowest elements in $q_i(u, v)$ to obtain $\mathsf{lca}\{u, v\}$.

Input: (V, E) in topological order.

Output: Pairable nodes as well as their common ancestors (in q_i).

```
1  for i ∈ V do
2      for u ∈ i.to do
3          p_0(u) := i.to − {u}
4          for v ∈ i.to with u ≠ v do
5              q_0(u, v) := q_0(u, v) ∪ {i};
6          end
7      end
8  end
9  for i ∈ V do
10     for u ∈ i.to do
11         p_i(u) := p_{i−1}(u) ∪_u p_{i−1}(i)
12         for v ∈ p_{i−1}(i) do
13             q_i(u, v) := q_{i−1}(u, v) ∪ q_{i−1}(i, v);
14             q_i(v, u) := q_i(u, v);
15             p_i(v) := p_{i−1}(v) ∪_v (i.to)
16         end
17     end
18 end
```

Algorithm 3: Main steps for generating common ancestors for all and only pairable nodes. Here (V, E) is the input graph with V the set of nodes, and E the set of edges. $q_i(u, v)$ is the set of common ancestors for nodes u and v computed up to step i. Note that when $i = 0$, $q_0(u, v)$represents the initialization result (lines 1-8), computed before the main phase (lines 9-18).

3.1.4. *Illustrative example*

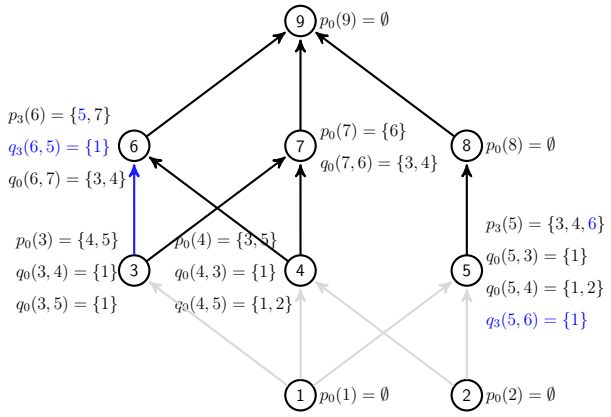

Figure 3: Updating up to node 3 and edge $(3, 6)$.

Although using only a small number of steps, the recursive nature involved in Algorithm 3 as well as the intricate behavior can be better demonstrated through an example. The following figures illustrate a step-by-step run of Algorithm 3 on the example in Figure 2. Edges being iterated and incremental value changes are highlighted in blue.

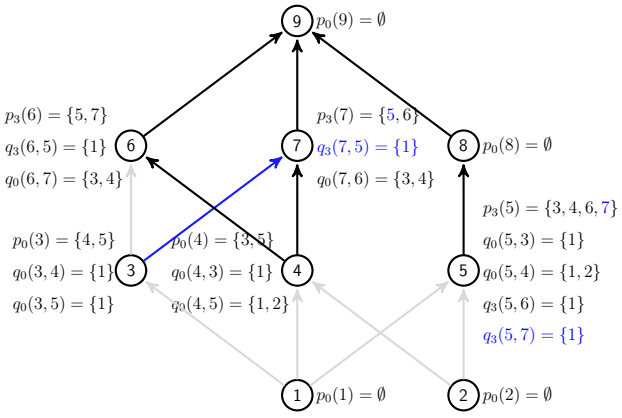

Figure 4: Step for node 3 and edge (3, 7).

Updating up to node 3 and edge $(3, 6)$ gives the result illustrated in Figure 3. Note that nothing gets updated when $i = 1, 2$. When $i = 3, u = 6, v = 4$, since nodes 6 and 4 are not pairable, nothing gets updated. When $i = 3, u = 6, v = 5$, since nodes 6 and 5 are pariable, we have $p_3(6) = \{5, 7\}, p_3(5) = \{3, 4, 6\}, q_3(6, 5) = q_3(5, 6) = \{1\}$.

As shown in Figure 4, for $i = 3, u = 7, v = 4$, since nodes 7 and 4 are not pairable, no updates took place. For $i = 3, u = 7, v = 5$, since $p_3(7) = \{5, 6\}$ and $p_3(5) = \{3, 4, 6, 7\}$, we

have $q_3(7,5) = q_3(5,7) = \{1\}$.

Figure 5 captures the snapshot for $i = 4$ and $u = 6$: when $v = 3$, we have nodes 6 and 3 are not pairable and no update is needed; when $v = 5$, we have $q_4(6,5) = q_4(5,6) = \{1,2\}$.

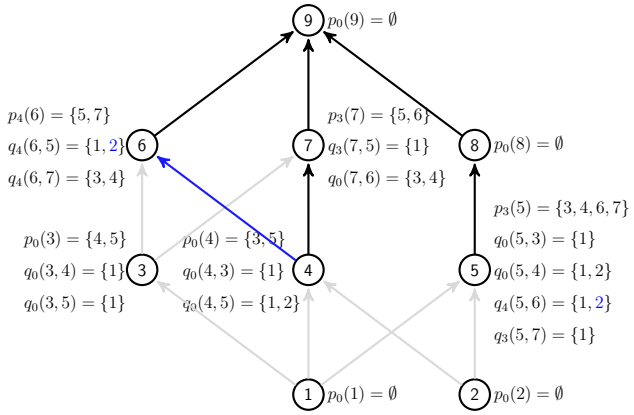

Figure 5: Step for node 4 and edge (4, 6).

Figure 6 shows the step for $i = 4$ and $u = 7$: when $v = 3$, we have nodes 7 and 3 are not pairable; when $v = 5$, the updated result is $q_4(5,7) = q_4(7,5) = \{1,2\}$.

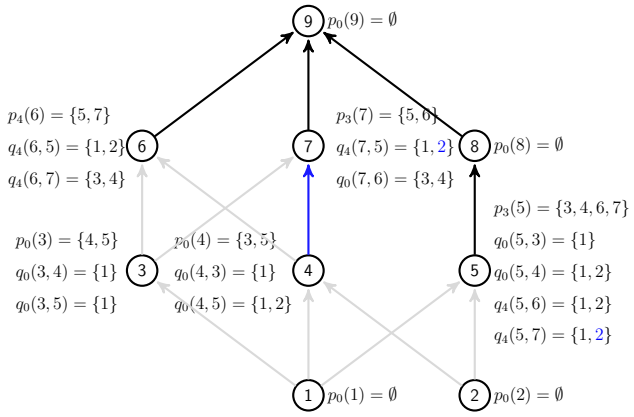

Figure 6: Step for node 4 and edge (4, 7).

Figure 7 captures the following configurations. $i = 5, u = 8, v = 3$: $p_5(3) = \{4,5,8\}, p_5(8) = \{3\}, q_5(3,8) = q_5(8,3) = \{1\}$; $i = 5, u = 8, v = 4$: $p_5(4) = \{3,5,8\}, p_5(8) = \{3,4\}, q_5(8,4) = q_5(4,8) = \{1,2\}$; $i = 5, u = 8, v = 6$: $p_5(8) = \{3,4,6\}, p_5(6) = \{5,7,8\}, q_5(8,6) = q_5(6,8) = \{1,2\}$; $i = 5, u = 8, v = 7$: $p_5(8) = \{3,4,6,7\}, p_5(7) = \{5,6,8\}, q_5(8,7) = q_5(7,8) = \{1,2\}$.

Finally, Figure 8 shows that for $i = 6, 7, 8$, nothing gets updated since node 9 is not pairable to any other node.

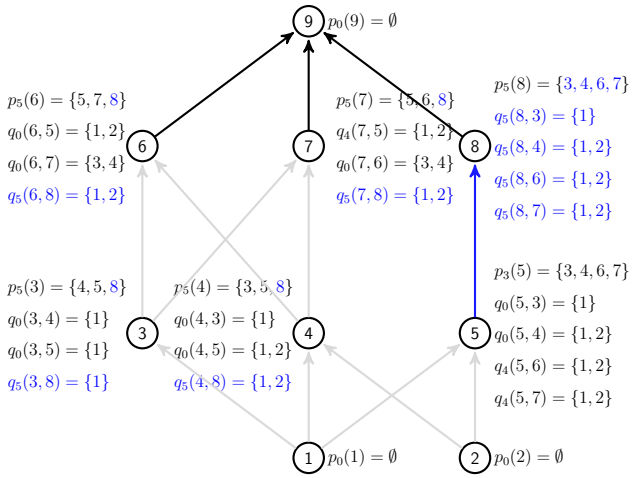

Figure 7: Step for node 5 and edge (5, 8).

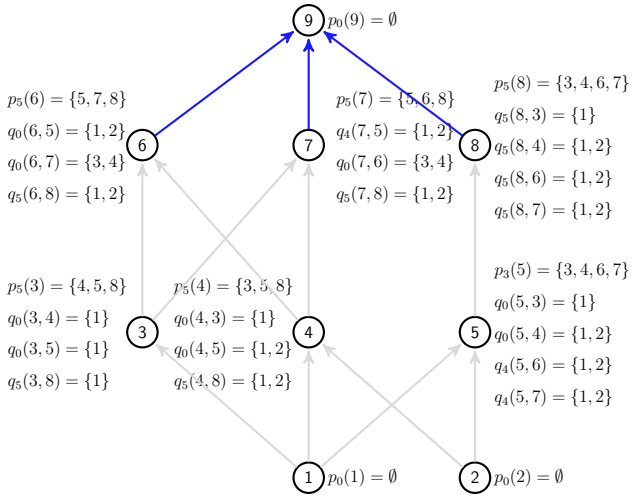

Figure 8: For node $i = 6, 7, 8$, nothing gets updated since node 9 is not pairable to any other node.

3.2. Correctness of the algorithm

We establish the correctness of Algorithm 2 and Algorithm 3 in a sequence of lemmas and theorems. Proof details are given in Appendix A for those who are interested.

With respect to a topologically sorted input graph (V, E), we distinguish the set $\pi(u)$ of all nodes pairable with u, and $p_i(u)$, the dynamic store of nodes pairable with u *at a stage* i *of the algorithm*. In the remainder of the paper we refer to nodes in V solely by their topological indices, integers that can also be incremented for algorithmic iteration in a while-loop.

According to Algorithm 2, $p_i(u)$ has the following straightforward properties:

- Monotonicity: for all $w \in V$, for all $i \leq j \in V$, we have $p_i(w) \subseteq p_j(w)$;
- Symmetry: for all $u, v \in V$, for all $i \in V$, $u \in p_i(v)$ implies $v \in p_i(u)$;
- Diagonality: for all $v \in V$, $p_v(v) = p_{v-1}(v)$.

Since Algorithm 2 initializes and grows $p_i(u)$ with only nodes pairable with u, we have

Theorem 1. *For all $u \in V$, for all $i \in V$,*

$$p_i(u) \subseteq \pi(u).$$

For proving containment in the other direction the next three lemmas serve as building blocks. Notationally, we use $[x, y]$ to stand for the closed integer interval $\{i \mid x \leq i \leq y\}$.

Lemma 1. *Suppose $b \in \mathsf{lca}(u, v)$ and $(b, u) \in E$. For $i \in [0, n]$, let $(v_i, v_{i+1}) \in E$ be edges such that $b = v_0$ and $v_n = v$. Then $u \in p_{v_{(i-1)}}(v_i)$ for all $i \in [1, n]$.*

Lemma 2. *Let $(v_i, v_{i+1}) \in E$ be edges in (V, E) for $i \in [0, n]$, with $b = v_0$ and $v_n = v$. Suppose $\mathsf{lca}(x, v) = b$ and $x \in \pi(v_i)$ for $i \in [0, n]$. If $x \in p_{v_k}(v_{k+1})$ for some k, then $x \in p_{v_j}(v_{j+1})$ for all $j \in [k, n]$.*

Lemma 3. *For all $0 < i < n$, we have $p_{v_i}(v_i) \subseteq p_{v_i}(v_{i+1})$, and moreover $p_{v_i}(v_i) \subseteq p_{v_{(i+1)}}(v_{i+1})$, by monotonicity.*

Lemmas 1, 2, and 3 show how pairability information is propagated along a path. Next we deal with the general situation of how this information is propagated to a pair of (pairable) nodes starting from the initial setting. To do so, consider a subgraph $D = A \cup B$ of (V, E), with $A = \{u_i \mid i \in [1, m]\}$ and $B = \{v_j \mid j \in [1, n]\}$ such that (u_{i-1}, u_i) and (v_{j-1}, v_j) are distinct edges with $i \in [1, m]$ and $j \in [1, n]$, where (see Figure 9)

1. $u_0 = v_0$, $u_m = u$, and $v_n = v$,
2. $u \in \pi(v)$ and $u_0 \in \mathsf{lca}(u, v)$, and
3. $A \cap B = \emptyset$.

Consider $W = \{w_i \mid i \in [1, m+n]\} = A \cup B$, with topological indices appearing in $A \cup B$ sorted in ascending order.

Definition 1. *The i-th alternation index for W is the index α_i, such that either $w_{\alpha_i} \in A$ but $w_{\alpha_i+1} \in B$, or $w_{\alpha_i} \in B$ but $w_{\alpha_i+1} \in A$.*

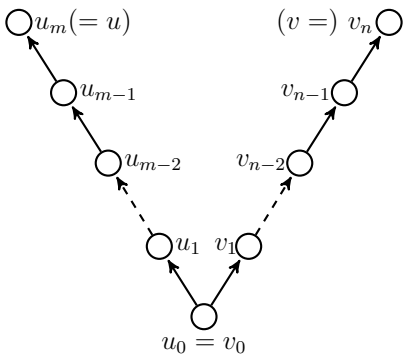

Figure 9: Subgraph with $A = \{u_i \mid i \in [1, m]\}$ and $B = \{v_j \mid j \in [1, n]\}$.

The next lemma, whose proof appears in Appendix A, characterizes how pairability information "jumps" from one branch (say A) to the other (say B) at critical junctures of an alternation index.

Lemma 4. *For any alternation index α_i, we have: 1. if $v_t = w_{\alpha_i}$ and $u_s = w_{\alpha_i+1}$ then $u_s \in p_{v_t}(v_t)$; 2. if $u_s = w_{\alpha_i}$ and $v_t = w_{\alpha_i+1}$ then $v_t \in p_{u_s}(u_s)$.*

The following Theorem 2, whose proof appears in Appendix A, deals with the opposite direction of Theorem 1. It allows us to conclude that for each $u \in V$, if $v \in \pi(u)$ then there exists $i \in V$, such that $v \in p_i(u)$ by choosing a large enough i. With it, all nodes pairable with u are accounted for by the function $p_i(u)$.

Theorem 2. *For all $i \in V$ and for all $w \leq i$, we have*

$$p_i(w) \supseteq \pi(w) \cap [1, i],$$

where $[1, i]$ stands for the integer interval $\{j \mid 1 \leq j \leq i\}$.

Similar to $p_i(u)$, the binary function $q_i(u, v)$ has the following properties, as can be directly derived from Algorithm 3:

- Monotonicity: for all $u, v \in V$, for all $i \leq j \in V$, we have $q_i(u, v) \subseteq q_j(u, v)$;
- Symmetry: for all $u, v \in V$, for all $i \in V$, we have $q_i(u, v) = q_i(v, u)$;
- Diagonality: For all $u, v \in V$, we have $q_u(u, v) = q_{u-1}(u, v)$.

By inspecting steps involved in Algorithm 3, we can establish this fact:

Theorem 3. *For each $i \in V$ and for each $u \in \pi(v)$, we have $q_i(u, v) \subseteq \mathsf{ca}\{u, v\}$.*

The next lemma shows how alternation indices help propagate the common ancestor information to all relevant pairs in the graph.

Lemma 5. *Suppose $x \in \mathsf{lca}\{u, v\}$, $u \in \pi(v)$, and suppose that (see Figure 9) (u_{i-1}, u_i) and (v_{j-1}, v_j) are distinct edges with $i \in [1, m]$ and $j \in [1, n]$, where $x = u_0 = v_0$, $u_m = u$, and $v_n = v$. For any alternation index α_i as given in Definition 1, we have 1. if $w_{\alpha_i} = v_t$ and $w_{\alpha_i+1} = u_s$, then $x \in q_{v_t}(v_t, u_s)$; 2. if $w_{\alpha_i} = u_t$ and $w_{\alpha_i+1} = v_s$, then $x \in q_{u_t}(u_t, v_s)$.*

Lemma 5 leads to the following theorem, which affirms the correctness of Algorithm 3.

Theorem 4. *Suppose $x \in \mathsf{lca}\{u, v\}$ with $u \in \pi(v)$. Then either $x \in q_u(u, v)$ or $x \in q_v(u, v)$.*

Theorem 4 shows that Algorithm 3 finds all lowest common ancestors of u, v in $q_i(u, v)$, for some i. It does not, however, guarantee that all common ancestors of u, v will eventually be included in $q_i(u, v)$. Neither does Algorithm 3 ensure that all elements in $q_i(u, v)$ are LCAs of u and v. Therefore, an additional straightforward step is needed to extract the lowest elements in $q_i(u, v)$ after the termination of Algorithm 3, to obtain $\mathsf{lca}\{u, v\}$.

4. Results

ANT-LCA was implemented in Java based on JDK7. Experiments on SNOMED CT and GO were performed on a MacBook Pro running Mac OS X Yosemite, with 16GB RAM and Intel Core i7 processor. The Java code is available through GitHub (`https://github.com/licongcui/nonlattice`).

4.1. SNOMED CT

We used 9 versions of SNOMED CT (International Version) from 2012 to 2017, dated 07/2012 (i.e., July 2012), 01/2013, 07/2013, 01/2014, 07/2014, 01/2015, 07/2015, 01/2016, and 01/2017. Table 1 summarizes the basic results about each version of SNOMED CT, including number of concepts, number of is-a relations, number of concept pairs that are pariable after the initialization step in Algorithm 1, number of all pairable pairs, number of non-lattice pairs, and the compute time for non-lattice pairs and non-lattice fragments.

The 07/2012 version contained 296,433 concepts, with 440,049 direct is-a relations connecting concepts. Among all possible concept pairs, 150,639 were identified as pairable after

Table 1: Summary of the basic statistics using ANT-LCA to process 9 versions of SNOMED CT. Initial Number of Pairable Pairs indicates the number of concept pairs that are pariable after the initialization step in Algorithm 1.

	07/2012	**01/2013**	**07/2013**	**01/2014**	**07/2014**	**01/2015**	**07/2015**	**01/2016**	**01/2017**
Total Number of Concepts	296,433	297,998	298,818	298,581	300,751	312,998	317,057	319,446	326,734
Total Number of is-a Relations	440,049	442,711	444,919	443,944	446,462	463,339	470,040	473,121	487,686
Initial Number of Pairable Pairs	150,639	151,996	153,892	153,645	153,934	158,488	161,346	162,689	171,966
Total Number of Pairable Pairs	1,383,888	1,397,332	1,420,284	1,425,848	1,428,870	1,475,826	1,502,108	1,523,325	1,641,853
Total Number of Non-lattice Pairs	578,237	583,433	593,498	594,076	594,106	614,018	625,484	633,307	683,744
Compute Time for Non-lattice Pairs (in seconds)	28	28	29	29	27	29	30	28	32
Compute Time for Non-lattice Fragments (in seconds)	524	527	548	524	502	512	541	554	747

the initialization step in Algorithm 1, a total of 1,383,888 were detected as pairable, among which 578,237 were found to be non-lattice pairs. It took 28 seconds to compute non-lattice pairs and 524 seconds to compute non-lattice fragments.

In general, it takes about 30 seconds for our algorithm to detect all non-lattice pairs for each version of SNOMED CT, consistent with our linear time analysis. We run each version 10 times and report the average time in row "Compute Time for Non-lattice Pairs" in Table 1.

The generation of all non-lattice fragments took less than 13 minutes for each version of SNOMED CT. This phase is more time-consuming than detection of non-lattice pairs because all nodes in-between a node in the non-lattice pair and the lowest common ancestors make up a fragment. For this part, we run each version 5 times and report the average time in row "Compute Time for Non-lattice Fragments" in Table 1.

4.2. Gene Ontology

We used 8 versions of GO from July 2015 to Febuary 2016. Table 2 shows the basic results about each version of GO. The 02/2016 version contained a total of 44,222 concepts, with 72,742 direct is-a relations connecting concepts. Among all possible concept pairs, 3,642 were identified as pairable after the initialization step in Algorithm 1, a total of 328,760 were detected as pairable, among which 102,948 were found to be non-lattice pairs. It took 3 seconds to compute non-lattice pairs and 32 seconds to compute non-lattice fragments.

Table 2: Summary of the basic statistics using ANT-LCA to process 8 versions of GO.

	07/2015	**08/2015**	**09/2015**	**10/2015**	**11/2015**	**12/2015**	**01/2016**	**02/2016**
Total Number of Concepts	43,330	43,507	43,654	43,758	43,880	43,980	44,049	44,222
Total Number of is-a Relations	70,826	71,167	71,443	71,700	71,926	72,153	72,268	72,742
Initial Number of Pairable Pairs	3,502	3,537	3,547	3,564	3,574	3,573	3,575	3,642
Total Number of Pairable Pairs	305,270	308,314	309,684	311,490	312,667	314,340	314,448	328,760
Total Number of Non-lattice Pairs	92,322	93,828	94,275	94,821	94,912	95,458	95,506	102,948
Compute Time for Non-lattice Pairs (in seconds)	2	3	3	3	3	2	2	3
Compute Time for Non-lattice Fragments (in seconds)	31	31	32	32	33	30	30	32

4.3. Experiments on random graphs

We also evaluated the performance of ANT-LCA on randomly generated, ontologically shaped DAGs. We implemented an algorithm (see Appendix B) to generate a random $DAG(N, d, C_{\min}, C_{\max})$, where N is the number of nodes, d is edge density of the DAG, and $C_{\min}/C_{\max}$ are the minimum/maximum number of children a node can have. The edge density is defined as the ratio of the extra edges (that will be added after a random tree is generated) to the number of edges in the tree.

Densities in real world ontologies tend to be smaller than 1, even though it can be as high as $\frac{N}{2}$. In our experiments, the edge density parameter was set between 0.02 and 1. This is a reasonable range to consider, since GO (02/2016 version) has $d = 0.64$ and SNOMED CT (01/2016 version) has $d = 0.48$ as their edge density, respectively.

Figure 10 is a plot of the average running time of ANT-LCA on randomly generated ontological structures of different sizes (number of nodes ranging from 100,000 to 1,200,000) and densities (0.02 to 1). The experimental results are consistent with our algorithmic analysis (see section 5): they show linear increase in time complexity across the density spectrum, with the slope (linear coefficient) getting larger for denser graphs.

Figure 11 is a 3D view which illustrates the trend of time increase with respect to graph size and density. The experiments were performed on a linux running the CentOS with 16GB RAM and Intel(R) Xeon(R) X3430 2.40GHz quad core CPU.

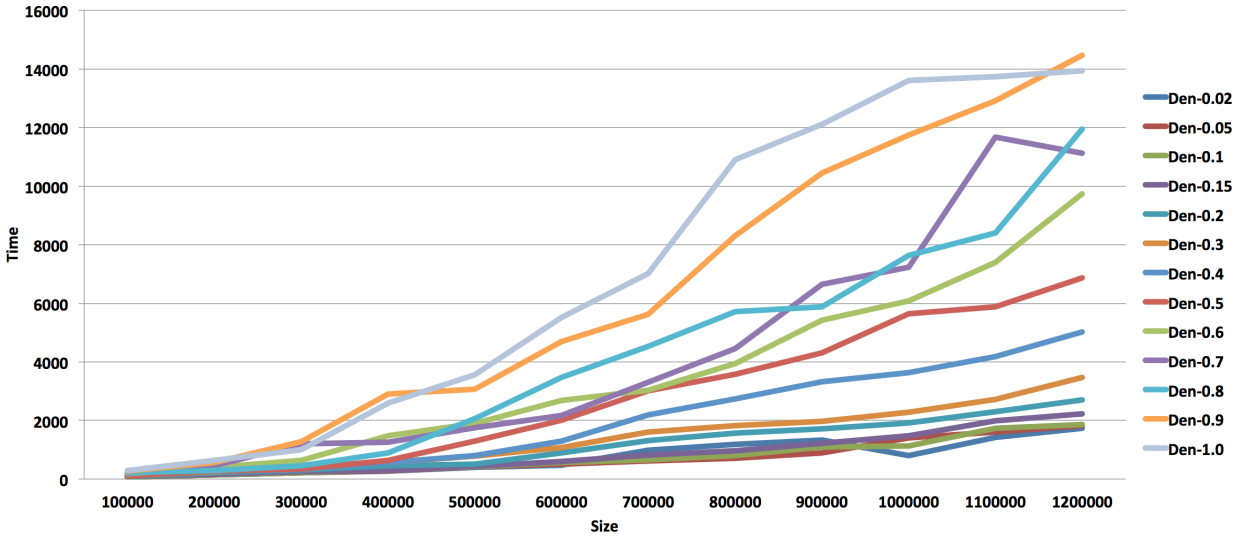

Figure 10: A plot of size vs. computational time in milliseconds. Different colors represent graphs of different density, with higher density requiring more computational time.

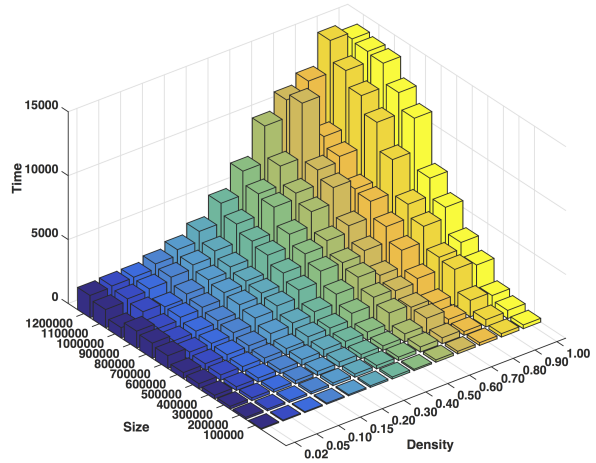

Figure 11: A 3D rendering showing the effect of size and density on required computational time.

5. Discussion

5.1. Time Complexity for Algorithm 2

Let σ_G be the *pairability degree* of graph G, defined as $\max_{u \in V} \pi(u)$, i.e. the maximum number of pairable nodes a single node can have in graph G. Algorithm 2 involves a main iteration process over all edges $(i, u) \in E$ of the input graph, as given in lines 1 and 2. Then

the time complexity for line 3 is (using set union complexity)

$$\sum_{(i,u)\in E} |p_i(u)|,$$

which is bounded by (with the assumption that the union cost is proportional to the size of the resulting set [12])

$$\sum_{(i,u)\in E} |\pi(u)|.$$

Hence,

$$\sum_{(i,u)\in E} |p_i(u)| \leq \sigma_G \cdot |E|.$$

Similarly, the time complexity for lines 4 and 5 is

$$\sum_{(i,u)\in E} \sum_{v\in p_{i-1}(i)} |p_i(v)|.$$

We have

$$\sum_{(i,u)\in E} \sum_{v\in p_{i-1}(i)} |p_i(v)| \leq \sigma_G^2 \cdot |E|.$$

Therefore, the overall time-complexity of Algorithm 2 is bounded by $\sigma_G^2 \cdot |E|$. Space complexity is similarly bounded, but less of a concern here due to the availability of sufficiently large, standard sizes of RAMs. For sparse graphs with small σ_G, Algorithm 2 performs well, as our experimental result in the next section shows. In the worst case $\sigma_G = |V|$, and the running time in the worst case is $O(|V|^2 \cdot |E|)$ and is the same as brute force search. In the best case σ_G is a constant, and the running time in the base case is $O(|E|)$. The actual time needed for the algorithm, $\sum_{(i,u)\in E} \sum_{v\in p_{i-1}(i)} |p_i(v)|$, is very close to the best case for the data set in our experiments. Even though σ_G may be in the thousands, the average size of $\pi(v)$, a more realistic estimation for the actual computational time, is below 50.

Intuitively, the more tree-like the input ontology is, the closer to the best case time-complexity of $O(|E|)$ our algorithm will achieve. The worst cases are when every pair of nodes is pairable, achievable when the ontology is dense with shared descendant concepts among its concept nodes.

5.2. Time Complexity for Algorithm 3

Note that we intentionally nested the for-loop in lines 4-6 of Algorithm 2, to faithfully account for the time-complexity for Algorithm 3. For Algorithm 3, the double nesting is necessary in order to compute pairable pairs while accumulating common ancestors (between u and v). If we are interested only in computing pairability, then the nesting in lines 4-6 of Algorithm 2 is not necessary, and we obtain a better time-complexity of $\sigma_G \cdot (|E| + |V|)$.

The key steps involved in Algorithm 3 can be captured by Algorithm 2 except for the accumulation of common ancestors in steps 13 and 14. We assume the computation required for these two steps to be a constant by keeping up to two LCAs, in order to provide a fair comparison with existing algorithms (which only output a representative LCA for each pair). Therefore, the time-complexity of Algorithm 2 is also bounded by $\sigma_G^2 \cdot |E|$. Therefore, the best case and worst case analyses for Algorithm 2 apply to Algorithm 3 as well.

5.3. Related work on LCA

Many attempts have been made on improving the efficiency of algorithms for the all-pairs all-LCA problem [13, 14], i.e., finding all LCAs associated with each pair of nodes. More recently, Dash et al. [15] presented an approach that combines the efficiency of existing LCA algorithms on trees with range-interval labeling scheme and an efficient matrix multiplication. This approach achieves near-linear time for tree-like, rooted DAGs, but query results are limited to a single representative LCA per each pair of nodes. This is a limit for applications that require all-LCAs as query results. In general, the all-pairs all-LCA problem remains to be super-quadratic, since its time-complexity is inherently tied to algorithms for matrix-multiplication [16, 14]. For many DAGs arising in real-world applications such SNOMED CT (with over 300,000 of nodes), existing algorithms become impractical.

In general approaches to the LCA problem, one distinguishes the off-line and online computations. Off-line computation serves to preprocess the input graph in order to speedup online LCA queries. Our paper focuses on off-line processing in order to support constant online query for a representative LCA, or online query for all LCAs (with performance parameterized in the size of the resulting set).

A key distinction of ANT-LCA from existing approaches is that it ensures computation

is performed on all and only non-trivial pairs. In fact, the time complexity of ANT-LCA is determined by the number of non-trivial pairs in the input graph, as our complexity analysis shows. Using the average size of pairable pairs for a give node, which is a more realistic reflection of the actual computational time, the time complexity for our experimental cases is approximately $(50)^2 \cdot |E|$.

Another distinction of our approach is that we compute all LCAs (of all non-trivial pairs) instead of a representative LCA. This makes our task more computationally intensive, and also makes many existing approaches to the LCA problem inapplicable. Our all LCA requirement is motivated by real-world application needs for implementing lattice-based approach to ontology quality assurance. Compared with the fastest all pairs representative LCA algorithm known to date with an $O(|V| \cdot |E|)$ time complexity [15], ANT-LCA provides a rough speed-up of three orders of magnitude for SNOMED CT. However, the worst time-complexity for our algorithm, $|V|^2 \cdot |E|$, is attained when virtually all nodes are pairable with all other nodes.

5.4. Related work using non-lattice subgraphs

This paper focused on an efficient algorithm to compute non-lattice pairs as a key part of step 1 in a 4-step non-lattice approach outlined in Introduction. More recent work has addressed other steps and reported specific application for improvements on SNOMED CT and NCI Thesaurus. In [7], a structural-lexical method was used to mine lexical patterns in non-lattice fragments in SNOMED CT to identify missing is-a relations and concepts. This method used 4 patterns to cover about 4% of all non-lattice fragments in SNOMED CT, with a solid precision rate (59%) of confirmed errors by domain experts. More recently, a new structural-lexical approach leveraged more existing knowledge in SNOMED CT by enriching the lexical attributes of each concept in non-lattice subgraphs to facilitate the identification of missing is-a relations [17]. This approach covered 7.4% of non-lattice subgraphs with higher precision (82.96%). Work reported in [9] demonstrated that the non-lattice approach can be applied to other ontologies than SNOMED CT (9.93% coverage of non-lattice fragments with 66% precision on identified errors in NCI Thesaurus).

Given such developments, it may seem reasonable to propose the reduced proliferation

of non-lattice substructures (i.e., the total number of non-lattice pairs) as a ontology quality metric. However, due to many factors that are involved in creating newer releases of an ontology, we found it not to be the case that newer releases would measure better than earlier releases. It may still be possible to use this method to measure and track the quality of specific sub-hierarchies where non-lattice fragments are unusually dense, or to demonstrate that a non-trivial portion of ontological changes between the releases involve non-lattice fragments.

6. Limitations

Since ANT-LCA is designed for detecting lowest common ancestors for all non-trivial pairs in a DAG, it is generally applicable to other ontologies or terminologies which are hierarchically organized in a DAG. We have applied it to SNOMED CT, Gene Ontology, and NCI Thesaurus for ontology quality assurance.

There are two types of limitations. One is specific to the ANT-LCA algorithm, and the other is related to the non-lattice approach. The limitation of the ANT-LCA algorithm is that, although it is efficient and suitable for ontological graph structures that are tree-like, it may not work well with other types of graph structures when all pairs of nodes are pairable.

Limitations of the non-lattice approach include the following. (1): The approach may not be efficient for ontologies that are "shallow," such as Ontology for General Medical Science (maximum depth 6), BRENDA Tissue and Enzyme Source Ontology (maximum depth 6), and Current Procedural Terminology (maximum depth 7), from BioPortal. (2): Our algorithm itself is agnostic to relation types, so it will still work for such relations as "part-of." However, the non-lattice approach is not applicable to other types of relations since this approach is only meaningful for the is-a hierarchy (of any ontology) due to its theoretical underpinning based Formal Concept Analysis. We are not aware of any (theoretical) reasons that indicate non-lattice fragments to be problematic for other types of relations. However, this should not diminish the value of our non-lattice approach.

7. Conclusions

To summarize, this paper introduced an efficient algorithm for detecting non-lattice pairs and generating non-lattice fragments, for ontology quality assurance work. Our algorithm overcomes a fundamental computational barrier in sub-graph based structural analysis of large ontological systems. It enables the implementation of a new breed of structural auditing methods that not only identifies potential problematic areas, but also automatically suggests changes to fix the issues.

Acknowledgements

This work was supported by the National Science Foundation through grants IIS-1657306 and ACI-1626364, and the National Institutes of Health (NIH) National Center for Advancing Translational Sciences through grant UL1TR001998. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH.

References

[1] SNOMED CT. `http://www.snomed.org/snomed-ct`, 2018 (accessed 13 January 2018).

[2] P. Zweigenbaum, B. Bachimont, J. Bouaud, J. Charlet, J.F. Boisvieux, Issues in the structuring and acquisition of an ontology for medical language understanding, Methods of information in medicine 34(1995) 15-24.

[3] B. Ganter, R. Wille, Formal concept analysis: mathematical foundations, Springer Science & Business Media, Berlin, Germany, 2012.

[4] G. Jiang, C.G Chute, Auditing the semantic completeness of SNOMED CT using formal concept analysis, J. Am. Med. Inform. Assoc. 16(1) (2009) 89-102.

[5] G.Q. Zhang, O. Bodenreider, Large-scale, exhaustive lattice-based structural auditing of SNOMED CT, In AMIA Annual Symposium Proceedings (2010) 922-926.

[6] L. Cui, S. Tao, G.Q. Zhang, Biomedical ontology quality assurance using a big data approach, ACM T. Knowl. Discov. D. 10(4) (2016) 41.

[7] L. Cui, W. Zhu, S. Tao, J.T. Case, O. Bodenreider, G.Q. Zhang, Mining non-lattice subgraphs for detecting missing hierarchical relations and concepts in SNOMED CT, J. Am. Med. Inform. Assoc. 24(4) (2017) 788-798.

[8] X. Zhu, J.W. Fan, D.M. Baorto, C. Weng, J.J. Cimino, A review of auditing methods applied to the content of controlled biomedical terminologies, J. Biomed. Inform. 42(3) (2009) 413-425.

[9] R. Abeysinghe, M.A. Brooks, J. Talbert, L. Cui, Quality Assurance of NCI Thesaurus by Mining Structural-Lexical Patterns, In AMIA Annual Symp Proc (2017) 364-373.

[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. Introduction to Algorithms. The MIT Press. Cambridge MA. 1990.

[11] C. Demetrescu, G.F. Italiano, Fully dynamic transitive closure: Breaking through the O(n^2) barrier, In 41st Annual Symposium on Foundations of Computer Science Proceedings (2000) 381-389.

[12] Z. Galil, G.F. Italiano, Data structures and algorithms for disjoint set union problems, ACM Computing Surveys (CSUR) (1991) 23(3) 319-44.

[13] M.A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, P. Sumazin, Lowest common ancestors in trees and directed acyclic graphs, J. Algorithm. (2005) 57(2) 75-94.

[14] A. Czumaj, M. Kowaluk, A. Lingas, Faster algorithms for finding lowest common ancestors in directed acyclic graphs, Theor. Comput. Sci. (2007) 380(1) 37-46.

[15] S.K. Dash, S.B. Scholz, S. Herhut, B. Christianson, A scalable approach to computing representative lowest common ancestor in directed acyclic graphs, Theor. Comput. Sci. (2013) (513) 25-37.

[16] Eckhardt S, Mühling AM, Nowak J. Fast lowest common ancestor computations in DAGs. In Algorithms ESA (2007) 705-716.

[17] L. Cui, O. Bodenreider, J. Shi, G.Q. Zhang. Auditing SNOMED CT hierarchical relations based on lexical features of concepts in non-lattice subgraphs. J. Biomed. Inform. https://doi.org/10.1016/j.jbi.2017.12.010

Appendix A: correctness proofs

7.1. Proof of Lemma 4

We prove this by induction on the i-th alternation index α_i.

Basis $i = 1$. Without loss of generality, suppose $v_j < u_1$ for all $1 \leq j \leq t$. Then $u_1 \in p_{v_0}(v_1)$. By monotonicity and Lemma 3 we have $u_1 \in p_{v_t}(v_t)$.

For the inductive step, assume $v_t < u_s$ defines the $(i+1)$-th alternation index, for some $s > 1$. We must have $u_{s-1} < v_t$. By Definition 1, the i-th alternation index must be of the form $u_{s-1} < v_\tau$, for some $\tau \leq t$. By induction hypothesis and the symmetry of the graph structure, we have $v_\tau \in p_{u_{s-1}}(u_{s-1})$.

By Algorithm 2, we have

$$p_{u_{s-1}}(u_s) = p_{u_{s-1}-1}(u_s) \cup_{u_s} p_{u_{s-1}-1}(u_{s-1}).$$

Since $p_{u_{s-1}}(u_{s-1}) = p_{u_{s-1}-1}(u_{s-1})$ by diagonality, we have $v_\tau \in p_{u_{s-1}}(u_s)$. Furthermore,

$$\begin{aligned} v_\tau \in p_{u_{s-1}}(u_s) &\implies u_s \in p_{u_{s-1}}(v_\tau) && (\textit{by symmetry}) \\ &\implies u_s \in p_{v_\tau}(v_\tau) && (\textit{by monotonicity}) \\ &\implies u_s \in p_{v_t}(v_t) && (\textit{by Lemma 3}) \end{aligned}$$

7.2. Proof of Theorem 2

Basis: $i = 1, 2$. We have $\pi(1) = \emptyset$, $1 \notin \pi(2)$, and $2 \notin \pi(2)$. Therefore, $\pi(1) \cap [1,1] = \pi(2) \cap [1,2] = \emptyset$.

For the inductive step, suppose, using course of value induction, that for all $j \leq k$, for all $w \leq j$,

$$p_j(w) \supseteq \pi(w) \cap [1, j].$$

We plan to prove that for $k+1$ we have that for all $u \leq k+1$,

$$p_{k+1}(u) \supseteq \pi(u) \cap [1, k+1].$$

For a given $u \leq k+1$, there are two possible cases for u:

$$\begin{aligned} &\text{Case 1:} \quad k+1 \notin \pi(u); \\ &\text{Case 2:} \quad k+1 \in \pi(u). \end{aligned}$$

We show that for each case, the desired set containment holds.

Case 1. If $k+1 \notin \pi(u)$, then we have

$$\pi(u) \cap [1, k+1] = \pi(u) \cap [1, k] \subseteq p_k(u) \subseteq p_{k+1}(u)$$

as needed, by induction hypothesis and monotonicity.

Case 2. Suppose $k+1 \in \pi(u)$, with $u < k+1$. Then there exists b with $b \in \mathsf{lca}(k+1, u)$ such that $(u_0, u_1), (u_1, u_2), \ldots, (u_{m-1}, u_m)$ and $(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$ are distinct edges of E with $b = u_0 = v_0$, $u_m = u$ and $v_n = k+1$.

We show that $u \in p_{v_n}(v_n)$. There are two cases: (a) $u_m < v_1$, and (b) $v_\alpha < u_m$ for some $\alpha \leq n-1$ (note that α cannot be n in this case).

For (a), by Lemma 4, we have $v_1 \in p_{u_m}(u_m)$. Furthermore,

$$\begin{aligned} v_1 \in p_{u_m}(u_m) &\implies v_1 \in p_u(u) && (u_m = u) \\ &\implies u \in p_u(v_1) && (\textit{by symmetry}) \\ &\implies u \in p_{v_1}(v_1) && (\textit{by monotonicity}) \\ &\implies u \in p_{v_n}(v_n) && (\textit{by Lemma 3}) \end{aligned}$$

For (b), assume α is the largest index such that $v_\alpha < u_m$. Then by Lemma 4, we have $u_m \in p_{v_\alpha}(v_\alpha)$. By Lemma 3, we have $u \in p_{v_n}(v_n)$. Hence $k+1 \in p_{k+1}(u)$, by symmetry.

By induction hypothesis, we have

$$\begin{aligned} p_{k+1}(u) &\supseteq p_k(u) \\ &\supseteq \pi(u) \cap [1, k]. \end{aligned}$$

Since $k+1 \in p_{k+1}(u)$, we have $p_{k+1}(u) \supseteq \pi(u) \cap [1, k+1]$ as needed.

7.3. Proof of Lemma 5

We prove this by induction on i, where α_i is the i-th alternation index.

Basis $i = 1$. If $w_{\alpha_1} = v_t$ and $w_{\alpha_i+1} = u_1$, then we have $v_j < u_1$ for all $1 \leq j \leq t$. We prove that $x \in q_{v_j}(v_j, u_1)$ for all $1 \leq j \leq t$. When $j = 1$, we have $u_1 \in p_{v_1}(v_1)$, by the fact

that $u_1 \in p_0(v_1)$ and by monotonicity. Since $x \in q_0(u_1, v_1)$ (by line 5 of Algorithm 3, we have $x \in q_{v_1}(v_1, u_1)$ by monotonicity and symmetry.

Suppose $x \in q_{v_j}(v_j, u_1)$ for some $1 \leq j < t$. Then $x \in q_{v_j-1}(v_j, u_1)$ by diagonality. Since $u_1 \in p_{v_j}(v_j)$ (by Lemma 3), we have $u_1 \in p_{v_j-1}(v_j)$, again by diagonality. Since $(v_j, v_{j+1}) \in E$, we have

$$q_{v_j}(v_{j+1}, u_1) = q_{v_j-1}(v_{j+1}, u_1) \cup q_{v_j-1}(v_j, u_1).$$

Therefore, $x \in q_{v_j}(v_{j+1}, u_1)$ and so $x \in q_{v_{j+1}}(v_{j+1}, u_1)$ (by monotonicity). This finishes the induction to give us $x \in q_{v_t}(v_t, u_1)$. A similar argument holds for the case when $w_{\alpha_1} = u_t$ and $w_{\alpha_i+1} = v_1$.

For the inductive step, let $v_t < u_s$ define the $(i+1)$-th alternation index. We must have $u_{s-1} < v_t$. By Definition 1, the i-th alternation index must be of the form $u_{s-1} < v_\tau$, for some $\tau \leq t$. In increasing order, we have the sequence $u_{s-1} < v_\tau \leq v_t < u_s$. By induction hypothesis, we have $x \in q_{u_{s-1}}(u_{s-1}, v_\tau)$, and so $x \in q_{u_{s-1}-1}(u_{s-1}, v_\tau)$ (by diagonality).

We now show that $x \in q_{v_j}(v_{j+1}, u_s)$ for each $j \in [\tau, t]$ by induction.

For basis $j = \tau$, we first show that **(a)**: $u_s \in p_{v_\tau-1}(v_\tau)$, and **(b)**: $x \in q_{v_\tau}(v_\tau, u_s)$, because if these are true, then by diagonality we have $x \in q_{v_\tau-1}(v_\tau, u_s)$, and Algorithm 3 gives us

$$q_{v_\tau}(v_{\tau+1}, u_s) = q_{v_\tau-1}(v_{\tau+1}, u_s) \cup q_{v_\tau-1}(v_\tau, u_s).$$

Therefore $x \in q_{v_\tau}(v_{\tau+1}, u_s)$, as needed.

(a): To show that $u_s \in p_{v_\tau-1}(v_\tau)$, note by Lemma 4, we have $v_\tau \in p_{u_{s-1}}(u_{s-1})$ ($u_{s-1} < v_\tau$ is an alternation). Furthermore,

$$\begin{aligned} v_\tau \in p_{u_{s-1}}(u_{s-1}) &\implies v_\tau \in p_{u_{s-1}}(u_s) && \text{(by Lemma 3)} \\ &\implies u_s \in p_{u_{s-1}}(v_\tau) && \text{(by symmetry)} \\ &\implies u_s \in p_{v_\tau-1}(v_\tau) && (u_{s-1} \leq v_\tau - 1) \end{aligned}$$

(b): To see that we have $x \in q_{v_\tau}(v_\tau, u_s)$, note that induction hypothesis gives us $x \in q_{u_{s-1}}(u_{s-1}, v_\tau)$. Also,

$$q_{u_{s-1}}(u_s, v_\tau) = q_{u_{s-1}-1}(u_s, v_\tau) \cup q_{u_{s-1}-1}(u_{s-1}, v_\tau)$$

by instantiating Algorithm 3 with $(u_{s-1}, u_s) \in E$ and $v_\tau \in p_{u_{s-1}-1}(u_{s-1})$. Therefore, $x \in$

$q_{u_{s-1}}(u_s, v_\tau)$. Furthermore,

$$\begin{aligned} x \in q_{u_{s-1}}(u_s, v_\tau) &\implies x \in q_{v_\tau}(u_s, v_\tau) && \text{(by monotonicity)} \\ &\implies x \in q_{v_\tau}(v_\tau, u_s) && \text{(by symmetry)} \end{aligned}$$

If $\tau = t$ then the proof is already compete. If $\tau < t$, this completes the induction basis $j = \tau$, because $x \in q_{v_\tau}(v_\tau, u_s)$ implies $x \in q_{v_\tau}(v_{\tau+1}, u_s)$.

For the inductive step, assume $x \in q_{v_j}(v_{j+1}, u_s)$ for *some* j such that $\tau \leq j < t-1$. Since $v_j \leq v_{j+1} - 1$, we have $x \in q_{v_{j+1}-1}(v_{j+1}, u_s)$. Since $u_s \in p_{v_\tau - 1}(v_\tau)$, we have $u_s \in p_{v_{j+1}-1}(v_{j+1})$ by diagonality and Lemma 3. By instantiating Algorithm 3 with $(v_{j+1}, v_{j+2}) \in E$ and $u_s \in p_{v_{j+1}-1}(v_{j+1})$,

$$q_{v_{j+1}}(v_{j+2}, u_s) = q_{v_{j+1}-1}(v_{j+2}, u_s) \cup q_{v_{j+1}-1}(v_{j+1}, u_s).$$

Therefore, $x \in q_{v_{j+1}}(v_{j+2}, u_s)$. By induction, we have $x \in q_{v_{t-1}}(v_t, u_s)$, and so $x \in q_{v_t}(v_t, u_s)$ (by monotonicity).

A similar argument holds for the case when the $(i+1)$-th alternation index is defined by $u_t < v_s$.

7.4. Proof of Theorem 4

If $u_m < v_j$ defines the last alternation index, then by Lemma 5, we have $x \in q_{u_m}(u_m, v_j)$. That is, $x \in q_u(u, v_j)$. If $n = 1$, then $j = n = 1$ and we have $x \in q_u(u, v)$.

We show that $x \in q_{v_k}(v_{k+1}, u)$ for each $k \in [j, n-1]$ by induction for the case when $n > 1$.

Basis: $k = j$. By Lemma 4, we have $v_j \in p_{u_m}(u_m) = p_u(u)$. Moreover,

$$\begin{aligned} v_j \in p_u(u) &\implies u \in p_u(v_j) && \text{(by symmetry)} \\ &\implies u \in p_{v_j}(v_j) && (u < v_j) \\ &\implies u \in p_{v_j - 1}(v_j) && \text{(by diagonality)} \end{aligned}$$

By instantiating Algorithm 3 with $(v_j, v_{j+1}) \in E$ and $u \in p_{v_j - 1}(v_j)$, we have

$$q_{v_j}(v_{j+1}, u) = q_{v_j - 1}(v_{j+1}, u) \cup q_{v_j - 1}(v_j, u).$$

Since $x \in q_u(u, v_j)$, we have $x \in q_{v_j - 1}(u, v_j) = q_{v_j - 1}(v_j, u)$ by monotonicity ($u \leq v_j - 1$) and symmetry. Therefore, $x \in q_{v_j}(v_{j+1}, u)$.

For inductive step, assume $x \in q_{v_k}(v_{k+1}, u)$ for *some* k such that $j \leq k < n-1$. Since $u \in p_{v_j}(v_j)$, we have $u \in p_{v_{k+1}}(v_{k+1})$ by Lemma 3, and so $u \in p_{v_{k+1}-1}(v_{k+1})$ by diagonality.

By instantiating Algorithm 3 with $(v_{k+1}, v_{k+2}) \in E$ and $u \in p_{v_{k+1}-1}(v_{k+1})$, we have

$$q_{v_{k+1}}(v_{k+2}, u) = q_{v_{k+1}-1}(v_{k+2}, u) \cup q_{v_{k+1}-1}(v_{k+1}, u).$$

Since $x \in q_{v_k}(v_{k+1}, u)$ (induction hypothesis), we have $x \in q_{v_{k+1}-1}(v_{k+1}, u)$ by monotonicity (with $v_k \leq v_{k+1} - 1$). Therefore, $x \in q_{v_{k+1}}(v_{k+2}, u)$.

This completes the induction, and we have $x \in q_{v_{n-1}}(v_n, u)$. Therefore, $x \in q_{v_n}(v_n, u)$ by monotonicity, i.e., $x \in q_v(v, u)$.

When $v_n < u_j$ defines the last alternation index, a similar argument gives $x \in q_u(u, v)$.

Appendix B: pseudocode for random graph generation

The random ontological graph generator algorithm (Algorithm 4) works in two steps:

1. Lines 7 to 18 generate a random rooted tree. Starting from the root, the number of children is randomly selected between $C_{\min}$ and $C_{\max}$. This process will continue until the number of nodes in the generated tree is equal to N.
2. Lines 19 to 26 add extra edges by selecting two random nodes. The number of edges that will be added is controlled by the density.

Input: $N, d, C_{\min}, C_{\max}$

Output: An randomly generated ontology.

```
1  q := new Queue();
2  root := new node;
3  E := (N − 1) × (1 + d);
4  q.enqueue(root);
5  numOfNodes := 0;
6  numOfEdges := 0;
7  edges := φ;
8  while numOfNodes < N do
9      p := q.dequeue();
10     c := random(C_min, C_max);
11     for i ≤ c do
12         child := new node;
13         edges := edges ∪ {(p, child)};
14         numOfNodes++;
15         numOfEdges++;
16         i++;
17     end
18 end
19 while numOfEdges < E do
20     src := random(0, numOfNodes);
21     dest := random(0, numOfNodes);
22     if src ≠ dest and (src, dest) ∉ edges then
23         edges := edges ∪ {(src, dest)};
24         numOfEdges++;
25     end
26 end
```

Algorithm 4: Procedure to generate a random ontology.