Efficient Concept-based Document Ranking

Anastasios Arvanitis* #1, Matthew Wiley #2, Vagelis Hristidis #3

University of California, Riverside, CA, USA

1tasos@cs.ucr.edu 2mwile001@cs.ucr.edu 3vagelis@cs.ucr.edu

ABSTRACT

Recently, there is increased interest in searching and computing the similarity between Electronic Medical Records (EMRs). A unique characteristic of EMRs is that they consist of ontological concepts derived from biomedical ontologies such as UMLS or SNOMED-CT. Medical researchers have found that it is effective to search and find similar EMRs using their concepts, and have proposed sophisticated similarity measures. However, they have not addressed the performance and scalability challenges to support searching and computing similar EMRs using ontological concepts. In this paper, we formally define these important problems and show that they pose unique algorithmic challenges due to the nature of the search and similarity semantics and the multi-level relationships between the concepts. In particular, the similarity between two EMRs is a function of the minimum semantic distance from each concept of one document to a concept of the other and vice versa. We present an efficient algorithm to compute the similarity between two EMRs. Then, we propose an early-termination algorithm to search for the top-k most relevant EMRs to a set of concepts, and to find the top-k most similar EMRs to a given EMR. We experimentally evaluate the performance and scalability of our methods on a large real EMR data set.

1. INTRODUCTION

Adoption and usage of Electronic Medical Records (EMRs) has become commonplace in healthcare organizations. An EMR contains systematic documentation of health care delivered to a patient over a period of time. Each medical record includes a variety of information recorded by health care providers, such as progress notes, lab results, discharge summaries, medication, problem lists etc. Figure 1 shows an excerpt of a clinical note describing a patient visit. A large part of an EMR is free text that contains numerous medical terms. In an effort to standardize EMRs many ontologies have been developed that describe medical concepts and their associations, like MeSH, RxNorm and SNOMED-CT. Links between a term that appears in an EMR and ontological concepts can be created using structured data entry tools [4] or by parsing the text of clinical notes using NLP tools like cTAKES [26] or MetaMap [1].

“Patient here for follow up ***diabetes*** *care. Computer print out of* ***blood sugar*** *shows average of 201 with 1.7 tests. There is* ***hypoglycemia*** *about 2-3 times a week. Current Medications: -* ***CELLCEPT 500MG po twice daily - FROSEMIDE 80MG po daily”***

Figure 1: Excerpt of a clinical note

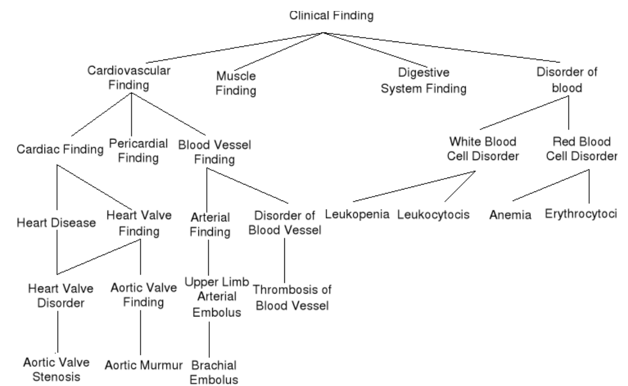


Figure 2: A subgraph of the SNOMED-CT ontology

Several types of relationships exist between these terms that are captured in the ontology structure. For example, in Figure 2 representing a small part of the SNOMED-CT ontology, a “heart valve finding” is a type of “cardiac finding”. Some medical terms can also be synonymous, e.g. “heart attack” and “myocardial infraction” represent the same ontology concept. Previous studies [18, 21] have shown that leveraging these concept associations can significantly improve the effectiveness of free-text search on EMRs. For instance, consider the query “aortic valve stenosis”. Intuitively, documents that do not contain the actual query terms, but contain similar concepts such as “thrombosis”, “embolus” or slightly more general ones such as “heart disease” or “heart valve finding” can be considered as relevant to the query. Thereby, documents are routinely viewed in medical literature as sets of concepts [10, 13, 17, 18, 21, 32] and several sophisticated measures have been proposed to quantify concept-concept similarity [3, 19, 20, 31].

Concept-based similarity search has proved to be beneficial in other domains as well. Lord et al. [14] compare genes and proteins based on the similarity of their functions that is captured in the Gene Ontology (GO), rather than based on their sequence similarity. The similarity between two gene representations can be used in order to predict gene functions or protein interactions [20].

Therefore, in this work we adopt the view of a document as a set of ontological concepts, as proposed in the biomedical literature, although we do recognize that also considering the free text that is not associated with concepts has the potential to further improve the retrieval quality. We study two query types: *relevance* and *similarity* queries, which are the most frequent in practice. As an ex-

*Anastasios Arvanitis is currently affiliated with the Vanderbilt Institute for Clinical and Translational Research (VICTR)

(c) 2014, Copyright is with the authors. Published in Proc. 17th International Conference on Extending Database Technology (EDBT), March 24-28, 2014, Athens, Greece: ISBN 978-3-89318065-3, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

ample of a relevance query, consider a clinical researcher searching an EMR database for patients that qualify to participate in a clinical trial for a new breast cancer treatment. Specific symptoms and past treatments for breast cancer, which can be represented as a set of concepts, may qualify the patient for the trial. Thus, the researcher wishes to find the most relevant patient records with respect to a set of medical concepts. As an example of a similarity query, a physician who wishes to be assisted in finding the right medical treatment for a patient can search a database of EMRs for patients with similar clinical indicators such as vital signs or medical history. Patient similarity assessment is also a very important task in the context of patient cohort identification for comparative effectiveness studies [28]. The key difference between the two aforementioned query types is that for relevance queries, we are not concerned with the concepts of the returned EMRs that are not related to the query concepts. In contrast, for similarity queries we care about the two-directional similarity between the query EMR and result EMRs.

In a relevance query (hereafter termed RDS for Relevant Document Search), the user specifies a query that consists of a set of concepts with the goal to retrieve the k most relevant documents (e.g., EMRs). In a similarity query (termed SDS for Similar Document Search), the user inputs a query document d_q with the goal to retrieve the k most similar ones. In order to evaluate each query type, we derive a ranking of documents that depends on the similarity between the query concepts (or query document respectively) with documents in the collection. This distance similarity is a function of the similarities of individual concepts. According to previous studies [17, 20], complex distance metrics do not clearly improve the correlation with the results provided by domain experts, whereas they affect efficiency [9]. Therefore, we adopt a simple distance metric represented as the shortest path connecting two concepts [23]. Further, for measuring the distance between two EMRs we use the document-document similarity measure proposed by Melton et al. [17], where the similarity between two documents is a function of the minimum semantic distance from each concept of the one document to a concept of the other and vice versa.

Each ontology may contain thousands of concepts that can be associated with several paths. For instance, SNOMED-CT ontology has a size of 300K concepts with up to 29 paths per concept; the UMLS metathesaurus contains over 2.9 million concepts. Thus, efficiency and scalability challenges arise. However, according to a recent work [9] and to the best of our knowledge, previous works on similarity metrics have serious limitations in terms of performance. Motivated by this, in this work we take a first step towards improving the efficiency and scalability of concept-based retrieval.

A baseline method is to precompute the distance of each concept with all documents in the collection and build an inverted file with a distance-based sorting for each postings list. After that, a top-k algorithm (e.g., Threshold Algorithm (TA) [7]) can be applied to find the k documents with the minimum distances. Although building such an index offline is feasible, applying TA for SDS queries is very inefficient. Due to the dual nature of the document-document distance, whenever TA examines a document, the postings lists for each concept in that document also need to be accessed and the distances from the query document determined. Since each posting list provides sequential access, in the worst case we would have to access all documents in each list (refer to Section 4.1 for details).

In an effort to address these shortcomings, we propose a uniform methodology to evaluate both RDS and SDS queries. Our query evaluation technique consists of two parts: (i) calculating query-document distances using the distances of the concepts they contain, and (ii) ranking documents based on their distance from the query. For the first part we propose an algorithm termed DRC that reduces the cost of query-document distance calculation from $O(n^2)$ to $O(nlogn)$ where n is the number of concepts in the query or examined document. Using a variation of the Radix Tree, our algorithm constructs a subgraph of the ontology that contains only the query and document concepts and uses this index to efficiently calculate the query-document distance.

For ranking documents, we present an algorithm for retrieving the k most relevant (resp. similar) documents, following a parallel branch and bound traversal of the ontology starting from the query concept nodes. Our processing strategy balances the costs associated with distance calculation and graph traversal by probing the DRC algorithm for a document only if it is highly likely that it will belong to the query results. For this purpose we propose an error estimation function for the semantic distances. Essentially, the error estimation measures how close the currently calculated distance of a document is to its actual distance based on the subset of query nodes already "covered" by the document. Thus, we avoid examining documents with large error estimate, such as those that "cover" only a few query concepts (recall that each document might contain hundreds or thousands of concepts in total).

An additional advantage of our method is that it does not require any distances precomputation. Our algorithm can integrate new documents into its computation on-the-fly; i.e., when a new patient arrives at the point-of-care, we can instantly add his or her EMR to our database. In contrast, TA would have to update every concept inverted index with the distance from the newly added EMR.

Contributions: The contributions of this paper are as follows:

- We define two important and challenging types of queries on concept-rich document corpuses, i.e., relevance and similarity.
- Based on a variation of a Radix tree, we propose an algorithm to reduce the cost of evaluating document-document distances from $O(n^2)$ to $O(nlogn)$.
- We present a threshold-based algorithm for efficiently identifying the k most relevant/similar documents.
- We provide a thorough experimental evaluation of our methods on a real EMR database. Our results show that our algorithms significantly outperform baseline strategies in terms of performance and scalability.

Outline: The rest of the paper is structured as follows. Section 2 gives a review of the related work and Section 3 provides some technical background and defines the semantic distances and semantic similarity queries. In Section 4, we present an algorithm for efficiently calculating document-query and document-document distances. We employ these distance methods for our algorithm presented in Section 5, which is used to evaluate both query types. Section 6 reports an experimental evaluation of our methods and Section 7 concludes the paper and discusses future work.

2. RELATED WORK

Fernandez et al. [9] provide a comprehensive survey of semantic-based searching approaches that have been proposed in the past. One classification of these approaches is based on the query model followed; some approaches utilize structured ontology query languages such as SPARQL, whereas others assume a keyword searching paradigm. In an effort to combine the flexibility of keyword search with the expressiveness of structured queries, Pound et al. [22] propose a hybrid approach where keyword queries are disambiguated to structured queries based on the vocabulary of the knowledge base. Our approach falls into the keyword search category. Additionally, we also address the case of semantic similarity queries where the input is a document instead of a set of keywords.

For keyword-based searching, ontology-based query expansion techniques have proved very beneficial for improving the retrieval quality [5]. For instance, Matos et al. [16] follow a concept-

oriented query expansion methodology to search biomedical publications by expanding gene concepts related to the query with related concepts such as protein and pathway names. Likewise, query expansion techniques have been applied by Lu et al. [15] on the PubMed database to significantly improve the results' precision.

In order to address some of the arising performance challenges, [2] and [29] propose to index together terms that appear frequently in common in user queries. Their approach requires additional space and does not consider the semantic distance between concepts, thus it cannot be used to rank documents based on their distances from the query terms, which is very useful if an ontology is available for the domain. Ding et al. [6] studied index optimization by grouping terms that appear in the subtree of a taxonomy. Concept-instance relationships were used to apply query substitutions, e.g., the query term "pet" may be replaced by "cat" or "dog". Compared to this work our focus is on query evaluation, rather than index maintenance. Further, our methods are not limited to concept-instance taxonomies but can be used in DAGs in general.

XOntoRank [8] considers keyword search against a corpus of XML documents with ontological references. XOntoRank returns subtrees that (i) either contain or (ii) are associated with the query terms through the ontological references. XOntoRank will not return any partial matches and it cannot be used on "bi-directional" distance functions such as the one proposed by Melton et al. [17]. Tao et al. address the problem of finding nearest neighbors in XML trees [27]. Given a query node q and a keyword w, a nearest keyword (NK) query returns the node that is the nearest to q among all nodes associated with w. The authors present an indexing scheme that allows answering NK queries efficiently. However, in our scenario the query keywords are not known apriori. Further, the proposed method cannot be applied for document-document similarity queries where bidirectional distance metrics apply.

In order to measure the semantic distance between ontology concepts, several metrics have been proposed; these metrics have been reviewed thoroughly [3, 19, 20, 31]. In [3] semantic measures are generally categorized as either: (i) *structured-based* or (ii) *information content-based*. Structured-based metrics exploit the geometrical structure of the ontology, such as the length of the shortest path connecting two concepts [23], or the depth of the concepts in the hierarchy [30], etc. Information content-based approaches capture the amount of information content shared by two concepts. Information content depends on the probability of occurrence of any descendant node of c [24]; i.e., it is proportional to the size of c's subtree including c. Resnik [24] and Lin [12] proposed different distances that measure the information content of the least common ancestor (LCA) of two nodes compared with the information content fully associated with the individual concepts. According to previous user studies with domain experts [17, 20], complicated distance metrics do not clearly improve the retrieval effectiveness, therefore in this work we adopt the shortest path distance metric as proposed by [23] for measuring concept-concept distance and the similarity metric proposed by [17] as a measure of similarity between documents that contain ontological concepts, since it has been shown to be effective for medical records.

3. PRELIMINARIES

In the following section we present the notation and the semantic distance measures that are used throughout this paper and we formally define the two query types targeted by our algorithms.

3.1 Ontologies and Radix Trees

Concept Ontology. Let $\mathcal{D}$ be a document corpus, where each document consists of terms derived from a vocabulary $\mathcal{V}$. Let $\mathcal{C} \subseteq \mathcal{V}$

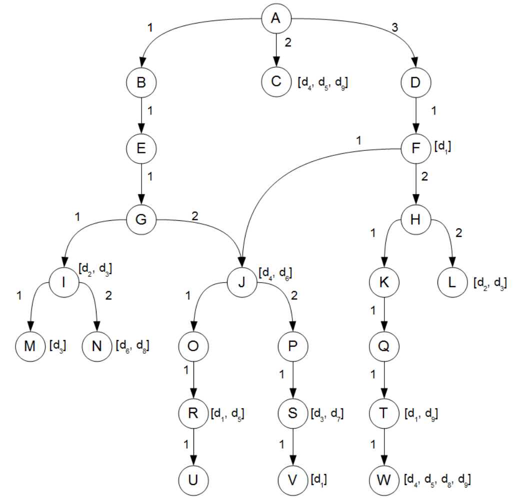


Figure 3: A labeled DAG representing an ontology

be the set of terms that are mapped to concepts derived from an ontology, where each $c_i \in \mathcal{C}$ is associated either with a single term or with several terms (synonyms) from $\mathcal{V}$.

In this work we will focus on domain ontologies that describe concept hierarchies, which is the type of ontology typically found in the medical domain. For instance, MeSH descriptors are organized in a hierarchical structure that allows searching at various levels of specificity, whereas the Gene Ontology is a Directed Acyclic Graph (DAG). In general a concept hierarchy is represented as a Directed Acyclic Graph (DAG) $G = \{C, E\}$, where C is the set of nodes representing concepts and E is a set of edges between concepts representing relationships such as *is-a*, *part-of*, etc.

In Figure 3 every path from the root to a concept $c_i \in \mathcal{C}$ is encoded using the Dewey Decimal Coding. Dewey is a prefix-based scheme where if a node c_j is a child of c_i and $l\{c_i\}$ is the label of a path from the root to c_i, then the path label of c_j is $l\{c_i\}.j$, where $j \in \{1, 2, \ldots, |children(c_i)|\}$.

Radix Trees. A trie index is a data structure used to store strings, where each path represents a unique string. In order to reduce the space consumption of tries, various techniques have been proposed including path compression or adaptive indexing of the internal nodes of the trie [11]. In case of path compression, nodes with only one child can be merged with their child, yielding a space-optimized index known as a Patricia or Radix Tree. In this paper we use a Radix index to represent a document as a set of concepts. Since our ontology is a DAG, each concept can be associated with several paths, therefore our index is not a tree but a DAG. The Radix DAG maintains the set of path labels to each concept in the document. Note that we only merge children that represent a concept in the document with parents that do not represent any concept in the document. Figure 4 shows the Radix tree for document $d = \{F, R, T, V\}$ using the ontology from Figure 3. The concepts contained in the document are denoted with squares. Nodes B, E, G, and J have been merged into one node with edge label 1.1.1.2. In Section 4, we describe a variation of the Radix DAG to speed up the calculation of distances between nodes in the ontology.

3.2 Semantic Distances

Let the semantic distance between $c_i, c_j \in \mathcal{C}$ be defined as $D(c_i, c_j)$. In this work we focus on the case where the semantic distance between two concepts is their shortest path distance, as proposed in [23] and evaluated on medical records in [3]. Note that we consider a path as valid, only if it passes through a common

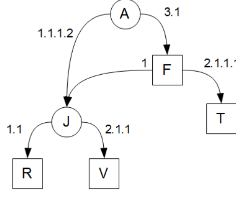


Figure 4: Indexing $d = \{F, R, T, V\}$ using a Radix DAG

ancestor of c_i, c_j. For instance, the shortest path distance $D(G, F)$ is not 2 but 5 because it has to pass through one of their common ancestors, A.

Next, we build on the concept-concept distance definition and define document-concept, document-query and document-document distances. First, we define the distance between a document $d \in \mathcal{D}$ and a concept $c \in \mathcal{C}$ as $D_{dc}(d, c)$[1]. $D_{dc}(d, c)$ is equal to the distance of c from the nearest concept in $\mathcal{C}$ that is associated with d:

$$D_{dc}(d, c) = \min_{c_i \in d} D(c_i, c) \quad (1)$$

Given a query consisting of a set of concepts $q = \{q_1, ...q_n\}$, define the distance of a document d from the query q as $D_{dq}(d, q)$[2,3]:

$$D_{dq}(d, q) = \sum_{i=1}^{n} D_{dc}(d, q_i) \quad (2)$$

We define the semantic distance between two documents as $D_{dd}(d_1, d_2)$. For this purpose we adopted the symmetric inter-patient distance function as proposed by Melton et al. [17], where we assumed that all concepts have equal weights. Thus, computing $D_{dd}(d_1, d_2)$ requires two calculations: one for deriving d_2 starting from d_1, and another for deriving d_1 starting from d_2; i.e., we calculate the distance of any concept in d_1 from the nearest concept found in d_2 and vice-versa, while normalizing by the number of concepts in the document:

$$D_{dd}(d_1, d_2) = \frac{\sum_{c_i \in d_1} D_{dc}(d_2, c_i)}{|C_1|} + \frac{\sum_{c_j \in d_2} D_{dc}(d_1, c_j)}{|C_2|} \quad (3)$$

where $|C_1|, |C_1|$ represent the number of concepts in documents d_1 and d_2 respectively. Note that unlike the document-query distance (Equation 2), Equation 3 is symmetric.

3.3 Similarity Queries

Now we introduce two important queries that arise when searching on a collection of documents that contain concepts derived from a domain ontology:

Definition 1 (Relevant Document Search - RDS)**.** *Given a set of query concepts $q = \{q_1, ..., q_n\}$, a document collection $\mathcal{D}$ and a positive integer k, determine the set $\mathcal{D}' \subset \mathcal{D}$, such that $|\mathcal{D}'| = k$ and $\forall d' \in \mathcal{D}', d \in \mathcal{D} - \mathcal{D}'$, $D_{dq}(d', q) \leq D_{dq}(d, q)$.*

Definition 2 (Similar Document Search - SDS)**.** *Given a query document d_q, a document collection $\mathcal{D}$ and a positive integer k, determine the set $\mathcal{D}' \subset \mathcal{D}$, such that $|\mathcal{D}'| = k$ and $\forall d' \in \mathcal{D}', d \in \mathcal{D} - \mathcal{D}'$, $D_{dd}(d', d_q) \leq D_{dd}(d, d_q)$.*

As mentioned in the introduction, RDS are suitable for exploratory queries, where the user is looking for documents relevant to a set of concepts. Recall the clinical researcher seeking qualifying candidates for a clinical trial. In this case, it is not important if a patient's record contains additional concepts not specified in the query, as long as the patient record is associated with some of the query concepts. On the other hand, SDS are appropriate for patient similarity queries, which have an inherent symmetric property.

[1] dc is used to denote that function D_{dc} measures document-concept distance as opposed to concept-concept distance for D, and is not related to variables d and c.

[2] dq denotes document-query distance.

[3] When merging the distances (scores) of documents produced by multiple queries (i.e. in query expansion) $D_{dq}(d, q_i)$ needs to be normalized with the size of q_i.

4. DISTANCE CALCULATION ALGORITHM

We now discuss document-query and document-document distance calculation. In Section 4.1 we describe the limitations of the baseline methods; in Section 4.2 we present a data structure, termed *D-Radix*, which we use in Section 4.3, where we propose our algorithm for calculating distances between documents efficiently.

4.1 Baseline Strategies

One approach for calculating document-query and document-document distances is to precompute all pairwise concept-concept distances. The space required to maintain these distances would be $O(|\mathcal{C}|^2)$. Even if it were possible to build this index, at query time, for each examined document we have to select the concepts with the minimum distances and calculate the distances based on Equations 2 or 3. Assuming n_q, n_d concepts in the query and the document, we must calculate $O(n_q n_d)$ distances for each examined document. Unfortunately, a typical EMR may contain thousands of concepts; in this case the naïve approach is not an option.

Another baseline method is to calculate offline the minimum distance of each concept from all documents in the collection based on Equation 1, which would require $O(|\mathcal{D}||\mathcal{C}|)$ space, where $|\mathcal{D}|$ is the size of the collection; $|\mathcal{D}|$ can be in the millions and $|\mathcal{C}|$ is 2.9 millions for the UMLS metathesaurus. Then we could build a postings list for each concept by sorting the (doc_id, distance) pairs in ascending order. After that, we could apply the threshold algorithm [7] to find the k documents with the minimum distances for the RDS query type. However applying a threshold algorithm for SDS queries poses several challenges. First, due to the dual nature of the document-document distance, whenever the threshold algorithm examines a document, the postings lists for each concept contained in that document also need to be accessed, and the distances from the query document determined. Since the postings lists provide sequential access, in the worst case for each list we should access $O(\mathcal{D})$ elements (documents). Further, the query document itself may contain thousands of concepts, thus we would have to traverse thousands of lists in parallel and maintain intermediate results in memory. Even worse, the lower bound threshold used by TA would assume that a partially examined document does not contain any concept other than those found so far, which does not allow for effective pruning in practice for the SDS query case.

4.2 The D-Radix Index

In order to address the scalability shortcomings of the baseline methods, in Section 4.3 we propose a more efficient algorithm for computing document-query and document-document distances. In contrast with baseline methods, our method does not require any precomputation of distances. Distance calculation is conducted at query time by utilizing a variation of the Radix that we introduce, termed *D-Radix DAG* (Distance-Radix DAG). Given a document d and a query q, a D-Radix DAG indexes all concepts that exist in either d or q. Additionally, each node contains the node's distance from the nearest node in d and q respectively. More formally:

Definition 3. *Given two sets of concepts d and q, a D-Radix DAG $T_{d,q}$ is a DAG $G(C[D_{dc}(d, c_i), D_{dc}(q, c_i)], E)$, where there is a node $c_i \in C$ for every common prefix found in $c \in d \cup q$ and if $\exists! e\{c_j, c_k\}$ and $c_j, c_k \notin d \cup q$ then c_j, c_k are merged into c_i. $D_{dc}(d, c_i)$ and $D_{dc}(q, c_i)$ represent the distances of node c_i from the nearest $c_d \in d$ and $c_q \in q$ respectively, as given in Equation 1.*

Example 1. *Figure 5(g) shows an example of a D-Radix DAG for a document $d = \{F, R, T, V\}$ and a query $q = \{I, L, U\}$. Document and query concepts are represented with squares and triangles respectively. Each node is associated with two numbers: the first number is the distance from the nearest document concept, and the second one is the distance from the nearest query concept.* □

Assuming that we have such an index structure available, then in the case of an RDS query, in order to calculate $D_{dq}(d, q)$ we can apply Equation 2 using the nearest document distance attached to each of the query nodes. Distances from the nearest query nodes are ignored. Hence, we get $D_{dq}(d, q) = D_{dc}(d, I) + D_{dc}(d, L) + D_{dc}(d, U) = 4 + 2 + 1 = 7$. Similarly, in the case of a SDS query with $d_q = \{I, L, U\}$, we can calculate $D_{dd}(d, d_q)$ based on Equation 3 where we use the distances from the nearest document node attached to each of the query nodes and the distances from the nearest query node attached to each of the document nodes.

Apart from having two distances associated with each node, a significant difference of the D-Radix index compared to the Radix Tree is that in a D-Radix two concept-nodes are not merged, even if there is no branch in any of the two nodes. In particular, we only merge children that represent a concept in the document or query with parents that do not represent any concept in the document or query. For instance, in a Radix Tree nodes R and U would have been merged; in the D-Radix they are kept separate.

4.3 The DRC Algorithm

DRC Overview. The DRC (D-Radix Construction) algorithm consists of a construction and a tuning phase. The construction phase builds a D-Radix DAG for indexing query and document concepts. All shortest distances are initially set to ∞ with one exception; if the inserted node is a document concept then the shortest distance from the document is set to 0, whereas if it is a query concept then the shortest distance from the query is set to 0. Once the index has been constructed, DRC propagates the shortest distance information by executing a bottom-up traversal followed by a top-down traversal. The distance information for a node is updated based on the minimum of (i) its distance, (ii) and the distance from its children or parents plus the length of the edge. We show that DRC calculates query-document and document-document distances in $O((|P_q| + |P_d|)\log(|P_q| + |P_d|))$ time, where P_q and P_d represent the number of paths leading to concepts from the query and the document respectively.

D-Radix DAG Construction. Constructing a D-Radix DAG is quite more complex compared to the construction of a Radix tree. The main reason is that, since we have to build a DAG rather than a tree, each step involves the *insertion of both a node and a path to that node*. Each inserted path has to be matched with edges that already appear in the index. Further, each partial match (path address) has to be checked against the set of nodes already inserted to the index, since it may define an alternative path to such a node. In that case, the insertion algorithm has to avoid adding a path twice, such that duplicate paths will not be propagated to the subtree. For the same reason, an already inserted edge may be split. We examine some of these cases based on a running example presented next. The details of inserting a path address are explained in the next paragraph. Algorithm 1 shows the complete pseudocode of DRC for the RDS query case. The SDS case is similar except that (i) we use the distances from both document and query and (ii) the distance is calculated based on Equation 3.

Path Insertion. Insertion for the D-Radix DAG is similar to that of Radix trees, except that a path may define a node already contained in $T_{d,q}$. $T_{d,q}$ is a hash of nodes, where each node contains zero or more pointers to other nodes in the hash; these pointers represent child edges. $T_{d,q}$ also contains a pointer to the root node, which

Algorithm 1: DRC Algorithm for RDS Queries

Input: d: a document, q: a query
Output: D_{dq}: document-query distance
Variables: $T_{d,q}$: a D-Radix DAG on d, q,
P_d: lexicographically sorted list of Dewey addresses for each $c_j \in d$,
P_q: lexicographically sorted list of Dewey addresses for each $q_i \in q$,
$l\{n_d\}$: next Dewey address from P_d, $l\{n_q\}$: next Dewey address from P_q

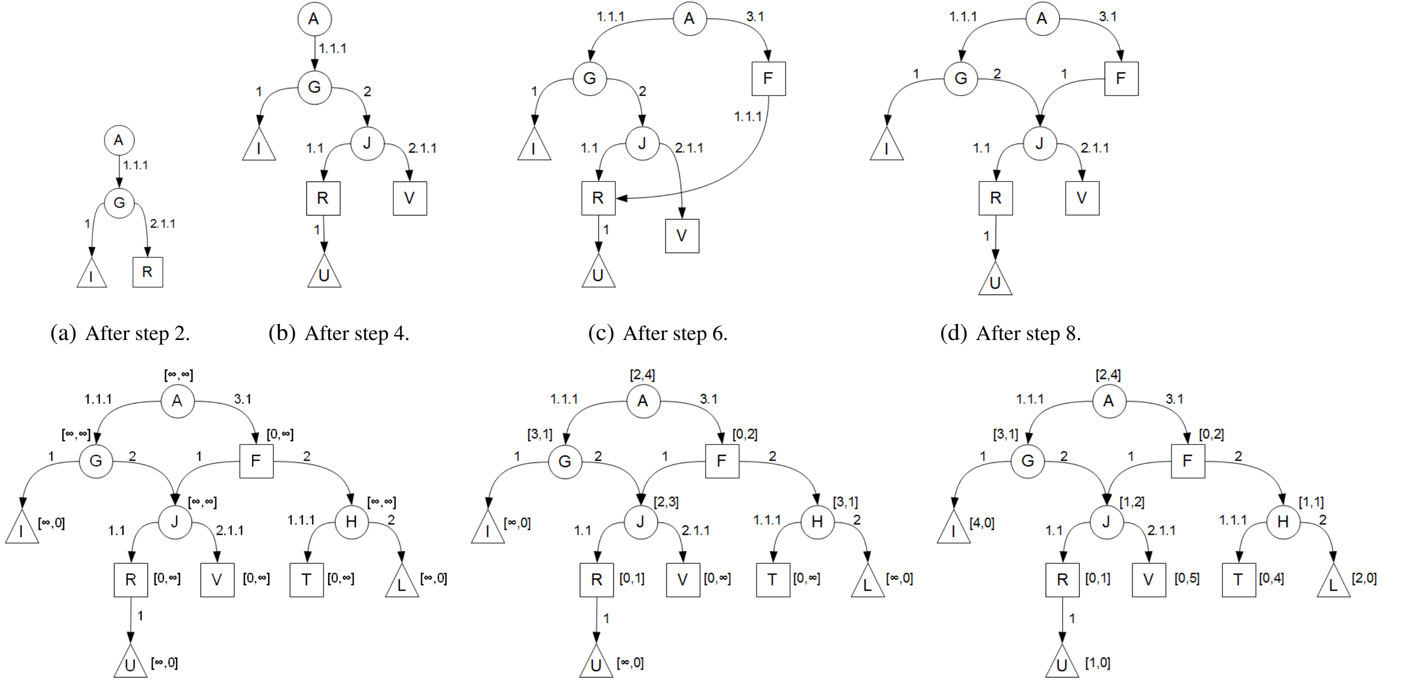
```
1  begin
2      //Index Construction Phase;
3      retrieve P_d; retrieve and sort P_q;
4      insert(root) into T_{d,q};
5      l{n_d} := P_d.first; l{n_q} := P_q.first;
6      while (P_d.hasNext or P_q.hasNext) do
7          if (l{n_d} ≤ l{n_q}) then
8              c_n := n_d;
9              InsertPath(l{n_d}, T_{d,q});
10             l{n_d} := P_d.next;
11         else
12             c_n = n_q;
13             InsertPath(l{n_q}, T_{d,q});
14             l{n_q} := P_q.next;
15         if c_n ∈ q then
16             D_q(q, c_n) := 0;
17         else
18             D_q(q, c_n) := ∞;
19     //Tuning Phase;
20     //Traverse T_{d,q} bottom-up;
21     foreach c_j ∈ T_{d,q} do
22         D_q(d, c_j) := min{D_q(d, c_j), min_{c_k}{D_q(d, c_k)+D(c_j, c_k)}};
23         for all c_k where c_k is a child of c_j;
24     //Traverse T_{d,q} top-down;
25     foreach c_j ∈ T_{d,q} do
26         D_q(d, c_j) := min{D_q(d, c_j), min_{c_k}{D_q(d, c_k)+D(c_k, c_j)}};
27         for all c_k where c_k is a parent of c_j;
28         if (c_j ∈ q) then
29             D_{dq}(d, q) := D_{dq}(d, q) + D_{query}(d, c_j);
30     return D_{dq}(d, q);
```

is created during initialization of Algorithm 1. Thus, the insertion algorithm starts at the root and traverses $T_{d,q}$ until all pointers have been updated correctly. Pseudocode for path insertion is given in Function InsertPath.

Let n_i be the node to be inserted with path address $l\{n_i\}$. Execution begins by initializing the variables: $u = \epsilon$, $v = l\{n_i\}$, and $c_n = T_{d,q}.root$ (lines 2-4). Function InsertPath maintains the invariant that u is a common prefix of $l\{n_i\}$, and v is the suffix of $l\{n_i\}$ not matched by u. c_n is used to keep track of the current node in the traversal (line 6); u defines a path to node c_n in $T_{d,q}$.

While variable v is not equal to ϵ, $l\{n_i\}$ has not been fully inserted into $T_{d,q}$ (line 5). Hence, we examine each child edge of c_n, seeking an edge that shares a common prefix with v (lines 5-10). Only one such edge may exist. If no such edge exists, then n_i is a child of c_n with edge label v (lines 11-13). Otherwise, v contains a prefix exactly equal to m, or v shares a prefix with m that is not equal to ϵ or m. If v contains a prefix exactly equal to m, then u, v, and c_n are updated to reflect traversal from c_n to n (lines 14-17). This is accomplished by concatenating m to u, removing the prefix of m from v, and setting $c_n = n$.

If v shares a prefix with m that is not equal to ϵ or m, then the edge between c_n and n must be modified to include the LCA of $l\{n_i\}$ and $T_{d,q}$. Thus, the child edge from c_n to m is removed (line 19). Let variable *lcp* be equal to the Longest Common Prefix (LCP) of v and m (line 20). The path defining the LCA is u concatenated with *lcp* (line 21). Once we have the Dewey address of the LCA, we look up its corresponding node identifier (line 22). Next, we add a child edge from c_n to the LCA with *lcp* as the edge label (line 23). Then an edge is added from the LCA to node n



(a) After step 2. (b) After step 4. (c) After step 6. (d) After step 8.

(e) The D-Radix DAG with the initial distances. (f) The D-Radix DAG after the bottom-up traversal. (g) The D-Radix DAG after the top-down traversal.

Figure 5: Running example of the DRC algorithm

with edge label $m.substring(lcp.length + 1)$ (line 24); plus one removes the "." trailing the lcp string. If the LCA is not equal to n_i, then we also add a child edge from the LCA to n_i with edge label $v.substring(lcp.length + 1)$ (lines 25-26); again, plus one removes the "." trailing the lcp string.

Example 2. *Consider* $d = \{F, R, T, V\}$ *from Figure 3 and* $q = \{I, L, U\}$. P_d *and* P_q *are listed in Table 1. First, DRC retrieves the lists and inserts a root node into* $T_{d,q}$. *In the first step, DRC processes* I *with address* 1.1.1.1. *A node for* I *is created along with an edge from* A *to* I. *Next, DRC processes* R *with address* 1.1.1.2.1.1. *After matching* 1.1.1.2.1.1 *with* 1.1.1.1, *DRC splits edge* 1.1.1.1 *into* 1.1.1 *(the common prefix) and* 1. *Thus, DRC will also insert node* G *with address* 1.1.1, *and insert the remaining path address to* R *as an edge* 2.1.1. *The resulting D-Radix DAG is shown in Figure 5(a). In the third step, DRC processes* U *with* 1.1.1.2.1.1.1, *which subsumes* 1.1.1.2.1.1 *(node* R*), thus it inserts an extra edge from* R *to* U *as well as node* U.

Node	Labels	Step #
P_d		
R	1.1.1.2.1.1	2
V	1.1.1.2.2.1.1	4
F	3.1	5
R	3.1.1.1.1	6
V	3.1.1.2.1.1	8
T	3.1.2.1.1.1	9
P_q		
I	1.1.1.1	1
U	1.1.1.2.1.1.1	3
U	3.1.1.1.1.1	7
L	3.1.2.2	10

Table 1: Dewey path address lists for DRC

In the fourth step, DRC has to process node V *with* 1.1.1.2.2.1.1. *This step splits the edge between* G *and* R *with* 1.1.1.2 *(node* J*) as the LCA;* V *(*2.1.1*) is added with* J *as the parent (Figure 5(b)). In the fifth step, node* F *with address* 3.1 *is added with the root as the parent. In the sixth step, DRC processes node* R *with address* 3.1.1.1.1. *Node* R *already exists in* $T_{d,q}$, *but an edge between* F *and* R *is missing. Thus, DRC adds this edge as shown in Figure 5(c). The seventh step processes node* U *with address* 3.1.1.1.1.1. *This address is completely matched in* $T_{d,q}$, *thus* $T_{d,q}$ *is not modified. The eighth step processes node* V *with address* 3.1.1.2.1.1. *By matching this address, the DRC algorithm decides that the edge between* F *and* R *has to be split into addresses* 3.1.1, 1.1 *and* 2.1.1. *DRC performs a lookup and finds out that* 3.1.1 *corresponds to node* J *which already appears in* $T_{d,q}$. *Therefore the edge between* F *and* R *is modified to be between* F *and* J *but no new node will be created. Also, DRC finds out that address* 2.1.1 *(node* V*) already appears in* $T_{d,q}$. *The result of this step is illustrated in Figure 5(d). The ninth step adds node* T *as a child of node* F. *Finally, in the tenth step DRC processes node* L *with address* 3.1.2.2. *By matching this address, DRC has to split the edge between* F *and* T *and insert* 3.1.2 *(node* H*) as a parent of* T *and* L. *The result of the tenth step with the initial distances assigned to each node is illustrated in Figure 5(e).* □

Distances Tuning. Obtaining the shortest distance for each node requires a bottom-up traversal followed by a top-down traversal. Let D_q symbolize the distance from the nearest query node; then the distance at each node is recursively updated as:

$$D_q(d, c_j) = \min\{D_q(d, c_j), \min_{c_k}\{D_q(d, c_k) + D(c_k, c_j)\}\} \quad (4)$$

$\forall c_k$ where c_k is a child of c_j for the bottom up traversal, and a parent of c_j for the top-down traversal. A similar formula is used for computing distances from document nodes for an SDS query.

Correctness. Let u, v be two nodes of $T_{d,q}$. One of the following conditions holds: (i) u is a descendant of v, or (ii) u is an ancestor of v, or (iii) u and v share a common ancestor. Executing a bottom-up and a top-down traversal always propagates the correct distance information for u and v in the first two cases. Further, recall from Section 3.1 that valid paths must pass through a common ancestor. Based on the order of the traversals all distance information is only propagated along valid paths that contain the common ancestor of two nodes. Since $T_{d,q}$ has only one root, the common ancestor of any u, v is always visited. Therefore in the third case u and v will always have the correct distance information.

Figure 5(e) shows the D-Radix DAG from Example 2 after the completion of the construction phase of DRC. The bottom-up traver-

Function InsertPath

```
Input: n_i: the node to insert, l{n_i}: Dewey address for n_i,
T_{d,q}: a D-Radix DAG on d, q
Variables: u: common prefix of l{n_i}, v: suffix of l{n_i} not contained by u,
c_n: node from T_{d,q} defined by address u
1  begin
2    u := ε;
3    v := l{n_i};
4    c_n := T_{d,q}.root;
5    while v ≠ ε do
6      m := null; n := null;
7      foreach edge e_j pointing to node n_child from c_n do
8        if e_j.sharesPrefixWith(v) then
9          m := e_j;
10         n := n_child;
11     if m is null then
12       c_n.addChild(n_i, v);
13       v := ε;
14     else if v.containsPrefix(m) then
15       u := u.concat("." + m);
16       v := v.substring(m.length + 1);
17       c_n := n;
18     else
19       c_n.removeChildEdge(m);
20       lcp := LCP(v, m);
21       LCA_path := u.concat("." + lcp);
22       LCA_node := FindNodeByDewey(LCA_path);
23       c_n.addChild(LCA_node, lcp);
24       LCA_node.addChild(n, m.substring(lcp.length + 1));
25       if LCA_node ≠ n_i then
26         LCA_node.addChild(n_i, v.substring(lcp.length + 1));
27       v := ε;
```

sal propagates these distances up to the root, as shown in Figure 5(f). The top-down traversal propagates these distances down to the leaves of the D-Radix DAG, as illustrated in Figure 5(g).

After finishing both the construction and tuning steps, the final distance is computed using Equation 2 or 3, depending on the query type. Distances are progressively calculated during the top-down traversal as each document and query node is visited.

Complexity Analysis. Let P_q and P_d be the sets of path addresses to concepts of the query and the document respectively. The D-Radix constructed by DRC will contain $O(|P_q|+|P_d|)$ nodes. The construction phase loops over each path address. Since the height of the D-Radix index is $\log(|P_q|+|P_d|)$, the construction phase takes $O((|P_q|+|P_d|)\log(|P_q|+|P_d|))$ time. The traversals required for the tuning phase are completed in $O(|P_q|+|P_d|)$. Hence, the total complexity of DRC is $O((|P_q|+|P_d|)\log(|P_q|+|P_d|))$.

5. K-NEAREST DOCUMENT SEARCH ALGORITHM

This section presents our algorithm for evaluating RDS and SDS queries, termed kNDS (k-Nearest Document Search Algorithm). We first present the challenges that our problem poses, and a general overview of the proposed algorithm. Next, we provide details regarding the algorithm's execution.

5.1 Baseline Methods

A naïve approach to evaluate RDS or SDS queries is to calculate the distances of all documents in the collection from the query (or query document); and then select k with the minimum distances. Clearly, this is prohibitively expensive and inefficient. Ideally, we would prefer to maintain a sorted list of documents ordered by their semantic distances from the query, such that unexamined documents would always have a larger distance, thus we could prune those documents. However, as we discussed in Section 4.1, this threshold-based approach would require precomputing the distance of each document in the collection from any concept in the ontology, i.e., $O(|\mathcal{D}||\mathcal{C}|)$ space, and it would not be useful for the SDS query due to the dual nature of the semantic distance of Equation 3.

5.2 Challenges and tradeoffs

In order to overcome this problem, we propose a solution that does not require any distance precomputation but exploits a threshold-based technique to prune irrelevant documents. Our algorithm, termed kNDS, is based on the idea of query expansion. In particular, we start our search by considering documents that contain the exact query terms and then we follow a breadth-first traversal of the ontology graph to retrieve documents that contain similar concepts. Our goal is the following: if at some point during the graph traversal we already have found k documents with *final* distances (i.e., we have covered all query nodes and the total distance of each document from the query has been determined), then we can prune all documents for which we have not calculated their exact distances, as long as their lower bound is greater than of the k already examined ones. Before delving into the details of kNDS we first explain how we calculate partial and lower bound distances.

Iteration $l+1$, $l \geq 0$ examines concepts having distance l from a query concept. Assume that during iteration $l+1$ for the breadth-first search starting from query node q_i we traverse a concept node c_j, such that c_j is contained in document d and c_j is the first concept for document d seen for query node q_i. Then, we know that $D_{dc}(d, q_i) = D(c_j, q_i) = l$. If no concept for document d is found, then the lower bound for the distance $D_{dc}(d, q_i)$, termed $D^-_{dc}(d, q_i)$ is equal to $l+1$.

Example 3. *Consider a query $q = \{I, L, U\}$ and document $d = \{F, R, T, V\}$. Then, starting a parallel breadth-first search from each query concept in Figure 3, in the second iteration we examine nodes: G, M, N, R, H. Only R is contained in d, thus the actual distance $D_{dc}\{d, U\}$ is 1. For the rest of the query nodes it holds that: $D^-_{dc}\{d, I\} = D^-_{dc}\{d, L\} = 2$.* □

Let $M_d(q_i, d)$ be a hash that maps a node q_i to a distance value l if during the breadth-first search starting from query node q_i a concept that belongs to document d has been found with distance l from q_i. For instance in our previous example, during the second step of the traversal, $M_d(q_i, d)$ would contain the element $\{U, 1\}$. Note that values for each key in M_d are only set once so that M_d maintains the minimum distance from each q_i. Then, we define the partial (currently calculated) $D^{partial}_{dq}(d, q)$ and the lower bound distance $D^-_{dq}(d, q)$ between a document and a query (for RDS) as:

$$D^{partial}_{dq}(d, q) = \sum_{q_i \in M_d} M_d(q_i, d) \tag{5}$$

$$D^-_{dq}(d, q) = \sum_{q_i \in M_d} M_d(q_i, d) + \sum_{q_i \notin M_d} (l+1) \tag{6}$$

Let $M'_d(c_i, q)$ be a hash for document d with a value for c_i if and only if a concept for document d has been found during any of the breadth first searches for any query node in q; values for each key in M'_d are only set once, hence M'_d contains the minimum distances. Then the partial and lower bound distances for SDS are:

$$D^{partial}_{dd}(d_1, d_2) = \frac{D^{partial}_{dq}(d_2, d_1)}{|C_1|} + \frac{\sum_{c_i \in M'_{d_2}} M'_{d_2}(c_i, d_1)}{|C_2|} \tag{7}$$

$$D^-_{dd}(d_1, d_2) = \frac{D^-_{dq}(d_2, d_1)}{|C_1|} + \frac{\sum_{c_i \in M'_{d_2}} M'_{d_2}(c_i, d_1) + \sum_{c_i \notin M'_{d_2}} (l+1)}{|C_2|} \tag{8}$$

kNDS proceeds in a branch-and-bound fashion. Starting from the query nodes, it performs a breadth-first traversal of the ontology, retrieves documents that contain the visited concept nodes and iteratively updates their partial distances using Equations 5 or 7, depending on the query type. Similarly, it calculates a lower bound distance based on Equations 6 and 8. Then, we can check whether some of the documents can be pruned by comparing their lower bound distance with the partial distances of already examined ones.

The challenge is that during graph traversal, it is highly unlikely to discover documents that would cover all query nodes early during the algorithm execution, especially if the query (or query document) contains many terms. Moreover, in general we would like to avoid calling the DRC algorithm to calculate actual (final) distances because this is an expensive operation. In fact there is a tradeoff between the distance calculation cost (DRC execution) and graph traversal. If we execute DRC too soon we may waste time to compute the distance of irrelevant documents. On the other hand if we wait until finding several concepts of a document before running DRC, this may explode the ontology traversal cost, since non-visited nodes are kept in a priority queue in memory.

Then when would it be preferable to calculate the actual distance of a document from the query in order to prune some documents? To answer this question, kNDS algorithm maintains an error estimate that compares the partial distance of the document with the document's lower bound distance based on the following formula:

$$\epsilon_d = 1 - \frac{D_{dq}^{partial}(d,q)}{D_{dq}^{-}(d,q)} \quad (9)$$

kNDS compares the calculated error with an error threshold ϵ_θ. If the error estimate is lower (i.e., the partial distance yields quite a good estimate of the actual distance), then kNDS probes DRC in order to compute the actual distance. Otherwise, kNDS continues the graph traversal until having a better distance estimation.

Note that determining a good error threshold ϵ_θ generally depends on several factors such as: (i) the query type (RDS or SDS), (ii) the query size, (iii) the ontology characteristics (fanout, average number of paths to each concept node, etc.), and (iv) the document collection statistics (e.g., if a document contains concepts that are close to each other in the ontology, the average number of concepts per document, etc.). Thereby, we use the error threshold as an input parameter to the algorithm. We include a detailed sensitivity analysis on this parameter in the experimental section (Section 6).

5.3 The kNDS Algorithm

We first describe the data structures used followed by the details of the algorithm execution.

Data Structures. kNDS maintains the following data structures:

- A queue, denoted as E_c, used to perform breadth-first traversal of the ontology, where each element contains a concept node and the corresponding query node from which the traversal originated, denoted as $\{c_j, q_i\}$.
- A list of documents, denoted as L_d, where each element contains a document d and its partial and lower distances.
- A binary heap H_k of the top-k most similar documents found so far and their respective distances from the query. This heap contains documents for which their distances have been determined; it is ordered in reverse $D_{dq}(d,q)$.
- A hashset S_d of documents that have been examined.

We also assume the availability of an index that allows us to traverse the ontology efficiently (this would typically fit in memory) as well as an inverted and a forward index that map concepts to documents and vice-versa (memory or disk-based).

Algorithm Execution. The algorithm execution consists of two steps: breadth-first expansion and distance calculation. In the following we provide details for each step. The complete pseudocode of the kNDS algorithm is given in Algorithm 2. Additional engineering optimizations are described at the end of this section.

Algorithm 2: kNDS Algorithm

Input: $\mathcal{D}$: a document collection, q: a query, G: a concept ontology, k: a positive integer, ϵ_θ: a distance error threshold, $D(c_j)$: inverted index on c_j
Output: the k most similar documents to q
Variables: E_c: nodes' queue, L_d: a list of documents, H_k: a heap of the k most similar documents to q, S_d: a hash of documents that have been examined, D_k^+: the distance of the k-th element in H_k from q, D^-: the lower bound of the distance from q of the first element in L_d

1 **begin**
2 $L_d := \emptyset$; $H_k := \emptyset$; $S_d := \emptyset$; $D^- := 0$; $D_k^+ := \infty$;
3 **foreach** $q_i \in q$ **do**
4 $E_c.push(q_i, q_i)$;
5 E_c.push($\emptyset, \emptyset$);
6 **while** *($D^- < D_k^+$ and $E_c \neq \emptyset$)* **do**
7 **while** *($E_c.head() \neq \{\emptyset, \emptyset\}$)* **do**
8 $E_c.pop() \rightarrow \{c_j, q_i\}$;
9 **foreach** c_l: $\exists E(c_l, c_j) \in G$ *or* $\exists E(c_j, c_l) \in G$ **do**
10 E_c.push(c_l, q_i);
11 **foreach** $d \in D(c_j)$ *and* $d \notin S_d$ **do**
12 calculate($D_{dq}^-(d,q)$);
13 L_d.push($d, D_{dq}^-(d,q)$);
14 $E_c.pop()$; E_c.push($\emptyset, \emptyset$);
15 sort(L_d);
16 calculateError(L_d.first) $\rightarrow \epsilon_d$;
17 **while** *($\epsilon_d \leq \epsilon_\theta$ and $L_d \neq \emptyset$)* **do**
18 L_d.removeFirst() $\rightarrow d$;
19 calculate($D_{dq}(d,q)$);
20 S_d.push(d);
21 **if** $|H_k| < k$ **then**
22 H_k.push($d, D_{dq}(d,q)$);
23 H_k.find-min() $\rightarrow D_k^+$;
24 **else if** $D_{dq}(d,q) < D_k^+$ **then**
25 H_k.delete-min();
26 H_k.push($d, D_{dq}(d,q)$);
27 H_k.find-min() $\rightarrow D_k^+$;
28 L_d.first() $\rightarrow D^-$;
29 calculateError(L_d.first) $\rightarrow \epsilon_d$;
30 **foreach** $d_i \in H_k$ **do**
31 **if** $D_{dq}(d_i, q) \leq D^-$ **then**
32 output d_i;
33 **foreach** $d_i \in H_k$ **do**
34 output d_i;

Breadth-first Expansion. Initially all data structures are empty (line 2). E_c is initiated by inserting each $q_i \in q$ into E_c (line 4). kNDS performs multiple breadth-first traversals of the ontology starting from each query node. For each node c_j in the queue, we maintain its distance to the query concept that was the source of c_j. We use this distance to compute the two document distances described above. In each iteration the following operations are conducted:

- The breadth-first traversal proceeds to the next depth level, e.g., at iteration l kNDS processes all nodes with distance (depth) l from any of the query nodes. Note that we enforce the traversal to follow only valid paths in the ontology (passing through a common ancestor), as we discussed in Section 3.1.
- For each traversed node c_j, the node's neighbors are inserted to E_c (lines 9-10). E_c maintains a natural ordering of elements via insertion. In order to distinguish elements that have different depths, we include a null insertion $\{\emptyset, \emptyset\}$ after finishing each iteration (lines 5 and 14). Note that a node can be visited several times during the ontology traversal. Labeling a visited node is more expensive, since it would require to maintain a large structure with all (c_j, q_i) already visited.

Iteration	S_d	L_d	E_c	H_k	D^-	D_k^+
0	$\emptyset$	$\emptyset$	$\{F,F\}\{I,I\}\{\emptyset,\emptyset\}$	$\emptyset$	0	∞
0	$\emptyset$	$\{d_1,1\}\{d_2,1\}\{d_3,1\}$	$\{D,F\}\{H,F\}\{J,F\}\{G,I\}\{M,I\}\{N,I\}\{\emptyset,\emptyset\}$	$\emptyset$	0	∞
1	$\{d_1,d_2\}$	$\{d_3,1\}$	$\{D,F\}\{H,F\}\{J,F\}\{G,I\}\{M,I\}\{N,I\}\{\emptyset,\emptyset\}$	$\{d_2,2\}\{d_1,4\}$	1	4
1	$\{d_1,d_2\}$	$\{d_3,2\}\{d_6,2\}\{d_4,3\}$	$\{A,F\}\{K,F\}\{L,F\}\{O,F\}\{P,F\}\{E,I\}\{J,I\}\{\emptyset,\emptyset\}$	$\{d_2,2\}\{d_1,4\}$	1	4
END	$\{d_1,d_2,d_3,d_6\}$	$\{d_4,3\}$	$\{A,F\}\{K,F\}\{L,F\}\{O,F\}\{P,F\}\{E,I\}\{J,I\}\{\emptyset,\emptyset\}$	$\{d_2,2\}\{d_3,2\}$	3	2

Table 2: Running example of the kNDS algorithm

- For each traversed node c_j, all documents that contain c_j and have not been examined before (i.e., they are not found in S_d) are inserted to L_d (lines 11-13). If the document already exists in L_d, its lower bound distance as well as the current distance are updated (line 12). For each document, we also maintain the query nodes from which the search originated, so that we do not increase a distance if the document is associated with a second concept that originated from a covered query node.

Distance Calculation. After completing a breadth-first expansion, kNDS proceeds to analyze collected documents. First, it sorts L_d by increasing $D^-_{dq}(d,q)$ (line 15). Then, it calculates the estimation error (ϵ_d) for the first element (line 16). If $\epsilon_d \leq \epsilon_\theta$, where ϵ_θ is the error threshold, then the document must be analyzed, i.e., the document is removed from L_d, added to S_d and the actual distance is calculated by calling upon DRC (lines 17-20). Otherwise, kNDS proceeds to the next breadth-first iteration. Each document for which the actual distance has been determined is compared with the documents contained in a min-heap H_k, where H_k contains the k documents with the currently lowest actual distances. If the new document's distance is lower than the distance of the k-th element of H_k (or $|H_k| < k$), then the new document replaces the last element of H_k (or it is inserted into H_k respectively) (lines 22-26). Documents from L_d are examined iteratively until either L_d is empty or $\epsilon_d > \epsilon_\theta$ (line 17) or D^- is higher than the distance of the k-th element in H_k (line 6); in the last case kNDS terminates and the contents of H_k are returned as the query results (lines 33-34).

Example 4. *Following Example 2, assume an RDS query with $q = \{F, I\}$, $\epsilon_\theta = 1$, and $k = 2$ on the document collection and the ontology depicted in Figure 3. Table 2 shows the contents of various data structures during the execution of kNDS. Every two rows represent one iteration of the main while loop. The first row shows the contents at the start of the respective iteration and the second row shows the contents after retrieving the neighbors for each node in E_c and updating L_d. kNDS begins by adding the query nodes to E_c, and initializing $D^- = 0$, $D_k^+ = \infty$ (row 1). The algorithm then pushes each neighbor of F and I into E_c, and initializes L_d (row 2). The top-2 documents (d_1 and d_2) are then analyzed and added to H_k and S_d; D^- is set to 1 using the lower bound distance of d_3, and D_k^+ is set to 4 using the actual distance of d_1 (row 3). Since $D^- < D_k^+$, kNDS continues to the next iteration. Next, kNDS processes E_c adding the respective neighbors and updates L_d (row 4). Note, node J has now been added twice to E_c; once for F and once for I. Also note that although G is a parent of J, the BFS for query node F did not push $\{G, F\}$ to E_c; this is due to the valid path rules discussed in Section 3.1. kNDS then examines L_d and sets D^- to 3 using the lower bound of d_4, and D_k^+ to 2 using the final distance of d_3. Since $D^- \geq D_k^+$, kNDS terminates and outputs the contents of H_k as the top-2 results.*

Correctness. We will show that kNDS algorithm always outputs the top-k documents with the lowest distances from the query. Any document $d \in \mathcal{D}$ can be in one of the following 3 states: (i) already examined, i.e. contained in S_d, (ii) partially visited, i.e. contained in L_d, or (iii) not visited yet. Recall that kNDS maintains a min-heap H_k with the documents found so far that have the lowest distances. Whenever the final distance of a new document is calculated (line 19), i.e. the documents moves from state (ii) to (i), if $D_{dq}(d,q) < D_k^+$ then the new document replaces the old one in H_k (lines 24-26). This step ensures that $\nexists d \in S_d - H_k : D_{dq}(d,q) < D_k^+$. Now recall that partially visited documents are kept in L_d sorted on their lower distance (lines 11-13). kNDS continues as long as the first document in L_d has $D^- < D_k^+$ (line 6). When kNDS terminates, since L_d is sorted in ascending lower distance, all documents in L_d will have $D^- > D_k^+$ so they can be safely discarded. Finally, let l be the distance of the concepts examined in the breadth-first traversal at the current iteration from q. Then for RDS it holds that $\forall d \in L_d$, $d' \in \mathcal{D} - \{S_d \cup L_d\}$, $D^- \leq D^-_{dq}(d,q) \leq |Q|(l+1) \leq D^-_{dq}(d',q) \leq D_{dq}(d',q)$. A similar inequality holds also for the SDS query based on Equations 7 and 8. Therefore, when kNDS terminates it holds $D_k^+ \leq D^- \leq D_{dq}(d',q)$. In other words, any not visited document will always have a greater distance than those already examined.

Complexity Analysis. The worst case for the cost of kNDS happens when the number of iterations (line 6) is maximized or all documents in the corpus have to be examined (each document's distance is computed). Each iteration performs a breadth-first step, so the maximum number of iterations is equal to the longest path in the ontology L. Normally $|\mathcal{D}| > L$. Further, based on our analysis in Section 4.3, each distance calculation has a $O((|P_q| + |P_d|)\log(|P_q| + |P_d|))$ cost where P_q and P_d represent the sets of path addresses to concepts of the query and the document respectively. Therefore, assuming $|\mathcal{D}|$ iterations in the worst case the complexity of kNDS will be $O(|\mathcal{D}|(|P_q|+|P_d|)\log(|P_q|+|P_d|))$. Note that the cost for the heap reorganization in each iteration (line 15) is dominated by the cost of the distance calculation, since in practice the number of documents kept in the heap is $|\mathcal{D}'| << |\mathcal{D}|$.

Optimizations. In order to speed up the algorithm execution we also apply the following optimizations:

- When updating the distances of a document in L_d, if the calculated lower distance grows larger than that of the k-th element in H_k, then the document is removed from L_d.
- Since the size of L_d might grow large, instead of sorting L_d after each iteration we build a partial sorted heap H_d that contains $n \geq k+1$ documents ordered by $D^-_{dq}(d,q)$. The reason for enforcing $n \geq k+1$ is that in the most favorable scenario, the first k elements in the heap will be the final query results. In that case, we need to know the lower bound distance of the next element in order to check the termination condition.
- As we discussed before, for each document d we maintain the number of distinct query concepts or their neighbors for which we have found that d is associated with. If all query nodes are found already then we can use the current distance instead of applying the DRC algorithm.
- kNDS can progressively output results from H_k during the algorithm execution. If the distance of a document d in H_k is lower than or equal to the lower bound distance of the first element in L_d (or H_d), then d must be in the top-k most similar documents and can be reported as a query result (lines 30-32).

6. EXPERIMENTAL EVALUATION

6.1 Experimental Setting

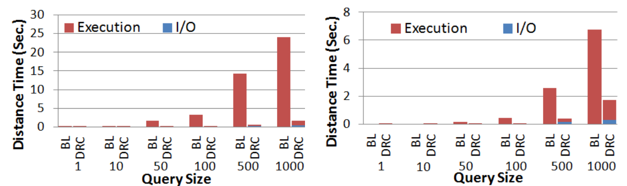
Dataset. Experiments were conducted using a subset of the MIMIC II clinical database [25]. This subset consists of 42,144 clinical notes over 983 patients. There are four different types of notes available for each patient: (i) MD Notes (816 documents), (ii)

	Patient	Radiology
Total Documents	983	12,373
Total Concepts	16,811	8,629
Avg. Tokens/Document	8,184	273.7
Avg. Concepts/Document	706.6	125.3

Table 3: Document Corpus Statistics

Parameter	Range
Number of Results (k)	3, 5, **10**, 50, 100
Query Size (Q)	1, **3**, 5, 10

Table 4: Values for parameters; default values shown in bold



(a) Time vs. n_q for SDS (PATIENT) (b) Time vs. n_q for SDS (RADIO)

Figure 6: Distance Calculation Time vs. Query Size n_q

Nursing Notes (28,133 documents), (iii) Radiology Reports (12,373 documents), and (iv) Discharge Summaries (822 documents).

For our experiments, we used two different document collections. Our purpose was to examine the performance of our methods on data sets with different characteristics in terms of size, average number of concepts contained per document, total number of distinct concepts in the collection, etc. The first document collection that we used consists of the Radiology Reports documents; we refer to this corpus as RADIO. For the second collection we constructed a patient records corpus. For this purpose, we treated all clinical notes associated with a patient as a single document. Since the new document includes all different types of notes, it contains more concepts and these concepts are more densely distributed in the ontology. On the other hand, RADIO contains fewer concepts per document and it less cohesive. Table 3 reports some statistics for the two document collections used in the experiments.

We used the SNOMED-CT ontology where we considered only edges that represent *is-a* relationships. In total, there are 296,433 concepts. Each node has an average of 4.53 children. On average there are 9.78 path addresses per concept with length equal to 14.1.

In order to link the medical documents with the SNOMED-CT ontology we applied the following procedure. First, we analyzed each document in order to identify and expand abbreviations based on a public list of medical abbreviations. Next, we used the MetaMap tool [1] in order to identify UMLS concepts associated with terms in the clinical notes. We indexed only UMLS concepts that correspond to SNOMED-CT concepts. Negation of concepts was identified using MetaMap as well. According to domain experts, negated concepts are not relevant when measuring inter-patient similarity [21]. Therefore we only consider concepts with positive polarity; e.g., we exclude concepts contained in phrases such as "absence of bradycardia". We have built an index of the ontology, an inverted index on concepts and a forward index to map documents to concepts. Depending on the collection and ontology sizes and memory availability, the indexes can be memory or disk-based. In our experiments the inverted and forward indexes where loaded into a MySQL database for indexing, thus we will also include performance analysis that measures the database access times.

Experimental setup. All experiments were carried out on an Intel i3 2.1 GHz CPU with 6 GB RAM running Windows 7 and MySQL 5.2.4. All algorithms were implemented in Java 7 with a 4 GB heap and a 64-bit JVM. In order to avoid memory overflow when inserting too many elements into the nodes' queue during a breadth-first expansion step, we set a maximum queue size of 50K elements. Whenever the size of the queue reaches this limit, the graph traversal halts and kNDS is forced to examine the collected set of documents. In practice, the queue size limit can be eliminated by implementing kNDS as a MapReduce job. Each mapper would be responsible for one iteration of the BFS traversal starting from one query node; reducers would do the book-keeping and execute the distance calculation algorithm, if needed.

Parameters. Table 4 describes the parameters under investigation; default values are shown in bold. For each experiment, we vary each parameter while keeping the rest in their default values. Additionally, we set a depth and a collection frequency (cf) threshold such that we exclude generic or very common concepts (such as "disease" or "blood" respectively). For depth threshold we used a default value of 4, i.e., we excluded all concepts in a depth level that is lower than 4. This includes over 99% of the concepts. We found that the number of concepts filtered by the cf threshold depends on the distribution of the dataset. Therefore we used $\mu + \sigma$ as the default cf threshold for each dataset, where μ is the estimated mean and σ is the estimated standard deviation; $\mu+\sigma$ includes about 92% of the concepts. In order to examine the statistical significance of our results, we ran a two-tailed t-test for the times reported in Figure 9 with two sample variances and found out that the execution times measured are statistically significant with p-value < 0.001.

6.2 Experimental Results

Previous works [17, 21] have studied the effectiveness of the distance metrics that we have used, hence our experiments will focus on efficiency. Our goal is to examine the performance of distance calculation separately from the document search algorithm. Thus we conducted two experiments: (i) the first one evaluates the performance on different algorithms for calculating document-document distances, (ii) the second one measures the benefit from our pruning strategy on query evaluation. We discuss which algorithms we compare at the beginning of the respective experiment.

Distance Calculation Experiments. The goal of the first experiment is to measure the scalability of the distance calculation methods against the query size, i.e., the number of concepts in the query document. As we discussed in Section 4.1, building a matrix for all concept-concept distances would impose a large space requirement. Thus, in order to have a fair comparison, we compared two methods that do not require index maintenance, i.e., DRC against a baseline that calculates the document to document distances at the query time by computing the respective minimum concept distances. Our experiments examine the scalability of the two methods when varying the query size n_q over a workload of 5000 randomly generated query documents with n_q concepts each. Figure 6 shows the average time required by the baseline (BL) and the DRC algorithm for the two document collections that we examined. As expected, in all experiments when the query size grows larger, the time required by the baseline methods grows quadratically. In contrast, DRC algorithm takes less than two seconds in the worst case, and grows with nlogn rate as shown in Section 4.3 (n_q is proportional to P_q).

Document Ranking Experiments. This experiment compares kNDS against a baseline method that does not apply any pruning of documents. In order to isolate the performance gains achieved because of the documents pruning that kNDS applies, *we used the DRC algorithm as the distance calculation component for both kNDS and the baseline method*. Note that we did not consider a TA [7] variation as a competitor algorithm since it is impractical for the SDS query due to the problems that we discussed in Section 4.1. We conducted experiments for both RDS and SDS queries. All query experiments measure the average times taken over 100 randomly generated queries; in the case of SDS, documents were randomly

picked from the corpus. Each experiment measures user time spent for distance calculations using DRC, ontology traversal time (applies only for kNDS) and the I/O time of each algorithm.

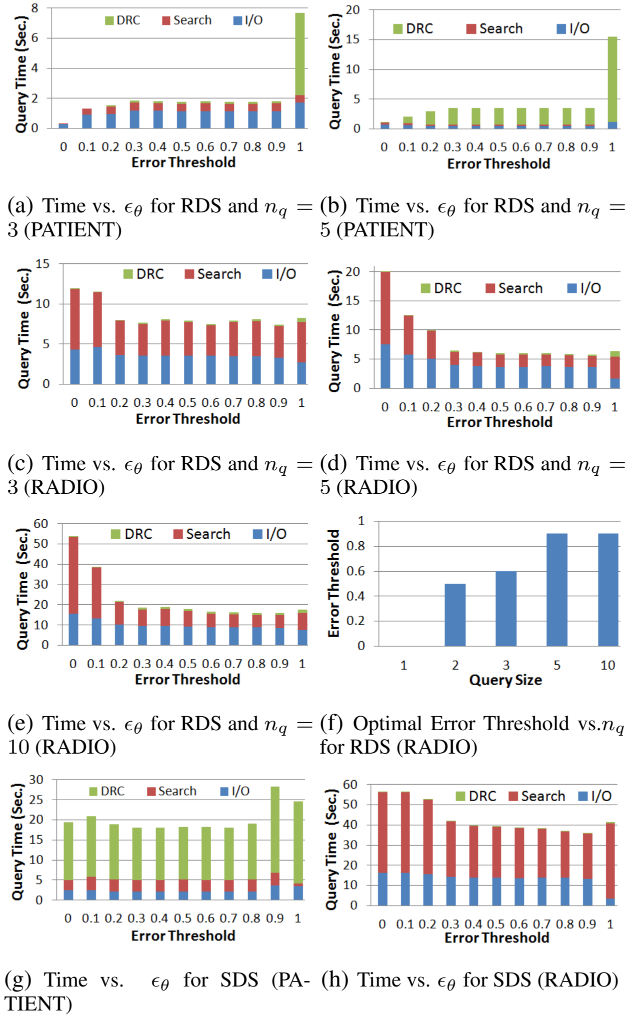
Sensitivity Analysis vs. Error Threshold. In the first set of experiments, we conduct a sensitivity analysis vs. the error threshold that is used an input parameter input of the kNDS algorithm. The examined range of values covers two extreme variations of kNDS; $\epsilon_\theta = 0$ represents a strategy where the algorithm waits until having visited all concepts of a document, i.e. it will calculate an exact distance for this document. On the other hand, when $\epsilon_\theta = 1$, then kNDS would directly calculate the actual distance of a document the first time it visits any concept node linked with the document.

We first examine the performance of kNDS for different values of ϵ_θ when varying the query size for the RDS query type. Plots 7(a)-7(b) show the measured times for the PATIENT collection. An interesting observation is that in this setting the optimal value for ϵ_θ is always 0, i.e., the best strategy is to find all query nodes before examining a document. The reason is that the PATIENT collection contains many concepts that are very close to each other. Thereby, it is highly likely that another document that contains a neighbor node may belong to the query results instead. Thus, in most of the queries, it is more efficient to wait until finding all query nodes in a document. Another important factor is that because of the large number of concepts contained in each document, the DRC calculation part is considerably expensive and dominates the total time for larger query sizes. This is another reason to avoid redundant distance calculations as much as possible.

Plots 7(c)-7(e) show the results for the RADIO collection. In contrast with PATIENT documents, we notice that in this case the query times are highly dependent on the error threshold and they are generally lower for larger thresholds. Further, the distance calculation cost is rather small. The reason is that RADIO documents contain fewer concepts. These concepts are generally sparsely distributed in the ontology graph. Thus, it is sufficient to find some documents that contain only a small subset of the query concepts in order to probe the distance calculation. Since the distance calculation is not expensive, making a false judgment does not affect the performance. As expected, the best error threshold is larger for larger query sizes (plot 7(f)) requiring less query nodes to be found before calculating distances. Plots 7(g) and 7(h) plot the query times measured for various error thresholds for SDS query.

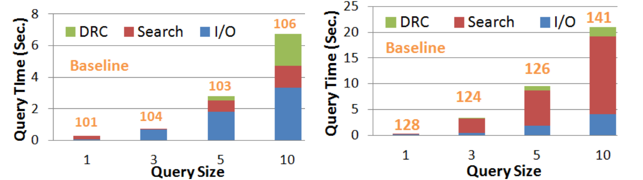
Regardless of the error threshold used, kNDS outperforms the baseline algorithm, where the baseline times are shown in plots 9(a)-9(d). However, the results of the above analysis allow us to find a good setting for the error threshold. In the following experiments we set the default error thresholds for the PATIENT and RADIO collections to 0.5 and 0.9 respectively. The percentage of examined documents (i.e. documents for which DRC was probed) that were eventually part of the top-k query results justifies our settings for the error threshold parameter. Specifically, for RDS in the PATIENT dataset, 99% of the documents for which the actual distance was calculated were returned in the top-k results. For SDS queries over 60% of the examined documents were reported as results; this percentage could be improved by increasing the node queue limit that may cause excessive calls to DRC.

Scalability vs. Query Size. Next we varied the query size n_q and measured the execution time needed by the baseline and kNDS on a workload of RDS queries. Results are depicted in plots 8(a)-8(b). As expected, processing times increase with the size of n_q with rate roughly nlogn, which supports the complexity analysis in Section 5.3 (n_q is proportional to P_q). Note that lower query sizes cause fewer calls to DRC so kNDS can often terminate before exceeding the queue limit. In all settings kNDS is the most efficient algorithm with a large performance gain over the baseline.



(a) Time vs. ϵ_θ for RDS and $n_q = 3$ (PATIENT)
(b) Time vs. ϵ_θ for RDS and $n_q = 5$ (PATIENT)
(c) Time vs. ϵ_θ for RDS and $n_q = 3$ (RADIO)
(d) Time vs. ϵ_θ for RDS and $n_q = 5$ (RADIO)
(e) Time vs. ϵ_θ for RDS and $n_q = 10$ (RADIO)
(f) Optimal Error Threshold vs.n_q for RDS (RADIO)
(g) Time vs. ϵ_θ for SDS (PATIENT)
(h) Time vs. ϵ_θ for SDS (RADIO)

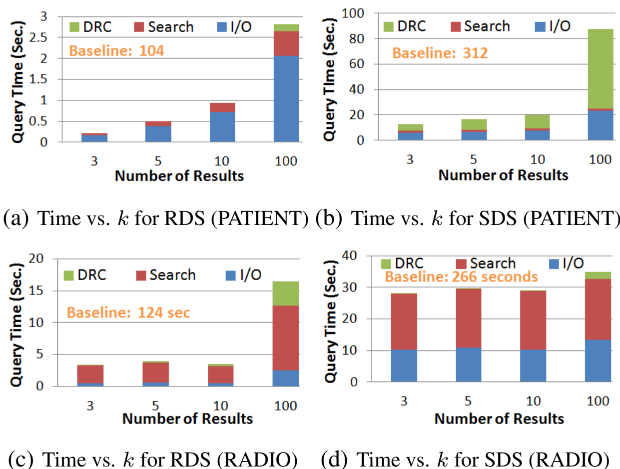
Figure 7: Query Time vs. Distance Error Threshold ϵ_θ



(a)Time vs. n_q for RDS (PATIENT) (b) Time vs. n_q for RDS (RADIO)

Figure 8: Query Time vs. Query Size n_q

Performance Analysis vs. Number of Results. Finally we examined the behavior of the algorithms for evaluating RDS and SDS queries when varying the number of results k. Plots 9(a)-9(d) show the results for the two document collections used. The baseline algorithm has to calculate the distances for all documents in the collection; thus its performance is independent from k whereas kNDS uses a termination condition in order to prune some documents. In all experiments, kNDS outperforms the baseline method with a broad margin. For example, for the default setting where $k = 10$ in the PATIENT collection, kNDS takes less than 1 sec to run, whereas the baseline method takes 104 secs. The performance gains of kNDS are more significant in SDS, e.g., for $k = 10$, kNDS is 99% faster. Again notice that for the PATIENT collection, most of the processing time is used for distance calculation; this is due to the large number of concepts contained in each patient record.



(a) Time vs. k for RDS (PATIENT) (b) Time vs. k for SDS (PATIENT)

(c) Time vs. k for RDS (RADIO) (d) Time vs. k for SDS (RADIO)

Figure 9: Query Time vs. Number of Results k

Finally, as shown in the plots, the performance of kNDS is not affected significantly by k. For instance, for $k = 100$ and a SDS query, the kNDS algorithm is 89% faster than the baseline.

7. CONCLUSION AND FUTURE WORK

In this work we studied two important and challenging types of queries arising when searching over concept-rich document collections, i.e., relevance and similarity queries. Such queries are frequently encountered in Electronic Medical Record (EMR) systems. We proposed an algorithm that reduces the cost of query evaluation from $O(n^2)$ to $O(nlogn)$ by using a variation of the Radix Tree. We presented an efficient early-termination algorithm to search for the top-k most relevant/similar documents that avoids redundant distance calculations following a branch and bound approach. We experimentally evaluated our algorithms against baseline strategies on real clinical data and we showcased the advantages of our methods in terms of efficiency and scalability. In our future work we plan to combine our methods with IR ranking and explore other semantic distances. We also aim to study how non *is-a* ontological edges can be incorporated into the similarity function and how this would affect the algorithms' performance.

Acknowledgments. This research has been partially supported by NSF Grant IIS-1216007.

8. REFERENCES

[1] A. R. Aronson. Effective mapping of biomedical text to the UMLS Metathesaurus: the MetaMap program. In *Proceedings of AMIA Symposium*, pages 17–21, 2001.

[2] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR*, pages 364–371, 2006.

[3] M. Batet, D. Sánchez, and A. Valls. An ontology-based measure to compute semantic similarity in biomedicine. *Journal of Biomedical Informatics*, 44(1):118–125, 2011.

[4] S. Bleeker, G. Derksen-Lubsen, A. van Ginneken, J. van der Lei, and H. Moll. Structured data entry for narrative data in a broad specialty: patient history and physical examination in pediatrics. *BMC Medical Informatics and Decision Making*, 6:29–35, 2006.

[5] C. Carpineto and G. Romano. A survey of automatic query expansion in information retrieval. *ACM Computing Surveys*, 44(1):1–50, 2012.

[6] B. Ding, H. Wang, R. Jin, J. Han, and Z. Wang. Optimizing index for taxonomy keyword search. In *SIGMOD*, pages 493–504, 2012.

[7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[8] F. Farfán, V. Hristidis, A. Ranganathan, and M. Weiner. XOntoRank: Ontology-aware search of electronic medical records. In *ICDE*, pages 820–831, 2009.

[9] M. Fernández, I. Cantador, V. Lopez, D. Vallet, P. Castells, and E. Motta. Semantically enhanced information retrieval: An ontology-based approach. *Journal of Web Semantics*, 9(4), 2011.

[10] N. C. Ide, R. F. Loane, and D. Demner-Fushman. Application of information technology: Essie: A concept-based search engine for structured biomedical text. *Journal of the American Medical Informatics Association (JAMIA)*, 14(3):253–263, 2007.

[11] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*, pages 38–49, 2013.

[12] D. Lin. An information-theoretic definition of similarity. In *ICML*, pages 296–304, 1998.

[13] J. J. Lin and D. Demner-Fushman. The role of knowledge in conceptual retrieval: a study in the domain of clinical medicine. In *SIGIR*, pages 99–106, 2006.

[14] P. W. Lord, R. D. Stevens, A. Brass, and C. A. Goble. Investigating semantic similarity measures across the Gene Ontology: The relationship between sequence and annotation. *Bioinformatics*, 19(10):1275–1283, 2003.

[15] Z. Lu, W. Kim, and W. J. Wilbur. Evaluation of query expansion using MeSH in PubMed. *Information Retrieval*, 12(1):69–80, 2009.

[16] S. Matos, J. Arrais, J. Maia-Rodrigues, and J. L. Oliveira. Concept-based query expansion for retrieving gene related publications from MEDLINE. *BMC Bioinformatics*, 11:212, 2010.

[17] G. B. Melton, S. Parsons, F. P. Morrison, A. S. Rothschild, M. Markatou, and G. Hripcsak. Inter-patient distance metrics using SNOMED-CT defining relationships. *Journal of Biomedical Informatics*, 39(6):697–705, 2006.

[18] R. Moskovitch, S. B. Martins, E. Behiri, A. Weiss, and Y. Shahar. Application of information technology: A comparative evaluation of full-text, concept-based, and context-sensitive search. *Journal of the American Medical Informatics Association (JAMIA)*, 14(2), 2007.

[19] T. Pedersen, S. V. S. Pakhomov, S. Patwardhan, and C. G. Chute. Measures of semantic similarity and relatedness in the biomedical domain. *Journal of Biomedical Informatics*, 40(3):288–299, 2007.

[20] C. Pesquita, D. Faria, A. O. Falcão, P. Lord, and F. M. Couto. Semantic similarity in biomedical ontologies. *PLoS Computational Biology*, 5(7), 2009.

[21] L. Plaza and A. Díaz. Retrieval of similar electronic health records using UMLS concept graphs. In *NLDB*, pages 296–303, 2010.

[22] J. Pound, I. F. Ilyas, and G. E. Weddell. Expressive and flexible access to web-extracted data: a keyword-based structured query language. In *SIGMOD*, pages 423–434, 2010.

[23] R. Rada, H. Mili, E. Bicknell, and M. Blettner. Development and application of a metric on semantic nets. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(1):17–30, 1989.

[24] P. Resnik. Using information content to evaluate semantic similarity in a taxonomy. In *IJCAI*, pages 448–453, 1995.

[25] M. Saeed, M. Villarroel, A. Reisner, G. Clifford, L. Lehman, G. Moody, T. Heldt, T. Kyaw, B. Moody, and R. Mark. Multiparameter intelligent monitoring in intensive care II MIMIC-II: A public-access intensive care unit database. *Critical Care Medicine*, 39:952–960, 2011.

[26] G. K. Savova, J. J. Masanz, P. V. Ogren, J. Zheng, S. Sohn, K. K. Schuler, and C. G. Chute. Mayo clinical text analysis and knowledge extraction system (cTAKES): architecture, component evaluation and applications. *Journal of the American Medical Informatics Association (JAMIA)*, 17(5):507–513, 2010.

[27] Y. Tao, S. Papadopoulos, C. Sheng, and K. Stefanidis. Nearest keyword search in XML documents. In *SIGMOD*, 2011.

[28] F. Wang, J. Sun, and S. Ebadollahi. Composite distance metric integration by leveraging multiple experts' inputs and its application in patient similarity assessment. *Statistical Analysis and Data Mining*, 5(1):54–69, 2012.

[29] H. Wang, Y. Liang, L. Fu, G.-R. Xue, and Y. Yu. Efficient query expansion for advertisement search. In *SIGIR*, pages 51–58, 2009.

[30] Z. Wu and M. Palmer. Verbs semantics and lexical selection. In *Proceedings of the 32nd Annual Meeting on Association for Computational Linguistics*, pages 133–138, 1994.

[31] X. Zhang, L. Jing, X. Hu, M. K. Ng, and X. Zhou. A comparative study of ontology based term similarity measures on PubMed document clustering. In *DASFAA*, pages 115–126, 2007.

[32] W. Zhou, C. T. Yu, N. R. Smalheiser, V. I. Torvik, and J. Hong. Knowledge-intensive conceptual retrieval and passage extraction of biomedical literature. In *SIGIR*, pages 655–662, 2007.