Efficient Association Discovery with Keyword-based Constraints on Large Graph Data

Mo Zhou
Indiana University, USA
mozhou@cs.indiana.edu

Yifan Pan
Indiana University, USA
panyif@cs.indiana.edu

Yuqing Wu
Indiana University, USA
yuqwu@cs.indiana.edu

ABSTRACT

In many domains, such as social networks, cheminformatics, bioinformatics, and health informatics, data can be represented naturally in graph model, with nodes being data entries and edges the relationships between them. The graph nature of these data brings opportunities and challenges to data storage and retrieval. In particular, it opens the doors to search problems such as semantic association discovery [13, 14, 15] and semantic search [2, 10, 11]. We study the application requirements in these domains and find that discovering *Constraint Acyclic Paths* is highly in demand. In this paper, we define the CAP problem and propose a set of quantitative metrics for describing keyword-based constraints. We introduce cSPARQL to integrate CAP queries into SPARQL. We propose a series of algorithms to efficiently evaluate core CAP, a critical fragment of CAP queries, on large scale graph data. Extensive experiments illustrate that our algorithms are efficient in answering CAP queries. Applying our technologies to scientific domains has draw interests from domain experts.

1. INTRODUCTION

RDF (Resource Description Framework) [13] is a W3C recommended language for describing linked data of the Semantic Web in the form of triples. Both RDF data and RDF schema can be represented by node and edge labeled graphs. The simplicity and flexibility of the graph-based data representation model facilitate the wide adoption of Semantic Web technologies in domains such as bioinformatics, cheminformatics, health informatics and social networks. Such applications in these domains pose challenges and opportunities for managing and searching Semantic Web data, as witnessed by new technologies proposed in semantic association discovery [13] and keyword search [10, 11].

1.1 Motivating Examples

The semantic association discovery problem aims at finding answers to the questions like "what are possible relationships between two entities"; the results of which are usually paths connecting the two nodes corresponding to the two entities in the graph [13]. The keyword search problem is to answer questions like "how do the data entities that match the given keywords relate to each other"; the results of which are usually trees/sub-graphs with the labels of their nodes and edges covering all of the keywords [10, 11].

Via the investigation of applications in various domains including cheminformatics and social networks, we confirmed that the discovery of relationships of two given entities under constraints is in great need. In particular we found that the problem of discovering acyclic paths between data entities under constraints such as appearance of nodes, edges and patterns, and the length of paths, is at the core of many applications.

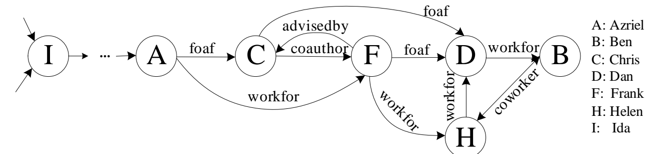

Figure 1: Graph Representation of a Sample RDF Data

EXAMPLE 1.1. *Figure 1 shows the graph representation of a sample RDF data representing the relationships among people in the social networks. Let's consider the following search requests:*
case1. *Find how* Azriel *connects to* Ben*;*
case2. *Find the close ties (within 3 steps) between* Azriel *and* Ben*;*
case3. *Find how* Azriel *connects to* Ben *through* Chris *or* Dan*;*
case4. *Find how* Azriel *connects to* Ben *through at least two people from* Chris, Dan *and* Ida *within four steps.*
case5. *Find* Azriel*'s close (within 4 steps) professional connections (e.g. relationships such as* workfor, coworker, coauthor*) to* Ben*;*
case6. *Find* Azriel*'s close (within 4 steps) semi-professional connections to* Ben *(i.e. half of the relationships in any tie should be professional);*
case7. *Find* Azriel*'s close (within 4 steps) semi-professional connections to* Ben *that involves the advisor of* Frank.

Query languages have been proposed to query data on the Semantic Web. SPARQL [5], the de facto standard query language for RDF, relies on graph patterns to identify data entities and relationships of interest. However, it lacks the ability to express arbitrary paths between data entities, such as those shown in Ex. 1.1. The notion of label-constraint reachability (LCR) [12] was proposed to describe the problem of finding the *connectivity* between two given nodes in a labeled graph under the constraint that all the edge labels along the path are in a given set. The semantic keyword search problem was defined to find trees in a labeled directed graph where the tree nodes cover all the keywords [8]. Combining these notions, several SPARQL extensions, with the introduction of path variables, were proposed [14, 16]. Such extensions are capable of expressing the search queries in cases 1-2 in Ex. 1.1, but not the other cases. There were also proposals for extending SPARQL with regular expressions [6, 7] to express complex patterns that satisfy strictly defined constraints such as cases 1,2,3,5. However, such

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.

SPARQL extensions still cannot express more relaxed constraints such as those in cases 4,6,7.

In this paper, we abstract the association discovery problem as the problem of *finding constrained acyclic paths (CAP) in directed labeled graphs*. We further specify the constraints in terms of the *length* of the resultant paths, the *coverage* of the keyword set by the resultant paths, and the *relevance* of the resultant paths with respect to the keyword set.

Expressing CAP search queries in a high-level query language greatly improves its usability in generic and domain-specific applications. We propose cSPARQL to incorporate CAP search into the framework of SPARQL, by introducing path variables and functions for defining the keyword-based constraints quantitatively.

1.2 CAP Discovery

Once the CAP search query is identified and formally defined, the imminent question is how to answer such search queries efficiently on graph data of large scale.

Significant amount of research has been done in RDF data storage, indexing and query evaluation for answering SPARQL queries efficiently [4, 9, 17, 20].

Many graph searching algorithms were proposed for answering keyword searches in graph data. In BANKS [8], BANKS2 [19] and BLINKS [11] *backward* search algorithms were proposed and improved. Other inventions in this area include a caching and buffering algorithm for searching large data graph residing on external storage [2] and using adjacency matrix to index sub-graphs and locate those satisfying search conditions [10].

Traversal-based approaches such as DFS, BFS, and bidirectional search were proposed for finding paths satisfying given regular expressions between two end nodes [6, 7]. A schema-based approach was later proposed to first search the RDF schema graph and use the findings to generate many SPARQL queries against the RDF data, and then execute these queries to find the desired paths [13]. However RDF schema is not always available, and the efficiency of this approach is not superb since the number of SPARQL queries generated may be very large, many have overlapping sub-queries, and indeed very few of these queries yield meaningful results. As a remedy, data graph was preprocessed and paths were encoded and indexed, then, such indices are used to find paths between two nodes [14, 15]. However preprocessing the whole graph and indexing all paths greatly affect the scalability of the algorithm.

In this paper, we propose to take advantage of the constraints on path length and keywords and push the result validation deep into the path discovery process, to prune unpromising intermediate results as early as possible. In particular, we propose two families of algorithms. ConstraintDFS (cDFS) and Enhanced ConstraintDFS (ecDFS) take advantage of projected value ranges on the constraint metrics to prune search branches efficiently. Search-and-Join (S&J) algorithm issues mini-searches to find exclusive path fragments (i.e. paths that do not pass through any keyword nodes) between pairs of nodes that contain keywords, then use *constrained sequence join* to concatenate the fragments to produce the final results. Careful bookkeeping allows us to use the partial results in one mini-search to limit the search space of many other mini-searches, and effectively reduces the overall search cost. Our experimental results indicate that our proposed algorithms can take advantage of constraints to efficiently answer CAP queries while the S&J algorithm outperforms the cDFS/ecDFS algorithms.

1.3 Contributions

Our contributions can be summarized as follows:

- We formally define the *constrained acyclic path* (CAP) discovery problem (Sec. 2).
- We propose cSPARQL, an extension to SPARQL [5] with the introduction of path variables and quantitative metrics for measuring keyword-based constraints, to express CAP search queries (Sec. 3).
- We propose two families of search algorithms to efficiently discover CAPs in large-scale graph data (Sec. 4- 5).
- We conduct extensive empirical study to understand the strength and limits of our algorithms (Sec. 6).

2. CONSTRAINT ACYCLIC PATH DISCOVERY PROBLEM

In this section, we formally define the *Constraint Acyclic Path Discovery* problem whose applications have been illustrated in Sec. 1.

2.1 Preliminary

Let $\mathcal{L}$ be an infinite set of literals and $\mathcal{U}$ be an infinite set of URIs disjoint with $\mathcal{L}$. We represent RDF data as a node and edge labeled directed graph $G = (V, E, \lambda)$ where V is a set of nodes, $E \subset V \times V$ is a set of edges, and λ is a labeling function that maps items in $V \cup E$ into a finite set of labels and literals.

We call a sequence of interleaving nodes and edges of graph G a *path fragment* (or *fragment*), represented by f, if

- for every adjacent (n, e, n') in f, it is the case that $n, n' \in V \wedge e = (n, n') \in E$;
- if (e, n) is a prefix of f, then there must exist $n' \in V$, such that $e = (n', n) \in E$; and
- if (n, e) is a suffix of f, then there must exist $n' \in V$, such that $e = (n, n') \in E$.

Given a fragment f, we use $f.head$ and $f.tail$ to represent the first item and last item in f. We use $nodes(f)$/$edges(f)$ to represent the set of nodes/edges in f, and $Length(f)$ (or $|f|$) the length of f, defined as $|edges(f)|$. We overload the mapping function λ to map a set of nodes/edges to their corresponding labels.

We are particularly interested in two special types of path fragment: e-fragment (denoted by f_e), in which $f.head, f.tail \in E$ [1], and en-fragment (denoted by f_{en}), in which $f.head \in E$ and $f.tail \in V$.

Given two nodes $n_s, n_d \in V$ as the source and destination nodes, the paths that link n_s to n_d in G are fragments in the form $f = (n_s, e_1, n_1 \ldots, n_{k-1}, e_k, n_d)$. In search queries that look for the paths between two nodes, frequently only acyclic paths are of interest to users. In the rest of the paper, we will focus only on such fragments. We use $\mathcal{F}_e(n_s, n_d)$ to represent all acyclic e-fragments between n_s and n_d and $\mathcal{F}_{en}(n_s, n_d)$ to represent all acyclic en-fragments from n_s to n_d (including n_d).

2.2 Set-based Constraints

As shown in Ex. 1.1, when users search for paths between a pair of nodes, they are frequently interested in using certain constraints to refine the results to be retrieved. Such constraints are usually given in the form of a *keyword set* (denoted by $\mathcal{S}$), where a keyword (denoted by l) is a label in $\mathcal{U}$. Certain constraints, such as presence constraints [14] on nodes and tight constraints on edges [12] are discussed in the existing works, but they are quite limited in terms of what can be in the keyword set and how the results are regulated by it, hence not sufficient to express some search queries, such as cases 3-7 in Ex. 1.1.

We generalize the keyword set to include keywords that can be mapped to labels of both nodes and edges and extend how the results are confined by a keyword set.

[1] Please note that it is possible that $f.head = f.tail$, in such case, the e-fragment contains only one edge.

DEFINITION 2.1. *Given a finite keyword set $\mathcal{S} \subseteq \mathcal{U}$ and an e-fragment $f_e \in \mathcal{F}_e(n_s, n_d)$,*

1. *if $\mathcal{S} \subseteq (\lambda(nodes(f_e)) \cup \lambda(edges(f_e)))$, we say f_e satisfies* presence constraint *w.r.t. $\mathcal{S}$;*
2. *if $\mathcal{S} \supseteq (\lambda(nodes(f_e)) \cup \lambda(edges(f_e)))$, we say f_e satisfies* context constraint *w.r.t. $\mathcal{S}$;*
3. *if $\mathcal{S} \cap (\lambda(nodes(f_e)) \cup \lambda(edges(f_e))) \neq \emptyset$, we say f_e satisfies* intersection constraint *w.r.t. $\mathcal{S}$.*

Based on the definition above, we can express the search request in Ex. 1.1 case3 as "find e-fragments from *Azriel* to *Ben* that satisfy the *intersection constraint* w.r.t. keyword set $\{Chris, Dan\}$".

2.3 Quantitative Metrics

Among a possible large number of resultant e-fragments of a search request, shorter e-fragments tend to express stronger and more meaningful relationship between the two end nodes than the longer ones do [11, 13]. The *length constraint* which restricts the length of the resultant e-fragments has been studied in [6, 7, 14], and can be used to express the search request in Ex. 1.1 case 2.

However, empowered by the constraints defined in Def. 2.1 and length constraint, we still cannot express the search problem in Ex. 1.1 cases 4-7 because they require a more subtle description of the relationship between an e-fragment and a keyword set than the *all-or-nothing* set-based constraints described in Def. 2.1. For this purpose, we introduce quantitative metrics *coverage* and *relevance*.

Intuitively, the *coverage* describes the fraction of the keyword set that appears in the label set of an e-fragment, while the *relevance* describes the fraction of the labels of an e-fragment that are in the keyword set. In an RDF graph, each node has its unique label, while more than one edge may have the same label. Therefore, we refine *coverage* and *relevance* further into *node-coverage* and *node-relevance* for keyword sets to be applied only on nodes, *edge-coverage* and *edge-relevance* for edges, and use *coverage* and *relevance* for the constraints in which keywords can be mapped to both nodes and edges. We use $cntE(l, f_e)$ to represent the number of appearance of a keyword l among the edges in an e-fragment f_e. Formally, the quantitative metrics can be defined as follows.

DEFINITION 2.2. *Given a graph G, two nodes $n_s, n_d \in V$ and a finite keyword set $\mathcal{S} \subseteq \mathcal{U}$, for an e-fragment $f_e \in \mathcal{F}_e(n_s, n_d)$,*

$$NodeCoverage(f_e, \mathcal{S}) = \frac{|\mathcal{S} \cap \lambda(nodes(f_e))|}{|\mathcal{S}|} \quad (1)$$

$$NodeRelevance(f_e, \mathcal{S}) = \begin{cases} \frac{|\mathcal{S} \cap \lambda(nodes(f_e))|}{|nodes(f_e)|}, & |f_e| > 1 \\ 0, & |f_e| = 1 \end{cases} \quad (2)$$

$$EdgeCoverage(f_e, \mathcal{S}) = \frac{|\mathcal{S} \cap \lambda(edges(f_e))|}{|\mathcal{S}|} \quad (3)$$

$$EdgeRelevance(f_e, \mathcal{S}) = \frac{\Sigma_{l \in \mathcal{S}} cntE(l, f_e)}{|edges(f_e)|} \quad (4)$$

$$Coverage(f_e, \mathcal{S}) = \frac{|\mathcal{S} \cap (\lambda(nodes(f_e)) \cup \lambda(edges(f_e)))|}{|\mathcal{S}|} \quad (5)$$

$$Relevance(f_e, \mathcal{S}) = \frac{|\mathcal{S} \cap \lambda(nodes(f_e))| + \Sigma_{l \in \mathcal{S}} cntE(l, f_e)}{|nodes(f_e)| + |edges(f_e)|} \quad (6)$$

Taking advantage of the quantitative metrics, the search requests in Ex. 1.1 case4-6 can be expressed more precisely: case 4, "find all e-fragments, e.g. f_e, from *Azriel* to *Ben* such that NodeCoverage(f_e, {*Chris*, *Dan*, *Ida*})≥ 0.6 and $|f_e| \leq 4$"; case 5, "find all e-fragments, e.g. f_e, from *Azriel* to *Ben* such that EdgeRelevance(f_e, {*coworker*, *workfor*, *coauthor*}) = 1 and $|f_e| \leq 3$"; case 6, "find all e-fragments, e.g. f_e, from *Azriel* to *Ben* such that EdgeRelevance(f_e, {*coworker*, *workfor*, *coauthor*}) ≥ 0.5 and $|f_e| \leq 4$".

All the constraints we defined in Def. 2.1 can be expressed using the quantitative metrics defined in Def. 2.2. Here, we introduce a set of boolean functions to express such constraints.

$$Presence(f_e, \mathcal{S}) \iff Coverage(f_e, \mathcal{S}) == 1 \quad (7)$$

$$Context(f_e, \mathcal{S}) \iff Relevance(f_e, \mathcal{S}) == 1 \quad (8)$$

$$Intersection(f_e, \mathcal{S}) \iff Relevance(f_e, \mathcal{S}) > 0 \quad (9)$$

Similarly, we can define the node/edge version of these functions.

2.4 Problem Definition

We define the Constraint Acyclic Path (CAP) search query:

A CAP search query $CAP(n_s, n_d, \tau)$ takes as input an RDF graph G, two end nodes $n_s, n_d \in V$, and constraint τ expressed using zero to many quantitative metrics functions that involve zero or many keyword sets, and returns the e-fragment(s) from n_s to n_d that satisfy τ.

In this paper, we will tackle two problems related to CAP search query: how to express CAP search queries in a structured query language, and how to evaluate CAP search queries efficiently on massive graph data.

3. cSPARQL

We propose cSPARQL to integrate the *CAP* search into the structured search of SPARQL [5] by introducing (1) *path variables* for expressing arbitrary e-fragments in a graph pattern; and (2) a set of quantitative metrics functions as defined in Sec. 2 for specifying the length and keyword-based constraints.

3.1 Syntax and Semantics

The basic construct of a SPARQL query is *simple access pattern* in the form of a triple (x, y, z) with $x, y \in \mathcal{U} \cup \mathcal{V}_n$ and $z \in \mathcal{U} \cup \mathcal{L} \cup \mathcal{V}_n$, where $\mathcal{V}_n$ is a set of variables disjoint with $\mathcal{U} \cup \mathcal{L}$ (variables are prefixed by either "?" or "$") . A *basic graph pattern* consists of a set of simple access patterns. Given an RDF graph $G = (V, E, \lambda)$, the core of the evaluation of a SPARQL query is to find mapping functions that map the basic graph pattern pt of the query to subgraphs of G. Formally, consider all mapping functions

$$M = \begin{cases} m(x) \in V \cup E & x \in \mathcal{V}_n \cup \mathcal{U} \\ m(x) \in V & x \in \mathcal{L} \end{cases}$$

the semantic of $pt(G)$ is to find all mappings in M such that for each simple access pattern $(x, y, z) \in pt$:

- $m(x) \in V$;
- $m(z) \in V$;
- $(m(x), m(z)) \in E$; and
- $\lambda((m(x), m(z))) = \lambda(m(y))$.

Besides the structural constraints specified using the graph pattern, the *FILTER* phrase can be used to further express value constraints on the variables.

In cSPARQL, we introduce a special type of variables $\mathcal{V}_p$, called *path variables*, prefixed by "??" as introduced in SPARQL2L [14]. $\mathcal{V}_p$ is disjoint with $\mathcal{V}_n \cup \mathcal{U} \cup \mathcal{L}$. We extend the basic construct of cSPARQL to include *path access pattern* in the form of a triple (x, p, z) with $x \in \mathcal{U} \cup \mathcal{V}_n$, $p \in \mathcal{V}_p$ and $z \in \mathcal{U} \cup \mathcal{L} \cup \mathcal{V}_n$. We define an *extended graph pattern* to be a set of simple access patterns and path access patterns. We extend the mapping functions to include the mapping of path variables to e-fragments in G,

$$M = \begin{cases} m(x) \in V \cup E & x \in \mathcal{V}_n \cup \mathcal{U} \\ m(x) \in V & x \in \mathcal{L} \\ m(x) \in \mathcal{F}_e & x \in \mathcal{V}_p \end{cases}$$

and extend the semantics of applying an extended graph pattern pt in cSPARQL on an RDF graph G to include mappings of each path access pattern $(x, p, z) \in pt$ such that

- $m(x) \in V$;
- $m(z) \in V$; and
- $m(p) \in \mathcal{F}_e(m(x), m(z))$;

We introduce a set of the numeric functions (formula 1-6 in Sec. 2) and boolean functions (formula 7-9 in Sec. 2) to be used in the FILTER phrase to express the length and keyword-based constraints on the path variables. For the convenience of the users, we introduce keyword *CONSTRAINTSET* in the form of

CONSTRAINTSET constraint-set-name $\{l_1, \ldots, l_s\}$

for users to create alias of a keyword set, which can be referred later in the functions.

EXAMPLE 3.1. *We illustrate the CAP query of Ex. 1.1 case7 in cSPARQL and more examples can be found in the appendix.*

```
SELECT ??p WHERE
{Azriel ??p Ben . Frank advisedby ?adv .
 CONSTRAINTSET ProfR {coworker, workfor, coauthor}.
 FILTER(Length(??p)<=4) .
 FILTER(EdgeRelevance(??p, ProfR)>=0.5) .
 FILTER(NodePresence(??p, ?adv)) }
```

Result: $\xrightarrow{foaf} C \xrightarrow{coauthor} F \xrightarrow{foaf} D \xrightarrow{workfor}$

4. CAP DISCOVERY

The cSPARQL proposed in Sec. 3 empowers users to express CAP search queries in the framework of SPARQL. As significant amount of research has been done on answering SPARQL queries efficiently [4, 9, 17, 20], here, we focus on the new components introduced in cSPARQL, that is to find all acyclic e-fragments between two given nodes under constraints, i.e., to answer the $CAP(n_s, n_d, \tau)$ query. We will first focus on the evaluation of a critical subset of CAP queries, *core CAP* queries, in which τ contains conjunctive predicates featuring only one keyword set. We use $\mathcal{S}$ to represent the single keyword set in τ, and use τ_l, τ_c, τ_r, τ_{nc}, τ_{ec}, τ_{nr} and τ_{er} to represent the length, coverage, relevance, node/edge coverage, node/edge relevance constraints respectively, each of which is defined as an interval, for example, $\tau_l = (\tau_{l_{min}}, \tau_{l_{max}})$.

In this section, we will present an enhanced DFS algorithm for answering core CAP queries. Then, we will present a Search-and-Join (S&J) algorithm in Sec. 5 to further improve the performance. Finally, we will discuss how to extend the algorithms we propose to answer CAP queries in general.

4.1 Basic Ideas

Certainly one solution, which we call *Search-Filter approach*, is to first find all acyclic e-fragments in $\mathcal{F}_e(n_s, n_d)$ (in fact, to find $CAP(n_s, n_d, \emptyset)(G)$), then eliminate those that do not satisfy the constraints specified in τ. However this approach is not practically efficient because generating $\mathcal{F}_e(n_s, n_d)$ is very time and space consuming, rendering the *search* phase costly, while it is frequently the case that $|CAP(n_s, n_d, \tau)(G)| \ll |CAP(n_s, n_d, \emptyset)(G)|$, rendering the high cost of the *search* phase mostly wasted.

Depth-First-Search (DFS) is a commonly adopted approach for generating paths between two nodes. In DFS, to generate an e-fragment $f_e \in \mathcal{F}_e(n_s, n_d)$, en-fragments of length ranging from 1 to $|f_e|$-1 are generated one step at a time, e.g. a set of en-fragments of length k+1 are generated by *extending* an en-fragment of length k in the k+1'th step of the DFS process. For any such en-fragment f_n generated as an intermediate result, if it is the prefix of a resultant e-fragment $f_e \in \mathcal{F}_e(n_s, n_d)$, we say that f_n is a *prefix* of f_e and f_e is an *extension* of f_n, denoted by $f_n \prec f_e$. We call the e-fragment f_e-f_n the *complement* of f_n w.r.t f_e, denoted $\overline{f_n}^{f_e}$, or $\overline{f_n}$ when there is no ambiguity about the specific f_e in question. Please note that more than one e-fragment in $\mathcal{F}_e(n_s, n_d)$ may share the same prefix en-fragment f_n, the whole set of which is denoted by $Ext(f_n)$, i.e. $Ext(f_n) = \{f_e | f_e \in \mathcal{F}_e(n_s, n_d) \wedge f_n \prec f_e\}$.

To minimize the DFS search space in computing core CAP query $CAP(n_s, n_d, \tau)$, we want to minimize the total number of en-fragments generated in the process. To be more specific, given an en-fragment f_n generated as an intermediate result, we want to stop the extension of f_n if we are certain $CAP(n_s, n_d, \tau)(G) \cap Ext(f_n) = \emptyset$, and we propose to do so by deriving tighter constraints using partial results generated in the DFS process.

4.2 Constraint Tightening

To better understand how the information about an en-fragment f_n can stop or limit its own extensions, we first take a look at the projected value ranges of the quantitative metrics of f_n's extensions, w.r.t. the keyword set.

LEMMA 4.1. *Given an en-fragment f_n generated in the DFS of $CAP(n_s, n_d, \emptyset)$ and a keyword set $\mathcal{S}$, for any e-fragment $f_e \in Ext(f_n)$ with $|f_e| > 1$,*

$$\frac{|\mathcal{S} \cap (\lambda(nodes(f_n))|}{|\mathcal{S}|} \leq NodeCoverage(f_e) \leq \begin{cases} \frac{|\mathcal{S} \cap (\lambda(nodes(f_n))| + |f_e| - |f_n| - 1}{|\mathcal{S}|} & * \\ 1 & \textit{Otherwise} \end{cases} \quad (1)$$

$$\frac{|\mathcal{S} \cap (\lambda(nodes(f_n))|}{|f_e| - 1} \leq NodeRelevance(f_e) \leq \begin{cases} \frac{|\mathcal{S} \cap (\lambda(nodes(f_n))| + |f_e| - |f_n| - 1}{|f_e| - 1} & * \\ \frac{|\mathcal{S}|}{|f_e| - 1} & \textit{Otherwise} \end{cases} \quad (2)$$

$$* \quad |f_e| \leq |\mathcal{S}| + (|f_n| - |\mathcal{S} \cap (\lambda(nodes(f_n))|) + 1$$

To prove this lemma, we consider the best and worst scenarios in the expansion from f_n to f_e. Taking the node coverage constraint as an example. The best case is that the nodes in $\overline{f_n}^{f_e}$ covers all keywords that were not yet covered by f_n, while the worst case is that it covers no keyword. Using these cases as the upper/lower bounds, the formula can be derived. We can apply the same approach to all the other constraints. The proof of Lemma 4.1 and other similar lemma (Lemma 4.2) are provided in the appendix.

Indeed, given an en-fragment f_n generated by DFS, the length of any e-fragment $f_e \in Ext(f_n)$ is also bounded. Its minimum value is $|f_n|$+1. Its maximum value (denoted by l_{max}) equals to the minimum value among the following: (1) $\tau_{l_{max}}$; (2) MAX($\frac{|\mathcal{S}|}{\tau_{nr_{min}}} + 1$, $|\mathcal{S}| \times \tau_{nc_{min}} + 1$, $|\mathcal{S}| \times \tau_{ec_{min}}$), with $\tau_{nr_{min}} > 0$, and (3) the diameter of the graph G. Hence, with the help of the projected bound on the length of the e-fragments in $Ext(f_n)$, we can obtain a tighter projected bound on the quantitative metrics of those e-fragments than the ones proposed in Lemma 4.1 and 4.2, and further identify if f_n is promissing or not to be expanded.

4.3 DFS-based Algorithms

We propose two DFS-based algorithms: constraintDFS (cDFS) prunes unpromising en-fragments at each DFS step; enhanced-cDFS (ecDFS) further saves unnecessary computation and verifications, while maintaining the same pruning power as cDFS.

constraintDFS cDFS is based on the non-recursive DFS. In cDFS, we start a DFS from the source node n_s. At each step in the DFS process, we can safely stop the expansion of an intermediate en-fragment f_n whenever the projected value ranges do not overlap with the value ranges specified in τ on any of the quantitative metrics. The pesudo-code of cDFS is shown in Alg. 1. We use a stack

Stk to keep track of all edges whose starting nodes have been expanded but ending nodes haven't. It is initialized with the nodes reachable from n_s via a single edge (L6-7). In each DFS step (L8-L28), fragments with loops are detected and eliminated (L14); results are identified (when the destination node n_d is reached and constraint τ is verified) and stored (L16-18); and unpromising en-fragment are identified and pruned (L27).

Algorithm 1 cDFS

Data: data graph G, sourse/destination node n_s, n_d, constraint τ.
Result: All acyclic e-fragments from n_s to n_d satisfying τ.
begin
1 Array $results \leftarrow \{\}$
2 HashTable $VisitedNodes \leftarrow \{\}$
3 $VisitedNodes.Add(n_s)$
4 Array <Edge> $f_n \leftarrow ()$
5 Stack $Stk \leftarrow \{\}$
6 **for** *e in n_s's outgoing edges* **do**
7   Stk.Push(e)
8 **while** *!Stk.isEmpty()* **do**
9   Edge $e \leftarrow Stk$.Pop()
10   **if** *e.startNode==null & e.endNode==null* **then**
11   Edge $tailEdge \leftarrow f_n.RemoveTailEdge()$
12     $VisitedNodes.Remove(tailEdge.endNode)$
13   Continue
14   **if** *!$VisitedNode$.Contains(e.endNode)* **then**
15     $f_n.Concatenate(e)$
16   **if** *e.endNode == B* **then**
17   **if** *f_n satisfies constraints* **then**
18         $results.add(f_n.clone())$
19   **else**
20   Array projRanges = computeProjRange(f_n, τ)
21   **if** *Overlapped(projRanges, τ)* **then**
22         $VisitedNodes$.Add(e.endNode)
23   Stk.Push((null, null))
24   **for** *e' in e.endNode's outgoing edges* **do**
25   Stk.Push(e')
26   Continue;
27     $f_n.RemoveTailEdge()$
28 Return results

Enhanced-cDFS In cDFS, the projected value ranges for the quantitative metrics are computed and compared with τ for every en-fragment generated. This is indeed unnecessary. Given an en-fragment f_n that is deemed promising in a DFS step, we are interested in predicting how many more steps forward, any extension of f_n is guaranteed to be promising and no more checking is needed.

LEMMA 4.3. *While applying cDFS algorithm to answer a core CAP query $CAP(n_s, n_d, \tau)$, if an en-fragment f_{n_m} is deemed promising at the m'th step, then in the next k steps, the en-fragment $f_{n_{m+k}}$ with $f_{n_m} \prec f_{n_{m+k}}$ is guaranteed to be promising if*

$$0 \leq k \leq MAX(0, SkippedStep(f_{n_m})) \quad with$$
$$SkippedStep(f_{n_m})$$
$$= MIN(l_{max} - |f_{n_m}|,$$
$$l_{max} + |\mathcal{S} \cap (\lambda(nodes(f_{n_m}))| - |\mathcal{S}| \times \tau_{nc_{min}} - |f_{n_m}| - 1,$$
$$l_{max} + |\mathcal{S} \cap (\lambda(edges(f_{n_m}))| - \tau_{ec_{min}} \times |\mathcal{S}| - |f_{n_m}|,$$
$$(1 - \tau_{nr_{min}}) \times l_{max} + |\mathcal{S} \cap (\lambda(nodes(f_{n_m}))| - |f_{n_m}| - (1 - \tau_{nr}),$$
$$l_{max} \times (1 - \tau_{er_{min}}) + \Sigma_{l \in \mathcal{S}} cntE(l, f_{n_m}) - |f_{n_m}|)$$

We provide a proof sketch for the component related to the node coverage in Lemma 4.3 by examining the worst case scenario of $f_{n_{m+k}}$ in the appendix. With the help of Lemma 4.3 we can predict the maximum number of steps we can extend safely after deciding to keep an en-fragment, without any additional bound computation and verification.

We propose an enhancement to the cDFS algorithm (ecDFS) to avoid unnecessary computation and verification of quality of the en-fragments. In ecDFS, whenever an en-fragment f_n is kept, we predict k=$SkippedStep(f_n)$ using the formula in Lemma 4.3. Then, verification can be skipped in the next k steps.

EXAMPLE 4.1. *Let's consider case 4 of Ex. 1.1, $CAP(A, B, \{NodeCoverage(??p, \{C, D, I\}) > 0.6 \& Length(??p) \leq 4\})$. Instead of generating all paths from A to B, cDFS will prune the en-fragments $\xrightarrow{foaf} C \xrightarrow{coauthor} H \xrightarrow{workfor} F$ and $\xrightarrow{workfor} F \xrightarrow{workfor} H$ because the projected node coverage value ranges of both are [0,0.3], which does not overlap with (0.6,1] given by τ. ecDFS will further save the validation on the en-fragment $\xrightarrow{foaf} C \xrightarrow{coauther} F$.*

5. LOCALIZED SEARCH AND JOIN

In both cDFS and ecDFS the search starts from the source node and the nodes matching the keywords do not contribute to the estimation of projected value ranges until they are reached. Whether to prune an intermediate en-fragment solely depends on the best/-worst cases foreseeable from the en-fragment itself. In this section, we propose to use the local information around nodes matching the keywords to produce more accurate projected ranges of the quantitative metrics for more efficient pruning.

5.1 Constrained Sequence Join

We call the nodes whose labels are in $\mathcal{S}$ the *constraint nodes*, denoted by $\mathcal{S}_n$. We consider constraint nodes, together with source and destination nodes, the *query nodes*: $\mathcal{Q} = \mathcal{S}_n \cup \{n_s, n_d\}$, and we call $\mathcal{S}_n \cup \{n_s\}$ the starting query nodes $\mathcal{Q}_s$, while $\mathcal{S}_n \cup \{n_d\}$ the ending query nodes $\mathcal{Q}_d$. We call a node sequence $(q_0, q_1, \ldots, q_u, q_{u+1})$ with $0 \leq u \leq |\mathcal{S}_n|$ a *query node sequence (QNS)* of a given $\mathcal{Q}$ if $q_0 = n_s$, $q_{u+1} = n_d$ and $q_i \in \mathcal{S}_n$ for $0 < i < u+1$, and no two nodes in the sequence are identical. Given a QNS qns, we use $|qns|$ to denote the number of constraint nodes in qns.

An e-fragment between two query nodes is *exclusive* (or xe-fragment, denoted by f_{xe}) if it does not pass through any query node, i.e. $nodes(f_{xe}) \cap \mathcal{Q} = \emptyset$. Following the same naming tradition used in this paper, we use $\mathcal{F}_{xe}(n_1, n_2)$ to represent all xe-fragments between n_1 and n_2 with respect to a set of query nodes $\mathcal{Q}$. We use $\mathcal{M}_{\mathcal{F}_{xe}}$ to denote the matrix of all $\mathcal{F}_{xe}(q_i, q_j)$ with $q_i \in \mathcal{Q}_s$ and $q_j \in \mathcal{Q}_d$.

To efficiently evaluate a core CAP query, certainly computing the full $\mathcal{F}_{xe}(q_i, q_j)$ between a node pair (q_i, q_j) and the full $\mathcal{M}_{\mathcal{F}_{xe}}$ is not desirable. In fact, our goal is to compute as small a subset of them as possible in query answering. In the rest of the section, we use $\widetilde{\mathcal{F}_{xe}}(q_i, q_j)$ to represent a subset of $\mathcal{F}_{xe}(q_i, q_j)$, and we use $\mathcal{M}_{\widetilde{\mathcal{F}_{xe}}}$ to represent a matrix of such $\widetilde{\mathcal{F}_{xe}}$'s.

Given two xe-fragments $f_1 \in \mathcal{F}_{xe}(q_i, q_j)$ and $f_2 \in \mathcal{F}_{xe}(q_v, q_w)$ w.r.t. $\mathcal{Q}$, we say that f_1 and f_2 can be *concatenated* to form one acyclic e-fragment by concatenating f_1 and the node q_j (q_v) and f_2, if (1) $q_j = q_v$, and (2) $nodes(f_1) \cap (nodes(f_2) \cup \{q_v, q_w\}) = \emptyset$ and vice versa. We define the concatenation operation "$\bowtie$" between two sets of xe-fragments $\widetilde{\mathcal{F}_{xe}}(q_i, q_j)$ and $\widetilde{\mathcal{F}_{xe}}(q_v, q_w)$ to compute the concatenation of every pair of xe-fragments from the Cartesian product of $\widetilde{\mathcal{F}_{xe}}(q_i, q_j)$ and $\widetilde{\mathcal{F}_{xe}}(q_v, q_w)$. The result of

$\widetilde{\mathcal{F}_{xe}}(q_i, q_j) \bowtie \widetilde{\mathcal{F}_{xe}}(q_v, q_w)$ is a subset of $\mathcal{F}_e(q_i, q_w)$ with all e-fragments in it passing through q_j (q_v).

DEFINITION 5.1. *Given constraint τ of a core CAP query, assuming that qns is a QNS of query nodes $\mathcal{Q}$ defined by the keyword set of τ and $\mathcal{M}_{\widetilde{\mathcal{F}_{xe}}}$ is an xe-fragment set matrix based on $\mathcal{Q}$,*

$$\bowtie_\tau (qns, \mathcal{M}_{\widetilde{\mathcal{F}_{xe}}}) = \sigma_\tau(\Pi_{i=0}^{|qns|} \widetilde{\mathcal{F}_{xe}}(q_i, q_{i+1}))$$

We call this operation constrained sequence join.

The result of a *constrained sequence join* operation is a set of e-fragments from n_s to n_d that satisfy τ. Obviously the answer to a core CAP query is the union of the results from computing the constrained sequence join for all possible QNS on $\mathcal{M}_{\mathcal{F}_{xe}}$ that is the full matrix of xe-fragments based on $\mathcal{Q}$.

Selection-push-down is one of the most important query optimization technique for reducing the cardinality of participants of complex operations in order to improve the overall query evaluation efficiency. We propose a similar CAP query evaluation technique that reduces the cardinality of the participants of constrained sequence join operation in two aspects: (1) identify and eliminate QNSs whose constrained sequence join result is empty; and (2) for each QNS that may generate non-empty constraint sequence join result, identify and eliminate the xe-fragments that have no chance to contribute to the results.

We call a QNS qns *invalid* if $\bowtie_\tau (qns, \mathcal{M}_{\mathcal{F}_{xe}}) = \emptyset$. We can predict whether a QNS qns is *invalid* by computing the projected value ranges of the quantitative metrics and comparing those to the value ranges given by τ. The more precise the projected value ranges are, the better we can identify and eliminate invalid QNSs.

We use $MinLen(qns)$ to represent the accumulative length of the shortest xe-fragments between every pair of adjacent nodes in qns, i.e. $MinLen(qns) = \sum_{i=0}^{|qns|} minL(q_i, q_{i+1})$, where $minL(q_i, q_{i+1}) = (min_{f \in \mathcal{F}_{xe}(q_i, q_{i+1})} |f|)$.

LEMMA 5.1. *Given a core CAP query with constraint τ, Q is the query node set, then, QNS qns is guaranteed to be invalid if any of the following condition is NOT satisfied*

1. *$\mathcal{F}_{xe}(q_i, q_j) \neq \emptyset$ for all pairs of adjacent nodes in qns; or*
2. *$MinLen(qns) \leq \tau_{l_{max}}$; or*
3. *$\tau_{nc_{min}} \leq \frac{|qns|}{|S|} \leq \tau_{nc_{max}}$; or*
4. *$\tau_{nr_{min}} \leq \frac{|qns|}{MinLen(qns)-1} \leq \tau_{nr_{max}}$ when $MinLen(qns) > 1$.*

Before we can positively identify a QNS as *invalid*, we call it a *candidate* QNS.

In a candidate qns, not all the xe-fragments between all pairs of adjacent query nodes in qns contribute to the join results.

LEMMA 5.2. *Given a QNS qns, q_i and q_{i+1} are adjacent nodes in qns, an xe-fragment $f_{xe} \in \mathcal{F}_{xe}(q_i, q_{i+1})$ can contribute to the join result of $\bowtie_\tau (qns, \mathcal{M}_{\mathcal{F}_{xe}})$ only if*

$$|f_{xe}| \leq \tau_{l_{max}} - MinLen(qns) + minL(q_i, q_{i+1})$$

If an xe-fragment f_{xe} satisfies the condition in Lemma 5.2, we call it a *candidate* xe-fragment for the given qns.

Given a core CAP query $CAP(n_s, n_d, \tau)$, we use $cQNS$ to represent the set of all *candidate* QNSs based on $\mathcal{Q}$. We use $c\mathcal{M}_{\mathcal{F}_{xe}}$ to represent a special $\mathcal{M}_{\widetilde{\mathcal{F}_{xe}}}$ in which each xe-fragment set is a special $\widetilde{\mathcal{F}_{xe}}$, such that each xe-fragment in them is a *candidate* xe-fragment for at least one QNS in $cQNS$.

THEOREM 5.1. *Given a core CAP query $CAP(n_s, n_d, \tau)$,*

$$CAP(n_s, n_d, \tau)(G) = \bigcup_{qns \in cQNS} \bowtie_\tau (qns, c\mathcal{M}_{\mathcal{F}_{xe}})$$

5.2 Search & Join Algorithm

The *Search & Join* (S&J) algorithm has two phases: the *search* phase takes as input the data graph G and the query $CAP(n_s, n_d, \tau)$ and compute $cQNS$ and $c\mathcal{M}_{\mathcal{F}_{xe}}$; the *join* phase then produces the query result using the formula presented in Theorem 5.1. As the join phase can be accomplished easily and efficiently with the help of any relational engine, here we focus our discussion on generating minimum $cQNS$ and $c\mathcal{M}_{\mathcal{F}_{xe}}$ in the search phase.

The intermediate result of the search phase is an $m \times m$ matrix. The two dimensions are nodes in Q_s and Q_d and the value on each coordinate is $\widetilde{\mathcal{F}_{xe}}(q_i, q_j)$, which is initialized to be $\emptyset$ and is always a subset of the target $c\mathcal{F}_{xe}(q_i, q_j)$. We use $\mathcal{M}$ to represent this working matrix, and $\mathcal{M}(q_i, q_j)$ the corresponding $\widetilde{\mathcal{F}_{xe}}(q_i, q_j)$.

The focus in the design of the search phase is to generate minimum $cQNS$ and $c\mathcal{M}_{\mathcal{F}_{xe}}$ and minimize the number of intermediate search steps to produce such results. Rather than issuing one search in the graph to generate each $c\mathcal{F}_{xe}$, we issue one BFS from each node $q \in \mathcal{Q}_s$ to compute all $c\mathcal{F}_{xe}(q, *)$, i.e. a row in $\mathcal{M}$. The BFSs proceed in parallel, so that the intermediate search results of one BFS can be used to limit the search ranges of other BFS searches as well as prune the invalid QNSs, as discussed in Lemma 5.1 and 5.2. Careful bookkeeping is needed to accomplish this.

Given a pair of query nodes (q_i, q_j), we use $minl(q_i, q_j)$ to represent the length of the shortest xe-fragment between q_i and q_j to the best of our knowledge. It is precise and equals to $minL(q_i, q_j)$ if $\mathcal{M}(q_i, q_j)$ is non-empty. Otherwise, the value of $minl(q_i, q_j)$ is estimated by the current BFS search step plus one.

Given a pair of query nodes (q_i, q_j), if there exists a QNS qns, such that the current search step of the BFS starting from q_i is greater than or equal to $\tau_{l_{max}}$ - $MinLen(qns)$+$minL(q_i, q_j)$, based on Lemma 5.2, it is safe to conclude that all candidate xe-fragments between q_i and q_j for qns have been generated. Thus we say that $\mathcal{M}(q_i, q_j)$ generated so far is *complete w.r.t. qns*. We say that $\mathcal{M}(q_i, q_j)$ is *complete* if (1) (q_i, q_j) is not contained by any QNS; or (2) $\mathcal{M}(q_i, q_j)$ generated is complete w.r.t. all QNSs containing (q_i, q_j) such that $\mathcal{M}(q_i, q_j)$ is the desired $c\mathcal{F}_{xe}(q_i, q_j)$. If all $\mathcal{M}(q_i, *)$ are *complete*, it is safe to stop the BFS starting from q_i and we say q_i is *complete*.

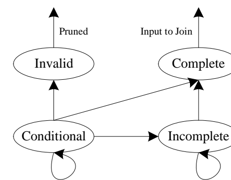

Figure 3: QNS Status Transition

We classify QNSs into four categories: *conditional, incomplete, complete*, and *invalid*. Given a QNS qns, if qns can be identified as *invalid* based on Lemma 5.1, we prune it. Otherwise if $\mathcal{M}(q_i, q_j)$ is complete w.r.t. qns for every adjacent node pair (q_i, q_j) in qns, we say that qns is *complete*. A *complete* qns is ready to be joined. Otherwise, if $\mathcal{M}(q_i, q_j)$ is non-empty for every adjacent node pair in qns, we say that qns is *incomplete*. An incomplete QNS is guaranteed to be a candidate QNS but is not yet ready for join. All other QNSs are *conditional*. In a conditional QNS qns, there is at least one pair of adjacent query nodes (q_i, q_j) in qns such that $\mathcal{M}(q_i, q_j)$ is empty. Thus the candidacy of qns depends on the precise value of $minl(q_i, q_j)$. We say that an incomplete or conditional QNS qns *depends* on a node pair (q_i, q_j) if $\mathcal{M}(q_i, q_j)$ is incomplete w.r.t. qns. The status transformation is illustrated in Fig. 3.

An outline of S&J algorithm is shown in Alg. 2. The search phase starts with $\mathcal{M}$ being empty, all bookkeeping variables initialized. Besides the data structures required by the traditional BFS, every starting query node q_i also maintains a *checklist* that is a set of query nodes q_j with $\mathcal{M}(q_i, q_j)$ being incomplete. The search phase ends when all QNSs are either invalid and pruned or com-

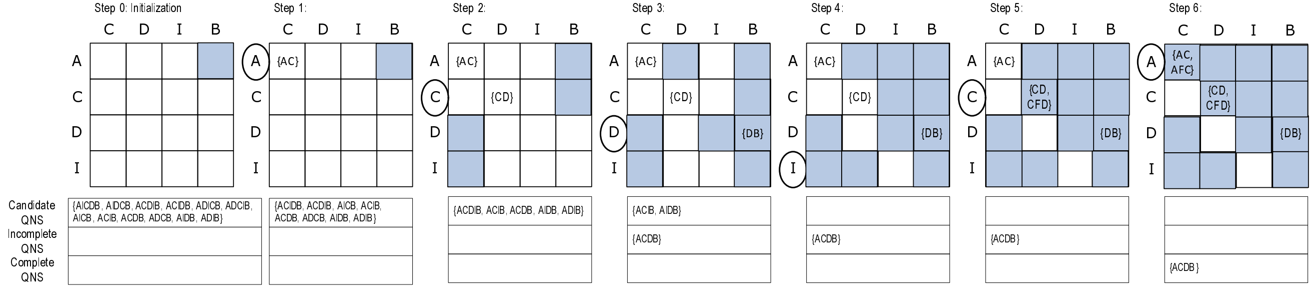

Figure 2: Example of the S&J algorithm

plete and ready for join, as well as $\mathcal{M} = c\mathcal{M}_{\mathcal{F}_{xe}}$. Then the join phase takes the results of the search phase, performs constraint sequence join, and outputs answers to the CAP query. The search phase has three critical operations: Pick, Expand and Adjust.

Algorithm 2 S&J (sketch)

Data: data graph G, source/destination nodes n_s, n_d, and constraint τ.
Result: All acyclic e-fragments from n_s to n_d satisfying τ.
begin
    // search phase
1   initialize $\mathcal{Q}_s$, $\mathcal{Q}_d$, $condQNS$, $incmpQNS$, $cmpQNS$, $\mathcal{M}$
2   **for** $n \in \mathcal{Q}_s$ **do**
3   start BFS from n
4   initialize $status$, $sRange$, $frontier$, $checklist$
5   **while** $(condQns \neq \emptyset\ \&\ incmpQns \neq \emptyset)$ **do**
6   Pick q
7   Expand(q)
8   Adjust
9   return csJoin($cmpQNS$, $\mathcal{M}$) // join phase

Pick: Each time, we pick a query node such that the BFS starting from this node has the potential to prune the maximum number of invalid QNSs and restrict the search ranges of the BFSs most sharply. Our strategy is to pick the query node that is depended by most conditional QNSs. To break a tie, we pick the node that is depended by most incomplete QNSs. If there is still a tie, we pick the most unexpanded node.

Expand: Given the picked node q, the operation expands the BFS search frontier by one step, followed by bookkeeping: if expanding to node n in q's checklist, add the xe-fragment to $\mathcal{M}(q, n)$; otherwise if n is in $\mathcal{Q}$, the branch is pruned; otherwise, insert n into q's next search froniter.

Adjust: After an Expand step on q, we use the information newly obtained to adjust the status of QNSs, the cells in $\mathcal{M}$ and the search ranges of BFSs and other status we maintained for each BFS.

Details of the algorithm are provided in the appendix.

EXAMPLE 5.1. *Let's consider the running example:* $CAP(A, B, \{NodeCoverage(??p, \{C, D, I\}) > 0.6\&Length(??p) \leq 4\})$. *Fig. 2 shows the execution of the search phase of S&J algorithm to answer this query. The nodes in circles are picked to be expanded in each step, and a shaded cell in* $\mathcal{M}$ *indicates that it is complete.*

Let's take step 3 as an example. D is picked with the checklist $\{B, I\}$. *BFS from D reaches node B. As B is in D's checklist, path DB is added to* $\mathcal{M}(D, B)$. *D's next search frontier becomes empty, so D becomes* complete *and will not be picked later. Among the current conditional QNSs, ADIB can be pruned because* $MinLen(ADIB)$ *=minl(A, D)+minl(D, I)+minl(I, B)= 2+2+1=5>* $\tau_{l_{max}}$. *Similarly we prune the ACDIB. As* $\mathcal{M}(A, C)$, $\mathcal{M}(C, D)$ *and* $\mathcal{M}(D, B)$ *are all non-empty, ACDB is upgraded from conditional to incomplete. A's search range is also tightened.*

In this example, the search ranges of A and C are 2 while the search ranges of D and I are both 1. Thus the total number of visited edges and the total number of intermediate results generated by S&J are both much smaller than those in cDFS and ecDFS.

5.3 Discussion

Due to the page limitation, we are only able to discuss the details of cDFS, ecDFS and S&J algorithms for answering core CAP queries. In fact, they can be extended easily to answer all CAP queries in general. For example, the S&J algorithm can be extended to support constraints on both nodes and edges as follows: In the search phase, formula similar to those in Lemma 5.1 and 5.2 can be introduced to calcuate projected value ranges of edge coverage/relevance; In the join phase, by maintaining the total number of keyword edges and number of distinct keyword edges in the xe-fragments, constrained sequence join can also be extended to verify edge coverage/relevance.

6. EXPERIMENTAL EVALUATION

We conducted extensive experiments to study the performance of our algorithms, constraintDFS (*cDFS*), enhanced-constraintDFS (*ecDFS*), and Search-and-Join(*S&J*), as well as existing Search-Filter algorithms based on DFS (*S&F-DFS*) and bidirectional search (*S&F-BIS* [19]). We implemented the push-down of the length constraint in both the S&F algorithms. The experiments were carried out on a desktop PC running Red Hat 4.1.2 with dual Intel(R) Core(TM)2 2.40GHz CPU and 4GB memory.

Datasets and Queries Our experiments were conducted on two RDF datasets that have been widely used in the literature: *DBpedia*[1] and *chem2bio2RDF*[3]. The chem2bio2RDF graph contains 139K nodes and 1.8M edges, while the DBpedia graph contains 1504K nodes and 5.4M edges. As experiments on both datasets showed similar trends, here we report only the experimental results on the chem2bio2RDF dataset.

We test the algorithms on many randomly generated CAP queries, varying source and destination nodes, keyword sets and value constraints on length and other quantitative metrics. As the selection of source/destination nodes does not have significant impact to the performance beyond reasonable impact of data distribution, we report our results on a group of CAP queries of the same source/destination nodes. Based on the constraint τ, we group the queries into the following categories to facilitate comparison:

- Q_l queries have a fixed $\mathcal{S}$ and $\tau_{nc_{min}} = 0.2$ and vary on τ_l;
- Q_{nc} queries have fixed $\tau_{l_{max}}$(=7) and $\mathcal{S}$ and vary on $\tau_{nc_{min}}$;
- Q_{nr} queries have fixed $\tau_{l_{max}}$(=7) and $\mathcal{S}$ and vary on $\tau_{nr_{min}}$;
- Q_{ks} queries have fixed $\tau_{l_{max}}$(=7) and fixed $\tau_{nc_{min}}$ (=0.4), and vary on keyword set $\mathcal{S}$, in terms of both $|\mathcal{S}|$ and the contents in $\mathcal{S}$.

Query Evaluation We compared the hot run of the algorithms. Please note that as our algorithms improve the performance over

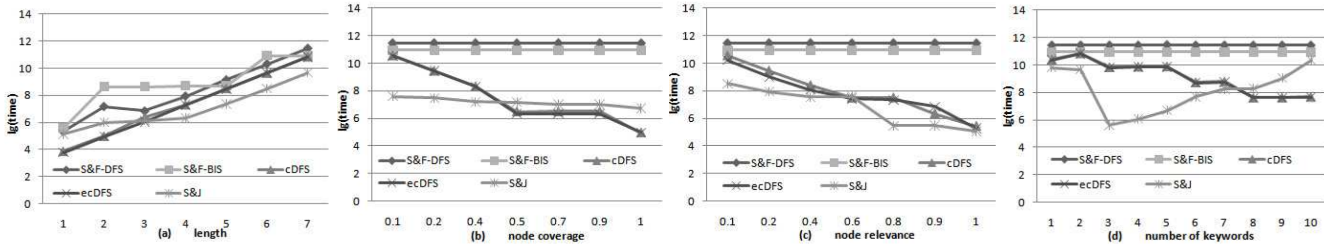

Figure 4: Performance Comparison

the S&F algorithms several orders of magnitude, to better illustrate the difference, we plot the results in logarithmic scale.

As shown in Fig. 4(a), even for queries with simplest keyword constraints, as those in the Q_l family, our algorithms are much more efficient than the Search-Filter algorithms. In addition, ecDFS is more efficient than cDFS because of the saving on validations. The S&J algorithm outperforms cDFS and ecDFS when $\tau_{l_{max}}$ is larger than 3 but is slightly more time-consuming than cDFS and ecDFS when $\tau_{l_{max}}$ is small due to the overhead of generating and maintaining query node sequences.

As shown in Fig. 4(b), keyword-based constraints have no impact on the performance of the Search-Filter algorithms but our algorithms, which take advantage of such constraints, significantly outperform them, especially when $\tau_{nc_{min}}$ is close to 1, as more intermediate results are pruned in such cases in our algorithms. It also worths noticing that our DFS-based algorithms and S&J algorithm behave differently when $\tau_{nc_{min}}$ changes. When the node coverage constraint becomes tight e.g. $\tau_{nc_{min}}$ is closer to 1, cDFS and ecDFS are very efficient, due to their strong pruning power and small overhead. But when the node coverage constraint is relatively relax, e.g. $\tau_{nc_{min}} < 0.5$, the S&J algorithm is able to take advantage of local information around the query nodes to limit the search ranges and thus is much more efficient (two orders of magnitude) than cDFS and ecDFS.

As shown in Fig. 4(c), again, all our algorithms take advantage of node relevance constraint to prune the search branches or limit the search ranges. The larger the $\tau_{nr_{min}}$, the larger the number of search branches can be pruned by our algorithms. Even when $\tau_{nr_{min}}$ is small, the performance of the S&J algorithm is significantly better than others, as it has much smaller search ranges thanks to the local information it takes into account.

As shown in Fig. 4(d), while other constraints are the same, cDFS and ecDFS algorithms benefit from larger number of keywords as constraints, e.g. pruning criteria. For the S&J algorithm, it is all about the tradeoff between the amount of location information that can be used to enhance pruning, and overhead. Many queries we tested indicate that S&J algorithm is at its best with three keywords, it still outperforms the DFS-based algorithms with 1-7 keywords, which in fact, is the size of keyword set of a typical CAP query, based on our study of the application domains.

7. SUMMARY AND FUTURE WORK

In this paper we propose the *Constraint Acyclic Path*(CAP) discovery problem for discovering acyclic paths between two given nodes in a directed graph under constraints. Specifically we propose to specify constraints in terms of path length, and coverage and relevance of resultant paths w.r.t. to a set of keywords. We introduce cSPARQL, an extension of SPARQL, to integrate CAP queries and the structured search on graph data.

We propose a family of algorithms for answering CAP queries: cDFS and ecDFS algorithms enhance DFS by efficiently pruning the search branches based on the projected value ranges of constraint metrics; S&J algorithm further improves the effectiveness of the pruning using local information of the constraint nodes. Our empirical evaluation proved that our algorithms outperform existing Search-Filter algorithms using both DFS and bidirectional search and improve the performance by several orders of magnitude.

We are looking forward to expand the research presented in this paper in the following directions: (1) extend algorithms to support CAP queries with multiple keyword sets and disjunctive/negative predicates; (2) further improve the scalability of our algorithms in terms of graph size, number of keywords and maximum length constraint; and (3) design the query optimization algorithm that optimizes SPARQL graph pattern matching and CAP query answering together to evaluate a cSPARQL query efficiently.

8. REFERENCES

[1] http://wiki.dbpedia.org

[2] B. Dalvi, *et al.* Keyword Search on External Memory Data Graphs). *VLDB Endowment*, 2008.

[3] C. Bin, *et al.* Chem2Bio2RDF: a Semantic Framework for Linking and Data Mining Chemogenomic and Systems Chemical Biology Data *BMC Bioinformatics*, 2010.

[4] D. Abadi, *et al.* SW-Store: a Vertically Partitioned DBMS for Semantic Web Data Management. *The VLDB Journal*, 2009.

[5] E. Prud'hommeaux, *et al.* SPARQL Query Language for RDF. *W3C Recommendation*, 2008.

[6] F. Alkhateeb, *et al.* Constrained Regular Expressions in SPARQL. In *SWWS*, 2008.

[7] F. Alkhateeb, *et al.* Extending SPARQL with Regular Expression Patterns (for Querying RDF). *Web Semant.*, 2009.

[8] G. Bhalotia, *et al.* Keyword Searching and Browsing in Databases using BANKS. In *ICDE*, 2002.

[9] G. Fletcher, *et al.* Scalable indexing of RDF graphs for Efficient Join Processing. In *CIKM*, 2009

[10] G. Li, *et al.* EASE: An Effective 3-in-1 Keyword Search Method for Unstructured, Semi-structured and Structured Data. In *SIGMOD*, 2008.

[11] H. He, *et al.* BLINKS: Ranked Keyword Searches on Graphs). *SIGMOD*, 2007.

[12] J. ruoming, *et al.* Computing Label Constraint Reachability in Graph Databases. In *SIGMOD*, 2010.

[13] K. Anyanwu, *et al.* ρ-Queries: Enabling Querying for Semantic Associations on the Semantic Web. In *WWW*, 2003.

[14] K. Anyanwu, *et al.* SPARQ2L: Towards Support for Subgraph Extraction Queries in RDF Databases. In *WWW*,2007.

[15] K. Anyanwu, *et al.* Structure Discovery Queries in Disk-Based Semantic Web Databases. In *DOI*, 2008.

[16] K. Kochut, *et al.* SPARQLeR: Extended Sparql for Semantic Association Discovery. In *ESWC*, 2007.

[17] L. Sidirourgos, *et al.* Column-store Support for RDF Data Management: not all swans are white. In *VLDB Endowment*, 2008.

[18] T. Jie, *et al.* Efficient Association Search in Social Network. In *WWW*, 2007.

[19] V. Kacholia, *et al.* Bidirectional Expansion For Keyword Search on Graph Databases. In *VLDB*, 2005.

[20] W. Kevin, *et al.* Efficient RDF Storage and Retrieval in Jena2. In *SWDB*, 2003.

Appendix

cSPARQL by Example

We now show how the search requests in Ex. 1.1 case1-6 can be expressed using cSPARQL.

Query 1. (CAP query without constraints)
Ex. 1.1 Case1: Find how *Azriel* connects to *Ben*;

```
SELECT ??p WHERE
{Azriel ??p Ben}
```

Results: The results are all the e-fragments connecting *Azriel* to *Ben* which are omitted due to space limitation.

Query 2. (CAP query with length constraint)
Ex. 1.1 Case2: Find the close ties (within 3 steps) between *Azriel* and *Ben*;

```
SELECT ??p WHERE
{Azriel ??p Ben .
 FILTER(Length(??p)<=3)) }
```

Results: omitted.

Query 3. (CAP query with node intersection constraint)
Ex. 1.1 Case3: Find how *Azriel* connects to *Ben* through *Chris* or *Dan*;

```
SELECT ??p WHERE
{Azriel ??p Ben .
 FILTER(NodeIntersection(??p, {Dan, Chris})) }
```

Result:
$\xrightarrow{foaf} C \xrightarrow{foaf} D \xrightarrow{workfor}$
$\xrightarrow{foaf} C \xrightarrow{coauthor} F \xrightarrow{foaf} D \xrightarrow{workfor}$
$\xrightarrow{foaf} C \xrightarrow{coauthor} F \xrightarrow{workfor} H \xrightarrow{workfor} D \xrightarrow{workfor}$
$\xrightarrow{workfor} F \xrightarrow{foaf} D \xrightarrow{workfor}$
$\xrightarrow{workfor} F \xrightarrow{workfor} H \xrightarrow{workfor} D \xrightarrow{workfor}$
$\xrightarrow{workfor} F \xrightarrow{advisedby} C \xrightarrow{foaf} D \xrightarrow{workfor}$

Query 4. (CAP query with node coverage constraint)
Ex. 1.1 Case4: Find how *Azriel* connects to *Ben* through at least two people from *Chris*, *Dan* and *Ida* within four steps.

```
SELECT ??p WHERE
{Azriel ??p Ben
 CONSTRAINTSET People
    {Chris, Dan, Ida} .
 FILTER(Length(??p)<=4) .
 FILTER(NodeCoverage(??p, People))>0.6 }
```

Results:
$\xrightarrow{foaf} C \xrightarrow{foaf} D \xrightarrow{workfor}$
$\xrightarrow{foaf} C \xrightarrow{coauthor} F \xrightarrow{foaf} D \xrightarrow{workfor}$
$\xrightarrow{foaf} C \xrightarrow{coauthor} F \xrightarrow{foaf} D \xrightarrow{workfor}$
$\xrightarrow{workfor} F \xrightarrow{advisedby} C \xrightarrow{foaf} D \xrightarrow{workfor}$

Query 5. (CAP query with edge context constraint)
Ex. 1.1 Case5: Find *Azriel*'s close (within 4 steps) professional connections (e.g. relationships such as *workfor*, *coworker*, *coauthor* to *Ben*;

```
SELECT ??p WHERE
{Azriel ??p Ben
 CONSTRAINTSET ProfSet
    {coworker, workfor, coauthor} .
 FILTER(Length(??p)<=4) .
 FILTER(EdgeContext(??p, ProfSet)) }
```

Results:
$\xrightarrow{workfor} F \xrightarrow{workfor} H \xrightarrow{workfor} D \xrightarrow{workfor}$

Query 6. (CAP query with edge relevance constraint)
Ex. 1.1 Case6: Find *Azriel*'s close (within 4 steps) semi-professional connections to *Ben* (e.g. half of the relationships in any tie should be professional);

```
SELECT ??p WHERE
{Azriel ??p Ben .
 CONSTRAINTSET ProfSet
          {coworker, workfor, coauthor} .
 FILTER(Length(??p)<=4) .
 FILTER(EdgeRelevance(??p, ProfSet)>=0.5) }
```

Results:
$\xrightarrow{foaf} C \xrightarrow{coauthor} F \xrightarrow{foaf} D \xrightarrow{workfor}$
$\xrightarrow{foaf} C \xrightarrow{coauthor} F \xrightarrow{foaf} D \xrightarrow{workfor}$
$\xrightarrow{workfor} F \xrightarrow{foaf} D \xrightarrow{workfor}$
$\xrightarrow{workfor} F \xrightarrow{advisedby} C \xrightarrow{foaf} D \xrightarrow{workfor}$
$\xrightarrow{workfor} F \xrightarrow{workfor} H \xrightarrow{workfor} D \xrightarrow{workfor}$

Lemma and Proof

LEMMA 4.1. *Given an en-fragment f_n generated in the DFS of $CAP(n_s, n_d, \phi)$ and a keyword set $\mathcal{S}$, for any e-fragment $f_e \in Ext(f_n)$ with $|f_e| > 1$,*

$$\frac{|\mathcal{S} \cap (\lambda(nodes(f_n))|}{|\mathcal{S}|} \leq NodeCoverage(f_e) \leq \begin{cases} \frac{|\mathcal{S}\cap(\lambda(nodes(f_n))|+|f_e|-|f_n|-1}{|\mathcal{S}|} & * \\ 1 & \textit{Otherwise} \end{cases} \quad (1)$$

$$\frac{|\mathcal{S} \cap (\lambda(nodes(f_n))|}{|f_e|-1} \leq NodeRelevance(f_e) \leq \begin{cases} \frac{|\mathcal{S}\cap(\lambda(nodes(f_n))|+|f_e|-|f_n|-1}{|f_e|-1} & * \\ \frac{|\mathcal{S}|}{|f_e|-1} & \textit{Otherwise} \end{cases} \quad (2)$$

$$* \quad |f_e| \leq |\mathcal{S}| + (|f_n| - |\mathcal{S} \cap (\lambda(nodes(f_n))|) + 1$$

PROOF. As $f_e \in Ext(f_n)$, we have $f_n \prec f_e$. The total number of keywords in $\mathcal{S}$ that is covered by f_e is

$$\begin{aligned} |\mathcal{S} \cap \lambda(nodes(f_e))| &= |\mathcal{S} \cap (\lambda(nodes(f_n)) \cup \lambda(nodes(\overline{f_n}))| \\ &= |\mathcal{S} \cap (\lambda(nodes(f_n))| + |\mathcal{S} \cap \lambda(nodes(\overline{f_n}))| \end{aligned}$$

Here, we took advantage of the fact that f_e is an acyclic path, hence $\lambda(nodes(f_n)) \cap \lambda(nodes(\overline{f_n})) = \phi$.

The two extreme cases of $\overline{f_n}$ are

1. $\overline{f_n}$ contains no keyword, e.g. $\mathcal{S} \cap \lambda(nodes(\overline{f_n})) = \phi$; and
2. $\overline{f_n}$ contains maximum number of keywords it can possibly have. In the case that the length of $\overline{f_n}$ is less than the number of keywords in $\mathcal{S}$ not yet covered by f_n, e.g. $|nodes(\overline{f_n})| \leq |\mathcal{S} - \lambda(nodes(f_n))|$ (the same as specified in *), the labels of all nodes in $\overline{f_n}$ are keywords, e.g. $\mathcal{S} \cap \lambda(nodes(\overline{f_n})) = \lambda(nodes(\overline{f_n}))$. Otherwise, $\overline{f_n}$ will cover all keywords not covered by f_n, rendering $|\mathcal{S} \cap \lambda(nodes(f_e))| = |\mathcal{S}|$.

Applying the formula in Def 2.2 to these cases, (1) and (2) are proved. □

Similarly, we can project the upper and lower bounds of edge coverage and relevance of any extension of a given en-fragment.

LEMMA 4.2. *Given an en-fragment f_n generated in the DFS of $CAP(n_s, n_d, \emptyset)$ and a keyword set $\mathcal{S}$, for any e-fragment $f_e \in Ext(f_n)$,*

$$\frac{|\mathcal{S} \cap (\lambda(edges(f_n))|}{|\mathcal{S}|} \leq EdgeCoverage(f_e) \leq \begin{cases} \frac{|\mathcal{S}\cap(\lambda(edges(f_n))|+|f_e|-|f_n|}{|\mathcal{S}|} & ** \\ 1 & \textit{Otherwise} \end{cases} \quad (1)$$

$$** \quad |f_e| \leq |\mathcal{S}| + (|f_n| - |\mathcal{S} \cap (\lambda(edges(f_n))|)$$

$$\frac{\Sigma_{l \in \mathcal{S}} cntE(l, f_n)}{|f_e|} \leq EdgeRelevance(f_e) \leq \frac{\Sigma_{l \in \mathcal{S}} cntE(l, f_n) + |f_e| - |f_n|}{|f_e|} \quad (2)$$

LEMMA 4.3. *While applying cDFS algorithm to answer a core CAP query $CAP(n_s, n_d, \tau)$, if an en-fragment f_{n_m} is deemed*

promising at the m'th step, then in the next k steps, the en-fragment $f_{n_{m+k}}$ with $f_{n_m} \prec f_{n_{m+k}}$ is guaranteed to be promising if

$$\begin{aligned}
&0 \leq k \leq MAX(0, SkippedStep(f_{n_m})) \quad with \\
&SkippedStep(f_{n_m}) \\
&= MIN(l_{max} - |f_{n_m}|, \\
&l_{max} + |\mathcal{S} \cap (\lambda(nodes(f_{n_m}))| - |\mathcal{S}| \times \tau_{nc_{min}} - |f_{n_m}| - 1, \\
&l_{max} + |\mathcal{S} \cap (\lambda(edges(f_{n_m}))| - \tau_{ec_{min}} \times |\mathcal{S}| - |f_{n_m}|, \\
&(1 - \tau_{nr_{min}}) \times l_{max} + |\mathcal{S} \cap (\lambda(nodes(f_{n_m}))| - |f_{n_m}| - (1 - \tau_{nr}), \\
&l_{max} \times (1 - \tau_{er_{min}}) + \Sigma_{l \in \mathcal{S}} cntE(l, f_{n_m}) - |f_{n_m}|)
\end{aligned}$$

PROOF. Let's consider, as an example, the evaluation of a core CAP query in which only $\tau_{nc_{min}}$ and $\tau_{l_{max}}$ are explicitly given. Assume that an en-fragment f_{n_m} is not pruned in the m'th step in DFS; this indicates that we estimate that all $NodeCoverage(f_e) \geq \tau_{nc_{min}}$ for all $f_e \in Ext(f_{n_m})$. Applying Lemma 4.1, we have

$$\frac{|\mathcal{S} \cap (\lambda(nodes(f_{n_m}))| + \tau_{l_{max}} - |f_{n_m}| - 1}{|\mathcal{S}|} \geq \tau_{nc_{min}}$$

e.g. $|f_{n_m}| \leq |\mathcal{S} \cap (\lambda(nodes(f_{n_m}))| + \tau_{l_{max}} - \tau_{nc_{min}} \times |\mathcal{S}| - 1$. k steps later, $f_{n_{m+k}} \succ f_{n_m}$ is generated by including k more nodes. The worse case is that none of the k nodes matches a keyword, thus $|\mathcal{S} \cap (\lambda(nodes(f_{n_{m+k}}))| = |\mathcal{S} \cap (\lambda(nodes(f_{n_m}))|$. For this worst-case $f_{n_{m+k}}$ to be deemed promising, it is required that

$$|f_{n_m}| + k \leq |\mathcal{S} \cap (\lambda(nodes(f_{n_m}))| + \tau_{l_{max}} - \tau_{nc_{min}} \times |\mathcal{S}| - 1$$

The fact that the worst-case $f_{n_{m+k}}$ is deemed promising indicates that all other en-fragments in $Ext(f_{n_m})$ of length m+k should be promising. Hence, at the m'th step, when f_{n_m} is deemed promising, if the formula above also holds, we can predict that within the next k steps, where $k \leq |\mathcal{S} \cap (\lambda(nodes(f_n))| - |f_n| + \tau_{l_{max}} - \tau_{nc_{min}} \times |\mathcal{S}| - 1$, no extension of f_{n_m} will be pruned.

Similarly criteria can be established for constraints on other quantitative metrics, e.g. other components in the formula. ☐

LEMMA 5.1. *Given a core CAP query with constraint τ, Q is the query node set, then, QNS qns is guaranteed to be invalid if any of the following condition is NOT satisfied.*

1. *$\mathcal{F}_{xe}(q_i, q_j) \neq \emptyset$ for all pairs of adjacent nodes in qns;*
2. *$MinLen(qns) \leq \tau_{l_{max}}$; or*
3. *$\tau_{nc_{min}} \leq \frac{|qns|}{|\mathcal{S}|} \leq \tau_{nc_{max}}$; or*
4. *$\tau_{nr_{min}} \leq \frac{|qns|}{MinLen(qns)-1} \leq \tau_{nr_{max}}$.*

PROOF. Condition (1) is straightforward.

We use $MinLen(qns)$ to represent the accumulative length of the shortest path among the xe-fragments between every pairs of adjacent nodes in qns, e.g. $MinLen(qns) = \sum_{i=0}^{|qns|-1} minL(q_i, q_j)$, where $minL(q_i, q_j) = (min_{f \in \mathcal{F}_{xe}(q_i, q_{i+1})} |f|)$. Therefore, qns is guaranteed to be invalid if $MinLen(qns) > \tau_{l_{max}}$.

Projected values ranges on other quantitative metrics can be considered and condition (3)(4) be proved in similar manner. ☐

LEMMA 5.2. *Given a QNS qns, an xe-fragment $f_{xe} \in \mathcal{F}_{xe}(q_i, q_{i+1})$ (i<$|qns|$) can contribute to the join result of $\bowtie_\tau (qns, \mathcal{M}_{\mathcal{F}_{xe}})$ only if*

$$|f_{xe}| \leq \tau_{l_{max}} - MinLen(qns) + minL(q_i, q_{i+1})$$

Algorithm 3 ecDFS

Data: data graph G, course node n_s, destination node n_d, constraint τ.
Result: All acyclic e-fragments from n_s to n_d satisfying τ.

```
begin
1    Array results ← {}
2    HashTable VisitedNodes ← {}
3    VisitedNodes.Add(n_s)
4    Array<Edge> f_n ← ()
5    Stack Stk ← {}
6    Stack Stk_skipped ← {}
7    cur_steps ← 1
8    num_skipped_step ← 0
9    for e in n_s's outgoing edges do
10       Stk.Push(e)
11   while !Stk.isEmpty() do
12       Edge e ← Stk.Pop()
13       if e==(null,null) then
14           Edge tailEdge ← f_n.RemoveTailEdge()
15           VisitedNodes.Remove(tailEdge.endNode)
16           cur_steps − −
17           if cur_steps ≤ 0 then
18               num_skipped_step = Stk_skipped.Pop()
19               cur_steps = num_skipped_step
20           Continue
21       if !VisitedNode.Contains(e.endNode) then
22           f_n.Concatenate(e)
23           booleanflag ← true
24           if e.endNode == B then
25               if f_n satisfies constraints then
26                   results.add(f_n.clone())
27           else
28               if cur_step > num_skipped_step then
29                   Array projRanges = computeProjRange(f_n, τ)
30                   if Overlapped(projRanges, τ) then
31                       Stk_skipped.Push(num_skipped_step)
32                       num_skipped_step ← SkippedStep(f_n)
33                       cur_steps = 0
34                   else
35                       flag ← false
36               if flag==true then
37                   VisitedNodes.Add(e.endNode)
38                   Stk.Push((null, null))
39                   for e' in e.endNode's outgoing edges do
40                       Stk.Push(e')
41                   cur_steps + +
42                   Continue;
43           f_n.RemoveTailEdge()
44   Return results;
```

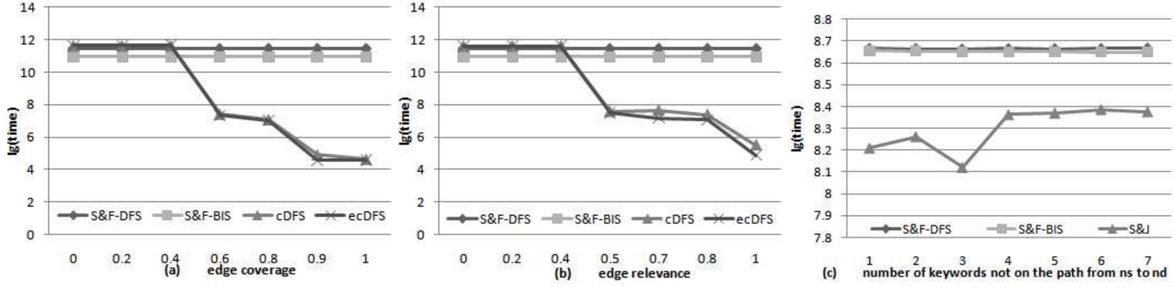

Figure 5: Additional Performance Comparison

PROOF. Given an xe-fragment $f_{xe} \in \mathcal{F}_{xe}(q_i, q_{i+1})$ where q_i and q_{i+1} are two adjacent query nodes in qns, assume $f_e \in\bowtie_\tau (qns, \mathcal{M}_{\mathcal{F}_{xe}})$ and f_{xe} is a subsequence of f_e. Based on the concatenation operation in Def. 5.1, $|f_e| > |f_{xe}| + \sum_{i=0}^{i-1} minL(q_i, q_{i+1}) + \sum_{i=i+1}^{|qns|-1} minL(q_i, q_{i+1})$. Based on the selection operation in Def. 5.1, $|f_e| < \tau_{l_{max}}$. Therefore, $|f_{xe}| \leq \tau_{l_{max}} - MinLen(qns) + minL(q_i, q_{i+1})$. □

Algorithm ecDFS

ecDFS Algorithm The ecDFS algorithm, as shown in Alg. 3, is based on the cDFS algorithm 1, with additional bookkeeping to enable the skip of computation and verification of some en-fragments.

S&J Algorithm

Here, we provide more details about the search phase of the S&J algorithm.

Associated with each BFS starting from a node q, besides the information to be kept for a classic BFS, such as current search step, the search frontier, and the intermediate results (in $\mathcal{M}$), we keep track of the following information with the sole purpose being minimizing final results ($cQNS$ and $c\mathcal{M}_{\mathcal{F}_{xe}}$) and minimizing intermediate steps in generating such results: (1) the *status* which is initialized to be incomplete, will become complete if all $\widetilde{\mathcal{F}_{xe}}(q,*)$'s are complete or its search frontier is empty; (3) *search range* which is initialized as $\tau_{l_{max}}$, will become more restricted as the BFS proceeds; and (3) *checklist* (CL_q) which includes all query node q_j such that $\mathcal{M}(q_i, q_j)$ is incomplete.

An outline of the search phase of the S&J algorithm is shown in Alg. 2. It starts with $\mathcal{M}$ being empty, all bookkeeping variables initialized as discussed above. It ends when all QNSs are either invalid and pruned or complete. We now discuss the details of the three critical operations: Pick, Expand and Adjust.

Pick: Each time, we pick a query node such that advancing the BFS starting from this node has the potential to prune the maximum number of invalid QNSs and restrict the search ranges of the BFSs of itself and other query nodes most sharply. Our strategy is to pick the query node that is depended by most conditional QNSs. To break a tie, we pick the node that is depended by most incomplete QNSs. If there is still a tie, we pick the most unexpanded node.

Expand: Given the picked node q, invoking the expand operation results in advancing one step further from every node in q's frontier in a BFS manner, followed by bookkeeping: if expanding to node n leads to the discovery of a new xe-fragment, e.g. n in q's checklist, add the xe-fragment to $\mathcal{M}(q, n)$; otherwise if n is in $\mathcal{Q}$, the branch is pruned; otherwise, insert n into q's next search froniter.

Adjust: After a Expand step on q, we use the information newly obtained to adjust the following status:

1. *The status of QNSs*: With an addition to $\mathcal{M}$, we may be able to upgrade a QNS from conditional to complete or from incomplete to complete. Any failure to add new e-fragment to $\mathcal{M}$ when a BFS reaches a certain depth may render a QNS invalid and pruned. We check all QNSs that depends on node pairs whose corresponding cell in $\mathcal{M}$ or whose $minl$ has been changed by the Expand process, and adjust the status of these QNSs and their dependency to node pairs accordingly.
2. *the status of a cell in $\mathcal{M}$ and the BFSs*: As the dependency of QNSs on node pairs changes, the cells in $\mathcal{M}$ that correspond to these node pairs may change from incomplete to complete. And in a cascade effect, mark a node to be complete and BFS to be complete and terminated.
3. *The search range and check list of BFS*: For any BFSs that are not yet to be terminated, we adjust search range and check list to tighten constraints on further expansion. For any new frontier n introduced in the Expand process, we check every QNS qns depending on (q, n) and update the search range of the BFS starting from all other nodes in qns. Please note that the search range of a BFS may be impacted by both node pair (q, n) and (q, n'), with both n and n' the newly added frontiers. In this case, we pick the tightest bound to be the new search range for the BFS. Checklist is updated to reflect the newly established/eliminated dependency between QNSs and node pairs.

Additional Experimental Results

Besides the results presented in the paper, we also conduct experiments on the following sets of queries to better understand the performance of the algorithms we propose.

- Q_{ec} queries queries have fixed $\tau_{l_{max}}$(=7) and $\mathcal{S}$, and vary on $\tau_{ec_{min}}$;
- Q_{er} queries have fixed $\tau_{l_{max}}$(=7) and $\mathcal{S}$, and vary on $\tau_{er_{min}}$;
- Q_{kp} queries have fixed length constraint ($\tau_{l_{max}} = 7$), fixed node coverage constraint ($\tau_{nc_{min}} = 0.4$) and fixed $\mathcal{S}$ with $|\mathcal{S}| = 10$, but vary on the keywords and include some that do not appear on any path from n_s to n_d.

The Fig. 5(a) and Fig. 5(b) show that our proposed approaches, cDFS and ecDFS, can take advantage of edge coverage and relevance constraint to efficiently evaluation CAP queries.

As shown in Fig. 5(c), the number of unrelated keywords increases, the performance of cDFS and ecDFS keeps the same, while the performance of S&J algorithm decreases because searching the neighborhood of unrelated constraint nodes adds to the cost but not contributing to the results.