LIGER – Link Discovery with Partial Recall

Kleanthi Georgala[1,2], Mohamed Ahmed Sherif[2], and Axel-Cyrille Ngonga Ngomo[1,2]

[1] Department of Computer Science, Paderborn University, Germany,
[2] Department of Computer Science, University of Leipzig, Germany
georgala@informatik.uni-leipzig.de
{mohamed.sherif,axel.ngonga}@upb.de

Abstract. Modern data-driven frameworks often have to process large amounts of data periodically. Hence, they often operate under time or space constraints. This also holds for Linked Data-driven frameworks when processing RDF data, in particular, when they perform link discovery tasks. In this work, we present a novel approach for link discovery under constraints pertaining to the expected recall of a link discovery task. Given a link specification, the approach aims to find a subsumed link specification that achieves a lower run time than the input specification while abiding by a predefined constraint on the expected recall it has to achieve. Our approach, dubbed LIGER, combines downward refinement operators with monotonicity assumptions to detect such specifications. We evaluate our approach on seven datasets. Our results suggest that the different implementations of LIGER can detect subsumed specifications that abide by expected recall constraints efficiently, thus leading to significantly shorter overall run times than our baseline.

1 Introduction

Sensor data is used in a plethora of modern applications, including condition monitoring and predictive maintenance in Industry-4.0 applications [3],[3] environmental protection applications, health monitoring systems[4] and traffic monitoring [8]. An increasing number of these machine implement intelligent predictive maintenance and condition monitoring by generating knowledge graphs in the Resource Description Framework (RDF) format [10]. This representation format facilitates the implementation of condition monitoring and predictive maintenance applications as explainable machine learning solutions [9], which learn and update OWL axioms periodically to detect (condition monitoring) or predict (predictive maintenance) error events [3]. A key step for learning axioms which generalize well is to learn them over across several machines. However, single machines generate independent data streams. Hence, time-efficient data integration (in particular link discovery, short LD) approaches must precede the machine learning approaches to render integrated machine learning over data streams from several machines possible. Given that new data batches are available periodically (e.g., every

[3] `https://katana.usu.de/en/smart-service/`
[4] `https://www.healthcare.siemens.co.uk/magnetic-resonance-imaging/options-and-upgrades/clinical-applications/interactive-realtime-imaging`

2 hours), practical applications of machine learning on RDF streams demand LD solutions which can guarantee the completion of their computation under constraints such as time (i.e., their total runtime for a particular integration task) or expected recall (i.e., the estimated fraction of a given LD task they are guaranteed to complete). In this paper, we address the *problem of integrating streams of RDF data under constraints pertaining to expected recall*. We dub this type of LD *partial-recall LD* (a formal definition is given in Section 2).

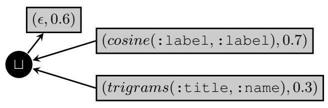
In this paper, we address the problem of LD with partial recall by proposing LIGER (Link discovery with Guaranteed Expected Recall), the first *partial-recall LD approach*. Given a link specification L that is to be executed, LIGER aims to compute a portion of the links returned by L efficiently, while achieving a guaranteed expected recall (see Section 2 for a formal definition). Our approach relies on a refinement operator, which allows the exploration of potential solutions to this problem efficiently. The main contributions of our work can be summarized as follows: (1) We present a downward refinement operator that allows the detection of subsumed LSs with partial recall. We use this operator to develop the first approach for partial-recall LD. (2) We use a monotonicity assumption to improve the time efficiency our approach even further. (3) We evaluate our approach using four benchmark datasets as well as three new datasets based on real data. In addition to an intrinsic evaluation, we also provide an extrinsic evaluation of our approach to quantify the effect of partial-recall LD on positive-only learning for LD.

2 Preliminaries

In this section, we introduce the fundamental concepts that are necessary to understand the rest of the paper. We present primarily the LD problem by providing a formal definition of its basic concepts in a fashion akin to [16]. A knowledge base K is a set of triples $(s, p, o) \in (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$, where $\mathcal{I}$ is the set of all Internationalized Resource Identifiers (IRIs) $\mathcal{B}$ is the set of all RDF blank nodes and $\mathcal{L}$ is the set of all literals. Given two sets of RDF resources S and T from two (not necessarily distinct) knowledge bases as well as a relation R, the main goal of LD is to discover the *mapping* $\mu = \{(s, t) \in S \times T : R(s, t)\}$. To achieve this goal, declarative LD frameworks rely on link specifications (LSs), which describe the conditions under which $R(s, t)$ can be assumed to hold for pairs $(s, t) \in S \times T$.

Several grammars have been used for describing a LS in previous works [12]. We borrow the approach used to specify query languages such as SPARQL and begin with the definition of the syntax of LSs. In general, these grammars assume that a LS consists of two types of atomic components: (1) *Similarity measures* m, through which property values of resources found in the input datasets S and T can be compared. Let M be the set of all similarity functions and $\mathcal{P}$ be the set of all properties. All similarity functions are either atomic or complex. We define an atomic similarity function $m \in M$ as a function $m : S \times T \times \mathcal{P}^2 \rightarrow [0, 1]$. A *complex* similarity function if it is a combination of two similarity functions using a *metric operator* (e.g., max, min). An *atomic LS* is a pair (m, θ) where $\theta \in [0, 1] \cup \{\varnothing\}$ and m is a similarity function. We denote the empty specification with $L_\emptyset$. *Filters* are pairs (f, τ), where (1) f is either empty (denoted

Fig. 1: Example of a complex LS dubbed L_0.



$(\epsilon, 0.6)$
$\sqcup$
$(cosine(\texttt{:label}, \texttt{:label}), 0.7)$
$(trigrams(\texttt{:title}, \texttt{:name}), 0.3)$

Table 1: Syntax and Semantics of LS.

L	$[[L]]$
(m, θ)	$\begin{cases} \{(s,t,m(s,t)) : (s,t) \in S \times T \wedge m(s,t) \geq \theta\} \text{ if } \theta \in [0,1] \\ \emptyset \text{ else.} \end{cases}$
(f, τ, L)	$\begin{cases} \{(s,t,r) \in [[L]] : r \geq \tau\} \text{ if } f = \epsilon \\ \{(s,t,r) \in [[X]] : f(s,t) \geq \tau\} \text{ else.} \end{cases}$
$L_1 \sqcap L_2$	$\{(s,t,r) \mid (s,t,r_1) \in [[L_1]] \wedge (s,t,r_2) \in [[L_2]] \wedge r = \min(r_1, r_2)\}$
$L_1 \backslash L_2$	$\{(s,t,r) \mid (s,t,r) \in [[L_1]] \wedge \neg \exists r' : (s,t,r') \in [[L_2]]\}$
$L_1 \sqcup L_2$	$[[L_1 \backslash L_2]] \cup [[L_2 \backslash L_1]] \cup \{(s,t,r) \mid (s,t,r_1) \in [[L_1]] \wedge (s,t,r_2) \in [[L_2]] \wedge r = \max(r_1, r_2)\}$
$L_\emptyset$	$\emptyset$

ϵ) or a similarity measure and (2) $\tau \in [0,1]$ is a threshold. *Specification operators op* combine two LSs L_1 and L_2 to a more complex specification $L = (f, \tau, op(L_1, L_2))$. In this paper, we limit ourselves to formalizing the specification operators $\sqcap$ (intersection of LSs), $\sqcup$ (union of LSs) and $\backslash$ (difference of LSs). We call L_1 the *left subspecification* and L_2 the *right subspecification* of L. For $L = (f, \tau, op(L_1, L_2))$, we call *op the operator* of L. We call (f, τ) the *filter of* L and denote it with $\varphi(L)$. The *size of* L, denoted $|L|$, is defined as follows: If L is atomic, then $|L| = 1$. For complex LSs $L = (f, \tau, \omega(L_1, L_2))$, we set $L = |L_1| + |L_2| + 1$. Consider the example specification L_0 depicted in Figure 1. The left child of L_0 is L_1 = (*trigrams*(`:title`, `:name`), 0.3). Note that L_1 is atomic. The operator of L_0 is $\sqcup$. The filter of L_0 is $(\epsilon, 0.6)$ and L_0 has a size of 3.

We define the mapping $[[L]] \subseteq S \times T$ of the LS L as the set of links that will be computed by L when applied to $S \times T$ as shown in Table 1. We denote the size of a mapping $[[L]]$ by $|[[L]]|$. Additionally, we define the selectivity of a link specification L as $sel(L) = |[[L]]|/|S \times T|$. The aim of sel is to encode the predicted value of $|[[L]]|$ as a fraction of $|S \times T|$. This is akin to the selectivity definition often used in the database literature. A specification L' is said to achieve a recall k w.r.t. to L if $k \times |[[L]]| = |[[L]] \cap [[L']]|$. If $[[L']] \subseteq [[L]]$, then the recall k of L' abides by the simpler equation $k \times |[[L]]| = |[[L']]|$. A specification L' with $[[L']] \subseteq [[L]]$ is said to achieve an expected recall k w.r.t. to L if $k \times sel(L) = sel(L')$.

Definition 1. *Partial-Recall LD. Given a specification L, the aim of partial-recall LD is to detect a rapidly executable LS $L' \sqsubseteq L$ with an expected recall of at least* $k \in [0,1]$, *i.e. a LS L' with* $sel([[L']]) \geq k \times sel([[L]])$*, where* $k \in [0,1]$ *is a minimal expected recall set by the user.*

3 Linking with Guaranteed Expected Recall

In this section, we present our approach to achieving a guaranteed expected recall when provided with an input LS L.

Definition 2 (LS Subsumption). *The LS L' is subsumed by the LS L (denoted $L \sqsubseteq L'$) when* $[[L]] \subseteq [[L']]$ *for any fixed pair of sets S and T.*

Note that $\sqsubseteq$ is a quasi-ordering (i.e., reflexive and transitive) on the set of all LS, which we denote $\mathcal{LS}$. A key observation that underlies our approach is as follows:

Proposition 1. $\forall \theta, \theta' \in [0,1]\ \theta > \theta' \rightarrow (m, \theta) \sqsubseteq (m, \theta')$.

Proof. This is due to the definition of the subsumption relation as $m(s,t) \geq \theta$ and $\theta > \theta'$ implies that $m(s,t) \geq \theta'$ by virtue of the strict ordering on numbers in $\mathbb{R}$.

This observation can be extended to LS as follows: (1) $L_1 \sqsubseteq L'_1 \rightarrow (L_1 \sqcup L_2) \sqsubseteq (L'_1 \sqcup L_2)$, (2) $L_1 \sqsubseteq L'_1 \rightarrow (L_1 \sqcap L_2) \sqsubseteq (L'_1 \sqcap L_2)$, (3) $L_1 \sqsubseteq L'_1 \rightarrow (L_1 \backslash L_2) \sqsubseteq (L'_1 \backslash L_2)$, and (4) $L_2 \sqsubseteq L'_2 \rightarrow (L_1 \backslash L'_2) \sqsubseteq (L_1 \backslash L_2)$

Definition 3 (Refinement Operator). *$(\mathcal{LS}, \sqsubseteq)$ is a quasi-ordered space. We call $\rho : \mathcal{LS} \rightarrow 2^{\mathcal{LS}}$ a* downward refinement operator *if $\forall L \in \mathcal{LS} : L' \in \rho(L) \rightarrow L' \sqsubseteq L$. L' is called a* specialisation *of L. We denote $L' \in \rho(L)$ with $L \rightsquigarrow_\rho L'$.*

Given L as input, the idea behind our approach is to use a refinement operator to compute $L' \sqsubseteq L$ with at least a given expected recall k w.r.t L. We define the corresponding refinement operator over the space $(2^{\mathcal{LS}}, \sqsubseteq)$ as follows:

$$\rho(L) = \begin{cases} \emptyset & \text{if } L = L_\emptyset, \\ L_\emptyset & \text{if } L = (m, 1), \\ (m, next(\theta)) & \text{if } L = (m, \theta) \wedge \theta < 1, \\ (\rho(L_1) \sqcup L_2) \cup (L_1 \sqcup \rho(L_2)) & \text{if } L = L_1 \sqcup L_2, \\ (\rho(L_1) \sqcap L_2) \cup (L_1 \sqcap \rho(L_2)) & \text{if } L = L_1 \sqcap L_2, \\ \rho(L_1) \backslash L_2 & \text{if } L = L_1 \backslash L_2. \end{cases} \quad (1)$$

This operator works as follows: If L is the empty specification $L_\emptyset$, then we return an empty set of LS, ergo, L is not refined any further. *If L is an atomic specification* with a threshold of 1, our approach returns $L_\emptyset$. By these means, we can compute refinement chains from $L = L_1 \sqcup L_2$ to $\{L_1, L_2\}$. If $\theta < 1$, our approach alters the threshold θ by applying the $next$ function. This $next$ function is based on the insight that for $\theta < 1$ and any pair of datasets S and T, if there is a smallest threshold $\theta' > \theta$ that leads to $[[(m, \theta')]] \subset [[(m, \theta)]]$, then θ' is a finite non-zero positive real number. This is exactly the value that $next(\theta)$ returns for each metric if θ' exists. If θ' does not exist, $[[(m, \theta)]]$ is the same as $[[(m, 1)]]$. Then, $next$ returns $\varnothing$, which is evaluated like $L_\emptyset$. Formally, let $N(m) = \{n : \exists (s,t) \in S \times T : n = m(s,t)\} \cup \{\varnothing\}$. $next$ returns the smallest value from $N(m)$ that is larger than θ. If no such value exists, then $\varnothing$ is returned. Note that $(m, next(\theta)) \sqsubseteq (m, \theta)$ always holds.

Definition 4 (Properties of refinement operators). *A refinement operator r over the quasi-ordered space $(\mathcal{S}, \preccurlyeq)$ can abide by the following criteria:* ***(1) Finiteness.*** *r is finite iff $r(s)$ is finite for all $s \in \mathcal{S}$;* ***(2) Properness.*** *r is proper if $\forall s \in \mathcal{S}, s' \in r(s) \Rightarrow s \neq s'$;* ***(3) Completeness.*** *r is said to be complete if for all s and s', $s' \preccurlyeq s$ implies that there is a s'' with $s'' \preccurlyeq s' \wedge s' \preccurlyeq s''$ such that a refinement chain between s'' and s exists;* ***(4) Redundancy.*** *A refinement operator r over the space $(\mathcal{S}, \preccurlyeq)$ is redundant if two different refinement chains can exist between $s \in \mathcal{S}$ and $s' \in \mathcal{S}$.*

Our refinement operator ρ is finite, incomplete, proper and redundant if L, S and T are finite. ρ being incomplete is not a restriction for our purposes given that we aim to find Ls that run faster and thus do not want to refine the input LS L to L' that might make our implementation of the operator slower. The meaning of the other characteristics for our implementation of ρ is as follows: Given that ρ is *finite*, we can generate ρ for any chosen node completely in our implementation. *ρ being redundant* means that after a refinement, we need to check whether we have already seen the newly generated LS. Hence, we need to keep a set of seen LS. Finally, the *operator being proper* means that while checking for redundancy, there is no need to compare LS with any of their parents.

4 Approach

Let L_0 be a LS and $\rho^*(L_0)$ represent the set of all LSs that can be reached from L_0 via ρ. The basic goal behind LIGER is to find the LS $\Lambda \in \rho^*(L_0)$ that achieves the lowest expected run time while (1) being subsumed by L_0 and (2) achieving at least a predefined excepted recall $k \in [0, 1]$. LIGER is based on ρ as described in Section 3.We present the LIGER algorithm and corresponding extensions in the subsequent paragraphs.

Algorithm 1 shows the steps of the basic implementation of LIGER. We dub this implementation C-RO. Our approach takes (1) a LS L_0, (2) one input source KB S and one input target KB T, (3) an oracle O which can predict the run times and selectivity of LS—such an oracle is presented in [6]—, (4) the minimal expected recall k and (5) a refinement time constraint *maxOpt* as input. We begin by asking O to provide the algorithm with estimations of the selectivity of L_0 (sel_{L_0}, see line 1 of Algorithm 1). We then initialize the best subsumed LS Λ with L_0 and the best run time rt_Λ with L_0's runtime estimation from O (lines 1, 2). The algorithm computes the desired selectivity value (sel_{des}) as a fraction of L_0's selectivity (line 3). Then, we add L_0 to the set $Buffer$ (line 1). This set serves as a buffer and includes LSs obtained by refining L_0 that have not yet been refined. All LSs that were generated through the refinement procedure as well as the input LS L_0 are stored in $Total$ (line 1). By keeping track of these LSs, we avoid refining a LS more than once and address the redundancy of our refinement operator.

The main loop starts in line 4 and runs until the termination criterion is met, i.e., until the refinement time has exceeded *maxOpt*, the refinement tree cannot be explored further ($Buffer = \emptyset$) or the selectivity of Λ returned by O is equal to sel_{des}. We define the refinement tree as follows: (1) it has L_0 as its root, (2) at each iteration of Algorithm 1, the set of refined LSs are added as children nodes to the currently refined LS, and (3) a leaf node is as a LS that can not be refined any further. At the beginning of each iteration, the algorithm selects the next node for refinement by calling the function $getNextNode(Buffer, O, S, T)$(Algorithm 5). $getNextNode(Buffer, O, S, T)$ computes a complete ordering of $Buffer$ with respect to runtime estimations. Algorithm 2 illustrates the steps of the algorithm. Initially, the $getNextNode(Buffer, O)$ algorithm retrieves the run time estimation (rt_L) of each LS $L \in Buffer$ (line 2) using O and adds the pair (L, rt_L) to the set $Nodes$. Then, if $Nodes$ is not empty, it orders the LS of $Nodes$ based on the run time scores in ascending order and sets to L_{next} the first element of the set (lines 6, 7 resp. of Algorithm 2). If $Nodes$ is empty, then Algorithm 1 terminates (line 6 of Algorithm 1).

Algorithm 1 then checks if the current LS L receives a better runtime score, the algorithm assigns L as the new value of Λ and changes rt_Λ accordingly (line 8).

Algorithm 1: LIGER Algorithm

```
1  $\Lambda \leftarrow L_0, Buffer \leftarrow \{L_0\}$,
     $Total \leftarrow \{L_0\}$
2  $rt_\Lambda \leftarrow O.getRT(L_0, S, T)$
3  $sel_{des} \leftarrow O.getSel(L_0, S, T) \times k$
4  while termination criterion not met do
5      $L \leftarrow$
         $getNextNode(Buffer, O, S, T)$
6      if $L = null$ then break ;
7      $rt_L \leftarrow O.getRT(L, S, T)$
8      if $rt_L < rt_\Lambda$ then $\Lambda \leftarrow L$,
         $rt_\Lambda \leftarrow rt_L$ ;
9      $newLSs \leftarrow refine(L, O, S, T)$
10     if $newLSs \neq \emptyset$ then
11         $update(newLSs, Total, O, Buffer,$
12         $sel_{des}, S, T)$
13     $Buffer \leftarrow Buffer \backslash L$
14 Return $\Lambda$
```

Algorithm 2: $getNextNode(Buffer, O, S, T)$ for C-RO

```
1  $Nodes \leftarrow \emptyset, L_{next} \leftarrow null$
2  foreach $L \in Buffer$ do
3      $rt_L \leftarrow O.getRT(L, S, T)$
4      $Nodes \leftarrow Nodes \cup (L, rt_L)$
5  if $Nodes \neq \emptyset$ then
6      $Nodes \leftarrow order(Nodes)$
7      $L_{next} \leftarrow Nodes.getTop()$
8  Return $L_{next}$
```

Algorithm 3: $getNextNode(Total, Buffer, O, S, T)$ for RO-MA

```
1  $L_r \leftarrow Total.getLatestNode()$
2  $lvl_{L_r} \leftarrow Total.getLevel(L_r)$
3  if $lvl_{L_r} == 0$ then
4      $L_{next} \leftarrow Buffer.getTop()$
5      Return $L_{next}$
6  $lvl \leftarrow lvl_{L_r} + 1$
7  $L_{next} \leftarrow null$
8  while $lvl \neq 0$ do
9      $SubTree \leftarrow \emptyset$
10     foreach $L \in Buffer$ do
11         if $lvl_L == lvl$ then
12             $SubTree \cup L$
13     if $SubTree \neq \emptyset$ then
14         $Nodes \leftarrow \emptyset$
15         foreach $L \in SubTree$ do
16             $rt_L \leftarrow$
                 $O.getRT(L, S, T)$
17             $Nodes \leftarrow$
                 $Nodes \cup (L, rt_L)$
18         $Nodes \leftarrow order(Nodes)$
19         $L_{next} \leftarrow Nodes.getTop()$
20         break;
21     else $lvl \leftarrow lvl - 1$ ;
22 Return $L_{next}$
```

In Algorithm 9, the algorithm calls the function $refine(L, O, S, T)$, which implements ρ. The results of the refinement are stored in $newLSs$. Once $newLSs$ has been retrieved (line 9), Algorithm 1 check which subsumed LS(s) of $newLSs$ can be refined in the future. To this end, it calls the function $update(newLSs, Total, O, Buffer, sel_{des}, S, T)$ (line 12). This methods checks that each LS $L' \in newLSs$ has not been explored before by checking if it already exists in set $Total$. Therewith, we ensure that LIGER does not explore LSs that have already been seen before. If L' is a new node, it is added to $Total$ and the algorithm proceeds in computing L''s selectivity. If $sel_{L'}$ is higher or equal to the desired selectivity, the algorithm updates $Buffer$ by adding L'. [5]. Finally, in Algorithm 1, after L has been refined, it is excluded from $Buffer$ (line 13), so it will not be refined in the future.

Extension of LIGER. One key observation pertaining to the run time of $L' \in \rho^*(L)$ is that by virtue of $L' \sqsubseteq L$, $RT(L') \leq RT(L)$ will most probably hold. By virtue of the transitivity of $\leq$, $L_1 \in \rho(L) \wedge L_2 \in \rho(L) \wedge RT(L_1) \leq RT(L_2) \rightarrow \forall L' \in$

[5] Both $Buffer$ and V are passed by reference and updated within the *addNewLS* function

$\rho^*(L_1)$: $RT(L') \leq RT(L_2)$ also holds. We call this assumption the *monotonicity of run times*. Since the implementation of Algorithm 2 for LIGER does not take this monotonicity into consideration, we wanted to know whether this assumption can potentially improve the run time of our approach. A direct consequence of this assumption would be the following:

Proposition 2. *Let L_1 be a leaf of the refinement tree at the distance d from the root of ρ's refinement tree, then for all leaves L_2 at a distance $d' > d$ from the root, $RT(L_1) \geq RT(L_2)$.*

To integrate it into LIGER, we created the extension of LIGER dubbed RO-MA (Refinement Operator with Monotonicity Assumption). RO-MA overwrites the `getNextNode (Buffer,O,S,T)` function with `getNextNode(Total,Buffer,O,S,T)` (line 5 of Algorithm 1) by using a hierarchical ordering on the set of unrefined nodes. By incorporating RO-MA as a search strategy, the refinement tree is expanded using a "top-down" approach until there are no nodes to be further explored in a particular path. Algorithm 3 describes the procedure used to find the next node for refinement. Initially, the algorithm retrieves the level (i.e., the distance from the root of the refinement tree, denoted lvl_{L_r}) of the most recently defined LS L_r and increases it by 1. The first time Algorithm 3 is called, the only node included in $Total$ will be L_0 with $lvl_{L_r} = 0$, since it is the root of the refinement tree. In this case, the algorithm will return L_0 to be refined (line 3). The main loop begins in line 8 and continues until lvl is equal to 0 (i.e., until the level of the root). Then, we find the set of nodes at level lvl, that have not be refined before (line 11). If such a subset exists (line 13), then the algorithm retrieves the run time estimation of each LS $L \in SubTree$ using O, and adds to the set $Nodes$. Then, it orders the LSs of $Nodes$ based on the run time scores in ascending order (line 18) and sets to L_{next} the first element of the set (line 19). If all the nodes of lvl have been refined before, then the search for the next LS continues at level higher. If the root's level has been reached, then Algorithm 1 terminates.

5 Evaluation

The aim of our evaluation was to address the following questions: Q_1. Is the combined run time of the search for the best subsumed LS Λ and the execution of said LS more time-efficient than the execution of the LS L_0? Q_2. To what extent do the different values of *maxOpt* influence the overall run time of partial-recall Link Discovery? Q_3. How do the strategies C-RO and RO-MA compare to each other? Q_4. How much does the sampling of links influence supervised machine learning for Link Discovery?

We evaluated our approach on seven datasets. The first four were the benchmark datasets for Link Discovery dubbed `Abt-Buy`, `Amazon-GP`, `DBLP-ACM` and `DBLP-Scholar` described in [13]. We created three additional datasets (`MOVIES`, `TOWNS` and `VILLAGES`) from the real datasets to explore the scalability of the approach presented herein. Information about the characteristics of the datasets and the LSs used during our experiments can be found in [7]. All LS used during our experiments were generated automatically by the unsupervised version of the genetic-programming-based ML approach EAGLE [14] as implemented in LIMES [15].

We conducted partial-recall experiments with all seven datasets describe previously to answer Q_1 – Q_4. We set the values of k to 0.1, 0.2 and 0.5 while the maximum times (*maxOpt*) for finding a partial-recall LS were set to 100, 200, 400, 800 and $1,600$ ms. As we mentioned in Section 4, an essential ingredient of the LIGER implementation is the selectivity and run time approximations, obtained from an Oracle O. LIGER assumes that it can (1) approximate its run time using a linear model described in [6], and (2) estimate its selectivity as follows: (1) For an atomic LS, the selectivity values were computed using $\frac{|[[L]]|}{|S|\times|T|}$, where $|[[L]]|$ is the size of the mapping returned by the LS L, $|S|$ and $|T|$ are the sizes of the source and target data. To do so, we pre-computed the real selectivity of atomic LSs that were based on a set of measures using the methodology presented in [16] for thresholds between 0.1 and 1. (2) For complex LSs, which are binary combinations of two LSs L_1 (selectivity: $sel(L_1)$) and L_2 (selectivity: $sel(L_2)$), the run time approximation was computed by summing up the individual run times of L_1,L_2, in addition to a constant value of 1 for each operator. The selectivity of operators was computed based on the selectivity of the mappings that served as input for the operators. We assumed the selectivity of a LS L to be the probability that a pair (s, t) is returned after applying L. Based on these assumptions, we derived the following selectivities: (1) $op(L) = \cap \rightarrow sel(L) = \frac{1}{2}sel(L_1)sel(L_2)$, (2) $op(L) = \cup \rightarrow sel(L) = \frac{1}{2}(1 - (1 - sel(L_1))(1 - sel(L_2)))$ and (3) $op(L) = \setminus \rightarrow sel(L) = \frac{1}{2}sel(L_1)(1 - sel(L_2))$. Note that we used a correction factor of 0.5 to correct for the dependence of the selectivity across properties like in previous works [1]. Finally, in all of our experiments, we used the open-source LD framework LIMES. as reference framework. LIMES was used to execute both the input LS L_0 and the partial-recall LS Λ. The results achieved with L_0 were our *Baseline*. We used LIMES as our baseline since it has outperformed the state-of-the-art approaches in previous publications [15]. The main goal of running *Baseline* was to compare LIGER's performance with a LD strategy that is not influenced by time and selectivity constraints.

To answer the questions mentioned in Section 5, we evaluated the performance of LIGER (i.e., C-RO and RO-MA) in terms of execution time (see Table 2). For the sake of comparison, we present the results of the *Baseline* for all seven datasets alongside with LIGER. As expected, all variations of LIGER require less execution time than the *Baseline*. This clearly answers our first question Q_1: LIGER produces more time-efficient LS, even when *maxOpt* is set to a high value. Table 2 also shows clearly that LIGER performs best on `VILLAGES` for $k = 0.1$ and $maxOpt = 0.4\,s$, where it can reduce the average run time of the 100 LSs we considered by 88%. On the smaller `BDLP-ACM` dataset, RO-MA performs best and achieves a time reduction of the run time by 77.5%. Table 2 also allows us to study the influence of *maxOpt* and k on the total runtimes. The behavior of our approaches on the datasets varies and depends on a combination of the size of the LSs (smaller LSs are easier to optimize and execute) and the difference between the execution times of the optimized LS and the original LS. For the `Amazon-GP`, `DBLP-Scholar`, `MOVIES` and `VILLAGES` datasets, we notice that for the same value of $maxOpt$, the runtime of both RO-MA and C-RO increases as k receives larger values. This is due to the LS set for these datasets consisting mostly of large complex LSs. For the `DBLP-ACM` and `TOWNS` datasets, we noticed that the behavior of LIGER is not highly influenced by the different combinations of our parameters.

The observation is based on the fact that the set of LSs for both datasets consists of a more balanced proportion of atomic and complex LSs, where the complex LSs are not large in size compared to the previous 4 datasets. And finally, for the `Abt-Buy` dataset, both RO-MA and C-RO have a less uniform performance for the different values of k and $maxOpt$. The `Abt-Buy`'s LS set consists mostly of atomic LSs. Consequently, the runtimes of the LS are lowest for $k = 0.2$. Overall, $maxOpt = 200\,ms$ produces the best execution times on average for both C-RO and RO-MA. To address question Q_3, we studied the overall run times for all experimental configurations (see Table 2). Our average results suggest that RO-MA outperforms C-RO on average. The statistical significance of these results is confirmed by a paired t-test on the average run time distributions (significance level = 0.95). Our intuition that the *monotonicity of run times* can potentially improve the run time of our approach is supported by the results on three out of the seven datasets (`Abt-Buy`, `DBLP-ACM` and `Amazon-GP`). On the remaining four datasets, RO-MA outperforms C-RO on average. Still, when C-RO outperforms RO-MA, the absolute differences are minute. Additionally, both subsumed LSs received the same selectivity. Hence, when the available refinement time is limited, RO-MA should be preferred when aiming to carry out partial-recall LD. The highest absolute difference between C-RO and RO-MA is achieved on the `DBLP-Scholar` dataset, where RO-MA is 1179.59 s faster than C-RO, while the highest relative gain of 776.28% by C-RO against the *Baseline* is achieved on `VILLAGES` (k=10%, $maxOpt = 400$), which is the largest dataset of our experiments. A particularity of the `DBLP-Scholar` is that the LSs generated by EAGLE are large, which leads to small optimization times being sufficient to find good LS. The different strategies followed by C-RO and RO-MA lead to different atomic measures being modified during the refinement process. Especially for `DBLP-Scholar`, RO-MA achieves this highest absolute difference because RO-MA is able to find subsumed LS for complex LSs more efficiently. Overall, we can answer Q_3 by stating that RO-MA is to be preferred over C-RO.

In this extrinsic evaluation, we aim to measure the loss of F-measure of a machine-learning approach when presented with the results of partial-recall LD in comparison with the F-measure it would achieve using the full results. W use WOMBAT, which is currently the only approach for learning LSs from positive examples. Our results show clearly that with an expected partial recall of 50%, WOMBAT achieves at least 76.6% of the F-measure that it achieves when presented with all the data generated by EAGLE (recall = 100%). More detailed results are provided in Figure 2. The overall loss in F-measure is on datasets where the achievable F-measure is smaller, (e.g., on `Amazon-GP`). Still, in the best case, we achieve more than 95% of the maximal F-measure with $k = 0.1$. This gives us a clear answer to Q_4, namely that while the sampling of links leads to smaller F-measures, the ratio between expected recall and portion of F-measure achieved speaks in favor of using partial-recall LD in machine-learning applications with time constraints.

6 Related Work

LD Time efficiency has been addressed by several approaches and frameworks. LIMES [15] focuses on reducing the number of comparisons necessary to compute mappings by us-

Table 2: Average execution times of *Baseline*, C-RO and RO-MA for the different combinations of k and *maxOpt* over 100 LS per dataset. All times are in seconds.

		`Abt-Buy`			`Amazon-GP`			`DBLP-ACM`			`DBLP-Scholar`		
	maxOpt	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**
$k = 0.1$	0.1	0.66	0.52	0.52	5.71	3.91	3.82	1.08	0.25	0.25	792.81	596.53	598.44
	0.2	0.66	0.55	0.54	5.71	3.81	2.89	1.08	0.26	0.26	792.81	545.22	546.01
	0.4	0.66	0.45	0.44	5.71	3.04	2.91	1.08	0.26	0.25	792.81	589.72	587.87
	0.8	0.66	0.55	0.53	5.71	3.27	3.15	1.08	0.28	0.26	792.81	598.54	599.03
	1.6	0.66	0.54	0.51	5.71	3.47	3.18	1.08	0.33	0.28	792.81	554.82	557.06
		`MOVIES`			`TOWNS`			`VILLAGES`					
	maxOpt	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**			
	0.1	4.05	1.89	1.89	44.52	31.15	31.20	123.58	15.32	15.26			
	0.2	4.05	1.75	1.76	44.52	32.23	32.19	123.58	15.65	15.71			
	0.4	4.05	1.91	1.90	44.52	34.21	34.09	123.58	14.10	14.12			
	0.8	4.05	1.77	1.76	44.52	34.08	34.10	123.58	14.69	14.52			
	1.6	4.05	1.93	1.89	44.52	34.38	34.00	123.58	15.65	15.17			
		`Abt-Buy`			`Amazon-GP`			`DBLP-ACM`			`DBLP-Scholar`		
	maxOpt	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**
$k = 0.2$	0.1	0.66	0.35	0.35	5.71	4.07	3.92	1.08	0.26	0.26	792.81	593.66	581.86
	0.2	0.66	0.34	0.34	5.71	3.38	3.23	1.08	0.26	0.26	792.81	590.91	587.89
	0.4	0.66	0.29	0.28	5.71	3.38	3.36	1.08	0.29	0.28	792.81	561.93	566.74
	0.8	0.66	0.33	0.32	5.71	3.17	3.15	1.08	0.29	0.26	792.81	579.14	580.28
	1.6	0.66	0.40	0.37	5.71	3.57	3.37	1.08	0.33	0.29	792.81	551.14	549.08
		`MOVIES`			`TOWNS`			`VILLAGES`					
	maxOpt	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**			
	0.1	4.05	1.98	1.99	44.52	34.49	34.42	123.58	20.60	20.64			
	0.2	4.05	1.99	1.99	44.52	32.28	32.36	123.58	20.01	19.98			
	0.4	4.05	2.01	1.98	44.52	32.56	32.46	123.58	19.40	19.42			
	0.8	4.05	1.97	1.97	44.52	33.65	33.79	123.58	21.23	20.97			
	1.6	4.05	1.99	1.99	44.52	33.98	34.04	123.58	21.15	20.80			
		`Abt-Buy`			`Amazon-GP`			`DBLP-ACM`			`DBLP-Scholar`		
	maxOpt	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**
$k = 0.5$	0.1	0.66	0.42	0.42	5.71	4.55	4.23	1.08	0.28	0.28	792.81	602.49	603.92
	0.2	0.66	0.44	0.43	5.71	4.02	3.64	1.08	0.28	0.28	792.81	554.38	555.24
	0.4	0.66	0.42	0.41	5.71	3.95	3.65	1.08	0.27	0.27	792.81	637.66	629.34
	0.8	0.66	0.42	0.41	5.71	3.66	3.63	1.08	0.29	0.27	792.81	590.48	581.50
	1.6	0.66	0.48	0.45	5.71	3.79	3.67	1.08	0.34	0.29	792.81	595.17	591.91
		`MOVIES`			`TOWNS`			`VILLAGES`					
	maxOpt	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**			
	0.1	4.05	2.83	2.81	44.52	32.05	31.96	123.58	25.50	25.43			
	0.2	4.05	2.81	2.78	44.52	34.79	34.59	123.58	29.231	29.23			
	0.4	4.05	2.81	2.84	44.52	34.57	34.08	123.58	25.60	25.58			
	0.8	4.05	2.92	2.89	44.52	31.84	31.87	123.58	29.21	29.17			
	1.6	4.05	2.91	2.91	44.52	34.69	34.57	123.58	28.49	28.52			

ing *PPJoin+* [22]. Another lossless LD framework that uses blocking is SILK [21]. With *MultiBlock* [12]. Given that LIMES was shown to outperform SILK [16], we chose to implement our approach based on LIMES. KNOFUSS [18] incorporates blocking approaches inspired from databases. Zhishi.links [19] is a framework that scales through an indexing-based approach. The problem of identifying appropriate LSs using machine learning techniques has been explored in various previous papers. For example, the approach in [11] focuses on generating LSs using genetic programming. EAGLE [14] and AGP [5] are some of the approaches that have used active learning in order to maintain a high level of accuracy by requesting less labeled examples by training. WOMBAT [20] uses only positive examples for finding accurate LS. Additionally, unsupervised methods and tools such as PARIS [17] require no training data but are based on specific assumptions about the characteristics of the matching pairs. Frameworks such as FEBRL [4] and MARLIN [2] rely on models such as Support Vector Machines and Decision Trees to identify appropriate classifiers. To the best of our knowledge, we are the first to address the problem of partial-recall Link discovery.

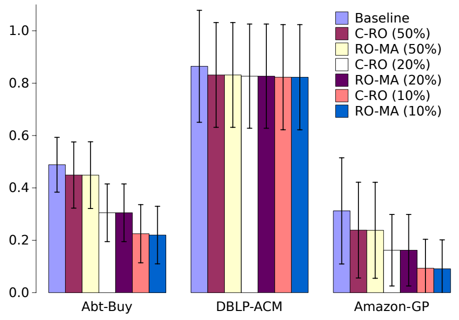


Fig. 2: F-measures achieved by WOMBAT,when provided with the results of LIGER as training data. The expected recall values set for C-RO and RO-MA are in brackets.

7 Conclusions and Future Work

In this work, we presented LIGER, the first (to the best of our knowledge) partial-recall LD approach. We provided a formal definition of a downward refinement operator with which we can detect subsumed LS with partial recall. We studied its characteristics and prove that our operator is finite, redundant, proper and incomplete. We used this operator to develop an algorithm for partial-recall LD. Additionally, we introduced an extension of said algorithm that takes into consideration the monotonicity of runtimes. We thus evaluated our approach on 7 datasets derived from real data and showed that our approach scales to large datasets. Our results show that by using our downward refinement operator, we are able to detect LS with guaranteed expected recall efficiently. Our extension of the LIGER algorithm with a monotonicity assumption pertaining to the run time of the LS was shown to be slightly better than the basic LIGER implementation. Also, we demonstrate that the results of partial-recall LD can be used to initialize supervised LD approaches without worsening their recall. In future work, we will build upon LIGER to guarantee the real selectivity and recall of our approaches with a given probability.

References

1. Abiteboul, S., Hull, R., Vianu, V. (eds.): Foundations of Databases: The Logical Level. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edn. (1995)
2. Bilenko, M., Mooney, R.J.: Adaptive duplicate detection using learnable string similarity measures. In: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 39–48 (2003)

3. Bin, S., Westphal, P., Lehmann, J., Ngonga, A.: Implementing scalable structured machine learning for big data in the sake project. In: 2017 IEEE International Conference on Big Data (Big Data). pp. 1400–1407. IEEE (2017)
4. Christen, P.: Febrl - an open source data cleaning, deduplication and record linkage system with a graphical user interface. In: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2008)
5. de Freitas, J., Pappa, G., da Silva, A., Gonçalves, M., Moura, E., Veloso, A., Laender, A., de Carvalho, M.: Active learning genetic programming for record deduplication. In: Evolutionary Computation (CEC), 2010 IEEE Congress (2010)
6. Georgala, K., Hoffmann, M., Ngomo, A.N.: An Evaluation of Models for Runtime Approximation in Link Discovery. In: Proceedings of the International Conference on Web Intelligence. pp. 57–64. WI '17, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3106426.3106428
7. Georgala, K., Obraczka, D., Ngonga Ngomo, A.C.: Dynamic planning for link discovery. In: The Semantic Web. pp. 240–255 (2018)
8. Gulisano, V., Jerzak, Z., Smirnov, P., Strohbach, M., Ziekow, H., Zissis, D.: The debs 2018 grand challenge. In: Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems. p. 191–194. DEBS '18, Association for Computing Machinery, New York, NY, USA (2018), https://doi.org/10.1145/3210284.3220510
9. Gunning, D.: Explainable artificial intelligence (xai). Defense Advanced Research Projects Agency (DARPA), nd Web **2** (2017)
10. Huber, M.F., Voigt, M., Ngomo, A.C.N.: Big data architecture for the semantic analysis of complex events in manufacturing. Informatik 2016 (2016)
11. Isele, R., Bizer, C.: Active Learning of Expressive Linkage Rules using Genetic Programming. Web Semantics: Science, Services and Agents on the World Wide Web **23**(0) (2013)
12. Isele, R., Jentzsch, A., Bizer, C.: Efficient Multidimensional Blocking for Link Discovery without losing Recall. In: Marian, A., Vassalos, V. (eds.) WebDB (2011)
13. Köpcke, H., Thor, A., Rahm, E.: Evaluation of Entity Resolution Approaches on Real-world Match Problems. Proc. VLDB Endow. **3**(1-2), 484–493 (Sep 2010), http://dx.doi.org/10.14778/1920841.1920904
14. Ngomo, A.C.N., Lyko, K.: Eagle: Efficient active learning of link specifications using genetic programming. In: Extended Semantic Web Conference. pp. 149–163. Springer (2012)
15. Ngonga Ngomo, A.C.: On Link Discovery using a Hybrid Approach. Journal on Data Semantics **1**(4), 203–217 (2012), http://dx.doi.org/10.1007/s13740-012-0012-y
16. Ngonga Ngomo, A.C.: HELIOS – Execution Optimization for Link Discovery, pp. 17–32. Springer International Publishing, Cham (2014)
17. Nienhuys-Cheng, S.H., van der Laag, P.R.J., van der Torre, L.W.N.: Constructing refinement operators by decomposing logical implication, pp. 178–189. Springer (1993), http://dx.doi.org/10.1007/3-540-57292-9_56
18. Nikolov, A., d'Aquin, M., Motta, E.: Unsupervised Learning of Link Discovery Configuration. In: Proceedings of the 9th International Conference on The Semantic Web (2012), http://dx.doi.org/10.1007/978-3-642-30284-8_15
19. Niu, X., Rong, S., Zhang, Y., Wang, H.: Zhishi.links results for OAEI 2011. Ontology Matching p. 220 (2011)
20. Sherif, M., Ngonga Ngomo, A.C., Lehmann, J.: WOMBAT - A Generalization Approach for Automatic Link Discovery. In: 14th Extended Semantic Web Conference. Springer (2017)
21. Volz, J., Bizer, C., Gaedke, M., Kobilarov, G.: Discovering and Maintaining Links on the Web of Data. In: Proceedings of the 8th International Semantic Web Conference. ISWC 2009 (2009), http://dx.doi.org/10.1007/978-3-642-04930-9_41
22. Xiao, C., Wang, W., Lin, X., Yu, J.X., Wang, G.: Efficient similarity joins for near-duplicate detection. ACM Trans. Database Syst. **36**(3), 15 (2011)