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. 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, combines downward refinement operators with monotonicity assumptions to detect such specifications. Our results suggest that our different implementations 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 Industry-4.0 applications such as condition monitoring and predictive maintenance. An increasing number of such machines generate knowledge graphs in the Resource Description Framework (RDF) format. A key step for learning axioms which generalize well is to learn them 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 integrate data streams from several machines. Given that new data batches are available periodically (e.g., every 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 LD with partial recall by proposing LIGER, 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. LIGER relies on a refinement operator, which allows the efficient exploration of potential solutions to this problem. The main contributions of our work are: (1) We present a downward refinement operator that allows the detection of subsumed LSs with partial recall. (2) We use a monotonicity assumption to improve the time efficiency of our approach. (3) We evaluate LIGER using benchmark datasets.

 This work has been supported by the EU H2020 project KnowGraphs (GA no. 860801) as well as the BMVI projects LIMBO (GA no. 19F2029C) and OPAL (GA no. 19F2028A).

2 Linking with Guaranteed Expected Recall

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 [3]. In general, these grammars assume that a LS consists of : (i) *Similarity measures* m ($m : S \times T \times \mathcal{P}^2 \rightarrow [0,1]$), through which property values of resources found in the input datasets S and T can be compared and (ii) *operators op*. An *Atomic LS* is a filter $f = (m, \tau)$, where m is a similarity measure and $\tau \in [0,1]$ is a threshold. *Operators* combine two LSs L_1 and L_2 to a more complex specification $L = (f, \tau, op(L_1, L_2))$. For $L = (f, \tau, op(L_1, L_2))$, we call *op the operator* of L. 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 $S \times T$. We denote the size of a mapping $[[L]]$ by $|[[L]]|$. 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 LS 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 $k \times sel(L) = sel(L')$. Formally, 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.

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: $\forall \theta, \theta' \in [0,1]\ \theta > \theta' \rightarrow (m, \theta) \sqsubseteq (m, \theta')$. 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) $L_2 \sqsubseteq L_2' \rightarrow (L_1 \backslash L_2') \sqsubseteq (L_1 \backslash L_2)$

We call $\rho : \mathcal{LS} \rightarrow 2^{\mathcal{LS}}$ a *downward refinement operator* if $\forall L \in \mathcal{LS} : L' \in \rho(L) \rightarrow L' \sqsubseteq L$, where $(\mathcal{LS}, \sqsubseteq)$ is a quasi-ordered space. 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:

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

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. 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, ρ *being proper* means that while checking for redundancy, there is no need to compare LS with any of their parents.

3 Approach

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 2.

The basic implementation of LIGER is dubbed C-RO. Our approach takes as input: a LS L_0, an oracle O which can predict the run times and selectivity of LS, the minimal expected recall k and a refinement time constraint *maxOpt*. We begin by asking O to provide the algorithm with estimations of the selectivity of L_0 (sel_{L_0}). We define a refinement tree with L_0 as its root. For each refined LS, the set of refined LSs are added as children nodes to the currently refined LS, and a leaf node is as a LS that can not be refined any further. We assign L_0 as the best subsumed LS Λ and the best run time rt_Λ with L_0's runtime estimation from O. The algorithm computes the desired selectivity value (sel_{des}) as a fraction of L_0's selectivity. Then, we add L_0 to the set $Buffer$, that 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 another buffer named $Total$. By keeping track of these LSs, we avoid refining a LS more than once and address the redundancy of our refinement operator. The refinement of L_0 stops when the refinement time has exceeded *maxOpt*, or if the $Buffer$ is empty or the selectivity of Λ returned by O is equal to sel_{des}. At each iteration, the algorithm selects the next node for refinement as follows: first, it retrieves the run time estimation of each LS L that belongs to $Buffer$ using O. Then, it selects the next LS for refinement as the LS with minimum run time estimation (L_{xt}). The algorithm then checks if L_{xt} receives a better runtime score, and assigns L_{xt} as the new value of Λ and updates rt_Λ accordingly. Finally, the algorithm refines L_{xt} by implementing ρ. For each subsumed LS, the algorithm checks if it already exists in set $Total$, to ensure that LIGER does not explore LSs that have already been seen before. Each remaining subsumed LS is added to $Total$ and the algorithm proceeds in computing its selectivity. If its selectivity is higher or equal to the desired selectivity, the algorithm updates $Buffer$ by adding the new LS.

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 $\leq$, $L_1 \in \rho(L) \wedge L_2 \in \rho(L) \wedge RT(L_1) \leq RT(L_2) \rightarrow \forall L' \in \rho^*(L_1)$: $RT(L') \leq RT(L_2)$ also holds. We call this assumption the *monotonicity of run times*. Since the implementation of C-RO does not take this monotonicity into consideration, we wanted to know whether this assumption can potentially improve the run time of our approach. We then

Table 1: Average execution times of *Baseline*, C-RO and RO-MA for $k = 0.1$ and different values of *maxOpt* over 100 LS per dataset. All times are in seconds.

$k = 0.1$

	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**
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

implemented an extension of LIGER with the monotonicity assumption (dubbed RO-MA). RO-MA uses 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.

4 Evaluation

We evaluated our approach on seven datasets [4, 2]. All LS used during our experiments were generated automatically by the unsupervised version of the genetic-programming-based ML approach EAGLE [5] as implemented in LIMES [6]. Regarding the Oracle O mentioned in Section 3, LIGER assumes that it can (i) approximate its run time using a linear model described in [1], and (ii) 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 [7] 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. Therefore, we derived the following selectivities: (I) $op(L) = \cap \rightarrow sel(L) = \frac{1}{2}sel(L_1)sel(L_2)$, (II) $op(L) = \cup \rightarrow sel(L) = \frac{1}{2}(1 - (1 - sel(L_1))(1 - sel(L_2)))$ and (3) $op(L) = \setminus \rightarrow sel(L) = \frac{1}{2}sel(L_1)(1 - sel(L_2))$. The results achieved with L_0 were our *Baseline*.

We first compared the execution time of C-RO and RO-MA (see Table 1) against the *Baseline* for all 7 datasets alongside with LIGER. As expected, all variations of LIGER require less execution time than the *Baseline*. As a result, LIGER produces more time-efficient LS, even when *maxOpt* is set to a high value. LIGER performs best on VILLAGES for $k = 0.1$ and $maxOpt = 0.4\,s$, where it can reduce the average runtime 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%. Futhermore, we studied how the strategies C-RO and RO-MA compare to each other (see Table 1). 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.

Finally, we wanted to measure the loss of F-measure of a machine-learning approach when presented with the results of partial-recall LD vs. the F-measure it would achieve using the full results. We use WOMBAT [8], which is currently the only approach for learning LSs from positive examples. Our results show 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%).

5 Conclusions and Future Work

We presented LIGER, the first partial-recall LD approach. We provided a formal definition of a downward refinement operator along with its characteristics,which we used to develop an algorithm for partial-recall LD. We thus evaluated our approach on 7 datasets and showed that by using our 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. In future work, we will build upon LIGER to guarantee the real selectivity and recall of our approaches with a given probability.

References

1. Georgala, K., Hoffmann, M., Ngomo, A.N.: An Evaluation of Models for Runtime Approximation in Link Discovery. In: Proceedings of the International Conference on WI (2017)
2. Georgala, K., Obraczka, D., Ngonga Ngomo, A.C.: Dynamic planning for link discovery. In: The Semantic Web. pp. 240–255 (2018)
3. Isele, R., Jentzsch, A., Bizer, C.: Efficient Multidimensional Blocking for Link Discovery without losing Recall. In: Marian, A., Vassalos, V. (eds.) WebDB (2011)
4. 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)
5. 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)
6. 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`
7. Ngonga Ngomo, A.C.: HELIOS – Execution Optimization for Link Discovery, pp. 17–32. Springer International Publishing, Cham (2014)
8. Sherif, M., Ngonga Ngomo, A.C., Lehmann, J.: WOMBAT - A Generalization Approach for Automatic Link Discovery. In: 14th Extended Semantic Web Conference. Springer (2017)