ALEX: Automatic Link Exploration in Linked Data

Ahmed El-Roby[*]
University of Waterloo
aelroby@uwaterloo.ca

Ashraf Aboulnaga
Qatar Computing Research Institute
aaboulnaga@qf.org.qa

ABSTRACT

There has recently been an increase in the number of RDF knowledge bases published on the Internet. These rich RDF data sets can be useful in answering many queries, but much more interesting queries can be answered by integrating information from different data sets. This has given rise to research on automatically linking different RDF data sets representing different knowledge bases. This is challenging due to their scale and semantic heterogeneity. Various approaches have been proposed, but there is room for improving the quality of the generated links.

In this paper, we present `ALEX`, a system that aims at improving the quality of links between RDF data sets by using feedback provided by users on the answers to linked data queries. `ALEX` starts with a set of candidate links obtained using any automatic linking algorithm. `ALEX` utilizes user feedback to discover new links that did not exist in the set of candidate links while preserving link precision. `ALEX` discovers these new links by finding links that are similar to a link approved by the user through feedback on queries. `ALEX` uses a Monte-Carlo reinforcement learning method to learn how to explore in the space of possible links around a given link. Our experiments on real-world data sets show that `ALEX` is efficient and significantly improves the quality of links.

Categories and Subject Descriptors

H.2.5 [**Heterogeneous Databases**]

Keywords

Linked Data; RDF; Knoweldge Bases; Automatic Linking; Federated Query Processing; Reinforcement Learning

1. INTRODUCTION

In recent years, advances in the field of information extraction have helped in automating the construction and publishing of large RDF knowledge bases. Some of these knowledge bases are general-purpose [3, 9, 11], and others focus on specific domains such as movies[1], geographic information[2], music[3], and city data[4]. Publishing these RDF data sets on the web was encouraged by the Semantic Web principle of making data on the web readable and processable directly by machines [5]. However, publishing these data sets alone is not sufficient. The true power of linked data is realized only when the data sets are linked to each other so that their semantic properties can be fully exploited [8]. Linking data sets is crucial for answering queries that cannot be answered using one RDF data set alone. For example, consider the query "Find all New York Times articles about the NBA's MVP of 2013". Articles about people are available from the New York Times RDF knowledge base. However, we need to know who the "NBA's MVP of 2013" is, since this information cannot be found in the New York Times data set. Another data set like DBpedia could have the information that "LeBron James" is the "NBA's MVP of 2013". One can use the Web Ontology Language (OWL)[5] to define an *owl:sameAs* relation linking the two entities representing "LeBron James" from both data sets. This relation indicates that the two entities refer to the same individual, and enables the system to return all articles about "LeBron James" from the New York Times data set.

Some work has been done on aligning RDF schemas [4, 13] and automatically linking equivalent entities from different data sets [6, 12, 14]. That work aims to automatically introduce *owl:sameAs* links between two data sets. However, automatic linking approaches are best effort in nature, with no guarantees on output quality. They try to infer semantics automatically based on syntax, which is a difficult task in the absence of human guidance. As such, automatic linking of RDF knowledge bases can greatly benefit from user feedback on the quality of the generated links.

In this paper, we introduce `ALEX` (Automatic Link Exploration in Linked Data), a system that improves the quality of links between linked data RDF data sets by utilizing feedback that users provide on answers to their queries. `ALEX` allows users to issue queries over multiple RDF data sets that are linked using any automatic linking approach. These queries can be answered using one or more data sets. When the query answer is produced using links between multi-

[*]Work done while author was at the Qatar Computing Research Institute

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2749428.

[1]`http://www.linkedmdb.org/`
[2]`http://www.geonames.org/`
[3]`http://musicbrainz.org/`
[4]`http://www.data.gov/opendatasites`
[5]`http://www.w3.org/2001/sw/wiki/OWL`

ple data sets, ALEX gives the user the opportunity to approve or reject the query answers. ALEX considers the approval/rejection of a query answer as an approval/rejection of the link(s) used to produce this answer, and it uses this feedback to improve the quality of links. ALEX uses a stochastic technique that generalizes the feedback provided by the user and is resilient to errors in that feedback. Errors in user feedback (approving a wrong answer or rejecting a correct answer) can arise due to errors in the data or errors made by the user.

ALEX starts with a set of automatically generated links that can be produced using any automatic linking algorithm (referred to as *candidate links*). ALEX removes incorrect links rejected by the user, but the main focus of ALEX is to find new links that are *similar* to the links approved by the user. The way ALEX finds similar links is as follows: An entity in an RDF data set is represented by a URI. Each entity has a set of attributes (RDF predicates), and values corresponding to these attributes (RDF objects). We represent a link between two entities from different data sets by a set of *features* made up of the attributes of the two entities. A feature is a pair of attributes where the first attribute comes from the first entity and the second comes from the second entity. Each feature has a value, which is the *similarity score* of the values of the two attributes. When a user approves a link by approving a query answer based on this link, ALEX chooses one feature and finds new candidate links for which the value of this feature (i.e., the similarity score) is within a (narrow) range around the value of the feature of the approved link.

An important question that ALEX needs to answer is: *Which feature to explore around for a given approved link?*. Exploring around a random feature is not effective since it incorrectly assumes that all features are of equal importance in determining whether the entities are equivalent. At the same time, ALEX has no prior knowledge of which features may be important, and the best feature to explore around can depend on the link being explored. For example, the title of the two entities may be a good feature to explore around for some link, while the date of birth may return better results for another link. Thus, ALEX needs a way to identify the feature to explore around for any approved link between any two entities.

In this paper, we propose that this problem can be solved using *Monte Carlo reinforcement learning* methods [22], where the system can *learn* which feature to explore around for different links. Using the terminology of reinforcement learning, ALEX aims at learning from interacting with the environment in order to learn the best action to take (feature to explore around) in order to maximize reward (positive user feedback). The feature is chosen by ALEX using a policy that is iteratively improved.

We also develop several optimizations that help ALEX to converge in fewer steps. These optimizations include reducing the search space of links, partitioning the data to exploit parallelism, using a blacklist to prevent known incorrect links from being proposed, and rolling back actions to undo actions that result in exploring many incorrect links.

Our experiments over real-world data sets show that ALEX can improve the quality of the initial set of candidate links, while not exposing the end-user to a large number of incorrect answers. Experiments also show that ALEX converges in a reasonable amount of time and is robust to changes in parameters values.

The contributions of this paper are as follows:

- To the best of our knowledge, we are the first to bridge the gap between automatic linking of data sets on one side and querying linked data on the other side by leveraging user feedback to discover new links between entities without prior knowledge of the data sets or how they were originally linked.
- We propose a reinforcement learning approach to find new links while preserving link precision.
- We develop optimizations to reduce execution time and converge in fewer steps.
- We prove that our approach is sound in terms of finding an optimal policy for links exploration.
- We demonstrate the validity of our approach by running experiments on large, real-world, multi-domain data sets.

The rest of the paper is organized as follows: Section 2 gives an overview of the related work. Section 3 gives some background on reinforcement learning and an overview of ALEX. We describe the details of ALEX in Section 4. In Section 5, we prove the soundness of ALEX. Section 6 shows our optimizations for faster convergence. In Section 7, we discuss our experiments. We conclude in Section 8.

2. RELATED WORK

Automatic Linking: Linked data has enabled seamless connections between open data sets [7]. The idea is to link entities from different data sets that are semantically equivalent to each other. There have been many works on semantic matching of entities, taking different approaches. The SILK framework [23] uses manually defined mapping rules that are applied on input data sets. New data sets require new mapping rules. OBJECTCOREF [14] uses training data to learn how to link entities. However, this requires having good training data that captures most aspects of the input data sets, which is difficult in practice. PARIS [21] is a powerful probabilistic holistic automatic linking algorithm that is fully automatic and does not require any prior information. It also produces better quality links than other approaches. Due to its generality and superior quality, we use PARIS in this paper as our automatic linking algorithm to produce the candidate links that are the starting point for ALEX. However, we emphasize that ALEX can work with any initial set of candidate links, regardless of how they were generated.

Incorporating User Feedback: User feedback has been used to improve schema matching in relational data. In [1], user feedback is used in an iterative exploratory process to guide the system towards the best data sources for the user, and the best mediated schema for these sources. In [15], user feedback is used to order candidate matches so that they can be confirmed by the user. The candidate matches are sorted based on their importance (i.e., they are involved in more queries or associated with more data). A user is asked to confirm or reject each match. In [16], the user is asked during the schema matching process about matches that the system is uncertain about in order to choose the correct match. In the Crowd ER system [24], users are also asked to confirm or reject entity matches. However, due to the large number of questions that can be asked, the system is more concerned with choosing questions that need to be answered first (questions that are more challenging to computers). In contrast to these approaches, ALEX does not ask the user to provide feedback on links but rather on answers to queries. The user wants to see the answers to the queries anyway, so providing feedback on query answers is easier for the user than providing feedback directly on links.

In [25], user feedback is obtained over the answers to a keyword search query. The feedback is represented as a constraint over the ordering of the returned answers, or as identification of good or bad answers. The feedback is then utilized to improve the ordering of answers to future queries. In contrast, `ALEX` does not expose the user to the ontology of the data sets or the details of the linked entities but rather directly improves the quality of links by utilizing user feedback on the answers to her queries.

In the context of linking open data, ZenCrowd [10] utilizes the crowd by forming micro-tasks using a probabilistic model for manual matching. Its goal is to link traditional web content to the Linked Open Data (LOD) cloud[6]. In this paper, we utilize user feedback to improve the quality of links *in* the LOD cloud to which ZenCrowd tries to link traditional web pages.

One goal of [2] that is related to our problem is to refine links in DBpedia by removing incorrect links to external web pages or resources. Users are shown a caption of the external source and determine if it matches the entity in DBpedia or not. In contrast to all the aforementioned approaches, the most distinct feature of `ALEX` is that it not only removes incorrect links from the set of candidate links, but also discovers new links that were not part of this set.

3. BACKGROUND AND OVERVIEW

3.1 Reinforcement Learning

In reinforcement learning [22], the learner and decision maker is called the *agent*. Everything else outside of the agent is considered to be the *environment*. A reinforcement learning system consists of four main components:

- *Policy*: The policy defines how a reinforcement learning agent interacts with the environment at a given state. It can be viewed as the mapping from an environment *state* to an *action* taken by the agent. The policy can be as simple as a lookup table, or it can involve extensive computations. It also can be either deterministic or stochastic. In `ALEX`, we use a stochastic policy. For example, consider a user providing positive feedback on the link (E_1, E_2), where E_1, and E_2 are entities. The policy π might state that when this link is encountered, explore around the feature *(E_1.label, E_2.name)* with probability 0.8, and around the feature *(E_1.birth, E_2.birthDate)* with probability 0.2. Formally, $\pi((E_1, E_2), (E_1.label,\ E_2.name)) = 0.8$ and $\pi((E_1, E_2), (E_1.birth, E_2.birthDate)) = 0.2$. When an action is chosen, say the one that has higher probability, then $\pi\left((E_1, E_2)\right) = (E_1.label, E_2.name)$. More details are presented in the following sections.
- *Reward Function*: The reward function defines the goal of the reinforcement learner. It can be viewed as a mapping from a state (or a state-action pair) to a *reward*. The goal of the agent is to maximize the total reward throughout its dynamic interactions with the environment. For example, assume a positive feedback over the link (E_1, E_2) resulted in the exploration of 5 links $\{link_1, \ldots, link_5\}$. Assume that positive feedback comes over $link_1$ and $link_5$, negative feedback comes on $link_3$, and no feedback is received over $link_2$ and $link_4$. In this case, the reward is $R(E_1, E_2) = 2 \times pReward - nReward$.
- *Value Function*: The value function defines what is good in the long run. A *value* of a state can be viewed as the total reward that can be collected from this state taking into account the states that are likely to follow and the rewards available from those states. The major difference between the reward function and the value functions is that the first indicates what is good in an immediate sense (next reward), whereas the second indicates the long-term value of a state (total rewards that can be collected in the future starting from the current state). The reward of a state may be low but its value can be high because other states that yield high rewards can be reached from this state.
- *Environment Model* (optional): This model simulates the behavior of the environment. For example, given a state and an action, the model can determine which state is next and the reward of the action. The model is used in planning because it enables the agent to determine which action to take without experiencing the state. However, there may be cases where a model is not available. In this case, the only way to determine the next state and reward given a state and action is by actually performing the action in the environment. In `ALEX`, the environment model is unknown because it is not possible to know in advance what feedback users will provide.

Reinforcement learning differs from other branches of machine learning in that the learning agent is not told what actions to take. The agent learns over time how to act by experiencing the return it gets from interacting with the dynamic environment. Reinforcement learning is more suitable than supervised learning [17] for interactive problems where it is impractical to obtain examples of the desired behaviors to use as training data that are correct and representative of most situations in the environment. The reinforcement learning agent learns how to act by trying different actions at different states and aiming to maximize the expected return at those states [22]. The challenge that faces the learning agent is finding a balance between the need to explore as many states as possible while exploiting the current knowledge to maximize the total reward. Also, reinforcement learning needs a way to evaluate the policy used to take actions in order to improve it. In `ALEX`, we use a Monte Carlo method to evaluate the policy through returns from interactions with the environment. This is a suitable approach in situations like ours where the model of the environment is not known.

3.2 Overview of ALEX

Before we present the details of `ALEX`, we present an end-to-end overview. `ALEX` can be integrated in a system that answers queries over multiple data sets of RDF linked data (e.g., data sets in the Linked Open Data cloud). In this paper, we use SPARQL[7] as the query language, although other query languages can be used. An application may send a SPARQL query that can only be answered using data from different data sets that are linked by *owl:sameAs* links. If a link is correct, answers returned based on this link should also be correct unless the data itself is erroneous (which is beyond the scope of this paper). The user evaluates the

[6] `http://lod-cloud.net/`

[7] `http://www.w3.org/TR/sparql11-query/`

returned answer and gives her feedback by marking the answer "correct" or "incorrect". If the feedback is positive, ALEX tries to find new *owl:sameAs* links that are similar to the approved link. If the feedback is negative, ALEX removes this link. ALEX works as follows:

- ALEX accepts the links between two data sets as input. These links are automatically generated by any automatic linking algorithm.
- Federated queries are issued over the linked data. A federated query is a query whose answer is not available in one data source, but can be answered using multiple data sources. The query is decomposed into subqueries for each involved data source. However, this fact is hidden from the user who issues the query as if all data is in one place. Answers are retrieved using a federated query processing system (e.g. FedX [19]). A user then gives her feedback on the returned answers. The feedback is as simple as approving or rejecting the returned answer. ALEX takes action immediately based on the feedback, since it assumes that this feedback is correct. However, ALEX has techniques to recover from incorrect feedback.
- ALEX represents each link as a state, starting from which it takes an action after receiving user feedback. The action, in the case of positive feedback, can be described as choosing a feature to explore around within some exploration distance. In the case of negative feedback, the wrong link is removed from the set of candidate links.
- The given feedback is translated into a reward in ALEX. This reward is positive in the case of an approved link (positive feedback), and negative in the case of a rejected one (negative feedback). The value of the reward can be the same for positive and negative feedback, or negative feedback can be penalized more.
- ALEX explores links in a space of feature sets. This space is populated in a pre-processing step, with a feature set for every pair of entities in the two data sets.
- Initially, ALEX chooses arbitrary actions whenever a state is encountered because there is no prior knowledge of what actions to take. Rewards are collected for each state-action pair encountered, and are aggregated to estimate the value of the state-action pair. This is called *policy evaluation*.
- Policy evaluation takes place until sufficient feedback is collected. We call this a feedback *episode*. At the end of an episode, *policy improvement* takes place. Policy improvement modifies the policy so that actions that maximize the reward are taken most of the time, while assigning a low but non-zero probability to other actions in order to ensure continuous exploration.
- These last two steps of policy evaluation and policy improvement are repeated until convergence. ALEX converges when the set of candidate links does not change after an iteration of policy evaluation - policy improvement or when a maximum number of iterations is reached. ALEX can also use a more relaxed convergence condition and stop if the change in the set of candidate links is less than 5%.

4. DISCOVERING NEW LINKS IN ALEX

Starting with the output of any automatic linking algorithm, ALEX's goal is to (1) discover new links that did not exist in the set of candidate links (which improves recall), and (2) preserve the correctness of links by quickly eliminating wrong links in the output (which improves precision).

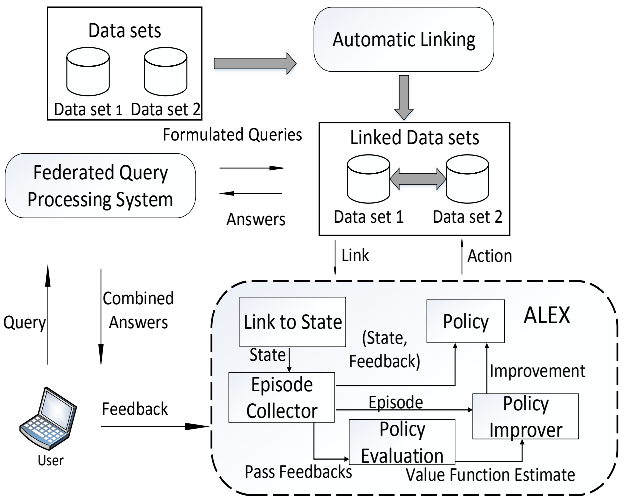


Figure 1: Architecture of a federated query system with ALEX.

Figure 1 shows the architecture of ALEX, in which a user issues a query over RDF data sets that are linked using some automatic linking algorithm. The query is answered by a federated query processing system that is able to answer queries that span multiple data sources (e.g., [18] or [19]). If a query answer is generated based on a link between two data sets, the user is given a chance to evaluate this answer and provide feedback about which answers are correct and which are not. This feedback is interpreted as feedback on the link that is used to generate the answer. That is, if the answer is correct then the link is correct, and if the answer is incorrect the link is incorrect. The feedback is sent to ALEX, which uses its current policy to take an action based on the current state. We emphasize that a user is not *required* to provide feedback on each query answer; if no feedback is provided on an answer, this answer will simply not trigger an action by ALEX. In ALEX, the state is represented by the feature set of the link used to answer the query, while the action is choosing one feature around which ALEX will explore to find new candidate links. The action results in adding these links to the set of candidate links.

An important aspect of reinforcement learning is improving the policy after using it for some time. In ALEX, this is achieved by grouping feedback on states into episodes, where each episode has a number of feedback items. The policy is evaluated while the episode of feedback is collected. Policy evaluation is defined as estimating the value of the states or state-action pairs encountered in an episode. Grouping feedback items into episodes makes the evaluation of the value of the states or state-action pairs more accurate. The larger the number of feedback items in an episode, the more accurate the estimation of the value. At the end of an episode, changes are made to improve the current policy to maximize the collected rewards, a process called policy improvement. The current policy defines the actions that ALEX takes within an episode, until the policy changes at the end of the episode through policy improvement. ALEX continues the iterations of policy evaluation and policy improvement until convergence.

4.1 States in ALEX

In a general reinforcement learning problem, the agent and the environment interact at every discrete time step, $t = 0, 1, 2, \ldots$. At each time step t, the agent perceives the state of the environment $s_t \in S$, where S is the set of all states in the environment. The agent then takes an action $a_t \in A(s_t)$, where $A(s_t)$ is the set of all actions available at the state s_t. As a result of the action taken, the agent receives a reward r_{t+1} and a new state s_{t+1} is reached. The agent's action is determined by the policy π_t where $\pi_t(s, a)$ is the probability that $a_t = a$ at $s_t = s$. In `ALEX`, the state is defined by the link that was approved or rejected by a user in a feedback item. The state is represented by the state feature set sf of many-to-many mappings between the attributes of the two entities that are linked by the link being considered. For example, consider the two entities E_1 and E_2 with n and m attributes, respectively, $E_1 = \{(p_{11}, o_{11}), (p_{12}, o_{12}), \ldots, (p_{1n}, o_{1n})\}$, and $E_2 = \{(p_{21}, o_{21}), (p_{22}, o_{22}), \ldots, (p_{2m}, o_{2m})\}$, where each attribute is the pair (predicate label, predicate value). An example entity is {(name, "LeBron James"), (birth date, 1984), (age, 29)}. We first construct a similarity matrix between the two entities using a similarity function that returns a score in the range $[0, 1]$. An element in the similarity matrix is $((p_{1x}, p_{2y}), score)$ where p_{1x} is a predicate from the first data set, p_{2y} is a predicate from the second data set, and $score$ is the similarity between the objects associated with these predicates, $score = sim(o_{1x}, o_{2y})$. Scores that are less than a specific threshold are discarded. `ALEX` uses a generic similarity function that depends on the type of the attributes to be compared (string, integer, float, date, etc.). The state feature set sf is then constructed by choosing the maximum value for each row in the similarity matrix if $n > m$ or each column if $m > n$. In this paper, we use the terms *state* and *link* interchangeably.

4.2 Actions in ALEX

Given a state s_t, `ALEX` takes an action a_t that is based on a policy π_t. The environment (the user in our case) then responds with a reward r_{t+1}. The ultimate goal of `ALEX` is to improve the policy π_t so that the maximum total reward is collected. The action of `ALEX` can be perceived as exploring an area surrounding the current state (the link between two entities) in a particular direction (one feature of the feature set). Given a state represented by a feature set sf of n features, the action a is also a feature set af of n features with a single non-zero feature that represents the offset by which `ALEX` should explore to discover new links. Formally, `ALEX` finds all the links that have similarity value between sf and $sf \pm af$. For example, consider the feature set $sf(E_1, E_2)$ ={((*label*, *name*), 0.8), ((*birth*, *year*), 0.6), ((*age*, *year*), 0.4)}. The first element of the feature set means that the predicate *label* from the first entity maps to the predicate *name* from the second entity, and the similarity between the predicate values is 0.8. A possible action can be represented by the action feature set $af(E_1, E_2)$ ={((*label*, *name*), 0.05), 0, 0}, which means that links that have a similarity score between attributes *label* and *name* in the range $[0.75, 0.85]$ should be added to the set of candidate links for future queries and possible feedback opportunities. The step size (0.05 here) is a parameter in `ALEX`.

`ALEX` can sometimes take an action that explores around a feature that has values that do not distinguish between entities. For example, it can decide to discover links around the feature (*rdf:type*, *rdf:type*) which has a categorical value *owl:thing*. Exploring around this feature and value is expected to return a large number of incorrect links because a large number of different entities share this attribute and value. `ALEX` can *learn* that this feature is not distinctive and avoid exploring around it in the future.

4.3 Rewards and Feedback

The goal of `ALEX` is to maximize the expected return, R_t, defined as the sum of all future rewards:

$$R_t = r_{t+1} + r_{t+2} + \ldots + r_T \tag{1}$$

where T is the final time step. In `ALEX`, the final time step is when a feedback episode ends. After that, the policy used during the episode is refined through policy improvement, and a new episode is started using the new policy.

In `ALEX`, the reward is the feedback given by the user. The feedback could be positive (approving a link) or negative (rejecting a link). The value of the reward can be equal in both cases, or we can severely penalize wrong links by giving them a negative value that is larger than the positive value of the approved link.

We also need to define the value of taking an action a at a state s under a policy π. The *action-value function* $Q^\pi(s, a)$ is defined by:

$$\begin{aligned} Q^\pi(s,a) &= E_\pi \{R_t | s_t = s, a_t = a\} \\ &= E_\pi \left\{ \sum_{k=t+1}^{T} r_k | s_t = s, a_t = a \right\} \end{aligned} \tag{2}$$

where E_π is the expected value given that `ALEX` follows policy π. A fundamental property of the value function is that it follows a recursive relationship between the current and future state-action values:

$$\begin{aligned} Q^\pi(s,a) &= E_\pi \left\{ \sum_{k=t+1}^{T} r_k | s_t = s, a_t = a \right\} \\ &= E_\pi \left\{ r_{t+1} + \sum_{k=t+2}^{T} r_k | s_t = s, a_t = a \right\} \\ &= \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + E_\pi \left\{ \sum_{k=t+2}^{T} r_k | s_{t+1} = s', a_t = a \right\} \right] \end{aligned} \tag{3}$$

where $P^a_{ss'}$ is the probability that the next state is s' when action a is taken at state s at time step t: $P^a_{ss'} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\}$, and $R^a_{ss'}$ is the expected value of the next reward when taking action a at state s to move to state s': $R^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$.

The relationship between the action-value of the current state and that of the next state can be obtained from Equation 3 as follows:

$$\begin{aligned} Q^\pi(s,a) &= \sum_{s'} P^a_{ss'} \Big[R^a_{ss'} + \sum_{a'} \pi(s',a') \times \\ &\quad E_\pi \left\{ \sum_{k=t+2}^{T} r_k | s_{t+1} = s', a_{t+1} = a' \right\} \Big] \\ &= \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \sum_{a'} \pi(s',a') Q^\pi(s',a') \right] \end{aligned} \tag{4}$$

where $\pi(s', a')$ is the probability of choosing action a' at state s' according to policy π. In ALEX, when a positive feedback item is received over a link (state) s, and an action a is taken based on a policy π with probability $\pi(s, a)$, a number of new links (states) is discovered and added to the set of candidate links. When one of these states s' is later visited and feedback (positive or negative) is received over it, we know then the value of $R^a_{ss'}$. We assume that all states that are generated by an action a have equal probability of being visited. Thus, $P^a_{ss'} = \frac{1}{|s'|}$.

A policy π' is considered to dominate another policy π if and only if $Q^{\pi'}(s, a) \geq Q^{\pi}(s, a)$ for all $s \in S$. In this sense, π^* is considered to be an *optimal policy* if its value function dominates the value functions of all other policies. There may exist more than one optimal policy. An optimal policy implies an optimal action-value function:

$$Q^*(s, a) = max_\pi \; Q^\pi(s, a) \qquad (5)$$

for all $s \in S$. Therefore, Equation 4 becomes:

$$Q^*(s, a) = \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + max_{a'} Q^*(s', a')\right] \qquad (6)$$

4.4 Iterative Improvement

After an episode of feedback is collected, ALEX improves the policy based on the value function evaluated during the episode. A new episode is then started and the policy evaluation - policy improvement iterations continue until ALEX converges. Convergence is defined by the candidate links not changing in an episode of feedback. In the following, we discuss how the policy is evaluated during an episode, and improved at the end of the episode.

4.4.1 Monte Carlo Policy Evaluation

The value function can only be evaluated through interactions between ALEX and the environment (the user and existing links). According to our definition of value, the value of a state or state-action pair can only be known if a user gives feedback on the current state and future states that follow. Since we need to evaluate the value function at the present time without waiting for future feedback, we need to estimate the value function according to the current state s, policy π, and action taken a. We use a Monte Carlo (MC) method for this estimation. Specifically, we use a Monte Carlo method to estimate the action-value of each state visited during each episode, while feedback is collected.

The existence of a state s in an episode is called a *visit*. We use a *first-visit* MC approach [22] for estimating the action-value function. In the first-visit MC approach, the average of returns following the first visit to s in which action a was taken is maintained. This means that if the state-action pair (s, a) is witnessed again during an episode, returns following that pair will not be considered. For example, if a state s_2 results from the state-action pair (s_1, a_1), and s_2 turns out to be a correct link, a positive reward is added to the return of state-action pair (s_1, a_1). Now, if from state s_2, action a_2 is taken and results in a wrong state s_3, a negative reward is added to (s_2, a_2) and (s_1, a_1). However, when state s_2 or s_3 is encountered again, no reward is added to the updated returns of the state-action pairs during the current episode. If a state, say s_2, is encountered in a future episode, that would be considered a new first visit. The first-visit MC approach converges asymptotically to $Q^\pi(s, a)$ [20].

The MC method requires π to be probabilistic. If π is deterministic rather than probabilistic at a state s, the same action a will always be taken. Thus, many relevant state-action pairs may never be visited. In such a case, there would be no need for learning how to choose among actions at any state. To compare the alternatives, we need to estimate the value of almost all actions from all witnessed states. ALEX ensures continual exploration to avoid this problem.

ALEX gives itself the option of choosing any action at any state. This means that at any state $s \in S$, and for all actions available in that state $a \in A(s)$, $\pi(s, a) > 0$. ALEX achieves this non-zero probability by using an ϵ-greedy policy so that it mostly chooses a greedy action that has the maximal estimated action value, but chooses a random action with low probability $\epsilon > 0$. In other words, with probability $1 - \epsilon + \frac{\epsilon}{|A(s)|}$ a greedy action is taken, and with probability $\epsilon - \frac{\epsilon}{|A(s)|}$, a non-greedy action is taken. This satisfies $\pi(s, a) \geq \frac{\epsilon}{|A(s)|} > 0$, which means that no action has zero probability of being selected by the current policy, thereby ensuring continuous exploration.

4.4.2 Policy Improvement

If the rewards, $R^a_{ss'}$, and the probabilities of moving to states given an action, $P^a_{ss'}$, are known in advance, Equation 6 has a unique solution since it is actually a system of N equations where N is the number of states in the environment. If the optimal value function is known, it is straightforward to determine an optimal policy: At any state s, choose the action that yields the maximum value of $Q^*(s, a)$ in Equation 6. In other words, a greedy policy with respect to the optimal evaluation function is the optimal policy. However, as explained earlier, the feedback on links and which states can be visited next are unknown in our problem. This means that the reward of the current action will not be known until the user gives feedback on the links discovered after the current action is taken. Also, the next state visited is not known in advance.

The previous section explained how a Monte Carlo method can be used to estimate Q^{π_k} for arbitrary probabilistic π_k, where k is the iteration number in the policy evaluation - policy improvement cycle. Policy improvement is done by making the policy greedy with respect to the current value function. That is, for any action-value function Q, the greedy policy is the one that, for all $s \in S$, chooses the action with the maximal Q value:

$$\pi(s) = argmax_a Q(s, a) \qquad (7)$$

Equation 7 can be used as the basis for policy improvement as follows:

$$\begin{aligned} Q^{\pi_k}(s, \pi_{k+1}(s)) &= Q^{\pi_k}(s, argmax_a Q^{\pi_k}(s, a)) \\ &= max_a Q^{\pi_k}(s, a) \qquad (8) \\ &\geq Q^{\pi_k}(s, \pi_k(s)) \end{aligned}$$

where, as explained above, π_{k+1} is the greedy policy with respect to Q^{π_k}. In Section 5, we prove π_{k+1} dominates π_k.

4.5 Interaction Between Policy Evaluation and Policy Improvement

The value function is repeatedly updated to approximate the actual value of the current state with respect to the current policy. Also, the policy is repeatedly improved with respect to the current value function. Iteratively, these two processes cause the policy to approach optimality, and the value function to approach its actual value.

As discussed before in Section 4.4.1, each evaluation step moves the value function Q^{π_k} towards its actual value. This

Algorithm 1: ALEX with ϵ-greedy policy

```
   input  : set of states $S$, set of actions $A$
   output : action-value function $Q(s,a)$, Policy $\pi(s)$
1  // Initialize
2  for all $s \in S$ do
3      for all $a \in A(s)$ do
4          $Q(s,a)$ = undefined;
5          $\pi(s)$ = arbitrary action;
6          $Returns(s,a)$ = empty list;
7      end
8  end
9  while set of candidate links different from last iteration
   do
10     // Policy Evaluation
11     while episode not complete do
12         receive feedback on a state $s'$;
13         if first visit of $s'$ then
14             append feedback value to all $Returns(s,a)$
               that led to $s'$;
15         end
16         $Q(s,a) = AVG(Returns(s,a))$;
17         if positive feedback then
18             $a' = \pi(s')$;
19         else
20             remove link;
21         end
22     end
23     // Policy Improvement
24     for all states s in episode do
25         $a^* = argmax_a Q(s,a)$;
26         for all $a \in A(s)$ do
27             if $a = a^*$ then
28                 $\pi(s,a) = 1 - \epsilon$;
29             else
30                 $\pi(s,a) = \frac{\epsilon}{|A(s)|}$;
31             end
32         end
33     end
34 end
```

value function converges to its actual value over many steps, at which point policy improvement can terminate. However, this process would require many feedback episodes for each policy improvement iteration. ALEX does not wait for complete policy evaluation before returning to policy improvement. In fact, only one episode of iterative policy evaluation is required between each two policy improvement steps.

Algorithm 1 shows how ALEX alternates between policy evaluation and policy improvement on an episode-by-episode basis. While collecting feedback in an episode, policy evaluation is done by estimating the action-value function $Q(s,a)$ (lines 11 to 22), the policy is then improved at all states visited in the episode by choosing the greedy action (line 25). It is not required for the policy to always be greedy. However, it is required to *move* towards a greedy policy. This will be proven to be sound in Section 5.

Algorithm 1 shows how policy improvement is done using an ϵ-greedy policy. When ALEX starts, it chooses arbitrary actions for new states visited for the first time or before the first policy improvement cycle (lines 2 to 8). Lines 24 to 33 show how policy improvement takes place after an episode by assigning the greedy action a probability of $1-\epsilon$ while non-greedy actions are assigned a probability $\frac{\epsilon}{|A(s)|}$. This is to ensure continuous exploration while at the same time moving towards a greedy policy. During the next episode, when a state that exists in the policy is encountered, a greedy action is taken with high probability, while other actions are explored with low probability.

5. SOUNDNESS OF ALEX

In this section, we prove that: (1) Policy improvement always yields a better policy unless the policy is already optimal. (2) This property applies for the ϵ-greedy policy used in ALEX.

The value function we discussed thus far is the action-value function $Q^\pi(s,a)$, which defines the expected return at a state s when choosing an action a following some policy π. Another value function, which we need in our proof, defines the expected return, $V^\pi(s)$, for a state s under policy π:

$$V^\pi(s) = E_\pi\left\{R_t | s_t = s\right\} = E_\pi\left\{\sum_{k=t+1}^{T} r_k | s_t = s\right\} \tag{9}$$

This value function is called the *state-value function* for policy π. It defines the expected value given that ALEX follows policy π at state s. Similar to Equation 4, the value function of state s can be represented as a recursive relationship with the value function of the next state, s' (derivation in Appendix A):

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + V^\pi(s')\right] \tag{10}$$

Also, similar to Equation 6, the optimality equation for state value V^* is given by (derivation in Appendix A):

$$V^*(s) = max_{a\in A(s)} \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + V^*(s')\right] \tag{11}$$

We now turn to proving that policy improvement always yields a better policy. Our proof will use the state-value function that we just defined. For ease of explanation, assume that the policy is deterministic. However, the concepts discussed here can be applied to probabilistic policies like the one used by ALEX.

The approach to proving that policy improvement always yields a better policy can be illustrated with an example: Assume that the policy currently being used is π, and that for some state s we want to change the policy to choose an action $\pi'(s) = a \neq \pi(s)$. The value of the state given that we follow policy π is given by $V^\pi(s)$. The question is whether it would be better or worse to change the policy so that it always chooses a at state s. The value of choosing a at s and then continuing for future states following the original policy π can be given by:

$$Q^\pi(s,a) = E_\pi\left\{r_{t+1} + V^\pi(s_{t+1}) | s_t = s, a_t = a\right\} = \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + V^\pi(s')\right] \tag{12}$$

If it turns out that following policy π' only at state s (i.e., choosing action a), and then following policy π for future states gives greater state-value than the value of the state when following policy π, this situation can be represented by the following inequality:

$$Q^\pi(s, \pi'(s)) \geq V^\pi(s) \tag{13}$$

We want to show that if Equation 13 holds, then always following policy π' at state s yields a greater value than following policy π. That is, we want to prove that

$$V^{\pi'}(s) \geq V^{\pi}(s) \tag{14}$$

This can be proved by starting from Equation 13 and expanding $Q^{\pi}(s, \pi'(s))$ using Equation 12. The proof is presented in Appendix A.

So far, we have shown that we can evaluate how a change in the policy at a single state to a particular action affects the state-value of this state, given the policy and the evaluation function. Greedily choosing a policy that increases the action-value function at state s improves the overall policy. This reasoning can be extended to all states and all possible actions by selecting the action a that yields the highest $Q^{\pi}(s, a)$ at each state s. This is the greedy policy π' defined by:

$$\begin{aligned} \pi'(s) &= argmax_a Q^{\pi}(s, a) \\ &= argmax_a E\left\{r_{t+1} + V^{\pi}(s_{t+1}) | s_t = s, a_t = a\right\} \\ &= argmax_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + V^{\pi}(s')\right] \end{aligned} \tag{15}$$

Thus far, we proved that a greedy policy π' is as good as, or better than, an arbitrary policy π. Now, if we assume that the greedy policy π' is as good as, but not better than, policy π (i.e.,$V^{\pi'} = V^{\pi}$), we get:

$$\begin{aligned} V^{\pi'}(s) &= max_a E\left\{r_{t+1} + V^{\pi'}(s_{t+1}) | s_t = s, a_t = a\right\} \\ &= max_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + V^{\pi'}(s')\right] \end{aligned} \tag{16}$$

This equation is the same as the optimality Equation 11. Thus, $V^{\pi'}$ must be V^*, and π and π' must be the optimal policies. This means that policy improvement must give a better policy unless the policy is already optimal. This also proves Equation 8 because each π_{k+1} is uniformly better than π_k, or equal to it if both are optimal policies:

$$\begin{aligned} Q^{\pi_k}(s, \pi_{k+1}(s)) &\geq Q^{\pi_k}(s, \pi_k(s)) \\ &\geq V^{\pi_k}(s) \end{aligned} \tag{17}$$

To show that policy improvement applies for the ϵ-greedy probabilistic policy used by ALEX, let π' be the ϵ-greedy policy. We show in Appendix A that:

$$Q^{\pi}(s, \pi'(s)) \geq V^{\pi}(s) \tag{18}$$

Equation 18 shows that policy improvement is sound for the ϵ-greedy policy. Using a greedy policy guarantees improvement on every step except when an optimal policy is reached. This analysis is independent of how the action-value functions are determined at each iteration.

6. OPTIMIZATIONS TO ALEX

In this section, we describe some optimizations to improve execution time and reduce the number of iterations required for the convergence of ALEX.

6.1 Filtering to Reduce the Search Space

ALEX searches in the space of all possible links between entities in the two data sets. It is computationally expensive to: (1) Construct the space of feature sets for each pair of entities from both data sets. (2) Search for candidate links in this space. It is important to reduce the search space to eliminate unlikely links since the number of correct links is considerably small compared to the number of all possible links.

When computing a value for a feature (i.e., a similarity score between two objects associated with two predicates), we require the value to pass a certain threshold θ. Feature values less than θ are set to zero. Feature sets that do not have any positive values are dropped. In our experiments, we use $\theta = 0.3$.

6.2 Partitioning the Search Space

The search conducted by ALEX in the space of possible links can be parallelized by partitioning the space into independent partitions that do not require communication. In order to achieve this, we partition the larger data set and generate feature sets between each partition and all entities in the smaller data set. Assume the first data set Ds_1 is larger than the second data set Ds_2. We partition Ds_1 into $\{Ds_{11} \cup Ds_{12} \cup \cdots \cup Ds_{1n}\}$. Feature sets are generated for each pair $\{(Ds_{11}, Ds_2), (Ds_{12}, Ds_2), \cdots, (Ds_{1n}, Ds_2)\}$. Feedback can then be directed to all partitions so that ALEX can take actions and explore new links in the partition. The different partitions can be independently explored in parallel, either on different CPU cores of the same machine or on multiple machines in a distributed setting.

ALEX uses a simple partitioning technique that we call *equal-size partitioning.* Equal-size partitioning divides the larger data set into equal-sized partitions in a round-robin fashion. That is, the ith entity is in partition $i\ mod\ n$, where n is the number of partitions. Equal-size partitioning enables parallelism that significantly reduces execution time without sacrificing the quality of candidate links.

6.3 Optimizations for Fast Convergence

ALEX's actions (exploring links that did not exist in the set of candidate links) lead to fast improvement in recall. However, ALEX can also generate incorrect links, which reduces precision. Negative feedback would eventually correct these errors by removing incorrect links. Based on negative feedback, ALEX learns that some action resulted in worse returns. During policy improvement, ALEX would change the policy so that this action is chosen only with small probability. However, relying only on policy improvement to remove incorrect links may result in slow convergence. In order to speed up convergence we develop two optimizations that improve precision without waiting for policy improvement: *blacklist* and *rollback*.

Blacklist: When negative feedback is received over a link, it is now known that the link is incorrect, so it is added to a list of incorrect links. The blacklist is used to prevent links that are known to be incorrect from being returned by ALEX when exploring links at any state in the future.

Rollback: The probabilistic nature of the ϵ-greedy policy used by ALEX allows it to choose incorrect actions at any state to learn how to make better choices when choosing future actions. Some actions may result in the discovery of a large number of incorrect links. When this happens, it is a wise choice to rollback and remove the links generated by such actions. ALEX traces feedback on links to know by which state-action pair these links were generated. When a sufficient number of negative feedback items is received over links generated by a specific state-action pair, a rollback process is initiated, and all links generated by this state-action pair are removed. However, links removed without being marked with negative feedback are not added to the blacklist since they may include some correct links. These links can be discovered later by another state-action pair with a better average return.

Data Set	Version	Field	Triples
DBpedia	3.5.1	Multi-domain	43.6M
OpenCyc	4.0	Multi-domain	1.6M
NYTimes	2010-01-13	Media	335K
Drugbank	2010-11-25	Life Sciences	767K
Lexvo	2013-02-09	Linguistics	715K
Semantic Web Dogfood	2014-05-29	Publications	337K
DBpedia (NBA)	3.5.1	Basketball Players	56K
OpenCyc (NBA)	4.0	Basketball Players	726

Table 1: Data sets used in the experiments.

The rollback optimization is particularly useful for handling incorrect feedback due to errors in the data or errors by the user. It may be possible to refine the feedback so that ALEX uses only high quality feedback obtained from a large number of users (e.g., using techniques from [16]). However, we should never expect feedback to be 100% correct, regardless of the measures taken to improve its quality. When incorrect feedback is received by ALEX, it will take an action that can be rolled-back if future feedback contradicts the incorrect feedback. A study of the effect of incorrect feedback is presented in Appendix C.

7. EXPERIMENTAL EVALUATION

7.1 Experimental Setup

Data sets: We use the real data sets shown in Table 1. DBpedia contains structured data extracted from Wikipedia, and OpenCyc[8] contains parts of the Cyc knowledge base of everyday knowledge. Both of these data sets cover multiple domains, and are in the center of the Linked Open Data cloud, with many links to other data sets. Each of our experiments aims to link one of these two data sets with one of the other data sets in Table 1, which are from different domains. NYTimes contains data about locations, people, and organizations. Lexvo[9] contains data about human languages. Semantic Web Dogfood contains data about conferences and workshops about the Semantic Web. Two subsets of the DBpedia and OpenCyc about NBA basketball players are extracted to evaluate ALEX in more domain-specific situations. In our experiments, we use the versions the of DBpedia, NYTimes, and Drugbank data sets from FedBench[10], a benchmark suite to test the efficiency of federated query processing systems, and we use OpenCyc and Lexvo from the Linked Open Data cloud.

Initial Set of Links: We use PARIS [21] as the automatic linking algorithm for generating the initial set of candidate links for ALEX. We chose PARIS because it was shown to outperform other techniques, and it is not domain specific. PARIS produces links where each link is associated with a score. In order to find better quality links from PARIS, we only consider links with score greater than 0.95. Lowering this threshold does not improve the recall value, but it lowers the precision.

[8]http://sw.opencyc.org/downloads/opencyc_owl_downloads/opencyc-latest.owl.gz

[9]http://www.lexvo.org/resources/lexvo_latest.rdf.gz

[10]http://fedbench.fluidops.net/

Ground Truth: The data sets that we use are part of the Linked Open Data cloud, which means that they are already linked. Some of the data sets are manually linked. Others are linked automatically and refined manually by human experts. We remove all existing links between the data sets and use them as our ground truth. In addition, we randomly inspect samples from the ground truth to remove any incorrect links. We then run PARIS over the pair of data sets to be linked to discover candidate links, which ALEX uses as initial candidate links. We manually inspect the initial candidate links generated by PARIS to find correct links that do not exist in the ground truth. If we find any such links, we add them to the ground truth.

Generating Feedback: We randomly choose a link out of the set of candidate links and compare it to the ground truth. If the link exists in the ground truth, a positive feedback item is returned to ALEX. If the link is incorrect (i.e., does not exist in the ground truth), a negative feedback item is returned.

Evaluation Metrics: We evaluate the efficiency of ALEX by comparing the candidate links it generates to the ground truth. We perform this comparison after each policy evaluation - policy improvement iteration, i.e., after each episode of feedback. We evaluate the quality of candidate links using precision $P = \frac{|C \cap G|}{|C|}$, recall $R = \frac{|C \cap G|}{|G|}$, and F-measure $F = \frac{2PR}{P+R}$, where C is the set of candidate links after each episode, and G is the ground truth.

We also measure the execution time that ALEX requires to converge. The execution time includes the exploration of new candidate links and improving the policy after each episode.

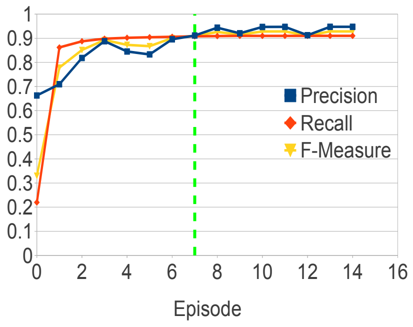
Default Settings: The step size in ALEX is the offset away from the feature score that ALEX searches around. For example, if a feature has a score of 0.8 and the step size is 0.05, ALEX will find links whose feature score is in the range [0.75, 0.85]. The default value of the step size is 0.05. The episode size is the number of feedback items collected before starting policy improvement. The default episode size is 1000. All our experiments use equal-size partitioning to partition the space of feature sets into 27 partitions.

Execution Environment: ALEX is implemented in Java. We ran our experiments on a shared server running Linux Ubuntu 12.04.3 with 64 AMD Opteron processors at 2.6 GHz and 256 GB of memory. Our memory usage for the largest data sets never exceeded 30 GB.

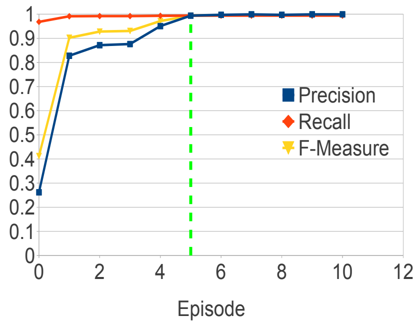
Section 7.2 evaluates the quality of the links discovered by ALEX, and Section 7.3 evaluates the efficiency of ALEX and the effect of the optimizations it employs.

7.2 Quality of Links

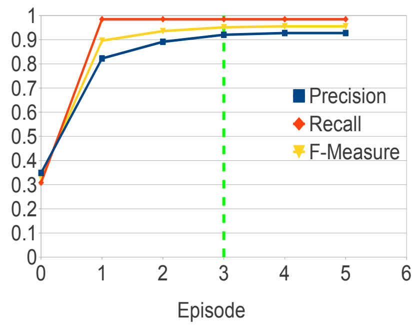
We envision ALEX being used in one of two settings: 1. Batch Mode: A service provider can give users the ability to query multiple, large linked RDF data sets. In this setting, the service provider collects feedback from many users over a large number of links between different parts of the data sets. ALEX applies the feedback in batches, using a large episode size, in order to ensure that there is sufficient feedback over different parts of the data sets. In this setting, we use the default episode size of 1000 (e.g., 1000 users providing 1 feedback item each). 2. Specific Domains: Individual users can develop applications that target more specific domains, either small data sets or subsets of large data sets. The user feedback is focused on a specific domain (e.g., a



(a) DBpedia - NYTimes

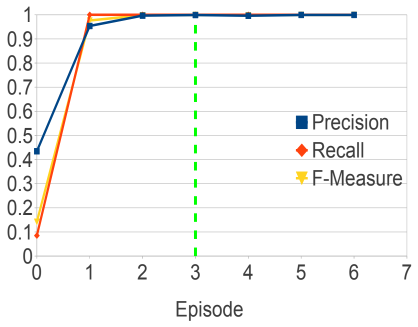


(b) DBpedia - Drugbank

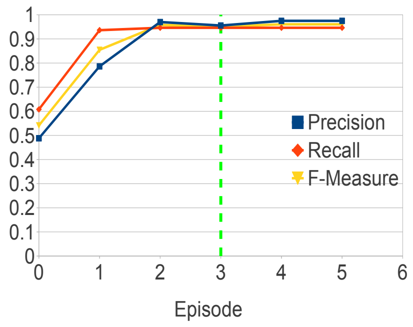


(c) DBpedia - Lexvo

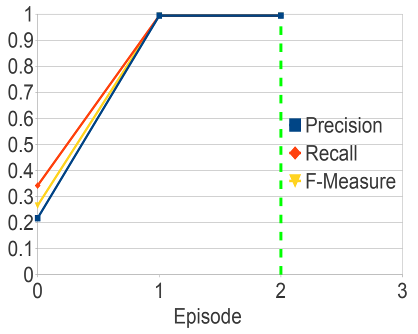
Figure 2: Quality of links between DBpedia and NYTimes, Drugbank, and Lexvo.



(a) OpenCyc - NYTimes



(b) OpenCyc - Drugbank



(c) OpenCyc - Lexvo

Figure 3: Quality of links between OpenCyc and NYTimes, Drugbank, and Lexvo.

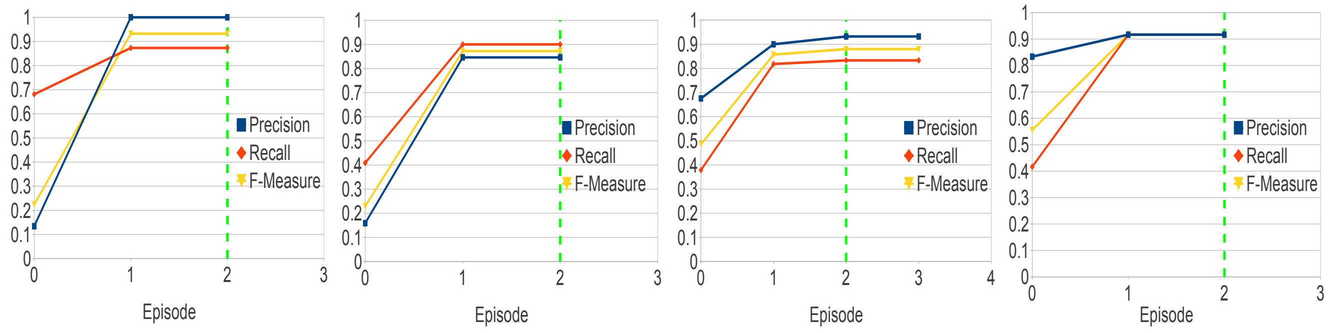
small part of the DBpedia data set), and she expects to see quick improvement in link quality based on her feedback. In this setting, we use a small episode size of 10. We evaluate the first setting in Section 7.2.1 and the second setting in Section 7.2.2. In both settings, we use a strict rule for convergence. `ALEX` stops when there is no change at all in the set of candidate links between episodes. We also show the episode at which fewer than 5% of the links change compared to the previous episode. This can be used as a more relaxed convergence rule.

7.2.1 *ALEX in Batch Mode*

Figure 2 shows how `ALEX` performs in batch mode on the following pairs of data sets: DBpedia - NYTimes, DBpedia - Drugbank, and DBpedia - Lexvo. Figure 2(a) shows the quality of links between DBpedia and NYTimes after each episode (i.e., iteration of policy evaluation - policy improvement). The figure shows how the recall value is significantly improved after the first episode from a low value of around 0.2 (i.e., most ground truth links are not included) to almost 0.9. This means that a large number of links have been discovered and added to the set of candidate links after only one episode. In some iterations, the precision is hurt by adding some incorrect links to the set of candidate links. However, `ALEX` recovers fast and keeps improving both precision and recall until it converges. The vertical green line in this and subsequent figures shows the episode at which the number of changed links from the previous episode is less than 5%. This relaxed convergence happens after 7 episodes in this experiment, while full convergence (i.e., no change in links) happens after 14 episodes. A total of 7568 new links were discovered by `ALEX`. The total number of links in the ground truth is 10968.

While linking DBpedia and NYTimes demonstrates a case in which PARIS is able to generate a set of initial candidate links with relatively good precision but with bad recall, linking DBpedia and Drugbank demonstrates a case in which the initial candidate links have very good recall but bad precision. Figure 2(b) shows how `ALEX` performs in this case. The figure shows that the automatically generated links have a low starting precision value (less than 0.3), and high recall value (over 0.95). `ALEX` is able to significantly improve the precision value after three episodes. The recall value is also improved. `ALEX` converges after 10 episodes in this experiment (5 episodes with the relaxed condition), reaching an F-measure of 0.99. The total number of links in the ground truth for this pair of data sets is 1514. A total of 70 new links were discovered by `ALEX`. This is a small number since recall was already high. Most of the work done by `ALEX` is in removing incorrect links.

Figure 2(c) shows a different case, where both precision and recall have low values. The number of links in the ground truth is 4364. A total of 3011 new links were discovered by `ALEX`. The figure shows that `ALEX` significantly improves the recall after the first episode and no longer improves it after the second episode. However, it improves



(a) DBpedia - Semantic Web Dogfood (b) OpenCyc - Semantic Web Dogfood (c) DBpedia (NBA) - NY-Times (d) OpenCyc (NBA) - NY-Times

Figure 4: Quality of links for specific domains: publications and NBA basketball players.

the precision for 3 more episodes until it converges after 5 episodes (3 episodes with the relaxed condition).

The results of similar experiments using OpenCyc instead of DBpedia are shown in Figure 3. `ALEX` performs as effectively in these experiments as it did in Figure 2. The number of ground truth links is 2965 for OpenCyc - NYTimes, 204 for OpenCyc - Drugbank, and 383 for OpenCyc - Lexvo.

7.2.2 *ALEX for Specific Domains*

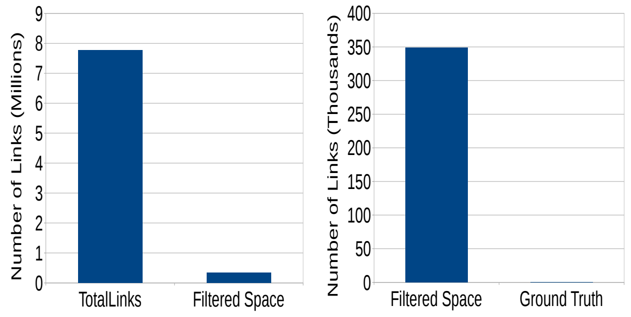
In this section we investigate the performance of `ALEX` in the single-user, specific domain setting. We use a small episode size of 10 feedback items since users expect quick improvement in this setting. Figure 4(a) shows the performance of `ALEX` between DBpedia and the Semantic Web Dogfood data set which has information about conferences and workshops about the Semantic Web. The number of ground truth links between DBpedia and Semantic Web Dogfood is small (461), although the Semantic Web Dogfood data set has more triples than the NYTimes data set. The links mainly connect universities and technical companies from both datasets. A total of 84 new links were discovered by `ALEX`. Figure 4(a) shows that `ALEX` achieves very good quality of links and converges in 2 episodes (i.e., significant improvement after 10 feedback items, and full convergence after 20).

Figure 4(b) shows a similar experiment, replacing DBpedia with OpenCyc. In this case, there are 110 links in the ground truth, and `ALEX` discovers 51 new links.

In addition, we extract data about NBA basketball players from DBpedia and OpenCyc to demonstrate another domain-specific situation where an application is interested in finding all news about NBA basketball players (active or retired). Figures 4(c) and 4(d) show the results of this experiment. There are 93 ground truth links in Figure 4(c), and `ALEX` discovers 43 new links after one episode. The number of ground truth links in Figure 4(d) is 35, and `ALEX` discovers 19 new links.

7.3 Efficiency of ALEX

In this section, we investigate the efficiency (i.e., running time) of `ALEX`, and the effect of the different optimizations to speed up convergence, which were described in Section 6. For these experiments, we use the DBpedia and NYTimes data sets. These data sets are challenging for `ALEX` since they contain data from more heterogeneous domains than



(a) Total links vs. filtered search space (b) Filtered search space vs. ground truth

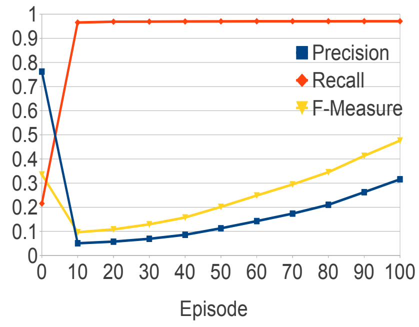
Figure 5: Comparing number of links: total possible links vs. filtered search space vs. ground truth.

the rest of data sets. In addition, they are the data sets for which PARIS was able to discover only a small fraction of the links, as seen in Figure 2(a). This pair of data sets also has the largest number of links between a multi-domain data set and a data set from any other domain. Only the two multi-domain data sets have more links between them, and we use them to test efficiency in Appendix B.

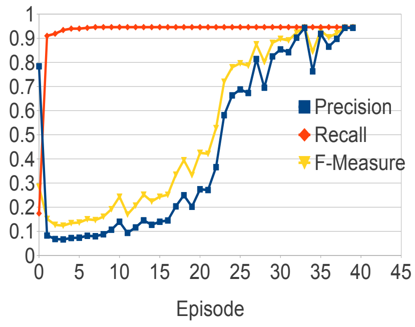
Filtering to Reduce the Search Space: Figure 5(a) shows the total number of links that can be generated between the first partition of the DBpedia data set and the whole data set of NYTimes, compared to the number of links after filtering using a threshold $\theta = 0.3$. The figure shows that filtering reduces the search space by 95%.

Figure 5(b) shows the number of filtered links compared to number of the ground truth links that can be generated out of this partition. The figure shows that ground truth represents only 0.2% of the filtered links, demonstrating the efficiency of `ALEX`, which is able to discover correct links in such a large space.

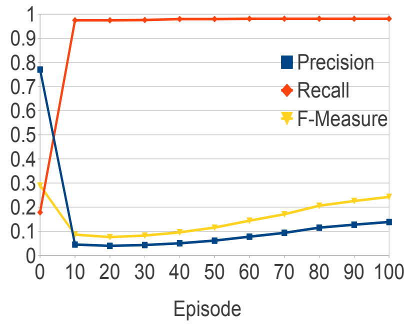
Blacklist: Figure 6 shows a comparison between `ALEX` with and without the blacklist optimization. Figure 6(a) shows that using a blacklist gives a slight improvement in F-measure over not using it. However, Figure 6(b) shows that using a blacklist significantly decreases the fraction of negative feedback, which the user provides on incorrect links



(a) Quality without rollback

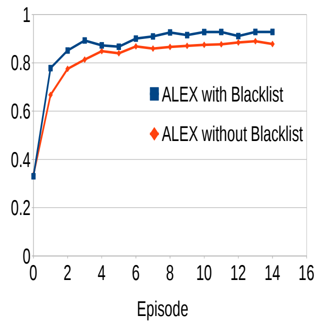


(b) A partition that converges

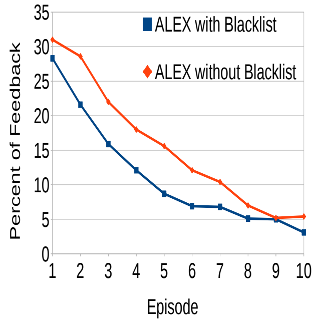


(c) A partition that does not converge

Figure 7: Effect of rollback: (a) quality without rollback, (b) a partition that converges, and (c) a partition that does not converge.



(a) F-measure



(b) Negative feedback

Figure 6: Effect of the blacklist: (a) F-measure, and (b) negative feedback.

between the data sets. Using a blacklist does not affect the execution time of ALEX. Intuitively, a black list is useful because when a user provides negative feedback on a link she should not need to provide this feedback again.

Rollback: ALEX learns by interacting with the environment. This means that it can sometimes make wrong decisions. These wrong decisions can result in exploring a large number of incorrect candidate links, significantly reducing the quality of the discovered links. If rollback is not used, ALEX can recover from wrong decisions only very slowly. Figure 7 shows the importance of the rollback optimization. Figure 7(a) shows the quality measures of ALEX without using the rollback optimization. This figure should be contrasted to Figure 2(a), which shows ALEX with rollback (the default). The figure shows that after the first episode, precision drops to a very low value. The figure also shows that it is hard to recover from the wrong decisions made during the first episode. After 100 episodes, which is the maximum number of iterations allowed by ALEX, precision is a little over 0.3. Figure 7(a) shows the overall quality of all partitions. If we examine partitions independently, we find that some partitions are able to recover from wrong decisions made by ALEX, while others are not. Figure 7(b) shows an example of a partition that is able to recover from the wrong decisions and converges in 40 episodes. The same partition converges in 7 episodes when rollback is applied. However, another partition, shown in Figure 7(c), cannot recover from wrong decisions.

Execution Time: In the experiment with DBpedia and NYTimes shown in Figure 2(a), ALEX finishes execution in 97 minutes, which is the execution time of the slowest partition. This is approximately 7 minutes per episode. The average execution time of all partitions is approximately 64 minutes. In the specific domain experiment shown in Figure 4(c), ALEX finishes in approximately 4 seconds. This is approximately 1.3 second per episode (10 feedback items). The faster convergence in the specific domain setting is because the data sets and the amount of feedback are smaller. Thus, in batch mode ALEX takes a few minutes per episode, while in interactive mode it takes a few seconds. We consider this to be acceptable efficiency.

Additional experiments are presented in the appendices. We study the efficiency of ALEX when linking the multi-domain data sets in Appendix B. We demonstrate that ALEX can recover from incorrect feedback in Appendix C. And we demonstrate that ALEX is not overly sensitive to parameter values in Appendix D.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented ALEX, a system that utilizes user feedback on queries over linked data to remove incorrect links and discover new links that did not exist between the data sets. When a user provides positive feedback on a link, ALEX finds new links that are similar to this link. This exploration is conducted by taking one pair of attributes from the two data sets and exploring around the similarity value between these two attributes. ALEX uses a probabilistic policy to choose the attributes to explore around. This policy is learned and improved using a Monte Carlo reinforcement learning approach. Using this approach, ALEX learns how to find new links as the user continues giving feedback on query answers. ALEX uses several optimizations to speed up convergence. Our experiments with real world data sets show the effectiveness and efficiency of ALEX.

An important direction for future work is confirming the effectiveness of ALEX through user studies using real applications on the Linked Open Data cloud. In such studies, users are likely to generate some incorrect feedback, which should enable us to validate the robustness of ALEX beyond our current set of experiments.

9. REFERENCES

[1] A. Aboulnaga and K. El Gebaly. μbe: User guided source selection and schema mediation for internet scale data integration. In *IEEE Int. Conf. on Data Engineering (ICDE)*, 2007.

[2] M. Acosta, A. Zaveri, E. Simperl, D. Kontokostas, S. Auer, and J. Lehmann. Crowdsourcing linked data quality assessment. In *Proc. Int. Semantic Web Conf. (ISWC)*. 2013.

[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. DBpedia: A nucleus for a web of open data. In *Proc. Int. Semantic Web Conf. (ISWC)*. 2007.

[4] D. Aumueller, H.-H. Do, S. Massmann, and E. Rahm. Schema and ontology matching with COMA++. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2005.

[5] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 2001.

[6] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *ACM Trans. on Knowledge Discovery from Data (TKDD)*, 2007.

[7] C. Bizer, T. Heath, and T. Berners-Lee. Linked data-the story so far. *Int. Journal on Semantic Web and Information Systems*, 5(3), 2009.

[8] C. Bizer, T. Heath, K. Idehen, and T. Berners-Lee. Linked data on the web. In *Proc. Int. World Wide Web Conf. (WWW)*, 2008.

[9] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2008.

[10] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. ZenCrowd: leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. In *Proc. Int. World Wide Web Conf. (WWW)*, 2012.

[11] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Web-scale information extraction in KnowItAll (preliminary results). In *Proc. Int. World Wide Web Conf. (WWW)*, 2004.

[12] A. Ferrara, D. Lorusso, and S. Montanelli. Automatic identity recognition in the semantic web. In *Proc. Int. Workshop on Identity and Reference on the Semantic Web (IRSW)*, 2008.

[13] J. Gracia, M. d'Aquin, and E. Mena. Large scale integration of senses for the semantic web. In *Proc. Int. World Wide Web Conf. (WWW)*, 2009.

[14] W. Hu, J. Chen, and Y. Qu. A self-training approach for resolving object coreference on the semantic web. In *Proc. Int. World Wide Web Conf. (WWW)*, 2011.

[15] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2008.

[16] R. McCann, W. Shen, and A. Doan. Matching schemas in online communities: A web 2.0 approach. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, 2008.

[17] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning.* The MIT Press, 2012.

[18] B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. In *The Semantic Web: Research and Applications*. Springer, 2008.

[19] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization techniques for federated query processing on linked data. In *Proc. Int. Semantic Web Conf. (ISWC)*. 2011.

[20] S. P. Singh and R. S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1-3), 1996.

[21] F. M. Suchanek, S. Abiteboul, and P. Senellart. PARIS: probabilistic alignment of relations, instances, and schema. *Proc. VLDB Endow. (PVLDB)*, 5(3), 2011.

[22] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, 1998.

[23] J. Volz, C. Bizer, M. Gaedke, and G. Kobilarov. Silk-A link discovery framework for the web of data. In *Proc. Workshop on Linked Data on the Web (LDOW)*, 2009.

[24] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. *Proc. VLDB Endow. (PVLDB)*, 6(6), 2013.

[25] Z. Yan, N. Zheng, Z. G. Ives, P. P. Talukdar, and C. Yu. Actively soliciting feedback for query answers in keyword search-based data integration. *Proc. VLDB Endow. (PVLDB)*, 6(3), 2013.

APPENDIX

A. DERIVATIONS AND PROOFS

A.1 State Value Function (Equation 10)

$$V^{\pi}(s) = E_{\pi}\left\{\sum_{k=t+1}^{T} r_k | s_t = s\right\}$$
$$= E_{\pi}\left\{r_{t+1} + \sum_{k=t+2}^{T} r_k | s_t = s\right\}$$
$$= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + E_{\pi}\left\{\sum_{k=t+2}^{T} r_k | s_{t+1} = s'\right\}\right]$$
$$= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + V^{\pi}(s')\right]$$

A.2 Optimal Value Function (Equation 11)

$$V^*(s) = max_a Q^{\pi^*}(s,a)$$
$$= max_a E_{\pi^*}\left\{R_t | s_t = s, a_t = a\right\}$$
$$= max_{a \in A(s)} E_{\pi^*}\left\{\sum_{k=t+1}^{T} r_k | s_t = s, a_t = a\right\}$$
$$= max_{a \in A(s)} E_{\pi^*}\left\{r_{t+1} + \sum_{k=t+2}^{T} r_k | s_t = s, a_t = a\right\}$$
$$= max_{a \in A(s)} E_{\pi^*}\left\{r_{t+1} + V^*(s_{t+1}) | s_t = s, a_t = a\right\}$$
$$= max_{a \in A(s)} \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + V^*(s')\right]$$

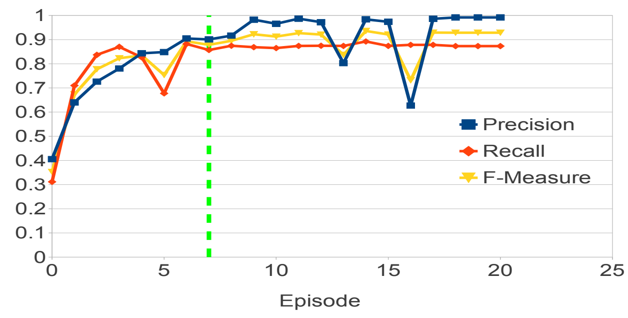


Figure 8: Quality of links between the two largest data sets: DBpedia and OpenCyc.

A.3 Policy Improvement Proof (Equation 14)

$$
\begin{aligned}
&V^{\pi}(s) \leq Q^{\pi}(s,\pi'(s))\\
&V^{\pi}(s) \leq E_{\pi'}\left\{r_{t+1}+V^{\pi}(s_{t+1})|s_t=s\right\}\\
&V^{\pi}(s) \leq E_{\pi'}\left\{r_{t+1}+Q^{\pi}(s_{t+1},\pi'(s_{t+1}))|s_t=s\right\}\\
&V^{\pi}(s) \leq E_{\pi'}\left\{r_{t+1}+E_{\pi'}\{r_{t+2}+V^{\pi}(s_{t+2})\}|s_t=s\right\}\\
&V^{\pi}(s) \leq E_{\pi'}\left\{r_{t+1}+r_{t+2}+V^{\pi}(s_{t+2})|s_t=s\right\}\\
&V^{\pi}(s) \leq E_{\pi'}\left\{r_{t+1}+r_{t+2}+r_{t+3}+V^{\pi}(s_{t+3})|s_t=s\right\}\\
&V^{\pi}(s) \leq V^{\pi'}(s)
\end{aligned}
$$

A.4 Proof of Improvement Using the ϵ-greedy Policy (Equation 18)

$$
\begin{aligned}
&Q^{\pi}(s,\pi'(s)) = \sum_a \pi'(s,a)Q^{\pi}(s,a)\\
&= \frac{\epsilon}{|A(s)|}\sum_a Q^{\pi}(s,a) + (1-\epsilon)max_a Q^{\pi}(s,a)\\
&\geq \frac{\epsilon}{|A(s)|}\sum_a Q^{\pi}(s,a) + (1-\epsilon)\sum_a \frac{\pi(s,a)-\frac{\epsilon}{|A(s)|}}{1-\epsilon}Q^{\pi}(s,a)
\end{aligned}
$$

The transition from the equality to the inequality is because the sum of the second term is the weighted average with non-negative weights summing to one ($\sum_a \frac{\pi(s,a)-\frac{\epsilon}{|A(s)|}}{1-\epsilon}$), and therefore must be less than or equal to the largest number averaged ($max_a Q^{\pi}(s,a)$).

$$
\begin{aligned}
&Q^{\pi}(s,\pi'(s)) \geq \frac{\epsilon}{|A(s)|}\sum_a Q^{\pi}(s,a) + \sum_a \pi(s,a)Q^{\pi}(s,a)\\
&-\frac{\epsilon}{|A(s)|}\sum_a Q^{\pi}(s,a)\\
&Q^{\pi}(s,\pi'(s)) \geq \sum_a \pi(s,a)Q^{\pi}(s,a)\\
&Q^{\pi}(s,\pi'(s)) \geq V^{\pi}(s)
\end{aligned}
$$

B. LINKING THE TWO MULTI-DOMAIN DATA SETS

We stress-test ALEX by using it to link the two multi-domain data sets in our experiments (DBpedia and OpenCyc), using the default episode size of 1000. Linking these two data sets is challenging because they are the largest in our experiments. Furthermore, they span multiple semantically diverse domains so they use a large number of labels for their predicates, and hence generate a large number of features. There are 41039 links between the two data sets in the ground truth, the largest in our experiments. Figure 8 shows that ALEX converges after 20 episodes with an F-measure greater than 0.9 (7 episodes with the relaxed condition). ALEX started with 12227 correct candidate links obtained from PARIS, and discovered 23476 additional correct links.

C. EFFECT OF INCORRECT FEEDBACK

In this paper, we assume that user feedback is always correct. However, in real-life scenarios, users may not agree on the correctness of query answers. Therefore, incorrect feedback items may be encountered. In order to evaluate ALEX in this context, we generate random incorrect feedback items such that 10% of the feedback items received by ALEX are incorrect. We evaluate ALEX with 10% incorrect feedback on the DBpedia - NYTimes data sets, using the default episode size of 1000. Figure 9 shows the precision, recall, and F-measure for ALEX with 10% incorrect feedback, and the corresponding values when all feedback is correct (from Figure 2(a)). The recall value in Figure 9(b) does not vary much, demonstrating that the reinforcement learning techniques used in ALEX to improve recall are robust to incorrect feedback. The precision values shown in Figure 9(a) are slightly worse when there is incorrect feedback, and this also affects the F-measure in Figure 9(c). ALEX improves precision by removing incorrect links from the set of candidate links, and these incorrect links can be removed only when negative feedback is received over them. When there is incorrect feedback, a constant stream of positive feedback is received over incorrect links, so these incorrect links stay in the set of candidate links. Nevertheless, the degradation in precision is relatively small, and ALEX is able to produce good results even in the presence of incorrect feedback.

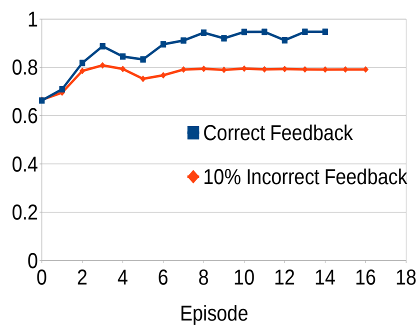
D. SENSITIVITY OF ALEX

We use the DBpedia - NYTimes data sets to test the sensitivity of ALEX to the step size and episode size parameters.

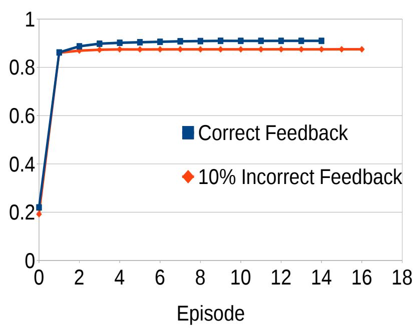
Step Size: In this experiment, we use the default episode size of 1000, and we vary the step size. Changing the step size slightly affects how ALEX performs. Increasing the step size means that we expand the search area around the feature score chosen by the current policy at the current state. Decreasing the step size means that we narrow it. Figure 10 shows ALEX with step sizes 0.01, 0.05 (default), and 0.1. Figure 10(a) shows that the F-measure does not vary significantly with higher step size, getting slightly better. Figure 10(b) gives a deeper insight by showing the variance in recall, where the gap between different step sizes is more obvious. The figure shows how increasing the search area around feature score results in more correct links being discovered. Precision is not reported because it has a trend similar to that of Figure 2(a).

However, increasing the search area does not come for free. The total execution time is determined by the partition that finishes last. ALEX with a step size of 0.1 has much higher execution time than with a step size of 0.05 or 0.1. The last partition to finish for step size 0.1 takes over 210 minutes, whereas it takes 97 minutes for a step size of 0.05 and 89 minutes for a step size of 0.01.

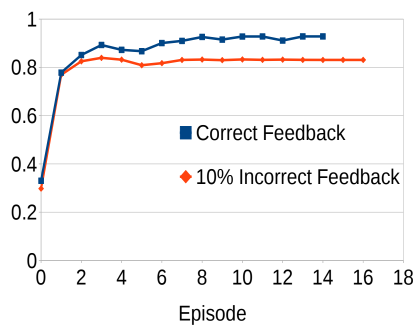
Another drawback of increasing the step size is that ALEX will discover more incorrect links in the search area. This is



(a) Precision

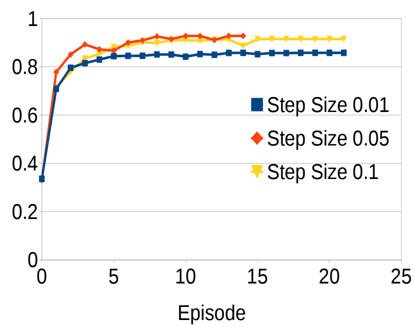


(b) Recall

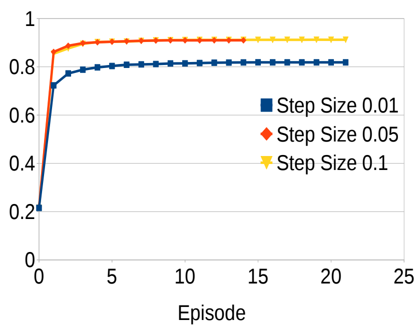


(c) F-measure

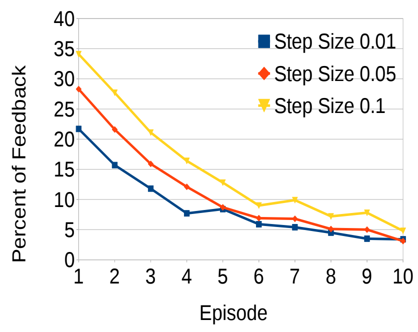
Figure 9: ALEX with correct feedback and with 10% incorrect feedback.



(a) F-measure



(b) Recall



(c) Negative feedback

Figure 10: ALEX with different step sizes: (a) F-measure, (b) recall, and (c) negative feedback.

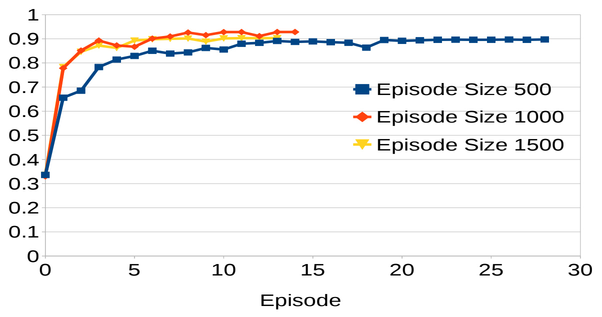


Figure 11: F-measure for ALEX with different episode sizes.

reflected directly in the number of negative feedback items received from the user. Figure 10(c) shows the percentage of feedback received in every episode that is negative. This negative feedback is caused by incorrect links. The figure shows this value for the first 10 episodes for different step sizes. The figure shows that during the first episode, ALEX with a step size of 0.1 receive almost 35% negative feedback of the 1000 feedback items in the episode, compared to less than 30% for a step size of 0.05 and a little over 20% for 0.01. For later episodes, ALEX improves its policy and learns to take better actions. This reduces the percentage of negative feedback. However, the trend that a bigger step size results in a larger percent of negative feedback continues.

Episode Size: Like changing the step size, changing the episode size slightly affects the quality of the discovered links. Figure 11 shows a comparison of ALEX using an episode size of 500, 1000 (default), and 1500. During each episode, policy evaluation uses the current policy to take action after every feedback item. At the end of an episode, the policy is improved for the next episode. Figure 11 shows the F-measure for the three episode sizes. The F-measures are very close to each other, with episode sizes 1000 and 1500 slightly outperforming episode size 500. A larger episode size results in ALEX taking fewer episodes to converge, since each episode has more feedback. ALEX converges in 26, 14, and 13 episodes for episode size 500, 1000, and 1500, respectively.

The experiments in this appendix show that ALEX is sensitive to changes in the values of its parameters. That is, the parameters have a noticeable effect on performance. However, the sensitivity is not so high as to make ALEX unstable and reliant on highly accurate parameter settings: the difference between the best case and the worst case performance is not too high, and reasonable choices of parameter values work well.