Focused Retrieval over Richly-Annotated Collections

Matthew W. Bilotti, Le Zhao, Jamie Callan and Eric Nyberg
Language Technologies Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, Pennsylvania 15235
{ mbilotti, lezhao, callan, ehn }@cs.cmu.edu

ABSTRACT

This paper introduces a theoretical framework for focused retrieval, based on a formalism called the annotation graph. Annotation graph-based retrieval provides a rich retrieval representation that directly supports query-time constraint-checking of arbitrary relations. This representation can support focused retrieval tasks, such as Question Answering systems, which often have constraint types as part of their information needs that can not be queried under many retrieval models. The problem of annotation graph-based retrieval is mapped onto existing XML element retrieval functionality in the Indri search engine. The remainder of the paper serves to identify and discuss the issues that emerged and illustrate by example what in our opinion constitutes the upcoming research challenges facing the focused retrieval community.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing; H.3.3 [**Information Search and Retrieval**]: Information Search and Retrieval

General Terms

Theory, Design

Keywords

Focused retrieval, question answering, type systems, annotation graphs

1. INTRODUCTION

The fundamental difference between focused retrieval tasks such as Question Answering (QA), XML Element Retrieval (XML-IR) and passage retrieval and the general *ad hoc* retrieval task is that the unit of retrieval is smaller than that of a document. The document retrieval paradigm forces users to read through the retrieved documents to locate the information that satisfies their information needs. Focused retrieval paradigms aim to ease this burden on the user by locating and retrieving the relevant information directly.

Text retrieval systems score by measuring the similarity between some representation of the text and a query, which is expressed in that same representation and encodes the constraints in the user's information need. Modern document retrieval systems use a lean text representation that makes indexing and retrieval convenient, but that is not capable of fully encoding all information needs, or even most of them. All they can represent is a notion of relevance based on ordering, proximity and frequency of terms. Though this representation can approximate many information needs well enough to be successful on the *ad hoc* retrieval task, it is not as well suited to the focused retrieval task.

The job of a focused retrieval system is significantly more difficult. Because it must assign scores to small units of text, it can not afford to ignore any of the constraints in the information need. Focused retrieval systems aim to provide a rich internal representation, both for the query and for the indexed texts, that reduces the mismatch between the information need and the query by directly supporting constraint-checking of more of the component constraints of the information need.

This paper introduces a formalism called the *annotation graph*, and a theoretical retrieval framework based on annotation graph retrieval. The annotation graph-based retrieval framework supports query-time constraint-checking of relations over information elements encoded in the annotation graph. We argue that this kind of rich information representation and powerful retrieval-time constraint-checking is necessary to support focused retrieval tasks, such as QA. This paper chronicles our experience mapping the annotation graph-based retrieval task onto existing XML element retrieval functionality in the Indri search engine. We conclude with a discussion of the issues that emerged, and a set of examples that illustrate what are, in our opinion, important research challenges in focused retrieval.

2. ANNOTATION GRAPH-BASED RETRIEVAL FRAMEWORK

This section presents a theory of focused retrieval based on a formalism called the annotation graph, which serves as a shared representation for not only information needs, but also the information content of texts.

2.1 Type Systems

Type System T	$T = \left(Te = \left\{te_1, te_2, ..., te_{\|Te\|}\right\}, Tr = \left\{tr_1, tr_2, ..., tr_{\|Tr\|}\right\}\right)$
Information Element Type te_i	$te_i = (name, parent) \wedge (parent \in Te \vee parent = \emptyset)$
Relation Type tr_i	$tr_i = (name, domain, range) \wedge domain, range \in Te$
Annotation Graph G	$G = \left(E = \left\{e_1, e_2, ..., e_{\|E\|}\right\}, R = \left\{r_1, r_2, ..., r_{\|R\|}\right\}, ts\right)$
Information Element e_i	$e_i = (type) \wedge type \in Te \wedge ts = (Te, Tr)$
Relation r_i	$r_i = (type, e_d, e_r) \wedge type = (name, domain, range) \in Tr \wedge$ $domain, range \in Te \wedge ts = (Te, Tr) \wedge e_d = (domain) \wedge e_r = (range)$
Collection C	$C = \left\{g_1, g_2, ..., g_{\|C\|}\right\}$
Query q	$q = f(g, C)$

Figure 1: Formal definitions for the theoretical retrieval framework based on annotation graphs

Type System

$$T_{BoW} = \left(\begin{array}{l} Te = \left\{ \begin{array}{l} (document, \emptyset), \\ (sentence, \emptyset), \\ (word, \emptyset), \\ (John, \emptyset), \\ (loves, \emptyset), \\ (Mary, \emptyset) \end{array} \right\}, \\ Tr = \left\{ \begin{array}{l} (encloses, document, sentence), \\ (encloses, document, word), \\ (encloses, sentence, word) \end{array} \right\} \end{array} \right)$$

Annotation Graph

$$G_{BoW} = \left(\begin{array}{l} E = \left\{ \begin{array}{l} d = (document), \\ s = (sentence), \\ j = (John), \\ l = (loves), \\ m = (Mary) \end{array} \right\}, \\ R = \left\{ \begin{array}{l} e_s = (encloses, d, s) \\ e_{j1} = (encloses, d, j) \\ e_{l1} = (encloses, d, l) \\ e_{m1} = (encloses, d, m) \\ e_{j2} = (encloses, s, j) \\ e_{l2} = (encloses, s, l) \\ e_{m2} = (encloses, s, m) \end{array} \right\}, \\ ts = TS_{BoW} \end{array} \right)$$

Figure 2: Example bag-of-words type system and annotation graph for the sentence, *John loves Mary*

Each annotation graph is defined with respect to a particular type system, which serves as a vocabulary for the types of information elements that can exist in the graph, and the relations that can be defined among those elements. A *type system* T is defined as a tuple consisting of two sets, Te and Tr, which contain all valid types for information elements and relations, respectively. An *information element type* te_i is defined as a name and an optional parent type pointer. A *relation type* tr_i is defined as a name, a domain type and a range type. Both the domain and range types must be defined in the same type system. See Figure 1 for the formal definitions of type systems, information element types and relation types.

2.2 Annotation Graphs

A piece of information of any type and complexity can be represented as an annotation graph consisting of a set of information elements (vertices) and a set of relations (edges) that hold between pairs of information elements. Although it is often convenient to think of the annotation graph as a representation of marked-up text, the formalism is general enough to represent audio, images, or any other type of data that can be viewed as a discrete set of elements with relations defined over them. An *annotation graph* is defined as a tuple consisting of a set of information elements E, a set of relations R and a pointer to the graph's type system ts.

Information element is a general term used to describe not only discrete units of raw content, such as the tokens in a text document or the notes in a musical composition, but also annotations representing higher-level or more complex content obtained through human or automatic analysis and mark-up of the raw content. An *information element* is defined with a single attribute, the name of a type defined within the type system of which the element is an instance.

Relations between pairs of elements are typed and asymmetric, holding between an information element of the domain type and an information element of the range type as declared in the type system. Many relations are overt in the raw content, for example, adjacency and ordering information between tokens in a text document or pixels in an image. Other types of relations come from the annotation process that adds higher-level information element types, such as syntactic information for a text document or musical phrase structure for audio data. See Figure 1 for the formal definitions of annotation graphs, information elements and relations.

An example of a simple bag-of-words type system and an annotation graph for the sentence, *John loves Mary*, can be found in Figure 2. The type system can represent words enclosed by sentences, which are, in turn, enclosed by documents. Words can also be directly enclosed by documents, which provides for not only transitivity of enclosure, but also

$$q\,(g=(E,R,ts),C) =$$
$$\frac{1}{|K|}\sum_{i=1}^{|K|}\begin{cases}\frac{tf(k_i,g)}{df(k_i,C)}, (encloses, document, k_i)\in R\\ 0, otherwise\end{cases}$$
$$K=\left\{k_1,k_2,...,k_{|K|}\right\}$$
$$tf(k_i,g)=\sum_{i=1}^{|R|}\begin{cases}1, r_i=(encloses, document, k_i)\\ 0, otherwise\end{cases}$$
$$df(k_i,C)=\sum_{i=1}^{|C|}\begin{cases}1, (encloses, document, k_i)\in R_{g_i}\\ 0, otherwise\end{cases}$$

Figure 3: Bag-of-words retrieval as an instance of the annotation graph-based retrieval framework

the case in which words, such as section titles, occur outside the boundaries of sentences identified in the document. Note that the type system can not represent ordering and proximity constraints between pairs of words, and that as such, it is not powerful enough to distinguish between the sentences *John loves Mary* and *Mary loves John*.

2.3 The Retrieval Process

Let a *collection* be defined as a set of annotation graphs sharing the same type system, each of which serves as a representation of a retrievable item that can be scored in response to a query. In this framework, a *query* is defined as an arbitrary function over an annotation graph and the collection. The query can represent any kind of operation, including but not limited to set operations, such as intersection and union; score combination techniques, such as sum or average; probabilistic operations such as Noisy-OR and not; and weighting schemes, such as *tf.idf* and language models. Query evaluation is a recursive procedure in which queries are broken down into sub-queries, the results of which are then combined by the query function. The queries at the leaves of the tree implement the matching of individual constraints against the annotation graph.

2.4 Example: Bag-of-Words Retrieval

This section describes an example instantiation of the above-proposed retrieval framework. Consider a simple bag-of-words retrieval system that scores the annotation graph g corresponding to a document based on whether the document encloses each k_i of a set of specific keyterms K. One possible implementation is shown in Figure 3. The query q scores the annotation graph g by an arithmetic average over the set of keyterms K, such that the value for each k_i is set to a weight if g contains an *encloses* relation between the document and k_i, and zero otherwise. In this case, the weight is a *tf.idf*-style weight, but any other weighting scheme could be used, or Boolean retrieval could be implemented by setting the weight to 1.

2.5 A Note about Implementation

The retrieval framework proposed in this section relies on a graphical representation, which, while convenient from a formal perspective, might seem to be inefficient both computationally at query-time and also in terms of the size of the index footprint on disk. Without getting into implementation-specific details, it bears emphasizing that this theoretical framework is a view of the retrieval problem meant to aid in understanding the process. The discussion of annotation graph-based retrieval should not be taken as an argument for implementing retrieval systems as full-blown graph similarity engines. The actual index structures in a real implementation could be thought of as a compilation or distillation of the annotation graph representation, and could be tuned to minimize space on disk and maximize the efficiency of query evaluation algorithms operating over them.

3. APPLICATION TO THE QA TASK

Question Answering (QA) is a focused retrieval task that aims to retrieve snippets of text satisfying certain constraints. It can be considered similar to a passage retrieval task, except that the passage size is smaller [2]. The constraint in a QA task is that the passages must contain answers to an input question. QA systems are often implemented as a cascade of document retrieval and an optional passage re-ranking step, followed by answer extraction. The core of a QA system can be thought of as focused retrieval application, bookended by language understanding tools, with question analysis at the beginning and answer selection, validation and presentation at the end.

All QA systems perform some kind of linguistic and semantic analysis on the input question as an initial step to determine how to proceed. The result of this analysis constitutes a specification for an answer to the question, phrased in terms of linguistic and semantic constraints that must hold over a piece of text for it to contain an answer to the question. This rich representation of the information need becomes the input to the focused retrieval task at the core of a QA system, and also, along with the retrieved results, to the answer selection, validation and presentation tasks at the end of the QA process.

Many QA systems, however, rely on a text retrieval component that can not handle this representation directly. These systems are forced to map their information needs into the query representation supported by their text retrieval components, potentially weakening the constraints. Recall the bag-of-words retrieval example introduced earlier. What if a QA system, trying to answer the question, *Who does John love?*, formulated a bag-of-words query consisting of the keyterms *John* and *love*. Under the bag-of-words retrieval model, pieces of text containing those two keyterms would be retrieved, but it is difficult to distinguish between relevant text, such as *John loves Mary* and non-relevant text, such as *Mary loves John*. To filter out these false positives, QA systems often perform on-the-fly linguistic and semantic analysis of the retrieved text and compare the result against the information need, discarding those results that do not satisfy the constraints.

Recently, there has been interest in building a text retrieval interface for QA applications that provides a richer query representation that can directly support retrieval-time checking of certain types of information need constraints [1]. Suppose that a *love* event is a primitive element in the semantics of a QA system to which the same question was asked, *Who does John love?* Provided the collection was properly annotated off-line for instances of *love* events, a text retrieval component capable of query-time constraint-checking would retrieve *John loves Mary*, but *Mary loves John* would not match the query, because *John* is not the actor of the *love* event.

Retrieval-time constraint-checking can also apply to QA systems that do not use a full-blown semantic representa-

[ARG0 *John*] [TARGET *loves*] [ARG1 *Mary*]

$$g = \left(\begin{array}{l} E = \left\{ \begin{array}{l} s = (sentence), \\ t = (target), \\ a0 = (arg0), \\ a1 = (arg1), \\ l = (loves), \\ j = (John), \\ m = (Mary) \end{array} \right\}, \\ R = \left\{ \begin{array}{l} e_t = (encloses, s, t), \\ e_{a0} = (encloses, s, a0), \\ e_{a1} = (encloses, s, a1), \\ e_l = (encloses, s, l), \\ e_j = (encloses, s, j), \\ e_m = (encloses, s, m), \\ e_{l2} = (encloses, t, l), \\ e_{j2} = (encloses, a0, j), \\ e_{m2} = (encloses, a1, m), \\ a_{a0} = (child, t, a0), \\ a_{a1} = (child, t, a1) \end{array} \right\}, \\ ts = TS_{SRL} \end{array} \right)$$

Figure 4: PropBank-style semantic analysis of the sentence, *John loves Mary*, with the corresponding annotation graph

tion internally. Consider a QA system that uses grammatical functions such as *subject*, *object* and *oblique* that can be obtained without semantic analysis. For the question, *Who does John love?*, an answer constraint would require that *John* be the subject of the verb *love*. The grammatical function-based representation is sufficient to distinguish between the relevant and non-relevant text in this case.

These examples can be related back to the annotation graph-based retrieval framework outlined in this paper. The internal representation for information needs used by a QA system is equivalent to a type system. As illustrated, the choice of a type system can range from a simple representation derived from syntax to a full-blown semantic representation, in addition to token-based representations widely supported by text retrieval technology. Analysis of the text in the collection would yield annotation graphs containing relations such as $(loves, John, Mary)$, or $(subject, John, loves)$ and $(object, Mary, loves)$, which could be checked at query time. Query functions could implement operators that can combine or weight individual constraints, as well as account for partial matches, such as $(adores, John, Mary)$.

4. IMPLEMENTATION

This section describes features of the freely-available Indri search engine [5], a part of the Lemur toolkit,[1] used to support annotation graph-based retrieval. The current version of Indri provides support for indexing and retrieval of arbitrary, hierarchical, overlapping fields that was originally added to address the needs of an XML-IR task [4].

Indri's existing fielded retrieval support can be thought of as an implementation of annotation graph retrieval for

[1]See: `http://www.lemurproject.org`

```
#combine[sentence]( john loves )

#combine[sentence]( #max( #combine[target]( loves
  #max( #combine[./arg0]( john ))))
```

Figure 5: Bag-of-words (upper) and PropBank-style (lower) Indri queries for the question, *Who does John love?*

a subset of type systems, those under which at most one relation type can be defined to hold over any pair of element types. This relation is implemented in the Indri index as field enclosure; if a field instance of the domain type encloses a field instance of the range type, it is said that the relation holds between the elements corresponding to those fields. At retrieval time, these relations can be checked using query operators that enforce enclosure constraints between fields.

For an XML-IR task, the type system would include the elements found in an XML document, but for a QA task, it is the linguistic and semantic annotations on text that the system uses to locate answers that need to be indexed so that constraints can be checked at retrieval time. One example of a type system for QA is the one used in [1], which supports verb predicate-argument structures with semantic role labels in the style of PropBank [3], as well as common named entity types. Some QA systems use this semantic representation internally throughout the system, annotating text retrieved by a bag-of-words retrieval system on-the-fly and comparing it against an analysis of the question to locate answers. The goal of supporting constraint-checking against this representation at the retrieval stage of the QA process is to reduce the occurrence of false positives, or text that scores well on a keyterm match, but that does not unify with the QA system's expectations for an answer.

Figure 4 shows the example sentence *John loves Mary* with its PropBank-style semantic analysis and the equivalent annotation graph. The root of a verb predicate-argument structure is identified as the TARGET. In the figure, the bracketed arguments are labeled as ARG0 and ARG1, which correspond to the agent or doer of the action, and the patient or the person for whom the action is done, respectively. Though all argument roles are verb-dependent, users of PropBank have found that ARG0 and ARG1 can be relied upon to have been consistently labeled across verbs.

Figure 4 introduces a new type of relation called *child* that relates the target verb to its arguments. Because the verb does not actually enclose its arguments, Indri supports an additional representation for relations between field instances. In [1], an extension that has subsequently been integrated into the main trunk was made allowing Indri to store an arbitrary pointer to another field instance, called the *parent* field, in the posting for a particular field instance. Representing a relation type in the index in this way is called a *parent-child* relation, as opposed to an *enclosure* relation.

When Indri indexes a corpus annotated off-line with PropBank-style predicate-argument structures, target verbs are represented as field instances of type TARGET in the index. There are also separate field types for each type of argument, including numbered complement arguments such as ARG0, ARG1 and ARG2, as well as adjunctive arguments such as

```
#combine[sentence](
 #max( #combine[target]( loves
  #max( #combine[./arg0](
   #max( #combine[person]( john ))))
  #max( #combine[./arg1]( #any:person )))))
```

Figure 7: This query requests *love* events in which *John* is the agent, and any person is the patient.

ARGM-TMP and ARGM-LOC, which represent temporal and locative modifiers, respectively. Arguments are related to their respective targets by use of the parent-child strategy, and enclosure relationships exist between sentences and target verbs, sentences and arguments, sentences and named entities, and arguments and any nested fields, which can include nested named entities as well as targets and arguments.

At query time, Indri provides two different pieces of query syntax to support checking of enclosure and parent-child constraints. Figure 5 shows two queries that might be formulated for the question, *Who does John love?* In the bag-of-words (upper) query, the `#combine[sentence]` operator is used to enforce that the terms *john* and *loves* must be enclosed by the same SENTENCE field instance. In the PropBank-style (lower) query, the nested `#combine` operators tell Indri to look for a SENTENCE enclosing a TARGET enclosing *loves*. The dot-slash syntax used in `#combine[./arg0]` requires that the target verb have a child ARG0 field instance enclosing *john*. Throughout, the `#max` operators are used to select the best field instance in the event that more than one match.

Both enclosure and parent-child constraints map to structured query operators in Indri's underlying inference network retrieval model [6]. The query operators restrict keyterm matches to occurring field instances of the specified type. When scoring an field instance, the score contribution of specific keyterm is the number of occurrences of that keyterm over the size of the field instance, smoothed by linear interpolation with a background model based on the document containing the field extent, and also with another model based on the collection as a whole. If the field instance contains no occurrences of the requested keyterm, its score defaults to the background score made up of the document and collection smoothing components.

5. CHALLENGES

The process of building and testing Indri's support for annotation graph-based retrieval revealed some non-trivial issues inherent in implementing this kind of retrieval solution. This section asks the questions as to why the obvious approach of mapping the task of retrieval for QA onto a field-based XML-IR approach did not work well, and what are the requirements a retrieval engine would have to satisfy to be able to support annotation graph-based retrieval. The narrative in this section is centered around a series of examples that illustrate the emergent issues and motivate the discussion. For convenience, our examples involve matching sentences in the sample text collection shown in Figure 6.

5.1 Partial Matching of Structures

Indri's current constraint-checking implementation penalizes missing constraints harshly. Consider the query shown in Figure 7, which describes a *love* event between *John* and some other person. Sentence s_1 is a complete match for this query, and is ranked first. If a requested argument role does not appear in a predicate-argument structure, a background score, which can be quite low, is combined into the overall score.

Linguistic and semantic analysis tools occasionally make mistakes, sometimes as a result of legitimate ambiguities in the language, such as prepositional phrase attachment, and other times because of a lack of coverage in a rule base or in a training data set. In sentence s_2, the named entity recognition tool failed to identify *Mary* as a person, giving the sentence a structure similar to that of, *John loves his dog.* When scoring s_2, the enclosed PERSON field instance is not found, so the score defaults to a background score made up of document-specific and collection-wide scores for *Mary* occurring inside field instances of type PERSON. In the current implementation, it is not possible to directly smooth with the enclosing ARG1 instance.

Sentence s_3 represents a role-labeling error in which the argument corresponding to *Mary* is labeled ARG2, which generally represents a recipient or beneficiary, as opposed to the correct ARG1, which indicates the patient. There is, in fact, no prescription for an ARG2 in the *love* frame in PropBank, which means that the training data for the semantic role labeling tool does not contain this example in its entirety, but errors like this can happen when the semantic role labeling process is decomposed into argument identification, attachment and labeling as separate steps to maximize use of training data. As with sentence s_2, the missing field instance causes a background score to be combined into the overall score. The current implementation will only score field instances of the requested type; it is not able to propose the ARG2 as a match for the `#combine[./arg1]` query clause with some discount factor despite the fact that the ARG2 field instances satisfies the `#any:person` constraint.

As partial matches, sentences s_2 and s_3 are ranked behind s_1. The relative order in which sentences s_2 and s_3 are ranked depends primarily on the document models used for smoothing; in this case, if the documents contain more occurrences of *Mary* tagged as a PERSON than as an ARG2, then s_2 would come first.

5.2 Combining Evidence from Partial Matches

Sentence s_4 is semantically similar, but not equivalent, to sentence s_1. Setting aside the issue of whether the source, in this case *John*, is to be believed when he asserts that he loves Mary, this sentence is clearly relevant to a QA system faced with determining an answer to *Does John love Mary?* or *Who does John love?*

Sentence s_4 is a partial match for the query shown in Figure 7 because it contains two distinct predicate-argument structures, and the query's constraints are distributed between the two structures. The outer structure satisfies the constraint that *John* occur inside a PERSON nested inside an ARG0, and the inner structure satisfies the constraint on the target verb and the attached ARG1 containing any field instance of type PERSON.

The query uses a `#max` operator to score a sentence based on the single best-matching predicate-argument structure it contains, because the current implementation has no way to

s_1	John loves Mary.	[ARG0 [PERSON *John*]] [TARGET *loves*] [ARG1 [PERSON *Mary*]]
s_2	John loves Mary.	[ARG0 [PERSON *John*]] [TARGET *loves*] [ARG1 *Mary*]
s_3	John loves Mary.	[ARG0 [PERSON *John*]] [TARGET *loves*] [ARG2 [PERSON *Mary*]]
s_4	John says he loves Mary.	[ARG0 [PERSON *John*]] [TARGET *says*] [ARG1 [ARG0 *he*] [TARGET *loves*] [ARG1 [PERSON *Mary*]]]
s_5	John adores Mary.	[ARG0 [PERSON *John*]] [TARGET *adores*] [ARG1 [PERSON *Mary*]]
s_6	Bill loves Jane.	[ARG0 [PERSON *Bill*]] [TARGET *loves*] [ARG1 [PERSON *Jane*]]
s_7	John gave Jane's book to Mary.	[ARG0 *John*] [TARGET *gave*] [ARG1 [PERSON *Jane's*] *book*] [to [PERSON *Mary*]]

Figure 6: Text collection referred to by the examples in Section 5. The verb predicate-argument structure with PropBank-style semantic role labels and named entity recognition is shown below each sentence.

aggregate belief across multiple structures. Therefore, s_4 is scored on the basis of the inner predicate-argument structure, because it satisfies more of the constraints. In fact, s_4 would get the same score even if it were *Jack said that he loves Mary*, because the `#max` operator hides the effect on the score corresponding to the query's *John* constraint.

5.2.1 Balancing Structural Constraints

The query shown in Figure 7 contains three unweighted constraints. Intuition would suggest that the person or QA system formulating the query intended that the constraints be equally important. The current Indri implementation of Jelinek-Mercer smoothing leads to an interesting phenomenon where s_5, which is a relevant partial match, is ranked behind s_6, which is not relevant at all, but satisfies the target verb constraint. This ranking suggests that, for some reason, Indri is considering the target verb constraint much more important than the other constraints in the query.

The reason for this behavior is that the vast majority of target verb field instances are of length one. The score contribution of the target verb constraint is computed by taking the smoothed count of the number of occurrences of the keyterm divided by the length of the field instance being scored. This means that if the target verb does match, a very high belief is combined into the overall score, and if the target verb does not match, the portion of the score corresponding to the target field instance is zero, resulting in a very low background score based on smoothing with the document and the collection. As a result of this scoring method, the target verb mismatch on s_6 pushes it below s_5, which has mismatches on two constraints.

It may be that certain smoothing methods that are appropriate to fielded retrieval in general may not be optimized for tasks in which assumptions can be made about the nature of the fields. A recent proposal to address this problem is two-level Dirichlet smoothing, an extension of Indri's existing Dirichlet smoothing method to include a smoothing component for the document. This method is less sensitive to the length of the field instance being scored, so the variance of the scores produced by the query operator corresponding to the target verb constraint is reduced. This results in a more sensible ranking that relaxes all constraints on partial matches simultaneously after all partial matches have been retrieved. Under two-level Dirichlet smoothing, sentence s_5 is correctly ranked ahead of sentence s_6.

5.3 Multiple Potentially Relevant Structures

In process of the question analysis, a QA system uses deep linguistic and semantic processing to build a fairly rich representation of the answer it is looking for. Sometimes, a system is able to posit multiple structures that can potentially contain answers. The queries shown in Figures 8, 9 and 10 attempt to retrieve more of the relevant sentences in a single pass.

5.3.1 Combining over Keyterms

The query shown in Figure 8 maintains the same predicate-argument structure as the one shown in Figure 7, but uses a controlled synonymy to specify target verb alternatives that are semantically related. The `#combine[target]` operator is intended to be able to match sentences such as s_1 and s_5 by being flexible about the target verb constraint.

```
#combine[sentence](
  #max( #combine[target]( loves adores
   #max( #combine[./arg0](
    #max( #combine[person]( john ))))
   #max( #combine[./arg1]( #any:person )))))
```

Figure 8: This query is the same as the one shown in Figure 7, except that it requests *adore* events in addition to *love* events.

The implementation of the `#combine` operator essentially performs an arithmetic average in log space over the two score components, each of which is computed as the smoothed count of matching term occurrences over the length of the field instance. As written, the query clause prefers *loves* and *adores* to any other verbs, but it also prefers both verbs to just one of them alone. Knowing how the text collection was annotated, it would be impossible for a TARGET field instance to contain both of those terms. In fact, the only

times when a TARGET is longer than length one is the case referred to as the phrasal verb, where a verb and a particle occur in the TARGET together. The query clause will not perform as intended because every TARGET field instance that matches one of the verbs will have a background score combined in from the other verb that does not match.

One potential mitigation for this phenomenon is to wrap the alternate keyterms in a `#syn` operator, which treats occurrences of each term equivalently. One side effect of this choice of operator is that the smoothing values are skewed, particularly for `#syn` operators with large numbers of arguments. Document frequency is computed as the union of the arguments of the `#syn` clause, which can have an affect on the ranking. Another operator choice is `#or`, which implements the probabilistic Noisy-OR. Rankings are a bit easier to understand with `#or`, but it does not really capture the intent of query. Some kind of exclusive-OR operator may be more appropriate, but the question as to how to build such an operator for this task is still open.

5.3.2 Combining over Full Structures

Figure 9 shows a query that wraps an outer `#combine` operator around two full predicate-argument structures in an attempt to match both sentences s_1 and s_4. This query will not perform as expected, because the scores coming out of the two inner `#combine[sentence]` clauses are not, in general, directly comparable. The variance in the document scores for a particular query operator is inversely related to the number of arguments that operator has. The more complex constraints translate to query operators with more arguments, which provide scores on a smaller scale that vary over a more narrow range than do the scores corresponding to simpler constraints.

```
#combine[sentence]( #max(
 #combine[sentence](
  #max( #combine[target](
   #max( #combine[./arg0](
    #max( #combine[person]( john ))))
   #max( #combine[./arg1](
    #max( #combine[target]( loves
     #max( #combine[./arg1]( #any:person ))))))))))
 #combine[sentence](
  #max( #combine[target](
   #max( #combine[./arg0](
    #max( #combine[person]( john ))))
   #max( #combine[./arg1]( #any:person )))))))
```

Figure 9: This query requests two potentially relevant structures, the simpler structure contained in sentence s_1, below, and the nested version shown in sentence s_4, above.

This type of query can be difficult to reason about. Consider sentence s_1, which is a complete match for the second `#combine[sentence]` clause in the query. It is also a partial match for the first such clause in the query, as it satisfies the ARG0 constraint on *John*. Although s_1 has an ARG1, that field instance does not enclose any TARGET instances, and so a background score will be combined into the overall score. Even though smoothing with the document will yield at least one match for *loves* inside a TARGET field instance, the background score is low enough in comparison to the matching scores produced by the query clauses corresponding to the constraints that are satisfied to significantly affect the overall score.

5.3.3 Combining over Partial Structures

Given the difficulty in scoring disjunctions over full structures, not to mention the linguistic and semantic analysis task inherent in positing those structures in the first place, a QA system might decide to try specifying a single structure, but allow variations to mitigate annotation error, improve recall or simply to be able to control the relaxation of the constraints. It seems natural to take this approach, because if one constraint fails to match, a background score could be avoided if there is an alternate path for belief in the query.

```
#combine[sentence](
 #max( #combine[target]( loves
  #max( #combine[./arg0](
   #max( #combine[person]( john ))))
  #max( #combine[./arg1]( #any:person ))
  #max( #combine[./arg2]( #any:person )))))
```

Figure 10: This query is the same as the one shown in Figure 7, except that it tries to compensate for annotation error by allowing the patient of the *love* event to occur in the ARG1 or ARG2 positions.

Figure 10 shows an example of a query that matches *love* events in which *John* is the agent, having another PERSON in an argument labeled ARG1 or ARG2. As written, however, the query does not express the system's intent. Because the ARG1 and ARG2 constraints are written in two separate `#max` clauses that are children of the `#combine[target]` clause, this query will rank any sentence satisfying both constraints, such as sentence s_7, ahead of any sentence satisfying only one of them, such as sentences s_1 and s_3.

The current implementation of Indri makes it difficult to encode the notion of alternative constraints. The background score coming from the non-matching query branch will drag the overall score down. An alternative query formulation would but the ARG1 and ARG2 constraints into a single `#max` clause within the `#combine[target]` clause. The `#max` operator has the benefit that the low-scoring query branch is pruned, but it must be used with caution. The query operators inside a `#max` operator must produce scores that are comparable; otherwise, one element may consistently win out in a way that does not necessarily reflect the quality of the match, but instead, an artifact of the scoring model. One way to address this would be to build in the notion of a discount factor, allowing the user to not only specify precedence over the alternatives in a `#max` operator, but also to potentially compensate for a mismatch in the scale of the scores produced by one of the elements of the `#max`.

6. CONTRIBUTIONS

This paper proposed a theory of focused retrieval based on a formalism called the annotation graph. We motivated the potential of the approach using a Question Answering

example, arguing that the annotation graph representation addresses the requirements of focused retrieval applications by providing for rich retrieval-time constraint-checking. We tested Indri's support for annotation graph-based retrieval, and discovered that the obvious approach of mapping the problem onto existing XML-IR machinery presented a number of challenges. The identification of these challenges, along with the discussion of specific examples, serves to bring attention to the fact that the focused retrieval problem is not solved and fertile research ground lies ahead. We share our experiences with other interested researchers in the hopes that they will prove useful to those grappling with similar problems.

7. REFERENCES

[1] M. Bilotti, P. Ogilvie, J. Callan, and E. Nyberg. Structured retrieval for question answering. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2007.

[2] L. Hirschman and R. Gaizauskas. Natural language question answering: The view from here. *Journal of Natural Language Engineering, Special Issue on Question Answering*, Fall–Winter 2001.

[3] P. Kingsbury, M. Palmer, and M. Marcus. Adding semantic annotation to the penn treebank. In *Proceedings of the 2nd International Conference on Human Language Technology Research (HLT 2002)*, 2002.

[4] P. Ogilvie and J. Callan. Hierarchical language models for retrieval of xml components. In *Proceedings of the Initiative for the Evaluation of XML Retrieval Workshop (INEX 2004)*, 2004.

[5] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft. Indri: A language model-based search engine for complex queries. In *Proceedings of the International Conference on Intelligence Analysis*, 2005.

[6] H. Turtle and W. B. Croft. Evaluation of an inference network-based retrieval model. *ACM Transactions on Information Systems*, 9(3):187–222, 1991.