AnswerFinder at TREC 2005

Diego MOLLA and **Menno VAN ZAANEN**
Centre for Language Technology, Macquarie University
Sydney,
Australia,
{diego,menno}@ics.mq.edu.au

Abstract

AnswerFinder has been completely redesigned for TREC 2005. The new architecture allows a fast development of question-answering systems for their deployment in the TREC tasks and other applications. The AnswerFinder modules use XML to express the services they provide, and they can be queried with XML for their services. The QA method now incorporates graph-based methods to compute the answerhood of a sentence and pin-point the answer. The system uses a set of graph-based patterns that are learnt automatically. Unfortunately the system could not be completed and debugged before the TREC deadline and the runs did not fare well. Currently we are debugging and evaluating the system.

1 Introduction

AnswerFinder is a research oriented question answering system that focuses on incorporating symbolic information in the process. One of the more interesting aspects we are currently investigating is how far automatically induced structural, symbolic information helps finding actual answers.

This article describes several aspects of this year's TREC QA submission. Firstly, the architecture of the system, which has been completely redesigned this year, will be introduced in section 2. This includes a description of the framework of the system and the different subsystems that can currently be used. Secondly, in Sections 3, 4 and 5 we discuss a new method of finding answers in sentences based on automatically inferred logical graphs. We will describe the graphs, how to learn them and how to apply them to sentences to find actual answers. In Section 6 we give an overview of the parameters we used in the two runs that have been submitted to the TREC competition. Running the system indicated some existing problems of the current implementation that will be solved for next year. In Section 7 the problems will be treated briefly and solutions will be provided.

2 The Architecture of AnswerFinder

2.1 Overview

The AnswerFinder system can be divided into several phases. The process is entirely question-driven and forming a pipeline architecture. We recognise the following phases:

Question Analysis The first phase of AnswerFinder is the analysis of the question. During this phase question type classification is performed to determine what sort of answer we are looking for, such as location, person, etc. Also, shallow semantics of the question are extracted.

Document Selection The next phase is the selection of the documents that are likely to contain the answer. This phase is performed based on the information of the question. Only the selected documents are considered in the following phases.

Sentence Selection Using the information extracted during the question analysis phase, sentences that are likely to contain the answer are extracted from the selected documents. Only these sentences are processed further.

Answer Selection The information taken from the question is matched against the selected sentences and based on this, the exact answer is extracted and returned.

In this project we mainly focus on the representation of questions and sentences in the text documents. We have implemented different question type classification methods (**?**) and are investigating different representations of the shallow semantics of the question and the texts. Previously, we have used minimal logical forms (Mollá and Gardiner, 2004) and in this article we will describe logical graphs as a representation of the semantics. In addition to computing overlap of semantic units (which was done in the past), the logical graphs are also used to find exact answers in the sentences.

2.2 Requirements

When redesigning AnswerFinder, we first recognised several requirements. In addition to performance requirements (speed, memory usage, accuracy of finding answers, etc.), we have identified two requirements that have a high impact on the design. These requirements are most important from the view that AnswerFinder is a research and development system.

Flexibility The system should be flexible in several ways. Although it is being developed for the TREC competition, it should also be possible to easily modify it to handle different situations. For example, different input and output formats (of documents, questions, and answers), types of questions and answers, and new algorithms (in all the phases) should be easily integratable in the system. This allows for easy testing of new ideas in different environments.

Configurability Having a system with many different algorithms that can be used in the phases, it should be easy to configure the system to run using specific parameters. Parameters in this case mean, not only the actual values needed in the algorithms, but also selecting a particular algorithm in a phase.

2.3 Usage

¿From the user's point of view, AnswerFinder works as follows. Firstly, AnswerFinder can be queried for the functionality it provides. When a functionality request is sent, AnswerFinder replies with an XML document containing all the functionality it provides. This document shows for each of the phases which algorithm can be used and it also indicates if multiple algorithms can be used in a certain phase at the same time. For example, sentence selection can be done based on different kinds of information, which is implemented by different algorithms.

Once the provided functionality is known, the user can create an XML request by simply selecting which functionality is needed. This request is then sent off to AnswerFinder. AnswerFinder will combine the right modules and apply it to the data. Finally, the answers found by AnswerFinder are sent back to the user.

2.4 Implementation

AnswerFinder is implemented using C++, where the object-oriented paradigm is extensively used. Not only data is represented by class instances, also all algorithms are contained in class hierarchies.

Each of the phases in AnswerFinder has a separate class hierarchy, where each algorithm that can be used in a particular phase is represented by a class. The different algorithms are registered to a builder, which is a class that can be requested to create objects it knows about. The builder can also be asked to provide information about the classes it knows of.

When a user request in the form of an XML document is received by the general AnswerFinder algorithm builder, this document is analysed and the appropriate parts of the document are sent to the different builders for each phase in the system. These builders create instances of the algorithms that are requested in the XML document and all these instances are put together by the AnswerFinder algorithm builder. This system is then applied to the data that is present in the request and the final answer is sent back to the user.

Adding a new algorithm for a particular phase in the system is easy. A new class needs to be defined in the particular class hierarchy for the phase and it should register itself with the builder. Since a user can ask the builder what algorithms are available, the new algorithm is automatically incorporated in the system and can be used directly.

2.5 Available Settings

To allow AnswerFinder to answer questions, the user has to select which functionality is required. AnswerFinder currently implements the following functionality (Figure 1).

Firstly, the relevant data should be selected. This is a reference to the documents the answers should be searched in and a reference to the questions. Next, the parameters of each of the phases can be selected. Currently, for the document selection phase, the pre-selected documents ranking provided by NIST is used. The question classification module is currently only implemented using a regular expression classifier (as described by ?)). Since we are mainly concentrating on the semantic representation of sentences and its use for question answering,

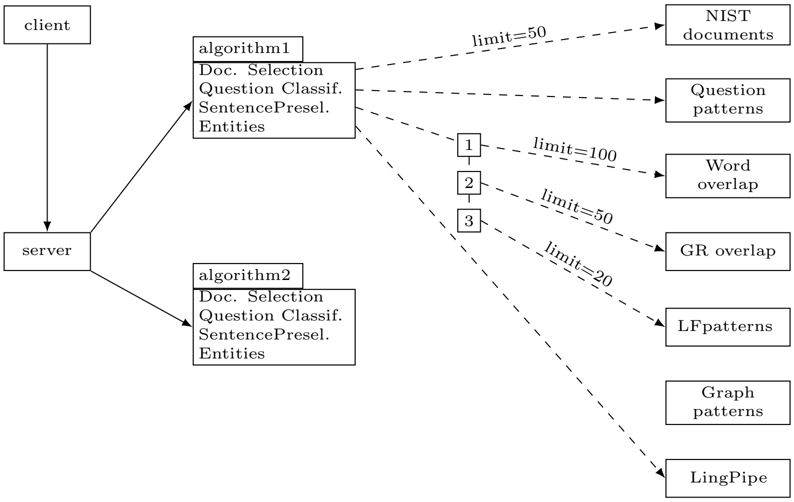


Figure 1: Functionality of AnswerFinder. The dashed arrows indicate a hypothetical configuration and its parameters.

several sentence selection algorithms are provided, which, when applied to sentences, may also be used to find possible answers. Four algorithms are implemented:

Word Overlap This algorithm includes a stop word list to indicate that certain words should not be used in the computation of the word overlap.

Grammatical Relations Overlap This algorithm uses the grammatical relations developed by **?**) and has been described by **?**).

Logical Form Overlap This algorithm uses the output of the Connexor parser[1] to produce logical forms and has been described by **?**).

Logical Graph Overlap This algorithm is new in our participation in TREC 2005 and will be described in this article.

All of these algorithms provide a score that indicates how well the question matches the sentence by considering the overlap of the representations. It is possible to combine several of these algorithms. Also, the system searches for named entities. We are using LingPipe[2], although we have also tried using Annie, the named entity recogniser from GATE[3].

3 Logical Graphs

An innovation in the version of AnswerFinder presented for TREC 2005 is the use of a graph notation for the representation of the logical contents of answer sentences. This is what we call the Logical Graphs (LGs). These Logical Graphs are inspired in Conceptual Graphs (Sowa, 1979), though in contrast with Sowa's approach, LGs do not attempt to encode the full semantics of a sentence. Following the principles of the logical forms of last year's AnswerFinder system (Mollá and Gardiner, 2004), LGs aim at representing the basic logical content required for question answering and they avoid the representation of well-known problematic concepts such as quantification, plurality, tense, and aspect.

Like Sowa's Conceptual Graphs, our Logical Graphs are directed, bipartite graphs with two types of vertices, namely concepts and relations:

Concepts Examples of concepts are objects *dog*, *table*, events and states *run*, *love*, and properties *red*, *quick*.

Relations Relations act as links between concepts. Examples of relations would be grammatical roles and prepositions. However, to facilitate the production of the Logical Graphs we have decided to use relation labels that are relatively close to the syntactic level. For example, instead of using labels related to thematic roles such as *agent*, *patient*, and so forth, we use syntactic roles *subject*, *object*, etc. Furthermore, to avoid resuming any debate about the

[1] http://www.connexor.com
[2] http://www.alias-i.com/lingpipe/
[3] http://gate.ac.uk/

possible names of the syntactic roles, we have decided to use numbers. Thus, the relation *1* indicates the link to the first argument of a verb (that is, what is usually a subject). The relation *2* indicates the link to the second argument of a verb (usually the direct object), and so forth.

Figure 2 shows various examples of LGs. These examples are taken from the examples used in Sowa's Conceptual Graphs website[4] and, while there is no space here to explain the differences and similarities between our LGs and Sowa's Conceptual Graphs, the interested reader may consult Sowa's examples and compare the graphs. Our first example shows the use of a relation labelled *1* to express the subject of the *go* event, and two relations, labelled *to* and *by*, that represent two prepositions. The second example shows the use of lattice structures to represent complex entities (such as the ones formed when a conjunction is used). This use of lattices is inspired on the treatment of plurals and complex events (Link, 1983; Mollá, 1997). Finally, the third example shows the expression of clauses and control verbs. These examples only cover a few of the linguistic features but we hope they will suffice to show the expressive power of the LGs.

The Logical Graphs are constructed automatically from the logical forms used in last year's AnswerFinder (Mollá and Gardiner, 2004) and they present a simplification of these logical forms. The conversion from a logical form to a LG is shown in Table 1.

4 Logical Graph Rules

To use the LGs to extract the answer, we have devised a method to learn Logical Graph Rules (LGRs) and apply the learnt rules to a question/answer candidate sentence pair. These LGRs are based on the concepts of *graph overlap* and *path* between two subgraphs in a graph (Mollá and van Zaanen, Draft).

Each rule r contains three components:

r_o An overlap between a question and its answer sentence.

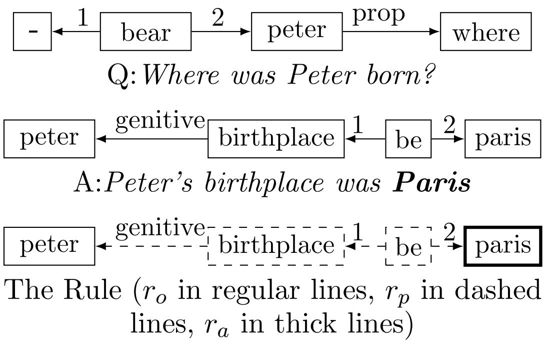
r_p A path between the overlap and the actual answer in the answer sentence.

r_a A graph representing the exact answer.

For further detail about the definition and properties of graph overlaps and paths, see (Mollá and van Zaanen, Draft). For the purposes of this article it suffices to say that, given the possible existence of repeated concept labels and relation labels in a LG, there may be several possible overlaps between two graphs. When this happens, the graph with the largest size is selected.

4.1 Learning of Logical Graph Rules

With the help of a training set of questions and sentences containing the answers, a set of LGRs can be learnt. Figure 3 shows an example of a rule learnt between two sentences. The graph notation has been simplified by replacing the relation vertices with labelled edges.



Q: *Where was Peter born?*

A: *Peter's birthplace was* ***Paris***

The Rule (r_o in regular lines, r_p in dashed lines, r_a in thick lines)

Figure 3: A logical graph rule

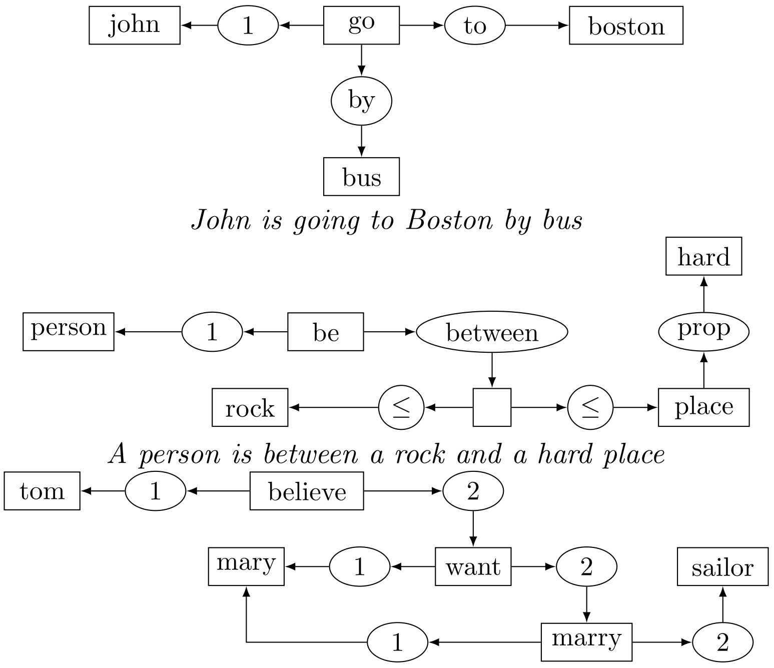
The algorithm for learning rules is fairly straightforward and is shown in Figure 4.

```
FOR every question/answerSentence pair
  $\mathcal{G}_q$ = the graph of the question
  $\mathcal{G}_s$ = the graph of the answer sentence
  $\mathcal{G}_a$ = the graph of the exact answer
  FOR every overlap $\mathcal{O}$ between $\mathcal{G}_q$ and $\mathcal{G}_s$
    FOR every path $\mathcal{P}$ between $\mathcal{O}$ and $\mathcal{G}_a$
      Build a rule $\mathcal{R}$ of the form
        $\mathcal{R}_o = \mathcal{O}$
        $\mathcal{R}_p = \mathcal{P}$
        $\mathcal{R}_a = \mathcal{G}_a$
```

Figure 4: Learning of graph rules

Rules learnt with this algorithm are very specific to the question/answer pair. For example, the rule in Figure 3 would only trigger for questions about Peter and it would not trigger, say, for the question *Where was Mary born?*. To generalise a rule we use a simple method:

[4] http://www.jfsowa.com/cg/index.htm



John is going to Boston by bus

A person is between a rock and a hard place

Tom believes that Mary wants to marry a sailor

Figure 2: Examples of Logical Graphs

- Concepts generalise to "_" (that is, concepts that would unify with anything).
- Relations do not generalise (relations express syntactic or semantic relations and it is not advisable to over-generalise them).

The generalisation of concepts applies to every concept except those that belong to a specific list of "stop concepts" (in analogy to the idea of stop words in Information Retrieval). The current list of stop concepts is:

> *and*, *or*, *not*, *nor*, *if*, *otherwise*, *have*, *be*, *become*, *do*, *make*

Rules are weighted according to the following formula. The weight $\mathcal{W}$ of a rule r is computed on the basis of its ability to detect the exact answer in the training corpus:

$$\mathcal{W}(r) = \frac{\text{\# correct answers found}}{\text{\# answers found}}$$

5 Graph-based Question Answering

To find if a sentence s with graph S answers a question q with graph Q, all learnt LGRs are tested. A rule r triggers iff its rule pattern r_o is a subgraph of Q (that is, the overlap between r_o and Q is r_o). When that happens, the graph of the question is expanded with the concepts and relations of the rule path r_p, producing a new graph Q_{r_p}. The resulting graph is more likely to produce a high overlap with an answer sentence similar to the one that generated the rule and, most importantly, the graph contains an indication of where the answer is located.

Once the graph of the question has been expanded with the rule path, one only needs to compute the overlap between this expanded graph and that of the answer sentence $ovl(Q_{r_p}, S)$. If the overlap retains part of the exact answer that was marked up by the graph rule r_a, then we have found a possible answer.

The above method will cover simple cases, but it needs to be extended to cover two special cases that arise from the fact that the question/sentence pairs that generated the rule are likely to be different from the actual question and sentence being tested. First of all, several rules may trigger, and each rule may extract a different answer. Consequently, there are several answer candidates and the system needs to choose one of them. Second, it is possible that the overlap between the extended graph and the sentence does not contain the complete answer but part of it. We will proceed to explain these two cases.

5.1 Answer Ranking

To identify the correct answer among a set of possible answers it is necessary to establish a measure of "answerhood" so that the correct

Table 1: Conversion from a Logical Form to a Logical Graph

1.	Convert all object predicates (e.g "object(john,o1,[x1])") into concepts labelled with the noun ("john") and indexed with the entity ("x1"). The index is required to allow the possibility of two different concepts having the same label. For example, the sentence *The big ball hit the small ball* would produce two different concepts labelled as "ball". The reification of the object ("o1") is ignored in the LG.
2.	Convert all event/state predicates (e.g. "evt(eat,e3,[x1,x2])") into concepts labelled with the verb ("eat") and indexed with the reification ("e3"). For every argument position, add one relation from the newly created concept to the concept indexed with the argument and labelled according to the position of the argument.
3.	Convert all property predicates introduced by adjectives and adverbs (e.g. "prop(hard,p1,[x2])") into concepts labelled with the adjective/adverb ("hard") and indexed with the reification ("p1"). Add one relation from the newly created concept to the concept indexed with the argument ("x2"). The relation is labelled "prop".
4.	Convert all other property predicates (e.g. "prop(to,p2,[e2,x4])") into relations labelled with the property label ("to"), which connect from the first argument ("e2") to the second argument ("x4"). The reification ("p2") is ignored.
5.	Convert all predicates of compound nouns (e.g. "compound_noun(x4,x5)") into relations labelled "compound_noun" that connect the first argument to the second argument.
6.	Convert all logical operators (e.g. "log_op(and,e2,[e3,e4])") into relations that connect the reification ("e2") to every argument, and labelled like the logical operator ("and").
7.	Convert all general dependencies (e.g. "dep(27000,d6,[x6])") into concepts labelled with the dependency label ("27000") and indexed with the reification ("d6"). Add a relation labelled "dep" that connects the newly created concept to the concept indexed with the argument ("x6").
8.	Convert all lattice relations (e.g. "x34<x35") and any remaining two-place predicates into relations labelled with the predicate name ("<") and connecting the first entity ("x34") to the second entity ("x35").

answer has a higher score than the other candidates. The rule weight gives an indication of the quality of the answer extracted. But we also need to keep in account the degree of similarity between the sentence that created the rule and the answer sentence being tested. For this we use the size of the overlap between the extended graph of the test question and the graph of the test answer $size(ovl(Q_{r_p}, S))$. Thus, the "answerhood" $\mathcal{A}(s)$ or likelihood that a sentence s with graph S contains the answer to a question q with graph Q is the product of the weight of the rule used $\mathcal{W}(r)$ and the size of the overlap:

$$\mathcal{A}(s) = \mathcal{W}(r) \times size(ovl(Q_{r_p}, S))$$

The size of a graph overlap is computed as the weighted sum of all concepts and relations in the overlap. The weight $\mathcal{W}_i$ of a concept or relation i in the overlap is determined using a variant of the Inverse Document Frequency (IDF) measure used in Document Retrieval. The actual formula that we use is:

$$\mathcal{W}_i = \frac{1}{\log N} \log \frac{N}{n}$$

n = total number of sentences using the concept (or relation) i

N = total number of sentences

The formula includes the constant factor $1/\log N$ to ensure that the value ranges between 0 and 1.

5.2 Answer Expansion

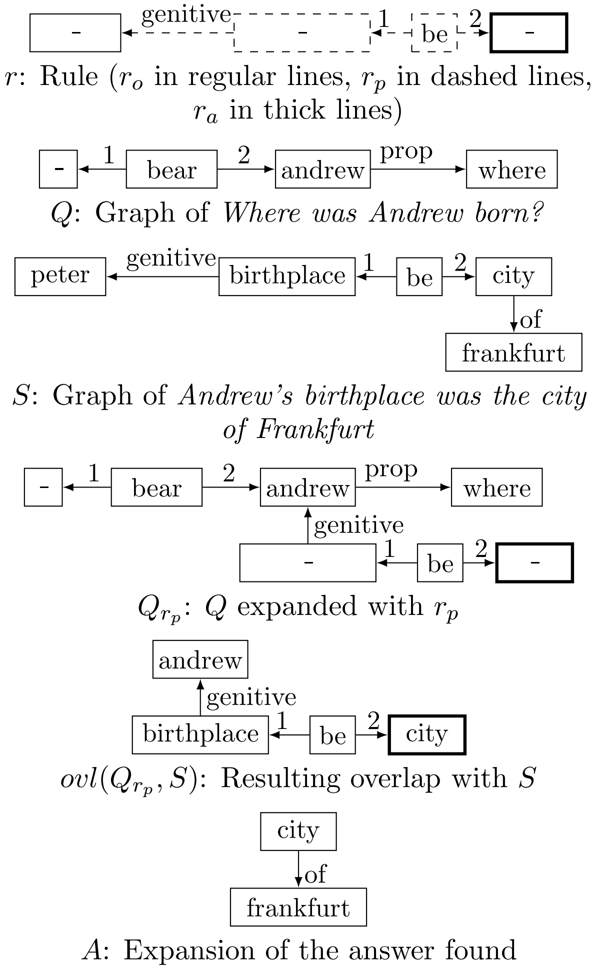
Sometimes the overlap between Q_{r_p} and S does not contain the complete answer but only the head of the answer. For example, suppose that the generalised rule of Figure 3 is used for the sentence pair:

Q: *Where was Andrew born?*

A: *Andrew's birthplace was the city of Frankfurt*

The overlap between the expanded graph of the question and the answer sentence (see Figure 5) would say that the answer is *city*, which is not correct. We need to expand the answer found. The process of answer expansion is very simple:

1. Start with the part of the exact answer marked up in the rule r_a that appears in $ovl(Q_{r_p}, S)$ and use it to form a new graph A.



r: Rule (r_o in regular lines, r_p in dashed lines, r_a in thick lines)

Q: Graph of *Where was Andrew born?*

S: Graph of *Andrew's birthplace was the city of Frankfurt*

Q_{r_p}: Q expanded with r_p

$ovl(Q_{r_p}, S)$: Resulting overlap with S

A: Expansion of the answer found

Figure 5: Graph-based Question Answering

2. For every concept in A, take the corresponding concept c in S and add to A all outgoing relations from c and their destination concepts.
3. Repeat step 2 until no more outgoing relations are found.

An example of answer expansion is shown in Figure 5. From this example we can see that the resulting answer may not be judged exact according to the NIST guidelines for the evaluation of answers. Still, we decided to keep the answer expansion on the grounds that otherwise no answer would have been found at all in cases like that of the example. We have not measured the impact of the answer expansion in the performance of AnswerFinder yet.

6 Parameters Used for the TREC 2005 Competition

This year, we submitted two runs to the TREC competition. The first run (af_run1) was used as a baseline, which used roughly the same parameters of the submission used in last year's competition. The second run (af_run2) tested the logical graphs as a replacement to the logical forms.

Both submissions used the same basic settings. This included document selection based on the documents pre-selected by the PRISE search engine and provided by NIST. For each of the questions, only the top 50 documents were used. Question type classification was done using a regular expression based classifier and LingPipe was used as the named entity recogniser.

The first run, which tested the use of minimal logical forms similarly to last year's submission, started off with selecting the 100 best matching sentences according to word overlap between the question and the sentences. These 100 sentences were converted into minimal logical forms, which finds some possible answers. The set of possible answers was then extended with the named entities present in the sentences.

The second run worked on the best 5 sentences selected based on word overlap. Logical graphs were computed on these sentences and matched with the logical graph of the question. This introduced some possible answers as described above. Again, named entities were used to expand the set of possible answers.

Finally, the set of possible answers found by the systems were ranked based on the score that was computed during the process of finding of possible answers. Answers found by the named entity recogniser increased the score by 1, similar to answers extracted by logical forms. Logical graphs increased the score by the product of the confidence of the rule and the graph measure, as described in Section 5.1.

Answers to factoid questions are then the single best answer according to the ranked possible answers. List type questions are answered by returning all the answers found and other type questions are simply considered to be a list type question where the question is "What is ⟨topic⟩?" Only answers that have a type that matches the question type are returned, but if none of them match, this requirement is dropped.

7 Results

The results of both runs in the TREC 2005 competition were very disappointing. The run af_run1 on the factoid questions had an accuracy of 0.028 (10 correct answers out of 362), the precision of recognising no answer was 0.077 and the recall of recognising no answer was 0.176. The answers to the list questions were not better: an average F score of 0.008. None of the "other" questions were correctly answered.

The run af_run2 performed even worse than af_run1. The accuracy on the factoid questions was 0.014 (5 correct answers), the precision of recognising no answer was 0.075 and the recall of recognising no answer was 0.235. The average F score on the list questions was 0 and again, none of the "other" questions had correct answers.

Generating the answers of both runs already indicated several problems with the current implementation. We rely on some external systems to compute intermediate results. For example, named entities are found by LingPipe, which is run as an external process. The way this is done now, is simply too time consuming. This restricted us to use only an extremely limited number of sentences to find answers in. The selection of sentences is based on word overlap, which may very well not be the best way to select them. If the answer is not present in the 100 (af_run1) or 5 (af_run2) sentences, the system will obviously not find a correct answer no matter how well these systems work.

Currently we are evaluating the system further. The low evaluation figures obtained indicate problems with the communication between modules in AnswerFinder. We are now exploring the cause of the poor performance.

A simplified system that uses only the Logical Graphs to find the answers and ignores any information about the question type and the named entities gives better results in our current evaluations. There may be several reasons for this. Firstly, the number of sentences selected by word overlap simply may not contain the answers, but we also suspect that the cause of the poor results is a bug in the system.

There are some more issues that create problems. Firstly, AnswerFinder currently tokenises all text it processes (questions and documents). The idea behind this is that it is easy to keep track of the possible answers in a standardised way. However, the external tools that are used (Connexor, LingPipe, Annie, etc.) perform their own tokenisation. This introduces problems since a mapping between the different tokenisation methods needs to be defined.

Another, related, problem is that the different methods that find possible answers may in fact return substrings of another possible answer. For example, one possible answer may be *the large house around the corner*, whereas another possible answer may be the substring *the large house*. The system currently treats these as completely different answers. The fact that substrings of another possible answer are also found as possible answers may count as extra evidence.

8 Conclusions and Future Work

In this article, we first presented the new AnswerFinder implementation. Because AnswerFinder is a research project, the system should be highly flexible. Systems are built dynamically and new methods can easily be integrated.

Next, a new representation of shallow semantics was discussed. Logical graphs allow a compact description of the semantics of questions and sentences. Not only are they visually clear (in contrast to logical forms), they can be used in the shape of logical graph rules to find possible answers in the text.

In the near future, we wish to understand why the results of this TREC competition are so much below our expectations. Although here we described some reasons why this is the case, more extensive testing is still under way. We also want to further investigate the learning of symbolic representations in the context of question answering. Logical forms and logical graphs are first steps in that direction. Of course, this also means that different scoring metrics need to be tested. Finally, the bugs and design issues of the current system will need to be fixed. This again requires more testing, but it also requires more significant modifications, such as offset annotation instead of tokenisation of text.

Acknowledgements

This research is funded by the Australian Research Council, ARC Discovery Grant no. DP0450750.

References

Godehard Link. 1983. The logical analysis of plurals and mass terms: a lattice-theoretical approach. In Rainer Bauerle, Christoph Schwarze, and Arnim von Stechov, editors, *Meaning, Use and Interpretation of Language*, pages 250–209. de Gruyter, Berlin.

Diego Mollá and Mary Gardiner. 2004. Answerfinder at TREC 2004. In Voorhees and Buckland (Voorhees and Buckland, 2004).

Diego Mollá and Menno van Zaanen. Draft. Learning of graph rules for question answering. Sydney. Submitted to ALTW 2005.

Diego Mollá. 1997. *Aspectual Composition and Sentence Interpretation: a formal approach.* Ph.D. thesis, University of Edinburgh.

John F. Sowa. 1979. Semantics of conceptual graphs. In *Proc. ACL 1979*, pages 39–44.

Ellen M. Voorhees and Lori P. Buckland, editors. 2004. *The Thirteenth Text REtrieval Conference (TREC 2004)*, number 500-261 in NIST Special Publication. NIST.