Web News Sentence Searching using Linguistic Graph Similarity

Kim Schouten and Flavius Frasincar

Erasmus University Rotterdam
PO Box 1738, NL-3000 DR
Rotterdam, the Netherlands
{schouten,frasincar}@ese.eur.nl

Abstract. As the amount of news publications increases each day, so does the need for effective search algorithms. Because simple word-based approaches are inherently limited, ignoring much of the information in natural language, in this paper we propose a linguistic approach called Destiny, which utilizes this information to improve search results. The major difference from approaches that represent text as a bag-of-words is that Destiny represents sentences as graphs, with words as nodes and the grammatical relations between words as edges. The proposed algorithm is evaluated using a custom corpus of user-rated sentences and compared to a TF-IDF baseline, performs significantly better in terms of Mean Average Precision, normalized Discounted Cumulative Gain, and Spearman's Rho.

1 Introduction

Nowadays, a significant portion of our mental capacity is devoted to the gathering, filtering, and consumption of information. With many things that are considered to be newsworthy, like updates from friends, twitter messages from people we follow, news messages on websites, and the more classical form of news like articles and news items, the amount of textual data (not to mention multimedia content) has become too large too handle. Even when considering only news items like articles, the number is overwhelming. And while some people can safely ignore lots of the news items, others are obliged to keep up with all the relevant news, for example because of their job.

While smart heuristics like skimming and scanning texts is of great benefit, it can only go so far. People, like investment portfolio managers, who have to monitor the stock of a certain group of companies, have to keep track of all news concerning these companies, including industry-wide news, but also that of competitors, suppliers, and customers. Therefore, being able to intelligently search news on the Web, for example to rank or filter news items, is paramount. Although text searching is very old, especially in computer science terms, the advance of new paradigms like the Semantic Web, has opened the way for new ways of searching.

This paper addresses one of these new search techniques, namely the search for news sentences. Searching for specific sentences enables the user to both search across and within documents, with the algorithm pointing the user to exactly the sentences that matches his or her query. With a previous publication [16] outlining the general concept of such a method, this paper aims to discuss the method in detail, providing additional analyses, and more insight into the actual workings of the algorithm.

2 Related Work

The over two decades worth of Web research has yielded several approaches to Web news searching. The most widely used approach is based on computing similarity by means of vector distances (e.g., cosine similarity). All documents are represented as a vector of word occurrences, with the latter recording either whether that word is in the document or not, or the actual number of times the word occurs in the document. Often only the stemmed words are used in these vector representations. The main characteristic of these methods is their bag-of-words character, with words being completely stripped of their context. However, that simplicity also allows for efficient and fast implementations, a useful trait when trying to provide a good Web experience. In spite of its simplicity, it has shown to perform well in many scenarios, for example in news personalization [1], but also in news recommendation [3]. Being the de facto default in text searching, TF-IDF [15], arguably the most well-known algorithm in this category, has been chosen to serve as the baseline for the evaluation of the proposed algorithm.

With the advance of the Semantic Web, a move towards a more semantic way of searching has been made. This includes the use of natural language processing to extract more information from text and storing the results in a formally defined knowledge base like an ontology. An example of such a setup can be found in the Hermes News Portal [8, 17], where news items are annotated using an ontology that links lexical representations to concepts and instances. After processing the news items in this way, querying for news becomes a simple matter of selecting the ontology concepts of interest and all news items being annotated with these concepts are returned. Comparable to this is Aqualog [12], a question answering application which is similar in setup as Hermes, and SemNews [9], a news platform like Hermes using its own knowledge representation.

Unfortunately, because searching is performed in the ontology instead of the actual text, only concepts that are defined in the ontology and correctly found in the text can be returned to the user. A deeper problem however is caused by the fact that ontologies are formally specified, meaning that all information in the text first has to be translated to the logical language of the ontology. While translation always makes for a lossy transformation, in this case it is worse as the target language is known to be insufficient to represent certain natural language sentences. Barwise and Cooper [2] proved that first-order logic is inadequate for some types of sentences, and most ontologies are based on propositional or description logics which have even less expressive power.

3 Problem Definition

Using the linguistic principles [6] of homophonic meaning specification and compositionality, a natural way of representing text is a graph of interconnected disambiguated words, with the edges representing the grammatical relations between words. While this representation is not as rich in semantics as an ontology, it avoids the problems of ontology-based approaches while at the same time providing more semantics than traditional word-based approaches.

With both the news items and the query represented by graphs, the problem of searching for the query now becomes related to graph isomorphism: the algorithm needs to rank all sentence graphs in the news database according to similarity (i.e., the measure of isomorphism) with the graph that describes the user query. Since we need a measure of isomorphism instead of exact graph isomorphism, we cannot simply implement Ullmann's algorithm [18].

This approximate graph isomorphism has a much larger search space than regular graph isomorphism which already is an NP-complete problem [4]. There are however some constraints that make the problem more tractable. Because all graphs are both labeled and directed, they can be referred to as attributed relational graphs, which are easier to deal with in this regard than unlabeled or undirected graphs. Furthermore, missing edges in the query graphs are allowed to be present in the news item graph (i.e., this is related to induced graph isomorphism), a characteristic which also makes the problem easier to solve since now the algorithm only has to check for the query's edges in the news sentence graph and not the other way around.

We have chosen to use an augmented version of the backtracking algorithm described in [13] to compute the graph similarities. The original algorithm iterates through the graph, checking with each step whether adding that node or edge to the partial solution can still yield a valid final solution. Because of this check, partial solutions that are known to be incorrect can be pruned, thus limiting the search space. Because parse graphs are labeled graphs, nodes can only be matched to nodes when their labels are identical, again limiting the search space. However, this will not work when considering measures of similarity or approximate matches. Then, its backtracking behavior is essentially lost as adding a node never renders a solution invalid, only less relevant. Because of this we can only speak of a recursive algorithm in the case of approximate matching. Such a recursive algorithm would assign similarity scores to all nodes and edges in the solution graph, and the sum of all these similarity scores would be the final score for this solution.

4 The Destiny Framework

The Destiny framework is the implementation that follows from the above discussion. It has two main tasks: first, it transforms raw text into a graph, and second, it ranks all graphs in a database based on similarity with a given user graph. In the current use case, news items are transformed into graphs and stored

in a database. The user graph represents the user query which is executed on the database.

4.1 News Processing

A natural language processing pipeline has been developed that transforms raw text into a grammatical dependencies-based graph representation. The pipeline consists of a set of components with a specific natural language processing task that are consecutively ordered, each processing the result of the previous component, sending the outcome as input to the next component in the pipeline. The same pipeline is used to process both news items and user queries. An overview of the pipeline design is given in Figure 1. The top half denotes the process of news transformation, whereas the bottom half denotes the process of searching the news.

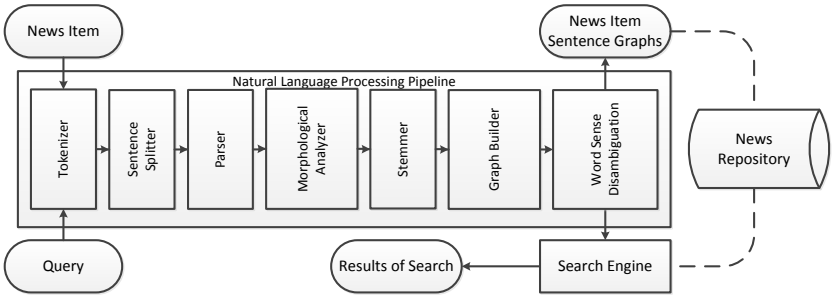

Fig. 1. Conceptual representation of framework

The pipeline is constructed on top of the GATE framework [5]. The same framework comes packaged with an extensive set of components, hence three out of the seven components are simply standard GATE components: the tokenizer to determine word boundaries, the sentence splitter to determine sentence boundaries, and the morphological analyzer to lemmatize words. While a default GATE component exists for the Stanford Parser [11], a slightly modified version is used to take advantage of a newer version of the parser itself. Porter's stemming algorithm [14] is used to determine the stem of each word.

The parser can be considered the main component of the pipeline, since it is responsible for finding the grammatical dependencies, thus directly influencing the graph output. Furthermore, it provides Part-of-Speech (POS) tags, essential information regarding the grammatical type of words (e.g., noun, verb, adjective, etc.). Based on the information extracted thus far, the graph builder component can construct a graph representation for all sentences. First, a node is generated for each word, encoding all known information about that word, like its POS, lemma, etc., in the node. Then, each syntactical dependency between words is used to generate an edge between the corresponding nodes, with the type

of syntactical dependency encoded as an edge label. Even though a word can appear more than once in a sentence, each instantiation of that word has its own unique grammatical role in the sentence. As such it has its own dependencies, and is therefore represented as a unique node in the resulting graph as well.

An example of a graph dependencies representation of a sentence is shown in Figure 2. As can be seen, some words are integrated into an edge label, in particular prepositions and conjunctions do not receive their own node. Integrating them in an edge label gives a tighter and cleaner graph representation.

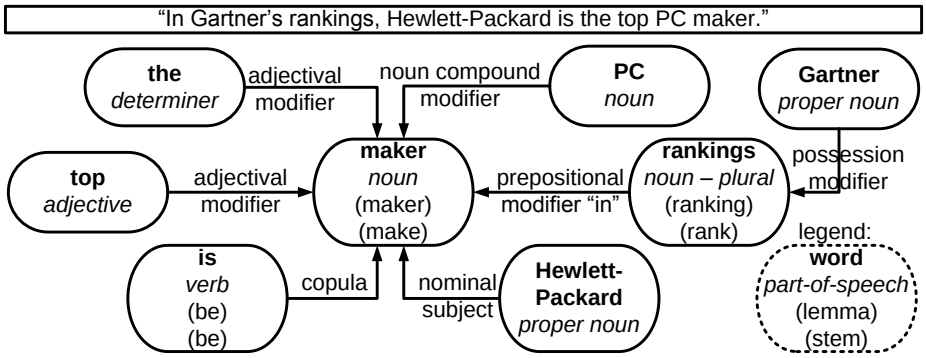

Fig. 2. Graph representation of the example sentence

The last step of this process is the disambiguation of words, where the correct sense of a word is determined and encoded in the corresponding node. Having the word senses allows the search process to compare words, not only lexically, which would not be very accurate in a number of situations, but also semantically. Even better, the search algorithm can effectively use this information to find relations of synonymy and hypernymy between words, something that would not be possible otherwise. Because the development of a word sense disambiguation algorithm is outside the scope of this paper, an existing, widely used, algorithm is implemented: the simplified Lesk algorithm [10].

4.2 News Searching

The news search algorithm is essentially a ranking algorithm, where all sentences in the database are ranked according to their similarity to the user query graph. As such, its core element is the part where the similarity between a sentence in the database and the user query is determined. This is the graph comparison, for which we decided to use a recursive algorithm.

However, an initial hurdle is the problem of finding a suitable starting point from where the graph comparison can commence. Since the structure of sentences can vary greatly, it would not suffice to simply start comparing at the root of both sentence graphs. On the other hand, comparing each node with every other

node would be too computationally expensive. As a compromise, each noun and verb is indexed by stem and are used as starting location for the graph comparison, the intuition being that nouns and verbs are the most semantically rich words in a sentence. In practice, this means that for each noun and verb in the query sentence, an index lookup is performed, returning a list of nodes that would be suitable to start searching from for that node in the query graph. The recursive graph comparison algorithm is then executed for each of those nodes, however, each pair of (query sentence, news sentence) is associated with (and thus ranked according to) only the highest score over all runs. Suboptimal scores are discarded. This process is described in Equation 1.

$$sentenceScore(query, sentence) = \max_{startNode_k \in sentence} score(query, startNode_k) \tag{1}$$

where $startNode_k$ denotes the kth starting node for this query. The implementation of this formula is represented in the pseudocode of method SEARCH in Algorithm 1. This algorithm makes use of COMPARE, which is described in Algorithm 2, to compute the raw scores. Being a recursive function, Algorithm 2 also calls itself with the next set of parameters to be compared. The object holding the raw score is forwarded as a parameter as well, so that each recursive loop will add some points to the overall raw score when applicable. Algorithm 2 uses two methods: SIMILARITYEDGE and SIMILARITYNODE, which compute the similarity scores for the edges and nodes, respectively.

Algorithm 2 compares the two graphs by first comparing the two starting nodes in the query graph and a news sentence graph. Then, using the edges of both nodes, the most suitable set of two nodes is determined to continue the graph comparison. This is done by looking one node ahead: the algorithm compares each connected node of the 'query node' with each connected node of the 'news node' to find the best possible pair. By means of a threshold, any remaining pairs with a preliminary score that is below the threshold are discarded. An additional effect of this policy is that when the preliminary score of a node is too low to be visited, its children will be discarded as well. While discarding regions of the graph that are likely to be irrelevant saves time, errors can also be introduced. As such this is a design choice, trading off a possible increase in accuracy against a decrease in running time. In the pseudocode, the process of looking ahead and finding the best pair of nodes to continue the graph comparison, if any, is encoded as a call to GETBESTSCORINGEDGE, which can be found in Algorithm 2. The recursive process will thus continue until either all nodes have been visited or no suitable matching pair is available for the remaining unvisited nodes that are connected to at least one visited node.

The similarity score of a news sentence with respect to the query sentence is essentially the sum of similarity scores of all selected pairs of nodes and pairs of edges. As such the actual score is determined by the similarity function of two nodes and the corresponding one for edges. While edges only have one feature, nodes have many aspects that can be compared and proper weighting of all

Algorithm 1 Pseudocode for the search algorithm

```
 1: function SEARCH(Document query, List of Documents processedDocuments) : SortedList
 2:     Initialize finalResults as SortedList of Scores
 3:     Initialize allScores as SortedList of Scores
 4:     Initialize matchedSentences as List of Strings
 5:     for all newsItem in processedDocuments do
 6:         Intialize queryStartNodes as List of Nodes
 7:         queryStartNodes = query.getStartNodes()
 8:         for all qStartNode in queryStartNodes do
 9:             Initialize newsStartNodes as List of Nodes
10:             newsStartNodes = newsItem.getNodes(queryStartNode.getStem())
11:             for all nStartNode in newsStartNodes do

    /* A new Score object is created. This object will be propagated through all recursive runs
    of the COMPARE method. */
12:                 Initialize score as Score

    /* The recursive method COMPARE as described in Algorithm 2 is started here. When it
    ends, the score collected over all recursive runs is saved. */
13:                 COMPARE(qStartNode,nStartNode,score)
14:                 allScores.add(score)
15:             end for
16:         end for
17:         finalResults.add(allScores.getHighestScore())
18:     end for
19:     return finalResults
20: end function
```

these features can substantially improve results. As such, all feature weights are optimized using a genetic algorithm, which was described in our previous paper [16].

The SIMILARITYEDGE function returns the similarity score for two edges. Since the only attribute edges have is their label, it returns a score only when the labels are identical. The exact score assigned to having identical edge labels is defined using a parameter which is optimized with the employed genetic algorithm. The SIMILARITYNODE function is slightly more complicated as nodes have more features that can be compared than edges. Each feature is again weighted using the genetic algorithm to arrive at a set of optimal weights for each of the features. The similarity score for nodes is computed as the sum of all matching feature scores that are applicable for the current comparison.

Nodes are compared using a stepwise comparison. First, a set of five basic features is used: stem, lemma, the full word (i.e., including affixes and suffixes), basic POS category, and detailed POS category. The basic POS category describes the grammatical word category (i.e., noun, verb, adjective, etc.), while the detailed POS category gives more information about inflections like verb tenses and nouns being singular or plural. For each feature, its weight is added to the score, if and only if the values for both nodes are identical.

If the basic POS category is the same, but the lemma's are not, there is the possibility for synonymy or hypernymy. Using the acquired word senses and

Algorithm 2 Pseudocode for the raw score computation

```
1: procedure COMPARE(currentQueryNode, currentNewsNode, score)
2:     Initialize nodeScore as double

   /* The two nodes are compared, and their similarity score is added to the total score. Both
   nodes are now marked as being visited. */
3:     nodeScore = SIMILARITYNODE(currentQueryNode,currentNewsNode)
4:     score.addScore(nodeScore)
5:     currentQueryNode.setVisited(true)
6:     currentNewsNode.setVisited(true)

   /* Now the parents and children of both nodes need to be compared. */
7:     Initialize queryEdges as List of Edges
8:     queryEdges = currentQueryNode.getEdges()
9:     Initialize newsEdges as List of Edges
10:    newsEdges = currentNewsNode.getEdges()

   /* Using SIMILARITYEDGE and SIMILARITYNODE the best possible route for the recursion
   is determined by comparing queryEdge with each possible newsEdge in newsEdges in
   GETBESTSCORINGEDGE. This method also makes sure that parents are compared only with
   parents and children only with children. If an edge exist that is good enough, the recursion
   will continue through that node. */
11:    for all Edge queryEdge in queryEdges do
12:        Initialize Edge bestEdge
13:        bestEdge = GETBESTSCORINGEDGE(queryEdge,newsEdges)
14:        if bestEdge ≠ ⊥ then
15:            double edgeScore = SIMILARITYEDGE(queryEdge,bestEdge)
16:            score.addScore(edgeScore)
17:            queryEdge.setVisited(true)
18:            bestEdge.setVisited(true)

   /* Recursion can only continue if there exists an unvisited node linked to bestEdge and one
   linked to queryEdge. */
19:            qNextNode = GETNEXTNODE(queryEdge)
20:            nNextNode = GETNEXTNODE(bestEdge)
21:            if !qNextNode.isVisited() then
22:                if !nNextNode.isVisited() then
23:                    COMPARE(qNextNode,nNextNode,score)
24:                end if
25:            end if
26:        end if
27:    end for
28: end procedure
```

WordNet [7], both nodes are first checked for synonymy and if so, the synonymy weight is added to the similarity score for this pair of nodes. If there is no synonymy, the hypernym tree of WordNet is used to find any hypernym relation between the two nodes. When such a relation is found, the weight for hypernymy, divided by the number of steps in the hypernym tree between the two nodes is added to the similarity score. In this way, very generic generalizations will not get a high score (e.g., while 'car' has a hypernym 'entity', this is so general it does not contribute much).

The last step in computing the similarity score of a node, is the adjustment with a significance factor based on the number of occurrences of the stem of that node in the full set of news items. For words which appear only once in the whole collection of news items, the significance value will be one, while the word that appears most often in the collection a significance value of zero will be assigned. Preliminary results showed that adding this significance factor, reminiscent of the inverse document frequency in TF-IDF, has a positive effect on the the obtained results. Equation 2 shows the formula used to compute the significance value for a sentence node.

$$significance_n = \frac{\log(\max \#stem) - \log(\#stem_n)}{\log(\max \#stem)} \tag{2}$$

where
n = a sentence node,
$\#stem_n$ = how often $stem_n$ was found in news,
$\max \#stem$ = the highest #stem found for any n.

Complexity Analysis As with any action that would require a user to wait for the results to be returned, the speed of the search algorithm is important. The query execution speed is highly dependent on the size of the data set, as well as on the size of the query. Furthermore, the higher the similarity between the query and the data set, the more time it will take for the algorithm to determine how similar these two are, as the recursion will stop when encountering too much dissimilarity between the query and the current news item, as defined in the threshold parameter. To give some insight into the scalability of the algorithm with respect to the size of the data set and the size of the query, the complexity of the algorithm (in the worst case scenario) is represented in the big-O notation:

$$\mathbf{f}(n, o, p, q, r) = \mathbf{O}(no^2pqr) \tag{3}$$

where
n = the # of documents in the database,
o = the # of nodes in the query,
p = the average # of nodes in the documents in the database,
q = the # of edges in the query, and
r = the average # of edges in the documents in the database.

In order to attain this (simplified) complexity, it is assumed that the number of nodes in a query is equal to the average number of nodes in the documents in the database, and the number of nodes is roughly equal to the number of edges for each sentence. In practice, a query will usually be much smaller than the average size of the documents in the database. Furthermore, this complexity, as it is a worst-case scenario, assumes it will have to compare each node of the query to all other nodes from the database. Again, this is usually not the case because of the threshold value limiting the recursion.

Interestingly, when scaling this up, the o^5 will quickly be dwarfed by n, the number of documents in the data set. We can therefore conclude that the algorithm is linear in the number of documents in the database.

Implementation Notes The system is developed in Java, using the Eclipse IDE (Helios). In order to have an easy and intuitive way of storing and retrieving the graph representations of text, we have chosen to use the object database provided by db4object (`www.db4o.com`). To access WordNet, the Java WordNet Library (`www.sourceforge.net/projects/jwordnet`) is used. This also provides convenient methods for determining synonymy and hypernymy relations between synsets.

5 Evaluation

In this section, the performance of the Destiny algorithm will be measured and compared with the TF-IDF baseline. First, some insight is given into the used data set. Then the performance in terms of quality, including a discussion on the used metrics, and processing speed are given. Last, a section with advantages of using Destiny is included, as well as a failure analysis based on our experiments.

5.1 Setup

Since Destiny searches on a sentence level (i.e., not only among documents but also within documents), a corpus of sentences is needed where each sentence is rated against the set of query sentences. From 19 Web news items, the sentences were extracted and rated for similarity against all query sentences. The news items yielded a total of 1019 sentences that together form the data set on which Destiny will be evaluated. From this set, ten sentences were rewritten to function as queries. The rewritten sentences still convey roughly the same meaning, but are rephrased by changing word order and using synonyms or hypernyms instead of some original words. Each sentence-query pair is rated by at least three different persons on a scale of 0 to 3, resulting in a data set of over 30500 data points. For each sentence-query pair, the final user score is the average of the user ratings. From these scores, a ranking is constructed for each query of all sentences in the database.

The inter-annotator agreement, computed as the standard deviation in scores that were assigned to the same sentence-query pair, is only 0.17. However, this includes a lot of pairs with a score of zero. As the majority of the sentences in the data set is completely dissimilar, a fact easily recognized by most people, the standard deviation is severely impacted by these scores. When we exclude all scores of zero and recompute the standard deviation, we attain a standard deviation of 0.83, which is slightly worse.

As discussed in the previous section, the weights are optimized using a genetic algorithm. In order to have a proper evaluation, the data set is split into a training set and a test set. The split itself is made on the query level: the genetic

algorithm is trained on 5 queries plus their (user-rated) results, and then tested on the remaining 5 queries. The results of the algorithm on those 5 queries are compared against the golden standard. This process is repeated 32 times, for 32 different splits of the data. All splits are balanced for the number of relevant query results, as some queries yielded a substantial amount of similar sentences, while others returned only a handful of good results.

5.2 Search Results Quality

The performance of Destiny is compared with a standard implementation of TF-IDF. As TF-IDF does not require training, the training set is not used and its scores are thus computed using the test set of each of the 32 splits only. The comparison is done based on three metrics: the Mean Average Precision (MAP), Spearman's Rho, and the normalized Discounted Cumulative Gain (nDCG) [12]. This gives a better view on the performance than when using only one metric, as each of these has its own peculiarities.

For this kind of data, the MAP is less suitable, as it assumes a Boolean similarity between query and candidates. A result is either similar, or it is not. This is in contrast with the graded similarity that is employed in this work. This means, that in order to compute the MAP, the user scores for all sentence-query pairs, ranging from 0 to 3, have to be mapped to either true or false. The cut-off value that will determine which user scores are mapped to dissimilar and which are mapped to similar is however rather arbitrary. We therefore made the choice to use a range of cut-off values, going from 0 to 3 with a stepsize of 0.1 and compute the MAP for each cut-off value. Hence, the MAP score reported in the next section is the average of these 30 computed MAP scores.

Both the nDCG and the Spearman's Rho do not suffer from the above problem and thus are more suitable metrics in this case. There are two concerns when computing Spearman's Rho. The first is that it computes the correlation between the ranked output of Destiny and the user scores over all sentence combinations in the list. This is not true in reality, as most users do not go through the whole list. Second, while the rankings are computed based on the degree of relevance, the latter is not directly used to compute the overall score. This means that it effectively assigns as much value to a top-ranking sentence being correct as to a lower- ranking sentence being correct.

In contrast, the nDCG only uses the first k number of results, and computes the added value of each of these k results for the total set by discounting for the position in the ranked results list. In this way the degree of relevance is also taken into account as results with a higher degree of relevance contribute more to the overall score than results with a lower degree of relevance.

To evaluate the performance of the Destiny algorithm, it is compared with the TF-IDF baseline on the ranking computed from the user scores. The results, shown in Table 1, clearly show that Destiny significantly outperforms the TFIDF baseline on the Spearman's Rho and nDCG ranking. The p-value is computed for the paired one-sided t-test on the two sets of scores consisting of the 32 split

scores for both Destiny and TF-IDF, respectively. The reported scores are the average scores over all 32 splits.

Table 1. Evaluation results

	TF-IDF mean score	Destiny mean score	rel. improvement	t-test p-value
nDCG	0.238	0.253	11.2%	< 0.001
MAP	0.376	0.424	12.8%	< 0.001
Sp. Rho	0.215	0.282	31.6%	< 0.001

5.3 Processing Speed

Query execution time is measured for the ten queries in our data set and compared with TF-IDF in Figure 3. The average time needed to search with Destiny is about 1570 milliseconds, while TF-IDF needs on average 800 millisecond to execute one query. As such, TF-IDF is on average approximately twice as fast as Destiny.

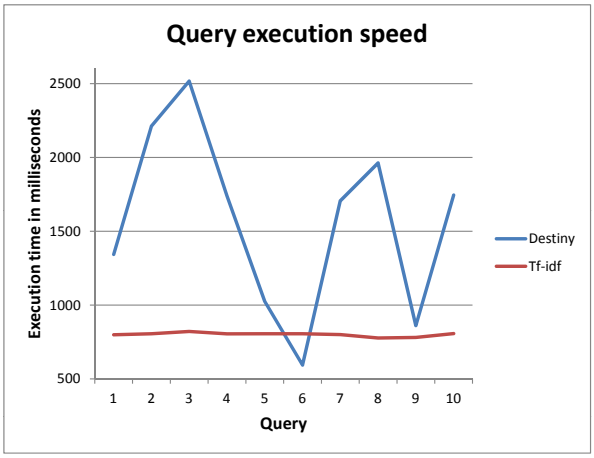

Fig. 3. Some query execution speed measures for Destiny and TF-IDF.

5.4 Advantages

Due to its focus on grammatical structure and word sense disambiguation, Destiny has some typical advantages compared to traditional search methods. The

first is the focus on sentences rather than separate words. When searching is based on word occurrence in a document, the document can get a high score even though different words from the query are not related at all but simply occur somewhere in that document. By focusing on sentences, words from the query are at least within the same sentence, making it much more likely that they are indeed semantically related.

Because grammatical relations are utilized when searching, users can actively use that to search for very specific information. While many different news items can be matched to the same bag-of-words, a group of words connected by a certain grammatical structure is much more specified. As such, it is more likely that a user will find his target when he can indeed specify his search goal by means of a sentence.

While grammar can be used to specify the query, the fact that the search algorithm employs synonyms and hypernyms improves the number of hits. Using synonyms and hypernyms, sentences can be found without explicit knowledge of the words in that sentence. This is obviously a great benefit compared to traditional word-based search algorithms which only take the literal word into account.

5.5 Failure Analysis

In order to analyze the errors made by Destiny and assess their origin, a failure analysis has been performed. This yielded a list of situations the algorithm is not able to handle well. These situations are summarized below.

With respect to dealing with named entities, Destiny is rather limited. Various versions of a name are for example not correctly identified as being the same, neither are different names belonging to the same concept. For example, “Apple” is not recognized to be the same as “Apple Inc.” or “Apple Computers Inc.”, nor is it matched properly to the ticker “AAPL”. Another example of the same problem would be the mismatch of the algorithm between “United States of America” and “U.S.A.” or just “USA”. Also, co-reference resolution is missing, so pronouns are not matched to the entity they are referring to. A graph-based approach like [11] seems particularly well suited for this work.

Also problematic in terms of semantics are proverbs, irony, and basically all types of expressions that are not to be taken literally. This caused some specific errors in the evaluation as in the data set many equivalent expressions are used for “dying”: “to pass away”, “to leave a void”, “his loss”, etc. While word synonyms can be dealt with, synonymous expressions are not considered.

Another issue is related to the fact that the search algorithm, when comparing two graphs, cannot cope well with graphs of varying size. Especially the removal or addition of a node is something the algorithm is unable to detect. When comparing Destiny with an algorithm based on graph edit distance [8], it can only detect substitution of nodes in a certain grammatical structure. Additional or missing nodes can thus break the iterative comparison, resulting in a significantly lower score than expected. For example, in the sentence “Microsoft is expanding its online corporate offerings to include a full version of Office”, it

is Microsoft that is the one who will include the full version of Office, but instead of Microsoft being the grammatical subject of "include", it is the subject of "is expanding", which in turn is linked to "include". When searching for "Microsoft includes Office into its online corporate offering", a full match will therefore not be possible.

6 Concluding Remarks

We have shown the feasibility of searching Web news in a linguistic fashion by developing Destiny, a framework that uses natural language processing to transform both query and news items to a graph-based representation and then searches by computing the similarity between the graph representing the user query and the graphs in the database. In the graph representation, much of the original semantics are preserved in the grammatical relations between the words, encoded in graph as edges. Furthermore, the search engine can also utilize semantic information with respect to words because of the word sense disambiguation component: words can be compared on a lexical level, but also on a semantic level by checking whether two words are synonyms or hypernyms.

While Destiny is slower than the TF-IDF baseline because of all the natural language processing, it is, nevertheless, better in terms of search results quality. For all three used metrics (e.g., Mean Average Precision, Spearman's Rho, and normalized Discounted Gain), Destiny yielded a significantly higher score.

Based on the failure analysis in the previous section, it would be useful to improve the accuracy of the search results by adding a module to match named entities with different spelling or using abbreviations. Also co-reference resolution might be beneficial, as sentences later in a news item often use pronouns to refer to an entity previously introduced, while a query, being only one sentence, usually features the name of the entity. Last, as discussed in the previous section, some form of graph edit distance might be implemented to mitigate the problem of important nodes not being present in both graphs.

While not within range of real-time processing speed, the processing and query execution times of the prototype provide an acceptable basis for further development. Currently, the system is entirely single-threaded, so a multi-threaded or even distributed computing system (e.g., processing news items in parallel) is expected to improve the speed.

Acknowledgment

The authors are partially supported by the Dutch national program COMMIT.

References

1. J. Ahn, P. Brusilovsky, J. Grady, D. He, and S. Y. Syn. Open User Profiles for Adaptive News Systems: Help or Harm? In *16th International Conference on World Wide Web (WWW 2007)*, pages 11–20. ACM, 2007.

2. J. Barwise and R. Cooper. Generalized Quantifiers and Natural Language. *Linguistics and Philosophy*, 4:159–219, 1981.
3. D. Billsus and M. J. Pazzani. User Modeling for Adaptive News Access. *User Modeling and User-Adapted Interaction*, 10(2-3):147–180, 2000.
4. S. A. Cook. The Complexity of Theorem-proving Procedures. In *Proceedings of the third annual ACM symposium on Theory of Computing*, STOC '71, pages 151–158. ACM, 1971.
5. H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan, N. Aswani, I. Roberts, G. Gorrell, A. Funk, A. Roberts, D. Damljanovic, T. Heitz, M. A. Greenwood, H. Saggion, J. Petrak, Y. Li, and W. Peters. *Text Processing with GATE (Version 6)*. University of Sheffield Department of Computer Science, 2011.
6. M. Devitt and R. Hanley, editors. *The Blackwell Guide to the Philosophy of Language*. Blackwell Publishing, 2006.
7. C. Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.
8. F. Frasincar, J. Borsje, and L. Levering. A Semantic Web-Based Approach for Building Personalized News Services. *IJEBR*, 5(3):35–53, 2009.
9. A. Java, T. Finin, and S. Nirenburg. SemNews: A Semantic News Framework. In *The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference (AAAI 2006)*, pages 1939–1940. AAAI Press, 2006.
10. A. Kilgarriff and J. Rosenzweig. English SENSEVAL: Report and Results. In *2nd International Conference on Language Resources and Evaluation (LREC 2000)*, pages 1239–1244. ELRA, 2000.
11. D. Klein and C. Manning. Accurate Unlexicalized Parsing. In *41st Meeting of the Association for Computational Linguistics (ACL 2003)*, pages 423–430. ACL, 2003.
12. V. Lopez, V. Uren, E. Motta, and M. Pasin. AquaLog: An Ontology-driven Question Answering System as an Interface to the Semantic Web. *Journal of Web Semantics*, 5(2):72–105, 2007.
13. J. J. McGregor. Backtrack Search Algorithms and the Maximal Common Subgraph Problem. *Software Practice and Experience*, 12(1):23–34, 1982.
14. M. F. Porter. An Algorithm for Suffix Stripping. In *Readings in Information Retrieval*, pages 313–316. Morgan Kaufmann Publishers Inc., 1997.
15. G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.
16. K. Schouten and F. Frasincar. A linguistic graph-based approach for web news sentence searching. In *Proceedings of the 24th International Conference on Database and Expert Systems Applications (DEXA 2013)*, pages 57–64. Springer, 2013.
17. K. Schouten, P. Ruijgrok, J. Borsje, F. Frasincar, L. Levering, and F. Hogenboom. A Semantic Web-based Approach for Personalizing News. In *ACM Symposium on Applied Computing (SAC 2010)*, pages 854–861. ACM, 2010.
18. J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *J. ACM*, 23(1):31–42, 1976.