Knowevo and Gravebook: Tracking the History of Knowledge

An Undergraduate Thesis

Aleksandar R. Gabrovski

May 23, 2012

Abstract

This thesis develops a framework for observing the evolution of various aspects of human knowledge over time. The framework uses several editions of *Encyclopedia Britannica*, as well as *Wikipedia*, to build a database for human knowledge corresponding to several time periods. The framework matches the same articles and topics across the different time periods and ranks them using various NLP and network analysis techniques. The data from *Encyclopedia Britannica* has been merged with Wikipedia to form a *Gravebook* - a website allowing scholars to query certain topics or articles and observe how the importance of the queried item evolved over time in terms of importance and content.

1 Introduction

Many humanitarian sciences study the evolution of their respective field, be it history, philosophy or literature. The study of the evolution of these various aspects of human knowledge likely holds significant potential for both scholarly and pedagogical insight. Yet, most of the developments in these fields took place before the revolution in information technology, making the data noisy and challenging to analyze computationally.

An encyclopedia can be viewed as a snapshot of human knowledge at a certain point in time. Thus, a study of the evolution of encyclopedias themselves will likely lead to similar insights as the study of every included topic separately. Such a study could be of considerable value to anyone interested in the evolution of human knowledge or certain parts of it, yet doing it by hand is a daunting task to say the least.

With the advent of various electronic encyclopedias and the readily available scans of older ones from initiatives, such as *Project Gutenberg*, *Jrank* and *Google Books*, the study of the evolution of encyclopedias today programatically seems feasible.

As a result, a research project was started by Professor Gronas and Professor Rumshisky to develop a framework for the analysis of the evolution of human knowledge. The author of this proposal shall attempt to develop the prototype for that framework as part of the larger research project of Gronas and Rumshisky by developing a suite of tools for matching, comparing and ranking the information contained in encyclopedias from different time periods.

The result will be a database containing snapshots of human knowledge at different points in time, and a set of tools to analyze and represent that data in a meaningful way. The end goal is to provide a website that will allow users to query and analyze the evolution of topics of interest.

2 Related Work

Previous works that look at the evolution and spreading of ideas in text include Gronas's work on analyzing the role of 19 century letter correspondence in the spreading of ideas [2], Rumshisky's work on the modeling and sense induction of corpuses of linguistic data [7], [6], [8] and Gronas's and Rumshisky's project in collaboration with Professor Bratus on visualization of conceptual and social networks in 19 century Russia [3]. The project also uses heavily Google's *PageRank* [5] for article ranking, *TF*IDF* [9] for article comparison and Levenshtein distance algorithm [4], [1] for fuzzy string matching.

3 Development Plan

The goal of this thesis is to analyze several editions of *Encyclopedia Britannica* and *Wikipedia* and build a website that tracks the evolution of topics within the encyclopedias. The author used the following sequence of milestones for achieving that goal:

4 Obtaining Encyclopedia Data

The author downloaded the OCR-ed text of edition 11 of *Encyclopedia Britannica* from *Jrank*, as well as from *Project Gutenberg*. The OCR-ed text of editions 9 and 3 have been acquired from *Google Books* with the help of Google's staff. Partial dumps of Wikipedia have been acquired using the *WikiMedia* API. The current edition (edition 15) of *Encyclopedia Britannica* was acquired by Prof. Gronas and Prof. Rumshisky as a dataset to the larger project to which this thesis is a part. There are parts of many of other editions found on *Google Books*, however, so far editions 3, 9, 11 and 15 are the only complete ones found by our team.

5 Parsing Encyclopedia Data

The parsing of the OCR-ed and otherwise formatted texts of the available encyclopedias has two goals. The first one is to split the text into separate article objects, each with its title and text. The second one is finding out references from one article to other articles in the same edition that will allow the construction of a reference graph of the edition.

The parsing of the current edition of *Encyclopedia Britannica* which was directly provided by the publisher in XML format was trivial for both the text and the references. Sam Kovaka converted all other editions to the same XML format used by the publisher in order to keep the consistency of the data.

Similarly, the parsing of the Wikipedia articles needed by the author (the selection of the *Wikipedia* articles is described later) was trivial due to the ease of use of the *WikiMedia* API and the consistent formatting of the internal Wikipedia format.

The parsing out of articles for edition 11 of *Encyclopedia Britannica* proved to be trivial since the articles were formatted as separate HTML files. However, the references provided by *Jrank* were incomplete and often wrong. Therefore, the author developed a heuristics for finding direct references from an article to articles in the same edition which proved to work much better. The heuristics first takes all "See also ..." parts of the plain text for each article and then uses a *Levenshtein Distance* function to find the articles referenced within the same edition. Below is a code snippet of the core function for the reference parser:

```
def getClosestReference(candid, titles, distanceFunction=getLevenNoPar):
    if candid in titles:
        return [(candid, 0)]
    else:
        res = []
        for i in xrange(len(titles)):
            res.append((titles[i], distanceFunction(candid, titles[i])))
        res.sort(key=lambda x: x[1])

        #return a lit of entities with the same score
        i = 0
```

```
        while i < len(res) and res[0][1] == res[i][1]:
            i = i+1
        return res[:i]


def getLevenshtein(a, b):
    '''
    From http://en.wikipedia.org/wiki/Levenshtein_distance
    '''
    lena = len(a)
    lenb = len(b)
    d = [[0 for x in xrange(lenb+1)] for y in xrange(lena+1)]
    for i in xrange(1, lena+1):
        d[i][0] = i
    for j in xrange(1, lenb+1):
        d[0][j] = j

    for i in range(1, lena+1):
        for j in range(1, lenb+1):
            if a[i-1] == b[j-1]:
                d[i][j] = d[i-1][j-1]
            else:
                insr = d[i][j-1]+1
                delt = d[i-1][j]+1
                repl = d[i-1][j-1]+1
                d[i][j] = min(insr, delt, repl)
    return d[lena][lenb]
```

The parsing of editions 3 and 9 was conducted by Sam Kovaka. Since these editions were obtained as plain OCR-ed text, the parsing relied on punctuation and whitespace to identify and split articles. Various heuristics were employed by Sam to deal with scanning errors and inconsistent formatting of the text. The parsing continues to be a work in progress. The heuristics developed by the author for edition 11 references is used in the parsing of editions 3 and 9.

6 Matching Articles Across Editions

One of the main goals of this project is to allow the comparison of the same articles across editions in terms of relative importance. Thus, matching articles on the same topic is critical. The author developed the following heuristics for matching articles.

6.1 Comparison Of Articles

6.1.1 Using TF*IDF

Initially under the supervision of Rumshisky and Gronas the author used *TF*IDF* to obtain weighted word vector representations of each article. Most values in the word vector for each article are 0 since the word vector is a mapping of all words in the corpus. Therefore, the author used a sparse representation of each vector where non-zero values are put in a hashtable of index to value and zero values are not included. The vector representation allowed the comparison of articles by means of computing the cosine between the vectors of the given articles.

However, *TF*IDF* does not take into account the position of a word in the text while assigning its weight within the vector of an article. As the author's advisers observe, this approach ignores the fact that words at the beginning of the article, i.e. the title and the introduction, usually seem to contain a concise version of the most important information laid out in the rest of the article. Thus, it seemed logical to modify *TF*IDF* to take into account this observation.

Furthermore, through exhaustive testing our team observed that words such as occupations (e.g. poet, politician), years of birth or death and people's names provide most of the insight necessary to correctly match and article.

Following the suggestions from Gronas and Rumshisky, the author developed the following modified version of *TF*IDF* which takes into account word positioning, as well as specific words:

```
def getWTFIDFd(words, params, occp, seenlimit=3):
    d = dict()
    ow = set()
    c = 0

    seen = dict()

    limit = params.getFirstWordsLim()
    for w in words:
        if w not in seen:
            seen[w] = 1
        else:
            seen[w]+=1

        if seen[w] > seenlimit:
            continue

        if w in d:
            d[w] = d[w]+1*(limit-c+1)*math.log(limit-c+math.e)
        else:
            d[w] = 1*(limit-c+1)*math.log(limit-c+math.e)


        #years
        if tools.isNumeric(w) and len(w) == 4:
            w = re.sub('o', '0', w)
            w = re.sub('i', '1', w)
            w = re.sub('G', '6', w)
            if w not in d:
                d[w] = 1

            if c < limit and w not in ow:
                d[w] *= params.getYearOw()
                ow.add(w)
                #print d[w], w

        #occupations
        if c < limit:
            if w in occp and w not in ow:
                d[w] *= params.getOccOw()
                ow.add(w)
                #print words[:2], d[w], w

        #title words only
        if c < params.getTitleWordsLim():
```

```
            if w not in ow:
                d[w] *= params.getTitleWordsOw()
                ow.add(w)
                #print d[w], w

        if c < limit:
            c+=1
    return d
```

The above code over-weighs the first 400 words of each article by adding to their frequency of occurrence using the following formula:

$$w_i = w_{i-1} + (l - p) * \ln(l - p + \exp) \tag{1}$$

where w_i is the weight for word w at occurrence i in the text, l is the limit of words to be over-weighed, p is the position of the word between 1 and $l + 1$. Choosing to overweigh the first 400 words rather than, say the first 200 or 500, was a rather arbitrary decision made by the author, but it still works better than plain *TF*IDF*.

The exact weighting shown above was reached through trial and error over the first 200 articles of edition 11.

The code overweighs any words found in a pre-compiles list of occupations (the list was compiled using Wikipedia occupations lists, nobility titles lists found here, and careers careers lists found online.

Titles of the the article (the first several words of each article) and years are also are overweighed. Since the scanned editions tend to parse incorrectly years a simple heuristic was added for fixing OCR errors for years - i are replaces with 1, G with 6, S or s with 5 and so on. The years are identified by two or more numeric characters present in a 3 or 4 letter word.

The exact weights for each overweighing were determined through testing over edition 9. A list of 200 human-proofed correct matches was compiled by our team. Then the testing framework went through more than 2000 different combinations of the overweighing

parameters and picked the one that gave the smallest error rate for the matching. The parameters retrieved in this fashion were:

Year Overweighing - 12 Title Overweighing - 8 Occupations Overweighing - 12 Number of first words to overweigh - 150

The error rate we received using the following parameters over the test sample was one %25. That was a big improvement over the original error rate of close to %50 but still very bad. Thus, the author added an additional step after matching to get rid of bad articles which will be described in two subsections.

6.1.2 Using Names, Occupations and Years

Since the *Weighted TF*IDF* approach still gave us a large error rate we tried reimplementing our previous insight for matching articles based on names, years and occupations only. The author developed a simple heuristics which just looks at keywords and adds up predetermined weights to come up with an overall matching score:

```
def getKeywordScore(art_text, cand_title, cand_text, occp):
    score_dict = dict()
    #links = re.findall('\[+(.+?)\]', cand_text)
    title_words = cand_title.lower().split('_')
    years = (re.findall('Category: *(\d+).+births', cand_text) +
             re.findall('Category: *(\d+).+deaths', cand_text))

    for year in years:
        score_dict[year] = 0

    for word in tools.splitToWords(cand_text)[:150]:
        if word in occp:
            score_dict[word] = 0

    nyear = 0
    nword = 0
    for word in tools.splitToWords(art_text)[:150]:
        nword+=1
        if tools.isNumeric(word):
            year = fixYear(word)
```

```
            nyear += 1
            if year in score_dict and nyear < 4:
                score_dict[year] = 100
            elif nyear < 3:
                for wyear in years[:4]:
                    try:
                        iwyear = int(wyear)
                        iyear = int(year)
                        if math.fabs(iwyear-iyear) < 10:
                            score_dict[year] = 80
                    except:
                        continue
        elif nword < 20 and word.lower() in title_words:
            score_dict[word.lower()] = 80
        elif word in occp:
            if word in score_dict:
                score_dict[word] = 20
    score = 0
    for k in score_dict.keys():
        score += score_dict[k]
    return score
```

In our case the simplistic technique of matching with keywords only actually tends to produce better results. Unfortunately the error rate is still quite significant. Thus the matching of articles remains an open issue.

6.2 Matching To *Wikipedia*

6.2.1 Getting the candidates

For each edition obtained every article is matched to a set of *Wikipedia* articles on the same topic. This is achieved in two separate ways developed by the author.

The first approach uses the WikiMedia and API-s for querying. The title of the article in questions is queried for within *wikipedia.org* and the first 5 results from *Wikipedia* and *Bing* are added to the list of candidate articles for matching. A second query is done with *Bing* in which not only the title but the first 5 words of the article are included. The first 5 results are again added to the candidate articles. Here is part of the code that achieves this:

```
#bing it
firstwords = '%20'.join(re.split('\s', artd['txt'])[:5])
q = ('site:wikipedia.org%20'+
    re.sub(' \(.*', '', artd['name'])+'%20'+firstwords)
links = filter(lambda x: 'en.wikipedia.org/wiki/'
    in x,bing.query(q, 5))
res = res | set(map(lambda x:
    wiki.reformat(x.split('/wiki/')[1]), links))

q = ('site:wikipedia.org%20'+
    re.sub(' \(.*', '', artd['name']))
links = filter(lambda x: 'en.wikipedia.org/wiki/'
    in x,bing.query(q, 5))
res = res | set(map(lambda x:
    wiki.reformat(x.split('/wiki/')[1]), links))


#wiki suggestions
res = (res |
    set(wiki.getSearchSuggestions(title, limit=5)))
```

The second approach uses a list of all article titles in *Wikipedia*. The list is sorted lexicographically. Then an insertion index is obtained for the spot where the title of the article in question can be inserted while preserving the sorted order of the list. Then the surrounding 6 articles around the insertion index are added to the candidate list. The author uses the Python *bisect* module to achieve this, which contains an algorithm similar to *Binary Search* for obtaining the insertion index. The code follows below:

```
#return titles from the vicinity in the range
title = reformat(artd['name'])
index = bisect.bisect_left(wikititles, title)
res = set()

for wt in wikititles[index-range:index+range]:
    if 'isambiguation' not in wt:
        res.add(wiki.reformat(wt))
```

Often the candidates retrieved are *Wikipedia* disambiguation pages or pages that have been taken down. Missing pages are removed from the candidate list, while disambiguation

pages are downloaded separately and all of their links (to articles that could refer to the disambiguated term) are added in our list of articles.

6.2.2 Evaluating the Candidates

The second stage of the matching algorithm uses the contextual comparisons techniques described above to assign a matching score to each candidate. For *Weighted TF*IDF* the algorithm downloads the text of all articles and creates *TF*IDF* word vectors for each of them. The same is done with the article in question. The author then computes the cosine between the article in question and each of the candidate articles. Following is a snippet of the described code:

```
def getTFScore(wordsa, wordsb, dfd, N, fnD):
    global CACHE

    if wordsa[0] == CACHE.name:
        da = CACHE.d
    else:
        da = fnD(wordsa)
        CACHE.d = da
        CACHE.name = wordsa[0]

    db = fnD(wordsb)
    getTFIDFVector(da, dfd, N)
    getTFIDFVector(db, dfd, N)
    try:
        res = tools.getCosOfDictVectors(da, db)
    except:
        res = 0
    return res


def getTFIDFVector(wordDF,dfd, N):
    for w in wordDF.keys():
        if w in wordDF:
            #add in dfd
            if w not in dfd:
                dfd[w] = 1
            #compute tfidf
```

```
            wordDF[w] = wordDF[w] * math.log(N/dfd[w])
```

For the keywords technique again the algorithm downloads all wikipedia text and constructs dictionaries with the total score for title, year and occupation keywords. The final score is the sum of all of the keywords' scores:

```
def getKeywordVectorScore(art_text, cand_title, cand_text, occp):
    score_dict = dict()
    title_words = cand_title.lower().split('_')
    years = (re.findall('Category: *(\d+).+births', cand_text) +
             re.findall('Category: *(\d+).+deaths', cand_text))

    for year in years:
        score_dict[year] = 0


    for word in tools.splitToWords(cand_text)[:150]:
        if word in occp:
            score_dict[word] = 0

    nyear = 0
    nword = 0
    for word in tools.splitToWords(art_text)[:150]:
        nword+=1
        if tools.isNumeric(word):
            year = fixYear(word)
            nyear += 1
            if year in score_dict and nyear < 4:
                score_dict[year] = 100
            elif nyear < 3:
                for wyear in years[:4]:
                    try:
                        iwyear = int(wyear)
                        iyear = int(year)
                        if math.fabs(iwyear-iyear) < 10:
                            score_dict[year] = 80
                    except:
                        continue
        elif nword < 20 and word.lower() in title_words:
            score_dict[word.lower()] = 80
        elif word in occp:
```

```
            if word in score_dict:
                score_dict[word] = 20

    score = {'years':0, 'titles':0, 'occp':0, 'other':0}
    for k in score_dict.keys():
        if k in years:
            score['years']  += score_dict[k]
        elif k in title_words:
            score['titles'] += score_dict[k]
        elif k in occp:
            score['occp']   += score_dict[k]
        else:
            score['other']  += score_dict[k]

    return score
```

6.3 Choosing From The Matched Candidates

Often the matching algorithm would include the correct match in the top candidates but, the correct match might not be the first one. We believe the reason for this erratic behavior is the fact that while traditional encyclopedias tend to have one article per topic, Wikipedia tends to have multiple articles on the same topic, with each article covering a tangent. For example, the Wikipedia article on *Abraham Lincoln* also includes links to separate Wikipedia articles such as *Early life and career of Abraham Lincoln*, *Abraham Lincoln in the Black Hawk War*, *Abraham Lincoln and slavery* and so forth. This tends to break the matching algorithm as we need the actual *Abraham Lincoln* article to match against.

In order to solve this issue, as well as the issue of plain wrong matches, the following filtering technique was developed by the author for the keyword comparison technique: If a candidate is a disambiguation page, skip it. If a candidate is missing skip it. If a candidate has a title and a year matched to the Britannica article, keep it. Else throw it away

The actual code is here:

```
def properFilterArts(arts, params=PARAMS):
    res = []
```

```
c = 0
MAX_DIFF = 80

for a in arts:
    matched = []
    first_name, first_score = a['candids'][0]
    if wiki.isDisambiguationPage(first_name):
            print name, 'dis'

    else:
        first_sum = sum(first_score.values())
        if first_score['years'] > 0 and first_score['titles'] > 0:
            matched = [wiki.getRedirect(first_name)]
            a['matched'] = matched
            res.append(a)
            continue

    #not first match
    for name, score in a['candids'][1:]:
        if wiki.isDisambiguationPage(name):
            print name, 'dis'
            continue

        if score['years'] > 0 and score['titles'] > 0:
            asum = sum(score.values())
            if first_sum - asum < MAX_DIFF:
                a['matched'] = [wiki.getRedirect(name)]
                res.append(a)
                break
            else:
                print a['name'],'DIFFBIG:',asum,name
                break
return res
```

In this case occupations matches don't actually affect the end result. The reason the author chose the following filtering technique is that although a lot good matches are thrown away, the ones that remain are actually not wrong. From the 200 articles in the edition 9 sample only 5 were matched incorrectly from the result that was to be put into the final database. The tradeoff was that 47 articles were eliminated entirely, shrinking the dataset considerably. The reasoning behind this choice is that it is better to have a smaller database with good matches than a big one with a lot incorrect matches.

6.4 Matching Across Editions

Once all matches are computed, the storage database is populated with all people articles from *Wikipedia*, a step described in detail in the section discussing the development of the *Gravebook*. The Django model for each Wikipedia article is the following:

```
CREATE TABLE "gravebook_article" (
    "name" varchar(512) NOT NULL PRIMARY KEY,
    "wid" bigint NOT NULL,
    "birth" integer,
    "death" integer,
    "image" varchar(512) NOT NULL,
    "vscore" double precision NOT NULL,
    "art_ed" integer NOT NULL,
    "text" text NOT NULL,
    "match_master_id" varchar(512),
    "match_count" integer NOT NULL
)
;
```

The key for each article is its name and each article contains a Foreign Key relationship to another article. The Britannica articles are inserted using the same model, with the ForeingKey relationship being populated to the appropriate *Wikipedia* article that was matched to. Thus, the database contains a graph in which *Wikipedia* articles are *sinks* to which all correspondingly matched *Encyclopedia Britannica* articles point.

7 Categorizing Articles

Currently, our team has come up with two approaches to categorization of the articles. The first one takes each article from each edition we have and assigns it the same categories that the *Wikipedia* articles have. The second approach uses the current *Encyclopedia Britannica* edition categories from the XML files and assigns them to all articles that were matched to a current edition article.

Wikipedia's categories hide many pitfalls. Not every article in *Wikipedia* has been assigned all categories that it should have, some categories seem to be randomly assigned and some categories are used only internally within *Wikipedia*. This causes a lot of errors when we apply the *Wikipedia* categories to our corpus. The author is still developing the best way to assign *Wikipedia*'s categories, the current results are promising but mixed.

The categories from the XML files of edition 15 are definitely a lot cleaner and more consistent. The interface, however, currently has only *Wikipedia* categories implemented due to the ease of their addition. The *Britannica* categories will be added in the future.

8 Measuring Article Importance

8.1 Using *PageRank*

8.1.1 Within Editions

Currently we are using Google's *PageRank* to measure article importance within each edition. The graph used for the *PageRank* is the direct reference graph that is obtained in the parsing part of the project. The author created an iterative *PageRank* implementation in Java whose inputs and outputs are plain text files representing the serialized direct reference graph and nodes with their *PageRank*-s respectively. The need for serialization became obvious due to Python's slow execution speed. The current solution is temporary until the research systems used for the project are updated to support *NumPy* or *Cython*.

8.1.2 Across Editions

Although *PageRank* tells us the importance of an article within its edition it cannot be used to compare articles from different editions directly. However, one can use the information the *PageRank* provides to construct a relative measure of article importance that goes beyond the article's own edition.

After obtaining the *PageRank* values, the author constructed separate rank measure for each article. First, all articles within an edition are sorted in descending order according to their *PageRank* values. Then for each article, its index in the sorted array is divided by the length of the array. This value represents the percentage of articles in the edition that are more important than the current article. In the actual application, the value is multiplied by 100,000 in order to avoid floating point precision problems which tend to be common across platforms after some number of decimal places. A snippet of the code follows:

```
def normalizeRank(arts, norm=100000):
    .....

    arts.sort(key=lambda a: float(a['rank']), reverse=True)
    for c in xrange(len(arts)):
        arts[c]['i_rank'] = c
        arts[c]['i_n_rank'] = norm*(c+0.0)/len(arts)
```

This measure is a meaningful comparison statistics for articles on the same topic in different editions.

8.2 Using Article Size

Unfortunately, we only have reliable reference graphs only for *Encyclopedia Britannica* Edition 15 and *Wikipedia*. Or previous parsing techniques for building the reference graphs for OCR-ed editions did not work reliably. Hongyu Chen has been working on a more reliable way to detect references, but his results have not been included in our front-end as of now due to the difficulty of producing reliable reference graphs for articles with myriad scanning errors.

Thus, the author developed a simpler importance measure for articles which relies on the article size. The main idea behind the algorithm is that if an article has more text than the average article in its edition, it is more important. Thus, one can compare articles in terms of importance by looking at how far their text size is from the average text size for the edition.

The above is computed using a simple *z-score* with the following formula:

$$vscore_{i,j} = (size_{i,j} - avg_sizej)/sd_j \tag{2}$$

Where $vscore_{i,j}$ and $size_{i,j}$ are the *size z-score* and size for article i in edition j, respectively; avg_sizej is the average size for articles in edition j and sd_j is the standard deviation for the size of articles in edition j.

The *size z-score* measure allows us the compute article importance independently for each edition (as the results is normalizes with respect to the average edition size) and thus compare articles from different editions.

9 Database

The author used *Django* with *PostgreSQL* for managing the interface. Thus the database model defined in Django on top of *PostgreSQL* is the following:

```
class Category(models.Model):
    name  = models.CharField(max_length=512, primary_key=True)
    size = models.IntegerField(blank=True, null=True)

    def __unicode__(self):
        return self.name

class Other(models.Model):
    name  = models.CharField(max_length=512, primary_key=True)


class Article(models.Model):
    name  = models.CharField(max_length=512, primary_key=True)
    wid   = models.BigIntegerField()
    birth = models.IntegerField(blank=True, null=True)
    death = models.IntegerField(blank=True, null=True)
    image = models.CharField(max_length=512, default='null')

    people = models.ManyToManyField('self')
```

```
peers  = models.ManyToManyField('self')

categories  = models.ManyToManyField(Category)
other_links = models.ManyToManyField(Other)

linked_by = models.ManyToManyField('self')

vscore = models.FloatField(default=0.0)

#100 means wikipedia. hacky slashy
art_ed = models.IntegerField(default=1000)
text = models.TextField(default='')
match_master = models.ForeignKey('self', blank=True, null=True)
match_count = models.IntegerField(default=0)

def __unicode__(self):
    return self.name
```

The primary key for each article is its name. This works well for *Wikipedia* which guarantees a unique name for each article. However, this assumption breaks once *Britannica* articles are included. Thus the author inserts all *Britannica* articles using the following trick:

$$name = j +_+ id_i +_+ name_{i,j} \tag{3}$$

where the name of each *Britannica* article in the database is concatenated to its edition j and id id_i to guarantee its uniqueness.

As described above, the field $match_master$ represents a *Foreign Key* relationship to another article. This field is used to keep track of the articles that *Britannica* articles were matched to. The rest of the fields are self-explanatory.

10 Front-End

The author developed the current front-end to the project by building an interface on top of *Wikipedia* called *Gravebook* and adding *Encyclopedia Britannica* articles on top to allow the tracking for the evolution of knowledge. The interface provides a page for each *Wikipedia*

article that was detected to be a person. On that page pictures of the person are hotlinked from *Wikipedia* where available, and the person's social graph is shown as described below. If *Britannica* articles were matched to that person, a chart showing the evolution of the article's importance is shown. The text for edition 3,9 and 11, as well as a *Wikipedia* stub can be read on the interface. Finally, each article contains a list of its *Wikipedia* categories. Each category has its own page that shows all articles with *Britannica* matches and a chart showing how the available matched articles evolved over time, within that category.

The front-end currently lives at:
http://hacktown.cs.dartmouth.edu/knowevo/gravebook/

It was developed using *Django 1.3*, *PostgreSQL 8*, *Django Chartit*, *Django South*, *Gephi*, *Vizster* and *JQuery 1.2* TODO. The project was developed entirely on Linux (ArchLinux for development and Ubuntu for deployment) and uses open-source software only.

The following few subsections describe the development of the interface with the above functionality in more detail.

10.1 *Wikipedia* Hotlinking

For each article in the *Gravebook* a portion of its text on *Wikipedia* is available for the use to see on the interface. The text is retrieved using the WikiMedia API TODO as an AJAX request to the server once the user clicks on the appropriate button for reading *Wikipedia's* text.

If an image is available on *Wikipedia* that image is hotlinked in the gravebook as well. Image names are retrieved by the *Grave-digger* described in the next section. The actual directory for the full URL to the image is determined by calling $md5$ on the image name and getting the first two hexadecimal characters of the resulting hash as the two nested directories in which the image is present. Thus the image called *Allan_Pinkerton-retouch.jpg* has a full URL of *aa6Allan_Pinkerton-retouch.jpg* on the server it is hosted on.

The actual URL to the image could be either to the actual *Wikipedia* or to *Wikipedia*

Commons. There is no way to predetermine which server hosts the file, and often images are moved back and forth. Thus, the server tries to request the image from both hosts. Since *Wikipedia* filters clients by the HTML user agent field, each of the requests is spoofed to be seen as a Mozilla Firefox request. This allows successful connection to be established to the correct URL. The attached code achieves the functionality described above:

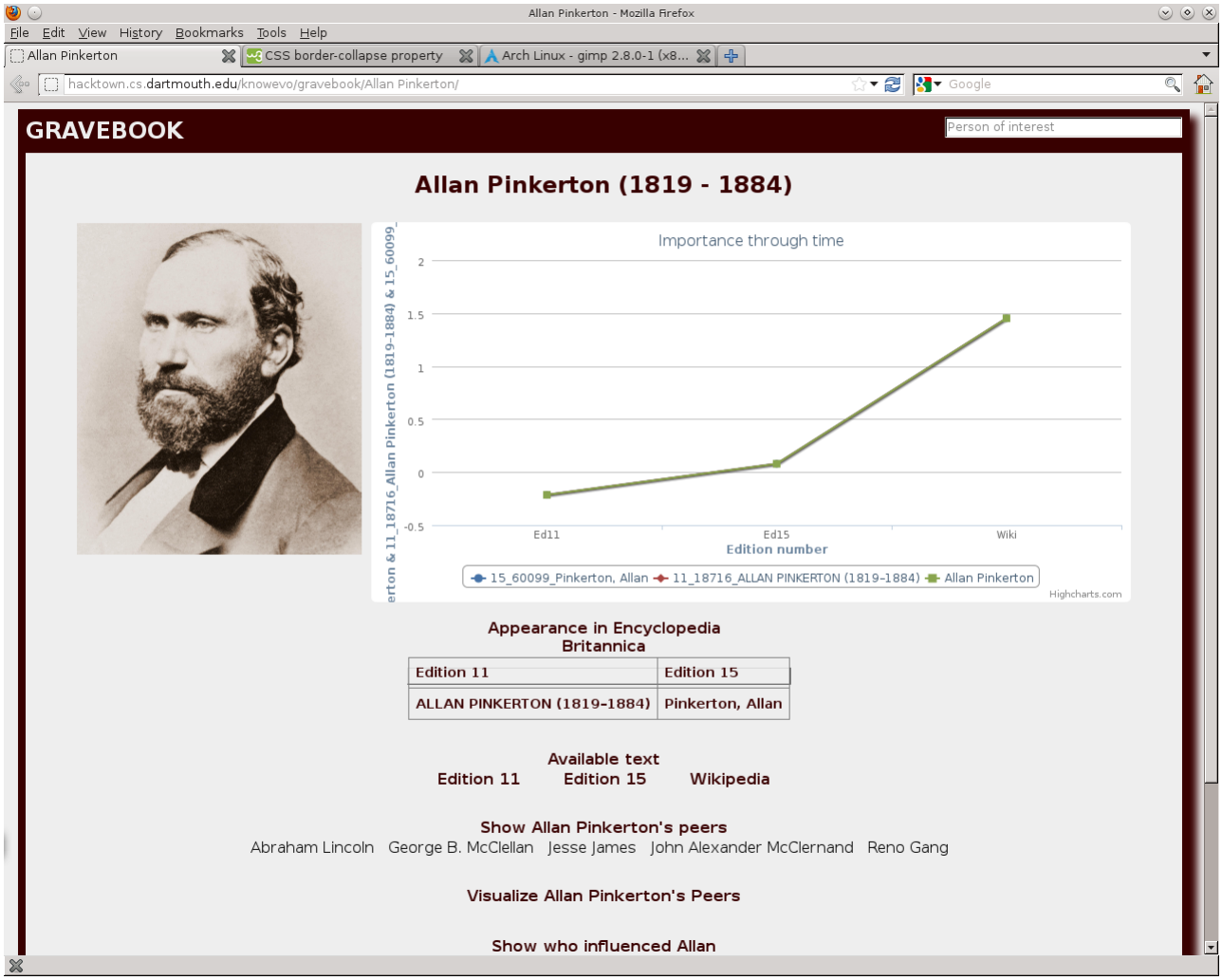
```
def prep_img_url(art):
    img = None
    if art.image != 'null':
        img = art.image.replace(' ', '_')
        img = img.split('|')[0]
        digest = md5.new(img).hexdigest()
        img = digest[0]+'/'+digest[:2]+'/'+img

        base = 'http://upload.wikimedia.org/wikipedia/'
        url = base+'commons/'+img
        headers={'User-Agent':"Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2
        try:
            r = urllib2.Request(url, headers=headers)
            f = urllib2.urlopen(r)
            f.close()
            return url
        except:
            pass

        url = base+'en/'+img
        try:
            r = urllib2.Request(url, headers=headers)
            f = urllib2.urlopen(r)
            f.close()
            return url
        except:
            pass
```

The URL that leads to an acknowledged request (which is dropped before the file is actually transmitted) is actually included in the webpage as the hotlink.

10.2 Evolution of Importance

Each article on the *Gravebook* has a chart depicting the way its importance changed through the available editions. The chart is created using *Django Chartit* which is somewhat inflexible but works for the prototype. The importance score used for the prototype and that is shown on the chart is the *size z-score* described in previous sections. The chart provides a nice visualization of the way articles evolved in terms of importance over time, where the data is available.

The attached screenshot shows the chart together with the main components of the interface, such as the hotlinked image from *Wikipedia* and the AJAX links for more data.



10.3 Grave-digging: Getting *Wikipedia* People

The first step to building the *Gravebook* was to get all people articles from *Wikipedia*. In order to achieve that, the author developed a Java parser for *Wikipedia* XML dumps. For each article the parser detects whether it is a person or not by looking through its list of categories. If the article has a category matching the pattern $(+)births|(+)death$ then that article is flagged as a person. The parser keeps track of the article's years of birth and death, title, size of text, links, categories and images and outputs an XML-formatted summary of people articles found in *Wikipedia*. While the parser is running all titles of person articles are added to a *Java HashSet* which is serialized at the end.

Then the Grave-digger does a second run through the XML output just produced. It loads the the HashSet previously computed and for each article splits its links in *people links*, if they were found to be present in the HashSet, or *other links*, respectively. The output is a different XML file containing all people articles, the people those articles references, categories, years of birth and death and so forth.

The final XML file is the imported in the database, with their text field set to null.

10.4 Peers, Influences and Mentees: Building *Wikipedia's* social graph

One of the initial goals of the project was to show the way the representation of the social network of articles changed over time, i.e. track the way references to people from people articles evolved through the different *Britannica* editions. The author was not able to complete this as robust references data for the OCR-ed text is still being generated. However, the groundwork for the analysis has been laid out by parsing *Wikipedia's* references.

For each person article in *Wikipedia* three groups of connected people are generated. The first one is a list of *peers*, i.e. reference to people who lived during the same time period as the article in question. We define two people as peers if at least one of them mentions the

other and they both lived during a 15-year overlap.

The second is group are *influences*, i.e. people that are mentioned in the article but lived before the person in question. The third and final group are *influenced* people, i.e. references to people who lived after the person in question.

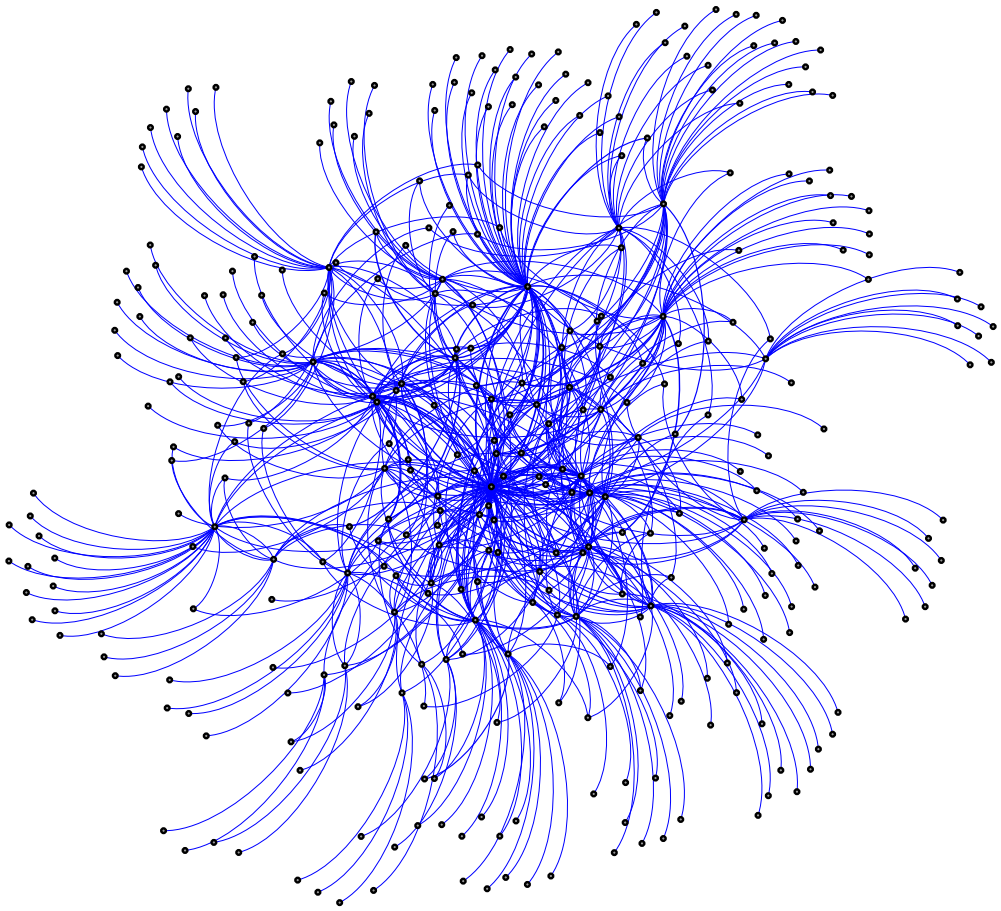
10.5 Weighing Social Relationships

Once we have discovered each article's social network we would like to assign each relationship a score that corresponds to its strength. A key observation allowing us to achieve the desirable weighing is that people who have a lot in common probably appear together in plenty of articles. Thus, for each pair of connected articles, we compute the number of articles in which both nodes are mentioned and assign that number as the weight of the edge between the nodes.

To achieve this the author added a *linked by* field to the database schema for each article that contains all articles that link to the current article. Thus, getting the cooccurrence weight of an edge between articles Alice and Bob requires a simple $COUNT$ of the intersection between Alice's and Bob's *linked by* articles.

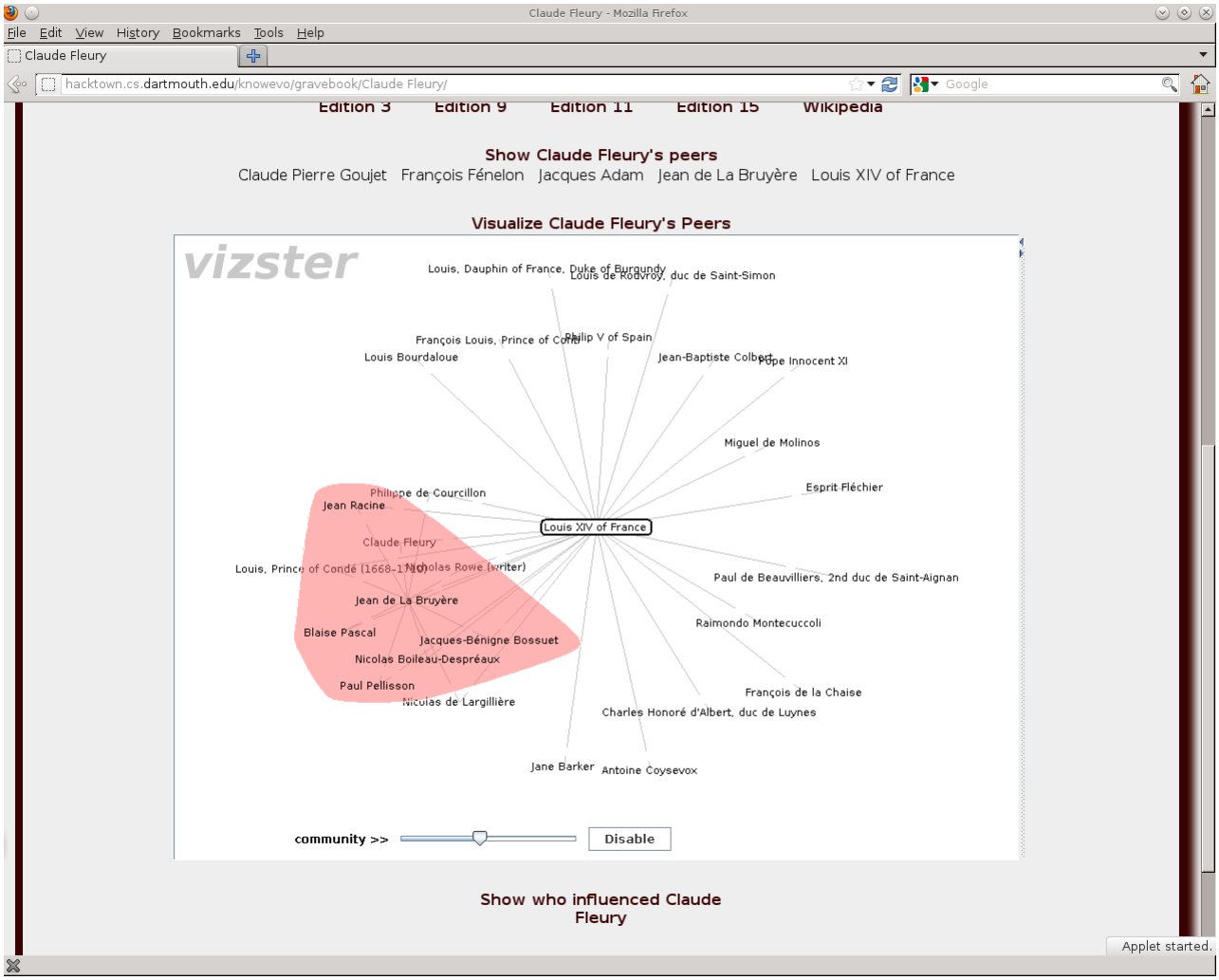
10.6 SpringBox-ing: Visualizing the social graph using Gephi or Vizster

After forming the social graphs for each article the author proceeded in finding a way to visualize them in a helpful manner. The author first tried using the *Gephi* package, an open-source Java framework for graph analysis. The author created a Java server to accept a connection on *localhost* for each page loaded. The server then invokes *Gephi* to produce a *SrpingBox* layout for the peers of the person in question. The output of the server is an SVG image file which is then show in the webpage requested. An example for Abraham Lincoln is shown.



The issue with *Gephi*, however, was the the produced images are static and hard to read. Rumshisky then pointed the author to the *Vizster* framework for Graph visualizations. *Vizster* is a desktop application for graph analysis and was never meant to be used on webpages. It uses a *Java JFrame* for its interface which cannot be embedded into HTML. Thus the author proceed to convert *Vizster* to a Java applet. The conversion as successful, but some of the functionality had to be sacrificed for the applet to be included into a webpage. The main problem is the *Lucene* package which *Vizster* uses for its search functionality. *Lucene* seems to cache results locally while performing searches which triggers the applet's security manager and turns the application off. The author disable the search functionality which was a requirement for several other features of *Vizster* that were also excluded. The result is an interactive Java applet with a nice representation of social graphs with up to 30-40 nodes.

A screenshot for the applet is included below:



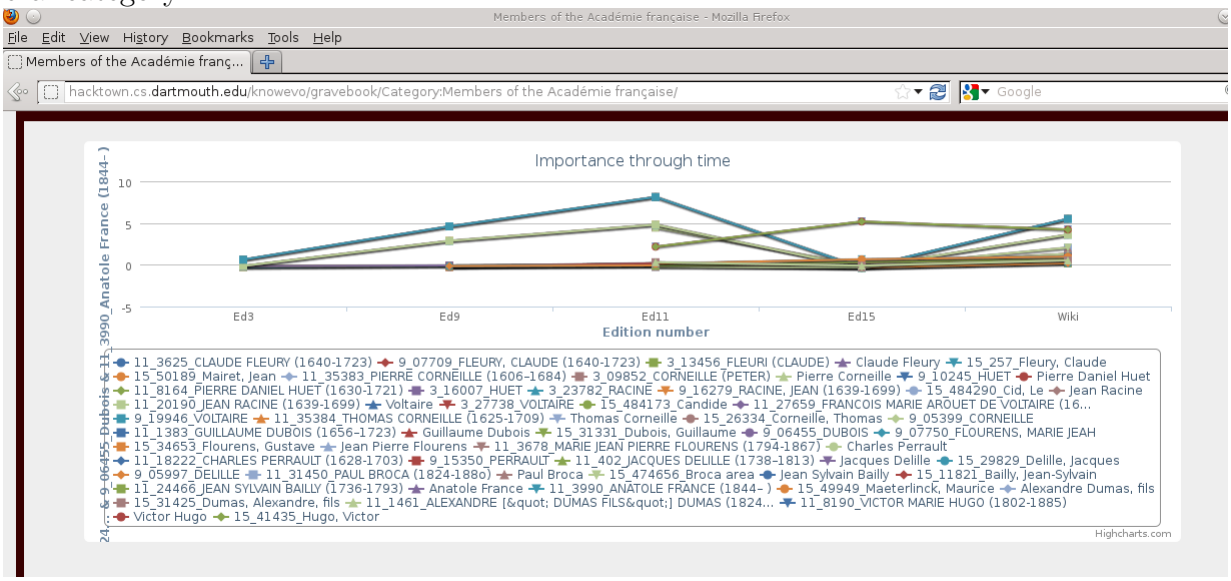
Currently, the social graph visualization is provided only for an article's peers. The social graph retrieved for each person in question is of depth 2, centered at the said person. *Vizster* uses the weights of the graph's edges to determine the distance between nodes in its visualization. Those edge weights are computed based on articles' cooccurrences, as described in the previous section.

Since a peer graph of depth 2 can get quite large, it is often impractical to show the full peer network in *Vizster*. Thus, the author developed a simple filter that takes each node's edges and sums up their weights to a *node score*. Then all nodes are sorted according to that *node score* and the top 30 nodes are actually show in the applet. The rationale for the sorting is that articles with many strong relationships are probably more important than

other articles. This allows for the visualization of the key members of a person's peer graph, without overloading the interface and the user with unnecessary information.

10.7 The Evolution of Categories

One of the major goals of the project is to allow the tracking of the evolution of general topics, or categories. The current interface does not provide for actual category evolution tracking due to the difficulty of implementing proper importance metrics without references and the limited data we have each category. The interface does provide charts showing how separate articles evolved within a category by looking at the changes in the *size z-score* measure described above. As the attached sreenshot makes it clear, it is hard to determine what is happening to the category by just looking at what is happening to its members. Still, the chart is useful for scholars interested in the evolution of separate articles within the same general category.



The development of a more useful category comparison is certainly possible, even using article size data only, and will hopefully be soon implemented. The difficulty of producing good matches is the biggest inhibitor for such macro analysis, as many matches have to be dropped due to their poor quality.

10.8 Performance Optimizations

Due to the large amount of data associated with each article, the website has been optimized to use as little memory as possible following the general *Django* recommendations. The current deployment server is hacktown.cs.dartmouth.edu which is an old desktop PC with barely 2 GB of memory making performance a big concern.

Even now, complex queries take a while to execute on the server. Therefore, the author converted the front-end from its original static form where all information was immediately loaded to the screen to a more dynamic AJAX front-end, using *JQuery*. Loading time was decreased significantly, as only information that the user wants is shown to the screen using AJAX requests.

11 TODOs

The goal of the thesis was to built the prototype for the ambitious research project started by Gronas and Rumshisky. From the beginning of the development gathering the data for it was the biggest and most frustrating challenge. This is where the current project has the most room for improvement, as every possible feature that the front-end could have depends on the data it is supposed to visualize. There are many different ways to attack parsing and matching scanned articles, some of which can be far more sophisticated and potentially better than the current approach. The use of artificial intelligence for matching or parsing articles is one way to go, however, it represents a large project on its own. The employment of people to do this by hand is another solution. Although the current version works decently for a prototype, a more robust solution to the data mining problem is needed for the current project to take off.

The project's interface could also be developed further. The biggest gap as of now is the lack of a way to compare categories in terms if importance. This problems presents both a technical and a conceptual challenge, as one needs to define what makes a category important

before the development can continue. The biggest roadblock for the development of better category comparisons techniques is the sad fact that any issues with the data, such as missing articles or the quality of matching, become painfully clear during any macro analysis, leading to doubts for the quality of its results.

12 Acknowledgments

The idea for this project came entirely from Professor Gronas and Professor Rumshisky who hired me as a Presidential Scholar to work on the project. They also convinced *Encyclopedia Britannica* to provide us with a copy of its current edition in XML format which was invaluable for the project. The advising Professors Rumshisky, Gronas and Bratus provided me with during the length of the project has been extremely helpful for the successful completion of the checkpoints outlined above.

I would also like to thank Dartmouth's Hacker Club ran by Parker Phinney for allowing me to use their server to host this database-intensive project, as well as for the advice and ideas that they gave me during the project's development.

I would also like to thank Sam Kovaka and Hongyu Chen for their help in the development process, as well as the staff of Research Computing for their support in running some of the more intense computations for the thesis project.

References

[1] L. Allison. Lazy dynamic programming can be eager. *Inf. Proc. Letters*, 43:207–12, 1991.

[2] M. Gronas. Pushkin and the art of the letter. *The Cambridge Companion to Pushkin*, pages 6–22, 2006.

[3] M. Gronas, A. Rumshisky, and S. Bratus. Digital pushkin: Computational visualizations of pushkin's social and political networks. In *41st National Convention of the American Association for the Advancement of Slavic Studies (AAASS 2009)*, 2009.

[4] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics - Doklady*, 10:707–10, 1966.

[5] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[6] A Rumshisky. Resolving polysemy in verbs: Contextualized distributional approach to argument semantics. *Distributional Models of the Lexicon in Linguistics and Cognitive Science, special issue of Italian Journal of Linguistics / Rivista Linqguistica.*, 2008.

[7] A. Rumshisky and V. A. Grinberg. Using semantics of the arguments for pedicate sense induction. In *Proceedings of the 5th International Conference on Generative Approaches to the Lexicon (Gl2009)*, 2009.

[8] A. Rumshisky and J. Pustejovsky. Between chaos and structure: Interpreting lexical data through a theoretical lens. *Special Issue of International Journal of Lexicography in Memory of John Sinclair*, 2008.

[9] G. Salton, E. Fox, and H. Wu. Extended boolean information retrieval. *Communications of the ACM*, 26:1022–1036, 1983.