Copyright Model for Collaboration

Literature Review

William Marsey
Imperial College London

June 2014

Contents

1 Introduction

This project will endeavor to evaluate individual contribution in the context of large-scale online collaboration. Given a collaboratively edited document, and information about each edit of which that article is consequent, may we define each collaborator in terms of their stake in the document? How do we algorithmically define the value of a contribution in this context?

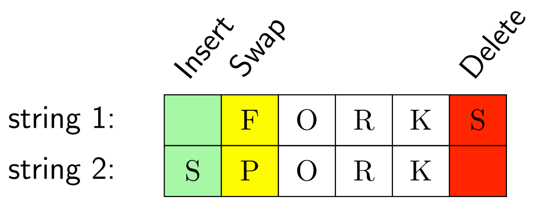
Taking a Wikipedia article and its revision history, this project aims to assign each editor a weighted value, according to the algorithmic analysis of the version generated by each edit. One of the main aims of this project is to produce a system that will allow for a visualisation of the distribution of this 'wealth' amongst the users.

In this literature review we begin with a brief overview of the concept of edit distance, including the various algorithms that have been devised in order to measure it. We then give an overview of existing research into Wikipedia revision histories that leverage these algorithms. We conclude with a general overview of the goals of this project, including the metrics by which we will measure contribution.

2 Previous work

2.1 Edit difference algorithms

To measure difference between different text revisions, we will refer edit distance. Edit distance between two texts, as first defined in the research of Levenshtein,[20] can be defined as the minimum amount of insert, delete and substitutions operations needed to transform one text into another.



forks → spork, edit distance: 3

Figure 1: An edit distance example using all three operations

Levenshtein's characterisation of this distance is given as:

for the function $\text{lev}_{a,b}(|a|, |b|)$:

$$\text{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } min(i,j) = 0 \\ \min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & else \end{cases}$$

$$\text{when } a_i = b_j,\ 1_{(a_i \neq b_j)} = 1$$
$$\text{when } a_i \neq b_j,\ 1_{(a_i \neq b_j)} = 0$$

That is, the distance between two strings is characterised the minimum distance between three different pair-combinations of its substrings. A 'text-book' implementation of this algorithm can be represented by the pseudo-code below. (We present the dynamic-programming-style algorithm here, and will generally be working with dynamic programming implementations throughout the study.)

```
ed(x,y):
   #end base cases
   if |x| = 0: return |y|
   if |y| = 0: return |x|

   #end table initialisation
   d is a table [0..|x|][0..|y|]
   for i = 1 to |m|:
      d[i,0] = i
   for j = 0 to |y|:
      d[0,j] = j

   #dynamic computation
   for j = 1 to |y|:
      for i = 1 to |x|:
         c = [(x[i] == y[j]) ? 0 else 1]
         ins = d[i-1,j] + 1
         dlt =d[i,j-1] + 1
         kp_swp = d[i-1,j-1] + c
         d[i,j] = min(ins, dlt, kp-swp)

   #return last computed number
   return d[|x|,|y|]
```

Figure 2: Basic dynamic implementation of Levenshtein distance

We can see that on comparing strings x and y, a $|x|$ by $|y|$ table is created, and then filled with values. For this reason both the time and space complexity of the algorithm is $\theta(|x||y|)$.

Reducing the space needed for this computation is relatively easy, and can be done in a few different ways. One way is to simply disregard parts of the table already computed. We can see that, on each computation of $d[i,j]$ (as it appears above), we require only a small part of the matrix: $d[i-1,j-1]$, $d[i-1,j]$ and $d[i,j-1]$. At any iteration i, where is great than 1, we may disregard rows $0\ldots(i-2)$ inclusive.

There are more complicated techniques that allow us to also disregard unneccesary computation — a few implementations employ strategies that allow them to trace the table space diagonally, rather than iteratively, achieving a time complexities as low as $O(ed(x,y)^2)$.[4] Another harnesses bit vectors to achieve a time complexity of $O(nm/w)$ or $O(nmlog\Sigma/w)$ time where w the bit-word size of the machine, and Σ is the alphabet size.[30][15] These optimisations will be explored in this project.

Varieties of edit distance

Modifications can be made to the nature of the distance itself, in order to adapt the measure a variety of different and specific needs. Here is a brief overview of the main groups these extensions fall into:

- **Hamming distance.** This allows for substitutions only, comparing same-length strings, such that:
  $ed_{hamming}(\text{“abc”}, \text{“abd”}) = 1$,
  $ed_{hamming}(\text{“abc”}, \text{“bcd”}) = 3$,
  and $ed_{hamming}(\text{“abc”}, \text{“ab”})$ is undefined.[13]
- **Reversals.** The Damerau-Levenshtein distance defines an ‘swap’ operation, which is the reversal of two adjacent characters. It is particularly suited to spell-checking, and for analysing DNA-sequence variations. In this case:
  $ed_{damerau}(\text{“ab”}, \text{“ba”}) = 1$

- **Block distance.** This allows for displacements of entire blocks to count as one operation. For example:
  $ed_{block}(\text{“abcde”}, \text{“cdeax”}) = 2$
  One move of the block ‘cde’, one substitution of ‘b’ for ‘x’.[39]
- ***q*-grams distance.** q-grams are simply sub-strings, and this measure describes the similarity of two strings in terms of q-grams they share.[40]
  $ed_{q-gram}(x, y) = \sum_{v \in \Sigma^q} |G(x)[v] - G(y)[v]|$
  where $G(x)[v]$ returns the number of occurrences of q-gram v in string x, and Σ^q is all the possible q-grams in the alphabet (capped by string length). $|G(x)[v] - G(y)[v]|$ a large positive number every time a q-gram appears a large amount of times in one string, but not the other; it returns 0 if the substring apears the same number of times. So, the whole function measures this difference for all possible substrings, and sums them, returning a high number for difference, and a low number for similarity.

Other algorithms we may look at are those that, like the q-gram distance, principally concern themselves with finding common subsequences between the strings. The common subsequence problem relates to the edit distance problem by way of the heuristic that two similar strings will have similar subsequences — the q-gram algorithm, for instance, relies on this heuristic, and works well for most texts, it does not agree with all distance measures. For example, two strings that are very different according to this heuristic may be quite similar according to the Damrau-Levenshtein measure.

Optimal alignment

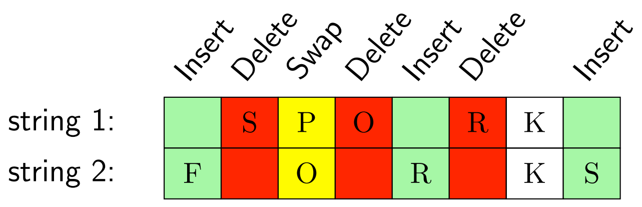
Another part of the problem of working out optimal edit distance is finding the ‘optimal alignment’ — the measures are closely related. We displace and arrange the characters of a string such that the set of operations to transform each character into its counterpart is minimal. For example, in figure 1, the alignment of the two strings “fork” and “spork” was:

s	p	o	r	k	-
-	f	o	r	k	s

However it could also conceivably have been:

s	p	o	r	k	-	or even	-	s	p	o	-	r	k		
f	-	o	r	k	s		f	-	o	-	r	k	-	s	

Here, the left-hand version results in an equivalent Levenshtein distance, but we can see how the distance for the right-hand example would be sub-optimal.



spork → forks, edit distance: 7

Figure 3: An sub-optimal edit distance example

The Smith-Waterman algorithm calculates optimal alignment by populating two tables — one like that in the pseudocode above, and also as a table of arrows. These arrows define a path from one corner of the table space to the other. The shape of this path defines how to align the two strings.[35] See figure 4 for an example.

This path may also be read as an edit operation. An arrow at the position $[i, j]$ in the table defines edit operations for $x[i]$ and/or $y[j]$ thus:

- $\nwarrow$ **at** $[i, j]$**, if** $x[i] \neq y[j]$
  Denotes a 'swap' between $x[i]$ and $y[j]$ (if $x[i] = y[j]$ then it denotes the lack of an operation).
- $\uparrow$ **at** $[i, j]$
  Denotes the deletion of $x[i]$
- $\leftarrow$ **at** $[i, j]$
  Denotes the insertion of $y[j]$

$$\left\{ \begin{array}{ccccccc}
 & & S & P & O & R & K \\
 & {\color{red}0} & {\color{red}0} & 0 & 0 & 0 & 0 \\
F & 0 & \nwarrow & {\color{red}\nwarrow} & \nwarrow & \nwarrow & \nwarrow \\
O & 0 & \uparrow & \nwarrow & {\color{red}\nwarrow} & \downarrow & \leftarrow \\
R & 0 & \uparrow & \uparrow & \uparrow & {\color{red}\nwarrow} & \leftarrow \\
K & 0 & \uparrow & \uparrow & \uparrow & \uparrow & {\color{red}\nwarrow} \\
S & 0 & \nwarrow & \uparrow & \uparrow & \uparrow & {\color{red}\uparrow}
\end{array} \right\}$$

(If the arrow reaches an edge before the left-hand corner, we trace along that edge, reading each shift as an arrow in the direction of the trace.)

Figure 4: Diagram showing Smith-Waterman traceback path (in red) on the edit operation forks $\rightarrow$ spork

Delta encoding

Finally, we may also look into Delta encoding algorithms. These describe ways of compressing the storage of a document's history — a format in which only the differences between each text is stored, not the entire version. These algorithms are of the same family of algorithms discussed above. In fact, we find that one of the fastest known algorithms,[1] *VDelta*, is a refinement of the block distance algorithm mentioned above. For a given version n of a document *doc* is defined as:

$$v_n = v_0 \cup \Delta(v_0, v_1) \cup \Delta(v_1, v_2) \cup \cdots \cup \Delta(v_{n-1}, v_n)$$

where $\Delta(v_i, v_j)$ is the difference between version i and version j of the document, and the union operation $\cup$ combines each version in a manner particular to the Δ data-type. Storing data in this way can be very efficient, resulting in a compression factors of five or ten on typical data.[25] It may also be relatively easy to maintain in our case, due to the linear nature of Wikipedia revision histories.

2.2 Wikipedia

In academia

Wikipedia is a free, open-source, publicly-editable online knowledge-base. The software is runs upon, MediaWiki, is also open-source, powering countless other online encyclopedias. Since it's launch in 2001, Wikipedia has become a the pre-eminent online source of reference. The website is ranked 6th globally in terms of website traffic, and is the highest-ranked reference website by far - most of the sites it shares the top spots with are portals, search engines, shopping mega-sites, and social media websites.[2] Despite early skepticism (particularly concern over the inherent chaos in the system: "...edits, contributed in a predominantly undirected and haphazard fashion by ... unvetted volunteers."[48]), it is widely claimed to be a success, 'the best-developed attempt thus far of the enduring quest to gather all human knowledge in one place'[28].

[1]According to Hunt's 1998 study[14]

[2]According to 'Alexa', an website ranking company.[3] Though, this may be an underestimation. Alexa may well be biased towards English speakers and Internet Explorer users, underestimating Wikipedia.org's popularity, since 'two thirds of all Wikipedia articles are in languages other than English'[44]

That Wikipedia has become a hub of research in many fields is also self-evident to anyone who has searched for articles on the subject. Mesgari et al, just quoted, has prepared a very recent 'systematic review of scholarly research on the content of Wikipedia', which gives an overview of 110 articles on the subject — attesting to his observation that Wikipedia has been 'irresistable point of unquiry for researchers from various fields of knowledge'. It will be a useful touching stone for this study, finding 82 out of the 110 surveyed articles to concern Wikipedia quality. Some of these are also referenced here, and many of the others will come to bear on the study as it progresses.

Other important general sources will be WikiLit,[42] AcaWiki[1] and WikiPapers[45], all of which are online repositories of academic research into Wikipedia and other Wikis.

Studies of Wikipedia revision history

Tangentialy related studies fall into two major groups: studies of Wikipedia article quality and studies of edit behaviour. It is from the first group that we find the most pertinent work — it is also one of the most fruitful areas of research.

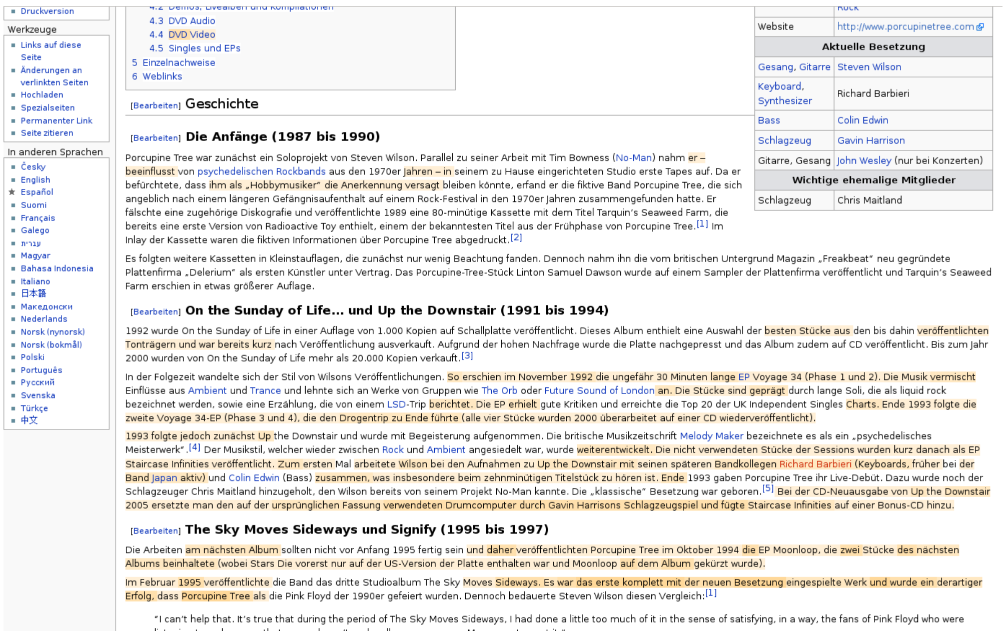
It is the metrics used to measure quality in these studies that are of most use to us here. We don't concern ourselves with the quality of the article on the whole, but many studies have endeavoured to find out what kind of article content can be automatically recognised. High numbers of Links, internal links, images and formulas have been found to indicate percieved quality,[24][27] and these are easy to identify using Wikipedia's markup language. Other useful meterics have been the age of the word,[6] the age and rate of change of the article in comparison to other articles,[50] and the recent activity of the article (an article undergoing a peak in edit changes may be 'unstable').[7] Another study of particular interest is that of Stvilia et al, which found metrics of article quality through factor analysis,[36] confirming much of the ideas already mentioned.

A landmark piece of work is the Wikitrust software.[2] Wikitrust was[3] a firefox plugin, designed to highlight the words of a Wikipedia article with different colors. The gradations of these colors relate to levels of trust, and the computations made to derive them were based upon the metrics mentioned above, with particular emphasis on a word's age. A screenshot can be seen in figure 5. The program was reviewed as recently as 2011,[23] and it was found to be basically flawed, with users not really seeing the use for it (it was found that, having read Wikipedia before, they already had a good idea of how to rate an article). However, the Wikitrust team's implementation of the quality measures described above will prove to be very useful study.

Another metric which may also affect the quality of an article as a resource, is logical structure. It was found in 2005 that this, if anything, was the clearest difference between Wikipedia and commercial encyclopedias,[12] supporting previous conjecture.[8] Not many studies have concerned themselves with structure, but later we will discuss how we may automatically recognise structural change.

A few other key studies present us with useful analyses of edit behaviour. Analyses of conflict between authors shows the possible reversion cases we will have to recognise. They reveal the high number of immediate 'undo'-type revisions, and also that malicious or unnecessary input may survive several versions before being undone. Some study these conflicts as a characterisation of normal editing behaviour,[19][17][18][32] while others look to controversial articles,[16] or articles recently cited in the press.[22] We find from these same studies that articles lead by small groups of 'leaders' produce articles of better quality than those with a more homogeneous contribution group, that a small group of editors contribute to most of Wikipedia, and that conflict and bureaucracy (increasing over time) are the major limiting factors in the growth of an article.[38] Knowledge of this context is vital in evaluating the edits we will eventually analyse.

[3]Defunct as of author's checks, Apr 2014



- Druckversion

Werkzeuge

- Links auf diese Seite
- Änderungen an verlinkten Seiten
- Hochladen
- Spezialseiten
- Permanenter Link
- Seite zitieren

In anderen Sprachen

- Česky
- English
- Español
- Suomi
- Français
- Galego
- עברית
- Magyar
- Bahasa Indonesia
- Italiano
- 日本語
- Македонски
- Nederlands
- Norsk (nynorsk)
- Norsk (bokmål)
- Polski
- Português
- Русский
- Svenska
- Türkçe
- 中文

Website	http://www.porcupinetree.com
Aktuelle Besetzung	
Gesang, Gitarre	Steven Wilson
Keyboard, Synthesizer	Richard Barbieri
Bass	Colin Edwin
Schlagzeug	Gavin Harrison
Gitarre, Gesang	John Wesley (nur bei Konzerten)
Wichtige ehemalige Mitglieder	
Schlagzeug	Chris Maitland

[Bearbeiten] Geschichte

[Bearbeiten] **Die Anfänge (1987 bis 1990)**

Porcupine Tree war zunächst ein Soloprojekt von Steven Wilson. Parallel zu seiner Arbeit mit Tim Bowness (No-Man) nahm er – beeinflusst von psychedelischen Rockbands aus den 1970er Jahren – in seinem zu Hause eingerichteten Studio erste Tapes auf. Da er befürchtete, dass ihm als „Hobbymusiker" die Anerkennung versagt bleiben könnte, erfand er die fiktive Band Porcupine Tree, die sich angeblich nach einem längeren Gefängnisaufenthalt auf einem Rock-Festival in den 1970er Jahren zusammengefunden hatte. Er fälschte eine zugehörige Diskografie und veröffentlichte 1989 eine 80-minütige Kassette mit dem Titel Tarquin's Seaweed Farm, die bereits eine erste Version von Radioactive Toy enthielt, einem der bekanntesten Titel aus der Frühphase von Porcupine Tree.[1] Im Inlay der Kassette waren die fiktiven Informationen über Porcupine Tree abgedruckt.[2]

Es folgten weitere Kassetten in Kleinstauflagen, die zunächst nur wenig Beachtung fanden. Dennoch nahm ihn die vom britischen Untergrund Magazin „Freakbeat" neu gegründete Plattenfirma „Delerium" als ersten Künstler unter Vertrag. Das Porcupine-Tree-Stück Linton Samuel Dawson wurde auf einem Sampler der Plattenfirma veröffentlicht und Tarquin's Seaweed Farm erschien in etwas größerer Auflage.

[Bearbeiten] **On the Sunday of Life… und Up the Downstair (1991 bis 1994)**

1992 wurde On the Sunday of Life in einer Auflage von 1.000 Kopien auf Schallplatte veröffentlicht. Dieses Album enthielt eine Auswahl der besten Stücke aus den bis dahin veröffentlichten Tonträgern und war bereits kurz nach Veröffentlichung ausverkauft. Aufgrund der hohen Nachfrage wurde die Platte nachgepresst und das Album zudem auf CD veröffentlicht. Bis zum Jahr 2000 wurden von On the Sunday of Life mehr als 20.000 Kopien verkauft.[3]

In der Folgezeit wandelte sich der Stil von Wilsons Veröffentlichungen. So erschien im November 1992 die ungefähr 30 Minuten lange EP Voyage 34 (Phase 1 und 2). Die Musik vermischt Einflüsse aus Ambient und Trance und lehnte sich an Werke von Gruppen wie The Orb oder Future Sound of London an. Die Stücke sind geprägt durch lange Soli, die als liquid rock bezeichnet werden, sowie eine Erzählung, die von einem LSD-Trip berichtet. Die EP erhielt gute Kritiken und erreichte die Top 20 der UK Independent Singles Charts. Ende 1993 folgte die zweite Voyage 34-EP (Phase 3 und 4), die den Drogentrip zu Ende führte (alle vier Stücke wurden 2000 überarbeitet auf einer CD wiederveröffentlicht).

1993 folgte jedoch zunächst Up the Downstair und wurde mit Begeisterung aufgenommen. Die britische Musikzeitschrift Melody Maker bezeichnete es als ein „psychedelisches Meisterwerk".[4] Der Musikstil, welcher wieder zwischen Rock und Ambient angesiedelt war, wurde weiterentwickelt. Die nicht verwendeten Stücke der Sessions wurden kurz danach als EP Staircase Infinities veröffentlicht. Zum ersten Mal arbeitete Wilson bei den Aufnahmen zu Up the Downstair mit seinen späteren Bandkollegen Richard Barbieri (Keyboards, früher bei der Band Japan aktiv) und Colin Edwin (Bass) zusammen, was insbesondere beim zehnminütigen Titelstück zu hören ist. Ende 1993 gaben Porcupine Tree ihr Live-Debüt. Dazu wurde noch der Schlagzeuger Chris Maitland hinzugeholt, den Wilson bereits von seinem Projekt No-Man kannte. Die „klassische" Besetzung war geboren.[5] Bei der CD-Neuausgabe von Up the Downstair 2005 ersetzte man den auf der ursprünglichen Fassung verwendeten Drumcomputer durch Gavin Harrisons Schlagzeugspiel und fügte Staircase Infinities auf einer Bonus-CD hinzu.

[Bearbeiten] **The Sky Moves Sideways und Signify (1995 bis 1997)**

Die Arbeiten am nächsten Album sollten nicht vor Anfang 1995 fertig sein und daher veröffentlichten Porcupine Tree im Oktober 1994 die EP Moonloop, die zwei Stücke des nächsten Albums beinhaltete (wobei Stars Die vorerst nur auf der US-Version der Platte enthalten war und Moonloop auf dem Album gekürzt wurde).

Im Februar 1995 veröffentlichte die Band das dritte Studioalbum The Sky Moves Sideways. Es war das erste komplett mit der neuen Besetzung eingespielte Werk und wurde ein derartiger Erfolg, dass Porcupine Tree als die Pink Floyd der 1990er gefeiert wurden. Dennoch bedauerte Steven Wilson diesen Vergleich:[1]

> "I can't help that. It's true that during the period of The Sky Moves Sideways, I had done a little too much of it in the sense of satisfying, in a way, the fans of Pink Floyd who were

Figure 5: Wikitrust in action, 2011

3 Conclusions

Now we have reviewed the existing work, we will begin to outline our intentions for the design of our own algorithm. Here we discuss the assumptions that we make about our data, our intentions with regards to analysing that data, and the metrics by which we will regard each contribution.

Assumptions

- **We assume that the final, or 'target' article is of 'good' quality.** There are many studies which concern themselves with verifying the accuracy and quality of Wikipedia articles — In this study we are specifically concerned with the quality of contribution, i.e. the quality of text within the article, relative to the domain of the article. Here, the quality of the article itself is a moot point.
- **We assume the article is well-formed.** As much as we do not concern ourselves with article accuracy, we also assume that the Wikipedia markup is also well-formed. We may also eventually check whether links are invalid, but for the moment we assume that they are.
- **We make no distinction between humans, bots and anonymous editors.**

Weighting contribution by markup

Given the extensive research regarding which features of a wikipedia article are most important, we may define the following features to have more weight than standard text from the outset:

- Links
- Images
- Equations

We may either preprocess the text to identify the each of these different 'flavours' of input by their Wikipedia mark-up conventions, or we may be able to more fluently work them into the main difference algorithm, raising and lowering flags during runtime.

Awarding restructuring

It has been found that, even in the most accurate articles, that the structure of Wikipedia article can be weak.[12] We should award attempts to reorganize articles, possibly looking out for lange block displacements.

Awarding the dense edits

We should give extra reward to the density of the changes. Perhaps we can propose the heuristic: A denser edit means more significant change. If we have a set of the indexes of an edit operation as $\{op_0, op_1, op_2, \ldots, op_n\}$, where op_i is the index of the ith operation, then we may evaluate it's density with a standard deviation of the edit itself, σ_{ed}, multiplied by the span of the edit itself in context of the wider article, and some weighting factor k. Something along the lines of:

$$ed_{density} = k \bullet \frac{(op_n - op_0)\sigma_{ed}}{|v_{ed}|}$$

where $|v_{ed}|$ is the overall length of resultant version. By implementing this carefully, we may achieve a gradient of weighting, with a lower weight values for things like spell-checks, and higher values for whole-paragraph changes.

Undone and partially undone operations

We consider three different ways of classifying an edit as valueless, or partially valueless. Two ways of classifying these kinds of edit are found in previous research, and are covered in figure 6.

$$\begin{tikzcd}
 & v_i \arrow[dr, "{ed(v_i, v_{i+1})}"] & \\
v_{i-1} \arrow[ur, "{ed(v_{i-1}, v_i)}"] \arrow[rr, "{ed(v_{i-1}, v_{i+1})}"'] & & v_{i+1}
\end{tikzcd}$$

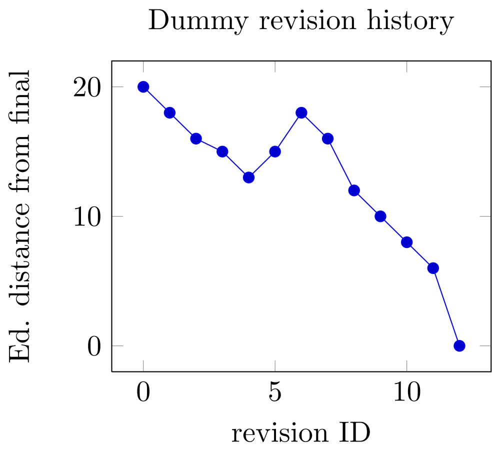
case a) if $ed(v_{i-1}, v_i) < ed(v_{i-1}, v_i) + ed(v_i, v_{i+1})$, then $ed(v_{i-1}, v_i)$ has been partially undone.
case b) if $ed(v_{i-1}, v_{i+1}) = 0$ $(v_{i-1} = v_{i+1})$, then $ed(v_{i-1}, v_i)$ has been completely undone.

Figure 6: Diagram showing identification of a partially or completely undone edit

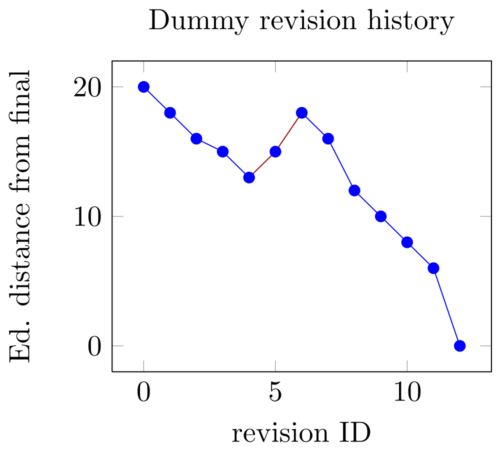
Figure 6 shows how we may identify undone or partially undone edits, when they are undone immediately, as layed out in Adler et al.[2] The triangle represents three consecutive revisions, and the arrows are the edit operations that transform one to another. By calculating the edit distance between the distance versions v_{i-1} and v_{i+1} we may characterise a longer history of revisions than usual, and use that context in order to re-characterise the edits it encompasses. In the figure above, case a describes v_i as a 'diversion'. If v_{i-1} can be transformed into v_{i+1} with less operations than the two edits that actually bridged that gap, then perhaps some of the edits in $v_{i-1} \to v_i$ were unnecessary, and undone by v_{i+1}. In this case, we may punish the edit $v_{i-1} \to v_i$ (for the diversion), reward $v_i \to v_{i+1}$, or both. Case b is an extreme version of a — the texts v_{i-1} and v_{i+1} are identical, so the changes in v_i must have been completely undone. These reverts are common in normal Wikipedia edit practice.[46]

This algorithm, however, is limited in its scope, and we may come across situations where reversions occur over a series of edits. Although the system may easily be extended to cover larger spans of history, to consider many nodes would require the edit-distance computation of very many different node pairs.

Let us propose a more efficient way of characterising redundant entries in terms of longer history spans. We must utilise the fact that we have take one article to be the ultimate destination of all previous edits in order to do so. Let us graph the entirety of a wikipedia's revision history in terms of the edit distance from this final version, thus:

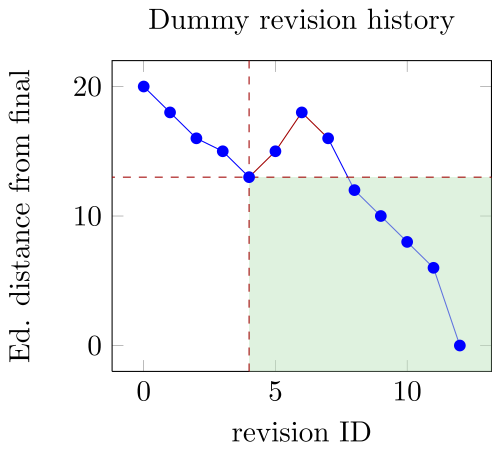


Each point represents a different version, and each line represents the the edit-distance between each version. Given this information, we may consider only those revisions that bring us closer to the final version. They appear on this graph as lines with a negative gradient (blue); those with a positive gradient take us further away from the final version (red):



We need not computer edit distances with a positive gradient (the two red lines). This is simple to implement. Given a version v_i, having computed it's immediate edit distance $ed(v_{i-1}, v_i)$, and it's edit distance from the final version, ed_{final_i}, we know our next computation must be $ed(v_{j-1}, ed_j)$, such that j is the smallest number that satisfies the qualities $j > i$ and $ed_{final_j} < ed_{final_{j-1}}$.

Another possible strategy would be to disregard all edit distances that, at any point, lie between two versions that are further away from final version than the version currently being considered.



In this graph, v_i, the currently considered version, is represented by the node at (4,13). The green rectangle shows the area in which we look to find v_j, and the blue lines are those edit distances we both to compute. In this case, after considering v_i, we move to the v_j such that j is the smallest number that satisfies to qualities $j > i$ and $ed_{final_j} < ed_{final_i}$. In this graph above we move from (4,13) to (8,12).

It is worth stating here that, in both these strategies, after discovering v_j, we always compute $ed(v_{j-1}, v_j)$ rather than $ed(v_i, v_j)$. We will look into the pros and cons of these different strategies further into the project.

Possible extensions

Visualisation

Given that we are distributing wealth according to a series of weight factors, it may be useful to devise a system that visualises how these weight factors affect distribution. This would simplybe a matter of expressing the final 'score' in terms of these variables, and producing some interactive graphs. Several good attempts have been made to visualise Wikipedia history data, with varying levels of success,[5][34][37][49][41] so the work would be well supported. We will look into the viability of this extension further into the project.

Extending data

Current intentions are to approach the data on an article-by-article basis, grabbing a particular version and tracing its history backwards. We grab the pages using HTTP requests, via Mediawiki's inbuilt API. We could, however, download Wikipedia in its entirety. The entire site is compressed and dumped monthly, and the dumps are free to download (though they are 800GB compressed).[43]

Further analysis

Given the over-arching nature of Wikis, it may be possible to derive other information about the articles we study. Past studies have used revision histories in order to figure out a variety of different things, including a 2012 study which used it to predict box-office success,[29] a 2009 study that was able to geo-locate editors by their edits.[21]

Given the by-products of our study, we may be able compare and contrast different categories of article, or editor. It would be interesting to contrast the actions of humans, and bots,[4] and perhaps look at the nature of edits made by different groups of editors.

Further subjects

The project may well extend to subjects beyond Wikipedia. A git project history, for instance, may be of interest for further study. We may combine the existing research with metrics that concern code in particular, such as Cyclomatic Complexity, which measures code flow complexity according to its logical operators.[26] It would also be interesting to figure out a way of changing our algorithm in order to regard non-linear revision histories.

4 Progress report

So far I have written two Python classes, which, together can fetch an entire history of a wikipedia article, and compute Levenshtein distances between the texts.

WikiRevisionScrape is a Python class which harnesses the Wikipedia API in order to download various pieces of information about articles and their histories. It is inspired by open Wikipedia

[4]It has been noted that there are around 700 bots registered on Wikipedia (as of 2014). Though not all of them make edits, those that do are very prolific, and are known to reverse malicious submisions in a matter of seconds.[47][31]

metadata classes such as 'Wikipedia Miner'[10], or the revision-fetching 'Java Wikipedia Library / Wikipedia Revision Library'.[9][11]

If the user doesn't specify a particular article title, we choose a random one, and trace it's history back. We can also set a parameter in order to pick a random article multiple times. There are various ways of improving the efficiency of this program (such as requesting multiple pages at once), and this will be introduced in a later version. At the moment the scraper is fully functional, though at the moment saves the data in CSV files. I will change this so that it uses a postgres database. **Code and example output can be found in appendix A.1 starting page 16.**

Another python class, LevDistBasic, is a naive implementation of Levenshtein distance (no space or speed optimisations). It is a first-attempt implementation of the algorithm, and it can return the Levenshtein distance, the computation table, the edit operation (in two different formats — human readable, and as a list of tuples). Future changes are many, such as including weightings for different kinds of string, more space-efficient and speed efficient implementations, etc. **Code and example output can be found in appendix B starting page 20.**

These two classes can be entwined manually, by fetching two pieces of data from the former and feeding them into the latter. The next step will be to build a class which autmatically builds a database of files, calculating and storing distances as it does so, but there is no point in writing such a class until I change the way the scraper deals with the data it fetches (it should instead returned at the end of a function — the use of CSV files is a temporary hack).

I choose Python principally for the ease at which it handles and passes around different kinds of data. However, even the optimised versions of the algorithms I will use will be fairly slow. If speed becomes an issue I will rewrite my code into C++, but since speed-efficiency is not really the goal of this project, this may not be a consideration.

References

[1] *AcaWiki*. `http://acawiki.org`. Accessed: 2014-04-31 (see p. 6).

[2] B. Thomas Adler and Luca de Alfaro. "A Content-driven Reputation System for the Wikipedia". In: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada: ACM, 2007, pp. 261–270. ISBN: 978-1-59593-654-7. DOI: `10.1145/1242572.1242608`. URL: `http://doi.acm.org/10.1145/1242572.1242608` (see pp. 6, 8).

[3] *Alexa About Us*. `http://www.alexa.com/about`. Accessed: 2014-04-31 (see p. 5).

[4] William I. Chang and Jordan Lampe. "Theoretical and Empirical Comparisons of Approximate String Matching Algorithms". In: *Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*. CPM '92. London, UK, UK: Springer-Verlag, 1992, pp. 175–184. ISBN: 3-540-56024-6. URL: `http://dl.acm.org/citation.cfm?id=647812.738270` (see p. 3).

[5] Ed H Chi, Bongwon Suh, and Aniket Kittur. "Providing social transparency through visualizations in Wikipedia". In: *Proceedings of the Social Data Analysis Workshop at CHI*. Vol. 8. 2008 (see p. 10).

[6] Tom Cross. "Puppy smoothies: Improving the reliability of open, collaborative wikis". In: *First Monday* 11.9 (2006). ISSN: 13960466. URL: `http://firstmonday.org/ojs/index.php/fm/article/view/1400` (see p. 6).

[7] Luca De Alfaro, Ashutosh Kulshreshtha, Ian Pye, and B. Thomas Adler. "Reputation Systems for Open Collaboration". In: *Commun. ACM* 54.8 (Aug. 2011), pp. 81–87. ISSN: 0001-0782. DOI: `10.1145/1978542.1978560`. URL: `http://doi.acm.org/10.1145/1978542.1978560` (see p. 6).

[8] Peter Denning, Jim Horning, David Parnas, and Lauren Weinstein. "Wikipedia Risks". In: *Commun. ACM* 48.12 (Dec. 2005), pp. 152–152. ISSN: 0001-0782. DOI: 10.1145/1101779.1101804. URL: http://doi.acm.org/10.1145/1101779.1101804 (see p. 6).

[9] Various developers. *Google Code — JWPL and the Wikipedia Revision Toolkit.* [Online; accessed 2014-06-03]. 2012. URL: https://code.google.com/p/jwpl/ (see p. 11).

[10] Various developers. *Wikipedia Miner.* [Online; accessed 2014-06-03]. 2012. URL: http://wikipedia-miner.cms.waikato.ac.nz/services/ (see p. 11).

[11] Oliver Ferschke, Torsten Zesch, and Iryna Gurevych. "Wikipedia Revision Toolkit: Efficiently Accessing Wikipedia's Edit History". In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: Systems Demonstrations.* HLT '11. Portland, Oregon: Association for Computational Linguistics, 2011, pp. 97–102. ISBN: 978-1-932432-90-9. URL: http://dl.acm.org/citation.cfm?id=2002440.2002457 (see p. 11).

[12] Jim Giles. "Internet encyclopaedias go head to head". In: *Nature* 438.7070 (2005), pp. 900–901. ISSN: 0028-0836. URL: http://dx.doi.org/10.1038/438900a (see pp. 6, 7).

[13] R. W. Hamming. "Error Detecting and Error Correcting Codes". In: *Bell System Technical Journal* 29.2 (1950), pp. 147–160. ISSN: 1538-7305. DOI: 10.1002/j.1538-7305.1950.tb00463.x. URL: http://dx.doi.org/10.1002/j.1538-7305.1950.tb00463.x (see p. 3).

[14] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. "Delta Algorithms: An Empirical Analysis". In: *ACM Trans. Softw. Eng. Methodol.* 7.2 (Apr. 1998), pp. 192–214. ISSN: 1049-331X. DOI: 10.1145/279310.279321. URL: http://doi.acm.org/10.1145/279310.279321 (see p. 5).

[15] Heikki Hyyrö. "A bit-vector algorithm for computing Levenshtein and Damerau edit distances". In: *Nordic Journal of Computing* (2003), p. 2003 (see p. 3).

[16] Takashi Iba, Keiichi Nemoto, Bernd Peters, and Peter A. Gloor. "Analyzing the Creative Editing Behavior of Wikipedia Editors: Through Dynamic Social Network Analysis". In: *Procedia - Social and Behavioral Sciences* 2.4 (2010). The 1st Collaborative Innovation Networks Conference - {COINs2009}, pp. 6441 –6456. ISSN: 1877-0428. DOI: http://dx.doi.org/10.1016/j.sbspro.2010.04.054. URL: http://www.sciencedirect.com/science/article/pii/S1877042810011122 (see p. 6).

[17] Aniket Kittur, Ed H. Chi, and Bongwon Suh. "What's in Wikipedia?: Mapping Topics and Conflict Using Socially Annotated Category Structure". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* CHI '09. Boston, MA, USA: ACM, 2009, pp. 1509–1512. ISBN: 978-1-60558-246-7. DOI: 10.1145/1518701.1518930. URL: http://doi.acm.org/10.1145/1518701.1518930 (see p. 6).

[18] Aniket Kittur and Robert E. Kraut. "Beyond Wikipedia: Coordination and Conflict in Online Production Groups". In: *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work.* CSCW '10. Savannah, Georgia, USA: ACM, 2010, pp. 215–224. ISBN: 978-1-60558-795-0. DOI: 10.1145/1718918.1718959. URL: http://doi.acm.org/10.1145/1718918.1718959 (see p. 6).

[19] Aniket Kittur, Bongwon Suh, Bryan A. Pendleton, and Ed H. Chi. "He Says, She Says: Conflict and Coordination in Wikipedia". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* CHI '07. San Jose, California, USA: ACM, 2007, pp. 453–462. ISBN: 978-1-59593-593-9. DOI: 10.1145/1240624.1240698. URL: http://doi.acm.org/10.1145/1240624.1240698 (see p. 6).

[20] VI Levenshtein. “Binary Codes Capable of Correcting Deletions, Insertions and Reversals”. In: *Soviet Physics Doklady* 10 (1966), p. 707 (see p. 2).

[21] Michael Lieberman and Jimmy Lin. *You Are Where You Edit: Locating Wikipedia Contributors through Edit Histories.* 2009. URL: `http://aaai.org/ocs/index.php/ICWSM/09/paper/view/204` (see p. 10).

[22] Andrew Lih. “Wikipedia as Participatory journalism: reliable sources? metrics for evaluating collaborative media as a news resource”. In: *In Proceedings of the 5th International Symposium on Online Journalism.* 2004, pp. 16–17 (see p. 6).

[23] Teun Lucassen and Jan Schraagen. “Evaluating WikiTrust: A trust support tool for Wikipedia”. In: *First Monday* 16.5 (2011). ISSN: 13960466. URL: `http://firstmonday.org/ojs/index.php/fm/article/view/3070` (see p. 6).

[24] Teun Lucassen and Jan Maarten Schraagen. “Trust in Wikipedia: How Users Trust Information from an Unknown Source”. In: *Proceedings of the 4th Workshop on Information Credibility.* WICOW ’10. Raleigh, North Carolina, USA: ACM, 2010, pp. 19–26. ISBN: 978-1-60558-940-4. DOI: `10.1145/1772938.1772944`. URL: `http://doi.acm.org/10.1145/1772938.1772944` (see p. 6).

[25] Josh MacDonald. “File system support for delta compression”. PhD thesis. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000 (see p. 5).

[26] Thomas J. McCabe. “A Complexity Measure”. In: *Proceedings of the 2Nd International Conference on Software Engineering.* ICSE ’76. San Francisco, California, USA: IEEE Computer Society Press, 1976, pp. 407–. URL: `http://dl.acm.org/citation.cfm?id=800253.807712` (see p. 10).

[27] Deborah L McGuinness, Honglei Zeng, Paulo Pinheiro Da Silva, Li Ding, Dhyanesh Narayanan, and Mayukh Bhaowal. “Investigations into Trust for Collaborative Information Repositories: A Wikipedia Case Study.” In: *MTW* 190 (2006) (see p. 6).

[28] Mostafa Mesgari, Chitu Okoli, Mohamad Mehdi, Finn Årup Nielsen, and Arto Lanamäki. ““The sum of all human knowledge”: A systematic review of scholarly research on the content of Wikipedia”. In: *Journal of the American Society for Information Science and Technology* (2014). This is a postprint of an article accepted for publication in Journal of the American Society for Information Science and Technology copyright © 2014 (American Society for Information Science and Technology). URL: `http://spectrum.library.concordia.ca/978618/` (see pp. 5, 14).

[29] Mårton Mestyån, Taha Yasseri, and Jånos Kertåsz. “Early Prediction of Movie Box Office Success based on Wikipedia Activity Big Data”. In: *CoRR* abs/1211.0970 (2012). URL: `http://dblp.uni-trier.de/db/journals/corr/corr1211.html#abs-1211-0970` (see p. 10).

[30] Gene Myers. “A Fast Bit-vector Algorithm for Approximate String Matching Based on Dynamic Programming”. In: *J. ACM* 46.3 (May 1999), pp. 395–415. ISSN: 0004-5411. DOI: `10.1145/316542.316550`. URL: `http://doi.acm.org/10.1145/316542.316550` (see p. 3).

[31] Daniel Nasaw. *BBC News — Meet the ’bots’ that edit Wikipedia.* [Online; accessed 2014-06-02]. 2012. URL: `http://en.wikipedia.org/w/index.php?title=Wikipedia:Bots&oldid=602889799` (see p. 10).

[32] Martin Potthast, Benno Stein, and Robert Gerling. “Automatic Vandalism Detection in Wikipedia”. In: *Advances in Information Retrieval.* Ed. by Craig Macdonald, Iadh Ounis,

Vassilis Plachouras, Ian Ruthven, and RyenW. White. Vol. 4956. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 663–668. ISBN: 978-3-540-78645-0. DOI: 10.1007/978-3-540-78646-7_75. URL: http://dx.doi.org/10.1007/978-3-540-78646-7_75 (see p. 6).

[33] *Python wikipedia 1.3.0.* https://pypi.python.org/pypi/wikipedia. "Wikipedia is a Python library that makes it easy to access and parse data from Wikipedia." Accessed: 2014-04-31 (see p. 16).

[34] Mikalai Sabel. "Structuring Wiki Revision History". In: *Proceedings of the 2007 International Symposium on Wikis.* WikiSym '07. Montreal, Quebec, Canada: ACM, 2007, pp. 125–130. ISBN: 978-1-59593-861-9. DOI: 10.1145/1296951.1296965. URL: http://doi.acm.org/10.1145/1296951.1296965 (see p. 10).

[35] T.F. Smith and M.S. Waterman. "Identification of common molecular subsequences". In: *Journal of Molecular Biology* 147.1 (1981), pp. 195 –197. ISSN: 0022-2836. DOI: http://dx.doi.org/10.1016/0022-2836(81)90087-5. URL: http://www.sciencedirect.com/science/article/pii/0022283681900875 (see p. 4).

[36] B. Stvilia, M.B. Twidale, L.C. Smith, and L. Gasser. "Assessing information quality of a community-based encyclopedia". In: *Proceedings of the International Conference on Information Quality.* 2005, pp. 442–454 (see p. 6).

[37] B. Suh, Ed H. Chi, B.A. Pendleton, and A. Kittur. "Us vs. Them: Understanding Social Dynamics in Wikipedia with Revert Graph Visualizations". In: *Visual Analytics Science and Technology, 2007. VAST 2007. IEEE Symposium on.* 2007, pp. 163–170. DOI: 10.1109/VAST.2007.4389010 (see p. 10).

[38] Bongwon Suh, Gregorio Convertino, Ed H. Chi, and Peter Pirolli. "The Singularity is Not Near: Slowing Growth of Wikipedia". In: *Proceedings of the 5th International Symposium on Wikis and Open Collaboration.* WikiSym '09. Orlando, Florida: ACM, 2009, 8:1–8:10. ISBN: 978-1-60558-730-1. DOI: 10.1145/1641309.1641322. URL: http://doi.acm.org/10.1145/1641309.1641322 (see p. 6).

[39] Walter F. Tichy. "The String-to-string Correction Problem with Block Moves". In: *ACM Trans. Comput. Syst.* 2.4 (Nov. 1984), pp. 309–321. ISSN: 0734-2071. DOI: 10.1145/357401.357404. URL: http://doi.acm.org/10.1145/357401.357404 (see p. 4).

[40] Esko Ukkonen. "Approximate string-matching with¡ i¿ q¡/i¿-grams and maximal matches". In: *Theoretical computer science* 92.1 (1992), pp. 191–211 (see p. 4).

[41] Fernanda B. Viégas, Martin Wattenberg, and Kushal Dave. "Studying Cooperation and Conflict Between Authors with History Flow Visualizations". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* CHI '04. Vienna, Austria: ACM, 2004, pp. 575–582. ISBN: 1-58113-702-8. DOI: 10.1145/985692.985765. URL: http://doi.acm.org/10.1145/985692.985765 (see p. 10).

[42] *WikiLit.* http://wikilit.referata.com. A temporary site, relating directly to [28]. Accessed: 2014-04-31 (see p. 6).

[43] *Wikimedia enwiki dump progress on 20140502.* http://dumps.wikimedia.org/enwiki/20140502/. Accessed: 2014-04-29 (see p. 10).

[44] *Wikimedia Wikipedia.org is more popular than...: Note on Alexa rankings.* https://meta.wikimedia.org/wiki/Wikipedia.org_is_more_popular_than...#Note_on_Alexa_rankings. Accessed: 2014-04-31 (see p. 5).

[45] *WikiPapers*. `http://wikipapers.referata.com`. Accessed: 2014-04-31 (see p. 6).

[46] Wikipedia. *Help: Reverting — Wikipedia, The Free Encyclopedia.* [Online; accessed 2014-06-02]. 2014. URL: `http://en.wikipedia.org/w/index.php?title=Help:Reverting&oldid=606567776` (see p. 8).

[47] Wikipedia. *Wikipedia:Bots — Wikipedia, The Free Encyclopedia.* [Online; accessed 2014-06-02]. 2014. URL: `http://en.wikipedia.org/w/index.php?title=Wikipedia:Bots&oldid=602889799` (see p. 10).

[48] Dennis M. Wilkinson and Bernardo A. Huberman. "Assessing the Value of Coooperation in Wikipedia". In: *CoRR* abs/cs/0702140 (2007) (see p. 5).

[49] Jianmin Wu and Mizuho Iwaihara. "Revision Graph Extraction in Wikipedia Based on Supergram Decomposition". In: *Proceedings of the 9th International Symposium on Open Collaboration.* WikiSym '13. Hong Kong, China: ACM, 2013, 10:1–10:6. ISBN: 978-1-4503-1852-5. DOI: `10.1145/2491055.2491065`. URL: `http://doi.acm.org/10.1145/2491055.2491065` (see p. 10).

[50] Honglei Zeng, Maher A. Alhossaini, Li Ding, Richard Fikes, and Deborah L. McGuinness. "Computing Trust from Revision History". In: *Proceedings of the 2006 International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services.* PST '06. Markham, Ontario, Canada: ACM, 2006, 8:1–8:1. ISBN: 1-59593-604-1. DOI: `10.1145/1501434.1501445`. URL: `http://doi.acm.org/10.1145/1501434.1501445` (see p. 6).

A Appendix A: Code progress

A.1 Python class for scraping a Wikipedia article's revision history

The following code is a first draft of a class which incrementally traces, parses, and stores the revision history of select articles. It chooses random articles up to the limit specified (default = 1). It traces the entire discoverable[5] history, unless a specific depth is specified by the user.

The class already yields workable data, but here is some immediate further work for this code:

- Allow the user to specify timeframe for revision history
- Allow for integration with a postgres database (at the moment the code saves the data in CSV format).

This leverages an existing wikipedia python class for some of the more trivial parts of fetching the article, and for picking a random title.[33]

```
import requests
import time
import json
import csv
import wikipedia
from bs4 import BeautifulSoup
from datetime import datetime, timedelta
from decimal import Decimal

WIKI_API_URL = 'http://en.wikipedia.org/w/api.php'
WIKI_USER_AGENT = 'wikipedia (https://github.com/goldsmith/Wikipedia/)'

class WikiRevisionScrape:
    par = {
        'format': 'json',
        'action': 'query',
        'prop': 'revisions'
        }
    head = {
        'User-Agent': WIKI_USER_AGENT
        }
    rand = True
    pagelimit = 1
    historylimit = -1
    rl = False
    rl_minwait = None
    rl_lastcall = None
    pageid = 0
    parentid = 0
    childid = 0

    #atm naively assuming headers, params, titles to be in correct format
    def __init__(self, pagelimit=1, historylimit=-1, _headers=None, _params=None,
        _titles=None):
        if(_params):
            params = _params

        if(_headers):
            self.head = _headers

        if(_titles):
```

[5]Using the Wikipedia API, articles can either be traced back to their origin (revision parent ID = 0), or to the point at which a loop is found in the revision history — this usually happens with older articles; not all articles can be traced all the way back to the origin.

```
            self.params['titles'] = _titles
            self.rand = False

        self.pagelimit = pagelimit
        self.historylimit = historylimit


    def scrape(self, indexfilename, contentsfilename):
        self._pace()
        index_f = open(indexfilename + ".csv", "ab") #HACK = needs to migrate to postrgres
        contents_f = open(contentsfilename + ".csv", "ab") #HACK = needs to migrate to
            postrgres
        index = csv.writer(index_f)
        contents = csv.writer(contents_f)
        index.writerow(["PAGEID","REVISION","USER","USERID","TIMSTAMP","SIZE","COMMENT"])
        contents.writerow(["PAGEID","REVISION","CONTENT"])

        for i in range(self.pagelimit):
            if 'rvprop' in self.par:
                del self.par['rvprop']
            if 'revids' in self.par:
                del self.par['revids']
            print "fetching page"
            if(self.rand):
                self.par['titles'] = wikipedia.random() #get random title
            self.childid = self._getlatest()
            r = requests.get(WIKI_API_URL, params=self.par, headers=self.head)
            self._rate()
            del self.par['titles']
            self._tracehist(index, contents)

    def _getlatest(self):
        r = requests.get(WIKI_API_URL, params=self.par, headers=self.head)
        r = r.json()

        #HACK = should grab multiple pages
        for key, value in r['query']['pages'].iteritems():
            self.pageid = key
        #HACK = chould grab multiple revisions (for each pageid)
        self.parentid = self.childid =
            r['query']['pages'][self.pageid]['revisions'][0]['revid']
        return self.childid

    def _tracehist(self, index, contents):
        ##We store revisions we've visited
        ##loops can occur in revision histories
        visited = []
        i = self.historylimit
        j = 0

        self.par['rvprop'] =
            'userid|user|ids|flags|tags|size|comment|contentmodel|timestamp|content'

        while (self.parentid not in visited) and i is not 0 and self.parentid is not 0:
            self.par['revids'] = self.parentid

            self._pace()

            r = requests.get(WIKI_API_URL, params=self.par, headers=self.head)
            r = r.json()
            self._rate()

            visited.append(self.childid)

            #print r

            self.childid = r['query']['pages'][self.pageid]['revisions'][0]['revid']
            self.parentid = r['query']['pages'][self.pageid]['revisions'][0]['parentid']
            user = r['query']['pages'][self.pageid]['revisions'][0]['user']
            userid = r['query']['pages'][self.pageid]['revisions'][0]['userid']
            size = r['query']['pages'][self.pageid]['revisions'][0]['size']
            timestamp = r['query']['pages'][self.pageid]['revisions'][0]['timestamp']
            comment = "" #comments sometimes don't return from old revisions...
            try:
                comment = r['query']['pages'][self.pageid]['revisions'][0]['comment']
            except:
                comment = ""
            content = r['query']['pages'][self.pageid]['revisions'][0]['*']

            index.writerow([self.pageid, self.childid, user.encode("UTF-8"), userid,
                timestamp, size, comment.encode("UTF-8")])
            contents.writerow([self.pageid, self.childid, content.encode("UTF-8")])

            if(self.historylimit > 0):
                print self.pageid, "fetch", j+1, "of", self.historylimit, ", revid",
                    self.childid, "timestamp", str(timestamp)
                i = i - 1
            else:
                print self.pageid, "fetch", j+1, ", revid", self.childid, "timestamp",
                    str(timestamp)
            j = j + 1
        print "limit reached"

    def _pace(self):
        if self.rl and self.rl_last_call and self.rl_lastcall + self.rl_minwait >
            datetime.now():
            wait_time = (self.rl_lastcall + self.rl_minwait) - datetime.now()
            time.sleep(int(wait_time.total_seconds()))

    def _rate(self):
        if self.rl:
            self.rl_lastcall = datetime.now()
```

A.2 Example output

```
$ python
>>> import WikiRevisionScrape

>>> singlescraper = WikiRevisionScrape.WikiRevisionScrape()

>>> singlescraper.scrape("filename1","filename2")
fetching page
searching for
{u'action': u'query', u'list': u'random', u'rnlimit': 1,
     u'rnnamespace': 0, u'format': u'json'}
25455543 fetch 1 , revid 553292956 timestamp 2013-05-03T03:01:26Z
25455543 fetch 2 , revid 550043052 timestamp 2013-04-12T18:59:57Z
25455543 fetch 3 , revid 503496279 timestamp 2012-07-21T21:52:51Z
```

```
.
. [skipping some output]
.
25455543 fetch 23 , revid 331902859 timestamp 2009-12-15T23:25:23Z
25455543 fetch 24 , revid 331902368 timestamp 2009-12-15T23:22:50Z
25455543 fetch 25 , revid 331902181 timestamp 2009-12-15T23:21:47Z
limit reached

>>> multiscraper = WikiRevisionScrape.WikiRevisionScrape(pagelimit=1000)

>>> multiscraper.scrape("multifilename1","multifilename2")
fetching page
searching for
{u'action': u'query', u'list': u'random', u'rnlimit': 1,
     u'rnnamespace': 0, u'format': u'json'}
7096591 fetch 1 , revid 472732138 timestamp 2012-01-23T03:00:01Z
7096591 fetch 2 , revid 416290467 timestamp 2011-02-28T00:06:47Z
.
. [skipping some output]
.
7096591 fetch 8 , revid 89546539 timestamp 2006-11-22T23:31:09Z
7096591 fetch 9 , revid 77039186 timestamp 2006-09-21T20:00:55Z
limit reached
fetching page
searching for
{u'action': u'query', u'list': u'random', u'rnlimit': 1,
      u'rnnamespace': 0, u'format': u'json'}
24830105 fetch 1 , revid 547881527 timestamp 2013-03-30T21:34:39Z
24830105 fetch 2 , revid 500160388 timestamp 2012-07-01T09:55:31Z
.
. [skipping some output]
.
[etc.]
```

B Appendix B: Python Levenshtein distance implementation

This Python class gives a basic implementation of Levenshtein distance. To compare strings x and y both the time and space complexity is $\Theta log(|x| \bullet |y|)$.

The class is instantiated with the twwo strings, or .txt files, it is to compare. Methods can then be accessed in order to examine the distance between the provided strings. The class provides command-line visualisations of the data — it can print out its table of computations, as well as instructions on how to transform one into the other. Please see the exmple output below. It does not currently give any information about optimal alignment, although information about this alignment is found in the table.

B.1 Code

```
import sys

class LevDistBasic:
    e = [] #edit operation array
    t = [] #grid array
    x = "" #string1
    y = "" #string2
    m = 0 #length string1
    n = 0 #length string2
    dist = 0 #Levenshtein distance
    ed = [] #the edit operation, calculated in _calculate()
    isFile = False

    def __init__(self, _x, _y, isFile=False):
        self.x = self._variablehandle(_x)
        self.y = self._variablehandle(_y)
        self.m = len(self.x)
        self.n = len(self.y)
        self.t = [[0]*(self.n+1) for _ in xrange(self.m+1)]
        self.e = [[" "]*(self.n+1) for _ in xrange(self.m+1)]
        self.dist = self._calculate()

    def __str__(self):
        return str(self.distance())

    def distance(self):
        return self.dist

    def strings(self):
        return self.x, self.y

    def table(self):
        return self.t

    def operation(self):
        return self.ed

    ##ADD WARNING for long strings / deal with them
    def showtable(self):
        result = ""
        for ch in self.y:
            result = result + ch + " "
        print "      ", result
        for r in range(len(self.t)):
            s = ' '
            if r:
                s = self.x[r-1]
            print s, ' ', self.t[r]

    def showop(self):
        for i, op in enumerate(self.ed):
            l = str(i) + ": "
            if op[0] == 'I':
                l += "insert " + op[-1]
            elif op[0] == 'K':
                l += "keep " + op[-1]
            elif op[0] == 'D':
                l += "delete " + op[-1]
            elif op[0] == 'S':
                l += "swap " + op[-1][0] + " for " + op[-1][-1]
            else:
                return "FAIL: incorrect operation"
            print l

    def _ed(self):
        i, j = len(self.e)-1, len(self.e[0])-1
        self._ed_recursive(i,j)

    def _ed_recursive(self,i,j):
        if self.e[i][j] == ' ':
            if i == 0 and j > 0:
                self.ed.append(('D', self.y[0]))
            if j == 0 and i > 0:
                self.ed.append(('D', self.x[0]))
            return
        if self.e[i][j] == 'K':
            self._ed_recursive(i-1, j-1)
            self.ed.append((self.e[i][j], self.x[i-1]))
        elif self.e[i][j] == 'S':
            self._ed_recursive(i-1, j-1)
            self.ed.append((self.e[i][j], (self.x[i-1] + ',' + self.y[j-1])))
        elif self.e[i][j] == 'D':
            self._ed_recursive(i-1,j)
            self.ed.append((self.e[i][j], self.x[i-1]))
        else:
            self._ed_recursive(i,j-1)
            self.ed.append((self.e[i][j], self.y[j-1]))

    def _variablehandle(self,v):
        if not isinstance(v, str):
            try:
                return v.read()
            except:
                try:
                    return str(v)
                except:
                    print "Argument cannot be of type" + type(v)
                    raise
                pass
        return v

    def _calculate(self):
        for i in xrange(self.m+1):
            self.t[i][0] = i
        for j in xrange(self.n+1):
            self.t[0][j] = j
        j = 1
        while j < self.n+1:
            i = 1
            while i < self.m+1:
                c = (self.x[i-1] != self.y[j-1])
                dl = self.t[i-1][j] + 1
                ins = self.t[i][j-1] + 1
                sbs = self.t[i-1][j-1] + c
                self.t[i][j] = min(ins, dl, sbs)
                if ins < dl and ins < sbs:
                    self.e[i][j] = 'I'
                elif dl <= sbs:
                    self.e[i][j] = 'D'
                else:
                    if(self.x[i-1] != self.y[j-1]):
                        self.e[i][j] = 'S'
                    else:
                        self.e[i][j] = 'K'
                i += 1
            j += 1
        self._ed()
        return self.t[self.m][self.n]
```

B.2 Example output

```
$ python
>>> import LevDistBasic
>>> test = LevDistBasic.LevDistBasic("bank","book")
>>> test.showtable()

         b   o   o   k
    [0, 1, 2, 3, 4]
b   [1, 0, 1, 2, 3]
a   [2, 1, 1, 2, 3]
n   [3, 2, 2, 2, 3]
k   [4, 3, 3, 3, 2]

>>> t = test.table()
>>> print t

[[0, 1, 2, 3, 4], [1, 0, 1, 2, 3], [2, 1, 1, 2, 3], [3, 2, 2, 2, 3],
[4, 3, 3, 3, 2]]

>>> s = test.strings()
>>> print s

('bank', 'book')

>>> test.showop()

0: keep b
1: swap a for o
2: swap n for o
3: keep k

>>> ed = test.operation()
>>> print ed

[('K', 'b'), ('S', 'a,o'), ('S', 'n,o'), ('K', 'k')]

>>> print test

2
```