Diff Strategies

by Neil Fraser, April 2006

Computing the differences between two sequences is at the core of many applications. Below is a simple example of the difference between two texts:

```
Text 1: Apples are a fruit.
Text 2: Bananas are also fruit.
Diff:   Apple Bananas are a also fruit.
```

This paper surveys the literature on difference algorithms, compares them, and describes several techniques for improving the usability of these algorithms in practice. In particular, it discusses pre-processing optimisations, strategies for selecting the best difference algorithm for the job, and post-processing cleanup.

---

1 Pre-processing Optimisations

Even the best-known difference algorithms are computationally expensive processes. In most real-world instances, the two sequences (usually text) being compared are similar to each other to a certain extent. This observation enables several optimisations that can improve the actual running time of an algorithm, and in certain cases, that can even obviate the need for running the algorithm altogether.

1.1 Equality

The most obvious and the simplest optimisation is the equality test. Since there is a non-trivial chance that the two sequences are identical, and the test for this case is so trivial, it is logical to test for this case first. One side effect of this test is that it may simplify subsequent code. After this test, there is guaranteed to be a difference; the null case is eliminated.

Sample Code

view plain print ?

```
01. if text1 == text2:
02.   return None
```

Show JavaScript | Show Java | Hide Python

1.2 Common Prefix/Suffix

If there is any commonality at all between the texts, it is likely that they will share a common substring at the start and/or the end.

```
Text 1: The cat in the hat.
Text 2: The dog in the hat.
```

This can be simplified down to:

```
Text 1: cat
Text 2: dog
```

Locating these common substrings can be done in O(log n) using a binary search. Since binary searches are least efficient at their extreme points and it is not uncommon in the real-world to have zero commonality, it makes sense to do a quick check of the first (or last) character before starting the search.

(This section generates a lot of email. The issue is that string equality operations (a == b) are typically O(n), thus the described algorithm would be O(n log n). However, when dealing with high level languages, the speed difference between loops and equality operations is such that for all practical purposes the equality opeation can be considered to be O(1). Further complicating the matter are languages like Python which use hash tables for all strings, thus making equality checking be O(1) and string creation be O(n). For more information, see the performance testing.)

Sample Code

view plain print ?

```
01.  def diff_commonPrefix(text1, text2):
02.    # Quick check for common null cases.
03.    if not text1 or not text2 or text1[0] != text2[0]:
04.      return 0
05.    # Binary search.
06.    pointermin = 0
07.    pointermax = min(len(text1), len(text2))
08.    pointermid = pointermax
09.    pointerstart = 0
10.    while pointermin < pointermid:
11.      if text1[pointerstart:pointermid] == text2[pointerstart:pointermid]:
12.        pointermin = pointermid
13.        pointerstart = pointermin
14.      else:
15.        pointermax = pointermid
16.      pointermid = int((pointermax - pointermin) / 2 + pointermin)
17.    return pointermid
18.
19.  def diff_commonSuffix(text1, text2):
20.    # Quick check for common null cases.
21.    if not text1 or not text2 or text1[-1] != text2[-1]:
22.      return 0
23.    # Binary search.
24.    pointermin = 0
25.    pointermax = min(len(text1), len(text2))
26.    pointermid = pointermax
27.    pointerend = 0
28.    while pointermin < pointermid:
29.      if (text1[-pointermid:len(text1) - pointerend] ==
30.          text2[-pointermid:len(text2) - pointerend]):
31.        pointermin = pointermid
32.        pointerend = pointermin
33.      else:
34.        pointermax = pointermid
35.      pointermid = int((pointermax - pointermin) / 2 + pointermin)
36.    return pointermid
```

Show JavaScript Show Java Hide Python

The GNU diff program (which does linear matching for prefixes and suffixes) claims in their documentation that "occasionally [prefix & suffix stripping] may produce non-minimal output", though they do not provide an example of this.

1.3 Singular Insertion/Deletion

A very common difference is the insertion or the deletion of some text:

```
Text 1: The cat in the hat.            | Text 1: The cat in the hat.
Text 2: The furry cat in the hat.      | Text 2: The cat.
```

After removing the common prefixes and suffixes one gets:

```
Text 1:                                | Text 1:  in the hat
Text 2: furry                          | Text 2:
```

The presence of an empty 'Text 1' in the first example indicates that 'Text 2' is an insertion. The presence of an empty 'Text 2' in the second example indicates that 'Text 1' is a deletion. Detecting these common cases avoids the need to run a difference algorithm at all.

Sample Code

```
view plain  print  ?
01.  if not text1:
02.    # Just add some text.
03.    return [(DIFF_INSERT, text2)]
04.  if not text2:
05.    # Just delete some text.
06.    return [(DIFF_DELETE, text1)]
```

In this and subsequent examples, the internal format for representing a set of differences is a list of tuples. The first element of each tuple specifies if it is an insertion (DIFF_INSERT), a ~~deletion~~ (DIFF_DELETE) or an equality (DIFF_EQUAL). The second element specifies the affected text.

Show JavaScript Show Java Hide Python

1.4 Two Edits

Detecting and dealing with two edits is more challenging than singular edits. Two simple insertions can be detected by looking for the presence of 'Text 1' within 'Text 2'. Likewise two simple deletions can be detected by looking for the presence of 'Text 2' in 'Text 1'.

```
Text 1: The cat in the hat.
Text 2: The happy cat in the black hat.
```

Removing the common prefixes and suffixes as a first step guarantees that there must be differences at each end of the remaining texts. It is then easy to determine that the shorter string ("cat in the") is present within the longer string ("happy cat in the black"). In these situations the difference may be determined without running a difference algorithm.

Sample Code

```
view plain  print  ?
01.  if len(text1) > len(text2):
02.    (longtext, shorttext) = (text1, text2)
03.  else:
04.    (shorttext, longtext) = (text1, text2)
05.  i = longtext.find(shorttext)
06.  if i != -1:
07.    # Shorter text is inside the longer text.
08.    diffs = [(DIFF_INSERT, longtext[:i]), (DIFF_EQUAL, shorttext),
09.             (DIFF_INSERT, longtext[i + len(shorttext):])]
10.    # Swap insertions for deletions if diff is reversed.
11.    if len(text1) > len(text2):
12.      diffs[0] = (DIFF_DELETE, diffs[0][1])
13.      diffs[2] = (DIFF_DELETE, diffs[2][1])
14.    return diffs
```

Show JavaScript Show Java Hide Python

The situation is more complicated if the edits aren't two simple insertions or two simple deletions. These cases may often be detected if the two edits are separated by considerable text:

```
Text 1: The cat in the hat.
Text 2: The ox in the box.
```

After removing the common prefixes and suffixes one gets:

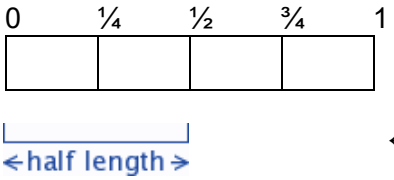
```
Text 1: cat in the hat
Text 2: ox in the box
```

If a substring exists in both texts which is at least half the length of the longer text, then it is guaranteed to be common. In this case the texts can be split in two, and separate differences carried out:

```
Text 1: cat     | Text 1: hat
Text 2: ox      | Text 2: box
```

Performing this test recursively may, in general, yield further subdivisions, although there are no such subdivisions in the above example.

Computing the longest common substring is an operation about as complex as computing the difference, which would mean there would be no savings. However, the limitation that the common substring must be at least half the length of the longer text provides a shortcut. As the diagram below illustrates, if a common substring of such a length exists, then the second quarter and/or third quarter of the longest string must form part of this substring.



← *Drag the 'half length' bar with your mouse.*

The smaller text can be searched for matches of these two quarters, and the context of any matches can be compared in both texts by looking for common prefixes and suffixes. The strings may be split at the location of the longest match which is equal to or greater than the half the length of the longer text. Due to the problem of repeated strings, all matches of each quarter in the smaller text must be checked, not just the first one which reaches the necessary length.

Sample Code

view plain print ?

```
01.  # Check to see if the problem can be split in two.
02.  hm = diff_halfMatch(text1, text2)
03.  if hm:
04.    # A half-match was found, sort out the return data.
05.    (text1_a, text1_b, text2_a, text2_b, mid_common) = hm
06.    # Send both pairs off for separate processing.
07.    diffs_a = diff_main(text1_a, text2_a)
08.    diffs_b = diff_main(text1_b, text2_b)
09.    # Merge the results.
10.    return diffs_a + [(DIFF_EQUAL, mid_common)] + diffs_b
11.
12.  def diff_halfMatch(text1, text2):
13.    # Do the two texts share a substring which is at least half the length of the
14.    # longer text?
15.    if len(text1) > len(text2):
16.      (longtext, shorttext) = (text1, text2)
17.    else:
18.      (shorttext, longtext) = (text1, text2)
19.    if len(longtext) < 10 or len(shorttext) < 1:
20.      return None  # Pointless.
21.
22.    def diff_halfMatchI(longtext, shorttext, i):
23.      seed = longtext[i:i + len(longtext) / 4]
24.      best_common = ''
25.      j = shorttext.find(seed)
26.      while j != -1:
27.        prefixLength = diff_commonPrefix(longtext[i:], shorttext[j:])
28.        suffixLength = diff_commonSuffix(longtext[:i], shorttext[:j])
29.        # print "%s|%s+%s|%s vs. %s|%s+%s|%s" %
30.        #     (my_suffix[0], my_suffix[2], my_prefix[2], my_prefix[0],
31.        #     my_suffix[1], my_suffix[2], my_prefix[2], my_prefix[1])
32.        if len(best_common) < suffixLength + prefixLength:
33.          best_common = (shorttext[j - suffixLength:j] +
34.              shorttext[j:j + prefixLength])
35.          best_longtext_a = longtext[:i - suffixLength]
36.          best_longtext_b = longtext[i + prefixLength:]
37.          best_shorttext_a = shorttext[:j - suffixLength]
38.          best_shorttext_b = shorttext[j + prefixLength:]
```

```
      j = shorttext.find(seed, j + 1)

    if len(best_common) >= len(longtext) / 2:
      return (best_longtext_a, best_longtext_b,
              best_shorttext_a, best_shorttext_b, best_common)
    else:
      return None

  # First check if the second quarter is the seed for a half-match.
  hm1 = diff_halfMatchI(longtext, shorttext, (len(longtext) + 3) / 4)
  # Check again based on the third quarter.
  hm2 = diff_halfMatchI(longtext, shorttext, (len(longtext) + 1) / 2)
  if not hm1 and not hm2:
    return None
  elif not hm2:
    hm = hm1
  elif not hm1:
    hm = hm2
  else:
    # Both matched.  Select the longest.
    if len(hm1[4]) > len(hm2[4]):
      hm = hm1
    else:
      hm = hm2

  # A half-match was found, sort out the return data.
  if len(text1) > len(text2):
    (text1_a, text1_b, text2_a, text2_b, mid_common) = hm
  else:
    (text2_a, text2_b, text1_a, text1_b, mid_common) = hm
  return (text1_a, text1_b, text2_a, text2_b, mid_common)
```

See earlier code for 'diff_commonPrefix' and 'diff_commonSuffix' functions.

Show JavaScript | Show Java | Hide Python

2 Difference Algorithms

Once the pre-processing optimisation is complete, the remaining text is compared with a difference algorithm. A brute-force technique would take $O(n_1*n_2)$ to execute (where n_1 and n_2 are the lengths of each input). Since this is clearly unscalable in practical applications where the text lengths are arbitrary, much research has been conducted on better algorithms which approach $O(n_1+n_2)$. However, these algorithms are not interchangeable. There are several criteria beyond speed which are important.

2.1 Input

There are three common modes for comparing input texts:

```
Text 1: The cat in the hat.
Text 2: The bird in the hand.
Char diff: The ~~cat~~bird in the ha~~t~~nd.
Word diff: The ~~cat~~bird in the ~~hat~~hand.
Line diff: ~~The cat in the hat.~~The bird in the hand.
```

Comparing by individual characters produces the finest level of detail but takes the longest to execute due to the larger number of tokens. Comparing by word boundaries or line breaks is faster and produces fewer individual edits, but the total length of the edits is larger. The required level of detail varies depending on the application. For instance comparing source code is generally done on a line-by-line basis, whereas comparing an English document is generally done on a word-by-word basis, and binary data or DNA sequences is generally done on a character-by-character basis.

Any difference algorithm could theoretically process any input, regardless of whether it is split by characters, words or lines. However, some difference algorithms are much more efficient at handling small tokens such as characters, others are much more efficient at handling large tokens such as lines. The

reason is that there are an infinite number of possible lines, and any line which does not appear in one text but appears in the other is automatically known to be an insertion or a deletion. Conversely, there are only 80 or so distinct tokens when processing characters (a-z, A-Z, 0-9 and some punctuation), which means that any non-trivial text will contain multiple instances of most if not all these characters. Different algorithms can exploit these statistical differences in the input texts, resulting in more efficient strategies. An algorithm which is specifically designed for line-by-line differences is described in J. Hunt and M. McIlroy's 1976 paper: An Algorithm for Differential File Comparison.

Another factor to consider is the availability of useful functions. Most computer languages have superior string handling facilities (such as regular expressions) when compared with array handling facilities. These more powerful string functions may make character-based difference algorithms easier to program. On the other hand, the advent of Unicode support in many languages means that strings may contain alphabet sizes as great as 65,536. This allows words or lines to be hashed down to a single character so that the difference algorithm can make use of strings instead of arrays. To put this in perspective, the King James Bible contains 30,833 unique lines and 28,880 unique 'words' (just space-delimited, with leading or trailing punctuation not separated).

2.2 Output

Traditional difference algorithms produce a list of insertions and deletions which when performed on the first text will result in the second text. An extension to this is the addition of a 'move' operator:

```
Text 1: The black cat in the hat?
Text 2: The cat in the black hat!
Ins & Del: The ~~black~~ cat in the black hat~~?~~!
...& Move: The ^cat in the black hat~~?~~!
```

When a large block of text has moved from one location to another, it is often more understandable to report this as a move rather than a deletion and an insertion. An algorithm which uses the 'move' operator is described in P. Heckel's 1978 paper: A technique for isolating differences between files.

An entirely different approach is to use 'copy' and 'insert' as operators:

```
Text 1: The black cat in the hat?
Text 2: The black hat on the black cat!
Copy & Ins: The black hat on the black cat!
```

This approach uses fragments from the first text, which are copied and pasted to form the second text. Much like clipping out words from a newspaper to compose a ransom note, except that the any clipped word may be photocopied and used multiple times. Any entirely new text is inserted verbatim. Copy/insert differences are generally not human-readable. However they are significantly faster to compute making them superior than insert/delete differences for delta compression applications. An algorithm which uses the 'copy' and 'insert' operators is described in J. MacDonald's 2000 paper: File System Support for Delta Compression.

2.3 Accuracy

No difference algorithm should ever return an incorrect output; that is, an output which does not describe a valid path of differences from one text to another. However, some algorithms may return sub-optimal outputs in the interests of speed. For instance, Heckel's algorithm (1978) is quick, but gets confused if repeated text exists in the inputs:

```
Text 1:  A X X X X B
Text 2:  C X X X X D
Optimal: ~~A~~C X X X X ~~B~~D
Heckel:  ~~A X X X X B~~C X X X X D
```

Another example of sacrificing accuracy for speed is to process the whole texts with a line-based algorithm, then reprocess each run of modified lines with a character-based algorithm. The problem with this multi-pass approach is that the line-based difference may sometimes identify inappropriate commonalities

between the two lines. Blank lines are a common cause of these since they may appear in two unrelated texts. These inappropriate commonalities serve to randomly split up edit blocks and prevent genuinely common text from being discovered during the character-based phase. A solution to this is to pass the line-based differences through a semantic cleanup algorithm (as described below in section 3.2) before performing the character-based differences. In cases involving multiple edits throughout a long document, performing a high-level difference followed by a low-level difference can result in an order of magnitude improvement in speed and memory requirements. However, there remains a risk that the resulting difference path may not be the shortest one possible.

Arguably the best general-purpose difference algorithm is described in E. Myers' 1986 paper: An O(ND) Difference Algorithm and Its Variations. One of the proposed optimisations is to process the difference from both ends simultaneously, meeting at the middle. In most cases this improves the performance by up to 50%.

---

3 Post-processing Cleanup

A perfect difference algorithm will report the minimum number of edits required to convert one text into the other. However, sometimes the result is too perfect:

```
Text 1: I am the very model of a modern major general.
Text 2: `Twas brillig, and the slithy toves did gyre and gimble in the wabe.
Diff:   -I+`Twas brillig, a-m+nd the -ver+slithy -mto-ve-l+s -of+did -a mod+gyre-rn -majo-r+nd gimble in the-r wa-l+be.
```

The first step when dealing with a new diff is to transpose and merge like sections. In the above example one such optimization is possible.

```
Old: -I+`Twas brillig, a-m+nd ...
New: -I+`Twas brillig, a-m+nd ...
```

Both diffs are identical in their output, but the second one has merged two operations into one by transposing a coincidentally repeated equality.

Sample Code

view plain print ?

```
def diff_cleanupMerge(diffs):
  # Reorder and merge like edit sections.  Merge equalities.
  # Any edit section can move as long as it does not cross an equality.
  diffs.append((DIFF_EQUAL, ''))  # Add a dummy entry at the end.
  pointer = 0
  count_delete = 0
  count_insert = 0
  text_delete = ''
  text_insert = ''
  while pointer < len(diffs):
    if diffs[pointer][0] == DIFF_INSERT:
      count_insert += 1
      text_insert += diffs[pointer][1]
      pointer += 1
    elif diffs[pointer][0] == DIFF_DELETE:
      count_delete += 1
      text_delete += diffs[pointer][1]
      pointer += 1
    elif diffs[pointer][0] == DIFF_EQUAL:
      # Upon reaching an equality, check for prior redundancies.
      if count_delete != 0 or count_insert != 0:
        if count_delete != 0 and count_insert != 0:
          # Factor out any common prefixies.
          commonlength = diff_commonPrefix(text_insert, text_delete)
          if commonlength != 0:
            x = pointer - count_delete - count_insert - 1
            if x >= 0 and diffs[x][0] == DIFF_EQUAL:
              diffs[x] = (diffs[x][0], diffs[x][1] +
                          text_insert[:commonlength])
            else:
```

```
              diffs.insert(0, (DIFF_EQUAL, text_insert[:commonlength]))
              pointer += 1
            text_insert = text_insert[commonlength:]
            text_delete = text_delete[commonlength:]
          # Factor out any common suffixies.
          commonlength = diff_commonSuffix(text_insert, text_delete)
          if commonlength != 0:
            diffs[pointer] = (diffs[pointer][0], text_insert[-commonlength:] +
                diffs[pointer][1])
            text_insert = text_insert[:-commonlength]
            text_delete = text_delete[:-commonlength]
        # Delete the offending records and add the merged ones.
        if count_delete == 0:
          diffs[pointer - count_insert : pointer] = [
              (DIFF_INSERT, text_insert)]
        elif count_insert == 0:
          diffs[pointer - count_delete : pointer] = [
              (DIFF_DELETE, text_delete)]
        else:
          diffs[pointer - count_delete - count_insert : pointer] = [
              (DIFF_DELETE, text_delete),
              (DIFF_INSERT, text_insert)]
        pointer = pointer - count_delete - count_insert + 1
        if count_delete != 0:
          pointer += 1
        if count_insert != 0:
          pointer += 1
      elif pointer != 0 and diffs[pointer - 1][0] == DIFF_EQUAL:
        # Merge this equality with the previous one.
        diffs[pointer - 1] = (diffs[pointer - 1][0],
                              diffs[pointer - 1][1] + diffs[pointer][1])
        del diffs[pointer]
      else:
        pointer += 1

      count_insert = 0
      count_delete = 0
      text_delete = ''
      text_insert = ''

  if diffs[-1][1] == '':
    diffs.pop()  # Remove the dummy entry at the end.

  # Second pass: look for single edits surrounded on both sides by equalities
  # which can be shifted sideways to eliminate an equality.
  # e.g: A<ins>BA</ins>C -> <ins>AB</ins>AC
  changes = False
  pointer = 1
  # Intentionally ignore the first and last element (don't need checking).
  while pointer < len(diffs) - 1:
    if (diffs[pointer - 1][0] == DIFF_EQUAL and
        diffs[pointer + 1][0] == DIFF_EQUAL):
      # This is a single edit surrounded by equalities.
      if diffs[pointer][1].endswith(diffs[pointer - 1][1]):
        # Shift the edit over the previous equality.
        diffs[pointer] = (diffs[pointer][0],
            diffs[pointer - 1][1] +
            diffs[pointer][1][:-len(diffs[pointer - 1][1])])
        diffs[pointer + 1] = (diffs[pointer + 1][0],
                              diffs[pointer - 1][1] + diffs[pointer + 1][1])
        del diffs[pointer - 1]
        changes = True
      elif diffs[pointer][1].startswith(diffs[pointer + 1][1]):
        # Shift the edit over the next equality.
        diffs[pointer - 1] = (diffs[pointer - 1][0],
                              diffs[pointer - 1][1] + diffs[pointer + 1][1])
        diffs[pointer] = (diffs[pointer][0],
            diffs[pointer][1][len(diffs[pointer + 1][1]):] +
            diffs[pointer + 1][1])
        del diffs[pointer + 1]
        changes = True
    pointer += 1

  # If shifts were made, the diff needs reordering and another shift sweep.
  if changes:
    diff_cleanupMerge(diffs)
```

Show JavaScript | Show Java | Hide Python

Transposition helps a little bit and is completely safe, but the larger problem is that differences between two dissimilar texts are frequently littered with small coincidental equalities called 'chaff'. The expected result above might be to delete all of 'Text 1' and insert all of 'Text 2', with the possible exception of the period at the end. However most algorithms will salvage bits and pieces, resulting in a mess.

This problem is most apparent in character-based differences since the small set of alphanumeric characters ensures commonalities. A word-based difference of the above example would be distinctly better, but would have inappropriately salvaged " the ". Longer texts would result in more shared words. A line-based difference of the above example would be ideal. However, even line-based differences are vulnerable to inappropriately salvaging blank lines and other common lines (such as "} else {" in source code).

The problem of chaff is actually one of two different problems: efficiency or semantics. Each of these problems requires a different solution.

3.1 Efficiency

If the output of the difference is designed for computer use (such as delta compression or input to a patch program) then depending on the subsequent application or storage method, each edit operation may have some fixed computational overhead associated with it in addition to the number of characters within that edit. For instance, fifty single-character edits might take more storage or take longer for the next application to process than a single fifty-character edit. Once the trade-off has been measured, the computational or storage cost of an edit operation may be stated in terms of the equivalent cost of characters of change. If this cost is zero, then there is no overhead. If this cost is (for example) ten characters, then increasing the total number of characters edited by up to nine, while reducing the number of edit operations by one, would result in a net savings. Thus the total cost of a difference can be computed as `o * c + n` where `o` is the number of edit operations, `c` is the constant cost of each edit operation in terms of characters, and `n` is the total number of characters changed. Below are three examples (with `c` set arbitrarily at 4) showing how increasing the number of edited characters can reduce the number of edit operations and reduce the overall cost of the difference.

First, any equality (text which remains unchanged) which is surrounded on both sides by an existing insertion and deletion need be less than `c` characters long for it to be advantageous to split it.

```
                    Operations   Characters   Cost
Text 1: ABXYZCD
Text 2: 12XYZ34
Diff:   ~~AB~~12XYZ~~CD~~34         4 * 4          + 8   = 24
Split:  ~~AB~~12~~XYZ~~XYZ~~CD~~34      6 * 4         + 14   = 38
Merge:  ~~ABXYZCD~~12XYZ34      2 * 4         + 14   = 22
```

Secondly, any equality which is surrounded on one side by an existing insertion and deletion, and on the other side by an existing insertion *or* deletion, need be less than half `c` characters long for it to be advantageous to split it.

```
                Operations   Characters   Cost
Text 1: XCD
Text 2: 12X34
Diff:   12X~~CD~~34       3 * 4          + 6   = 18
Split:  12~~X~~X~~CD~~34      5 * 4          + 8   = 28
Merge:  ~~XCD~~12X34      2 * 4          + 8   = 16
```

Both of these conditions may be computed quickly by making a single pass through the data, backtracking to reevaluate the previous equality if a split has changed the type of edits surrounding it. Another pass is made to reorder the edit operations and merge like ones together.

Sample Code

view plain print ?

```
def diff_cleanupEfficiency(diffs):
  # Reduce the number of edits by eliminating operationally trivial equalities.
  changes = False
  equalities = []  # Stack of indices where equalities are found.
  lastequality = ''  # Always equal to equalities[len(equalities)-1][1]
  pointer = 0  # Index of current position.
  pre_ins = False  # Is there an insertion operation before the last equality.
  pre_del = False  # Is there a deletion operation before the last equality.
  post_ins = False  # Is there an insertion operation after the last equality.
  post_del = False  # Is there a deletion operation after the last equality.
  while pointer < len(diffs):
    if diffs[pointer][0] == DIFF_EQUAL:  # equality found
      if (len(diffs[pointer][1]) < Diff_EditCost and
          (post_ins or post_del)):
        # Candidate found.
        equalities.append(pointer)
        pre_ins = post_ins
        pre_del = post_del
        lastequality = diffs[pointer][1]
      else:
        # Not a candidate, and can never become one.
        equalities = []
        lastequality = ''

      post_ins = post_del = False
    else:  # an insertion or deletion
      if diffs[pointer][0] == DIFF_DELETE:
        post_del = True
      else:
        post_ins = True

      # Five types to be split:
      # <ins>A</ins><del>B</del>XY<ins>C</ins><del>D</del>
      # <ins>A</ins>X<ins>C</ins><del>D</del>
      # <ins>A</ins><del>B</del>X<ins>C</ins>
      # <ins>A</del>X<ins>C</ins><del>D</del>
      # <ins>A</ins><del>B</del>X<del>C</del>

      if lastequality and ((pre_ins and pre_del and post_ins and post_del) or
                           ((len(lastequality) < Diff_EditCost / 2) and
                            (pre_ins + pre_del + post_ins + post_del) == 3)):
        # Duplicate record
        diffs.insert(equalities[len(equalities) - 1],
                     (DIFF_DELETE, lastequality))
        # Change second copy to insert.
        diffs[equalities[len(equalities) - 1] + 1] = (DIFF_INSERT,
            diffs[equalities[len(equalities) - 1] + 1][1])
        equalities.pop()  # Throw away the equality we just deleted
        lastequality = ''
        if pre_ins and pre_del:
          # No changes made which could affect previous entry, keep going.
          post_ins = post_del = True
          equalities = []
        else:
          if len(equalities):
            equalities.pop()  # Throw away the previous equality
          if len(equalities):
            pointer = equalities[len(equalities) - 1]
          else:
            pointer = -1
          post_ins = post_del = False
        changes = True
    pointer += 1

  if changes:
    diff_cleanupMerge(diffs)
```

See earlier code for 'diff_cleanupMerge' function.

Show JavaScript | Show Java | Hide Python

Although this is a good start, it is not a complete solution since it does not catch a third type of condition:

```
                  Operations   Characters   Cost
Text 1: ABCD
Text 2: A1B2C3D4
Diff:   A1B2C3D4         4 * 4          + 4   = 20
Split:  A1BB2CC3DD4     10 * 4         + 10   = 50
Merge:  ABCD1B2C3D4      2 * 4         + 10   = 18
```

In this and similar cases, each individual split would result in a higher total cost, yet these splits, when combined, result in a lower total cost. Computing this form of optimisation appears to be an $O(n^2)$ operation on selected regions of the difference (as opposed to the O(n) optimisation for the first two cases), thus it may be more costly than the savings themselves.

3.2 Semantics

3.2.1 Semantic Chaff

If the output of the difference is designed for human use (such as a visual display), the problem changes. In this case the goal is to provide more meaningful divisions. Consider these two examples:

```
Text 1: Quicq fyre      | Text 1: Slow fool
Text 2: Quick fire      | Text 2: Quick fire
Diff:   Quicqk fyire    | Diff:   SlowQuick foolire
Split:  Quicqk f fyire  | Split:  SlowQuick f foolire
Merge:  Quicq fyk fire  | Merge:  Slow foolQuick fire
```

Mathematically, these examples are very similar. They have the same central equality (" f") and they have the same number of edit operations. Yet the first example (which involves correcting two typographical errors) is more meaningful in its raw diff stage, rather than after splitting and merging the equality. Whereas the second example (which involves larger edits) has little meaning at its raw diff stage, and is much clearer after splitting and merging the equality. The primary distinction between these two examples is the amount of change surrounding the equality.

One solution for removing semantic chaff is to pass over the data looking for equalities that are smaller than or equal to the insertions and deletions on both sides of them. When such an equality is found, it is split into a deletion and an addition. Then a second pass is made to reorder and merge all deletions and additions which aren't separated by surviving equalities. Below is a somewhat contrived example showing the these steps:

```
Text 1:  Hovering
Text 2:  My government
Diff:    HMy governgment
Split 1: HMy governgment
Split 2: HMy goverovergment
Merge:   HoveringMy government
```

In this case "over" is four letters long, compared with only five and one letters of changes surrounding it, so it is left. However, "n" is only one letter, compared with one and five letters of changes surrounding it. Therefore "n" is split. Once an equality is split, the pass must backtrack to reevaluate the previous equality since its context has changed. In this case "over" is now surrounded by five and eight letters of changes, so it too is split. Finally all the pieces are collected together, resulting in an easily understandable difference.

Sample Code

view plain print ?

```
01. def diff_cleanupSemantic(diffs):
02.   # Reduce the number of edits by eliminating semantically trivial equalities.
03.   changes = False
04.   equalities = []  # Stack of indices where equalities are found.
```

```
  lastequality = None  # Always equal to equalities[-1][1]
  pointer = 0  # Index of current position.
  length_changes1 = 0  # Number of chars that changed prior to the equality.
  length_changes2 = 0  # Number of chars that changed after the equality.
  while pointer < len(diffs):
    if diffs[pointer][0] == DIFF_EQUAL:  # equality found
      equalities.append(pointer)
      length_changes1 = length_changes2
      length_changes2 = 0
      lastequality = diffs[pointer][1]
    else:  # an insertion or deletion
      length_changes2 += len(diffs[pointer][1])
      if (lastequality != None and (len(lastequality) <= length_changes1) and
          (len(lastequality) <= length_changes2)):
        # Duplicate record
        diffs.insert(equalities[-1], (DIFF_DELETE, lastequality))
        # Change second copy to insert.
        diffs[equalities[-1] + 1] = (DIFF_INSERT,
            diffs[equalities[-1] + 1][1])
        # Throw away the equality we just deleted.
        equalities.pop()
        # Throw away the previous equality (it needs to be reevaluated).
        if len(equalities) != 0:
          equalities.pop()
        if len(equalities):
          pointer = equalities[-1]
        else:
          pointer = -1
        length_changes1 = 0  # Reset the counters.
        length_changes2 = 0
        lastequality = None
        changes = True
    pointer += 1

  if changes:
    diff_cleanupMerge(diffs)
```

See earlier code for 'diff_cleanupMerge' function.

Show JavaScript | Show Java | Hide Python

This solution is not perfect. It has tunnel vision; it is unable to see beyond the immediate neighbourhood of each equality it evaluates. This can result in small groups of chaff surviving:

```
Text 1: It was a dark and stormy night.
Text 2: The black can in the cupboard.
Diff:   ItThe wblas a darck cand sin the cupboarmy nightd.
Split:  ItThe- wblaas a darck cand- sin tthe cupbooarrmy nightd.
Merge:  It was a darThe black cand stormy night in the cupboard.
```

A more comprehensive solution might compute a weighted average of differences further away from the equality in question.

3.2.2 Semantic Allignment

A separate issue with creating semantically meaningful diffs is aligning edit boundaries to logical divisions. Consider the following diffs:

```
Text 1: That cartoon.
Text 2: That cat cartoon.
Diff 1: That cat cartoon.
Diff 2: That cat cartoon.
Diff 3: That cat cartoon.
Diff 4: That cat cartoon.
Diff 5: That cat cartoon.
Diff 6: That cat cartoon.
```

All six diffs are valid and minimal. Diffs 1 and 6 are the ones most likely to be returned by diff algorithms. But diffs 3 and 4 are more likely to capture the semantic meaning of the diff.

The solution is to locate each insertion or deletion which is surrounded on both sides by equalities, and attempt to slide them sideways. If the last token of the preceeding equality equals the last token of the edit, then the edit may be slid left. Likewise if the first token of the edit equals the first token of the following equality, then the edit may be slit right. Each of the possible locations can be scored based on whether the boundaries appear to be logical. One scheme which works is:

- One point if a boundary is adjacent to a non-alphanumeric character.
- Two points if a boundary is adjacent to whitespace.
- Three points if a boundary is adjacent to a line break.
- Four points if a boundary is adjacent to a blank line.
- Five points if a boundary has consumed the entire equality.

This scheme would give scores of zero to diffs 1, 2, 5 and 6, while giving scores of four to diffs 3 and 4.

Sample Code

view plain print ?

```
def diff_cleanupSemanticLossless(diffs):
  # Look for single edits surrounded on both sides by equalities
  # which can be shifted sideways to align the edit to a word boundary.
  # e.g: The c<ins>at c</ins>ame. -> The <ins>cat </ins>came.

  def diff_cleanupSemanticScore(one, two):
    # Given two strings, compute a score representing whether the
    # internal boundary falls on logical boundaries.
    # Scores range from 5 (best) to 0 (worst).
    # Closure, but does not reference any external variables.

    if not one or not two:
      # Edges are the best.
      return 5

    # Each port of this function behaves slightly differently due to
    # subtle differences in each language's definition of things like
    # 'whitespace'.  Since this function's purpose is largely cosmetic,
    # the choice has been made to use each language's native features
    # rather than force total conformity.
    score = 0
    # One point for non-alphanumeric.
    if not one[-1].isalnum() or not two[0].isalnum():
      score += 1
      # Two points for whitespace.
      if one[-1].isspace() or two[0].isspace():
        score += 1
        # Three points for line breaks.
        if (one[-1] == "\r" or one[-1] == "\n" or
            two[0] == "\r" or two[0] == "\n"):
          score += 1
          # Four points for blank lines.
          if (re.search("\\n\\r?\\n$", one) or
              re.match("^\\r?\\n\\r?\\n", two)):
            score += 1
    return score

  pointer = 1
  # Intentionally ignore the first and last element (don't need checking).
  while pointer < len(diffs) - 1:
    if (diffs[pointer - 1][0] == DIFF_EQUAL and
        diffs[pointer + 1][0] == DIFF_EQUAL):
      # This is a single edit surrounded by equalities.
      equality1 = diffs[pointer - 1][1]
      edit = diffs[pointer][1]
      equality2 = diffs[pointer + 1][1]

      # First, shift the edit as far left as possible.
      commonOffset = diff_commonSuffix(equality1, edit)
      if commonOffset:
        commonString = edit[-commonOffset:]
```

```
          equality1 = equality1[:-commonOffset]
          edit = commonString + edit[:-commonOffset]
          equality2 = commonString + equality2

        # Second, step character by character right, looking for the best fit.
        bestEquality1 = equality1
        bestEdit = edit
        bestEquality2 = equality2
        bestScore = (diff_cleanupSemanticScore(equality1, edit) +
            diff_cleanupSemanticScore(edit, equality2))
        while edit and equality2 and edit[0] == equality2[0]:
          equality1 += edit[0]
          edit = edit[1:] + equality2[0]
          equality2 = equality2[1:]
          score = (diff_cleanupSemanticScore(equality1, edit) +
              diff_cleanupSemanticScore(edit, equality2))
          # The >= encourages trailing rather than leading whitespace on edits.
          if score >= bestScore:
            bestScore = score
            bestEquality1 = equality1
            bestEdit = edit
            bestEquality2 = equality2

        if diffs[pointer - 1][1] != bestEquality1:
          # We have an improvement, save it back to the diff.
          if bestEquality1:
            diffs[pointer - 1] = (diffs[pointer - 1][0], bestEquality1)
          else:
            del diffs[pointer - 1]
            pointer -= 1
          diffs[pointer] = (diffs[pointer][0], bestEdit)
          if bestEquality2:
            diffs[pointer + 1] = (diffs[pointer + 1][0], bestEquality2)
          else:
            del diffs[pointer + 1]
            pointer -= 1
      pointer += 1
```

Show JavaScript | Show Java | Hide Python

See an implementation and online demonstration of diff.
See also the companion paper on diff's counterpart: Patch