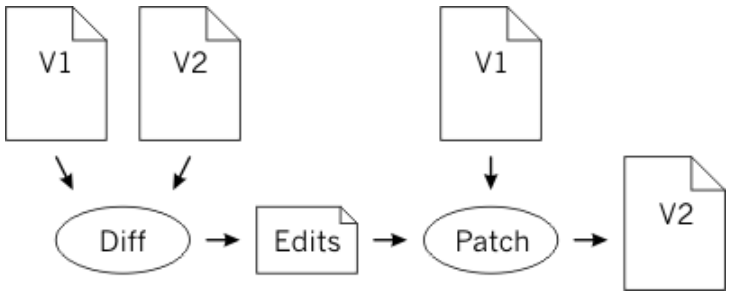
Fuzzy Patch

by Neil Fraser, May 2006

Patch is the process of modifying a document based on a set of edits created by a difference algorithm. A typical use case would be a developer who creates a new version of a document, uses *diff* to create a set of edits, then transmits those edits to a customer, who then applies them to their version of the document, thus recreating the new version.



A strict patch program is a trivial piece of software. It would just execute a scripted set of insertions and deletions on one version of the document to produce the other version. Assuming that both copies of the base document (in the above case labeled 'V1') are identical, then both copies of the second document (in the above case labeled 'V2') will also be identical. However, the program becomes more complicated when the two copies of the base document are not identical. In these cases the patch program must do the best job it can in applying the edits to the correct locations. This paper examines one method for building such a patch program.

GNU patch

The de facto standard for patch is GNU's implementation. The following example shows how edits for GNU's patch are structured. The first line contains line number information describing where the following edit should be performed. The edit is preceded and followed by three lines of context. The edit in this case involves the deletion of two lines and the insertion of three lines.

```
@@ -382,8 +481,9 @@
       function maxbits(){
          // Compute the number of bits in the highest int available.
          // i: maxint; b: maxbits
-         var b=1;
-         for(var i=2; (i+1)!=i; i*=2)
+         // No system will have less than 8 bits
+         var b=8;
+         for(var i=256; (i+1)!=i; i*=2)
            b++;
          return b;
       }
```

GNU's patch uses a fairly primitive matching system for inexact patches. Its documentation states:

> "[...] patch can detect when the line numbers mentioned in the patch are incorrect, and attempts to find the correct place to apply each hunk of the patch. As a first guess, it takes the line number mentioned for the hunk, plus or minus any offset used in applying the previous hunk. If that is not the correct place, patch scans both forwards and backwards for a set of lines matching the context given in the hunk. First patch looks for a place where all lines of the context match. If no such place is found, [...] then another scan takes place ignoring the first and last line of context. If that fails, [...] the first two and last two lines of context are ignored, and another scan is made."

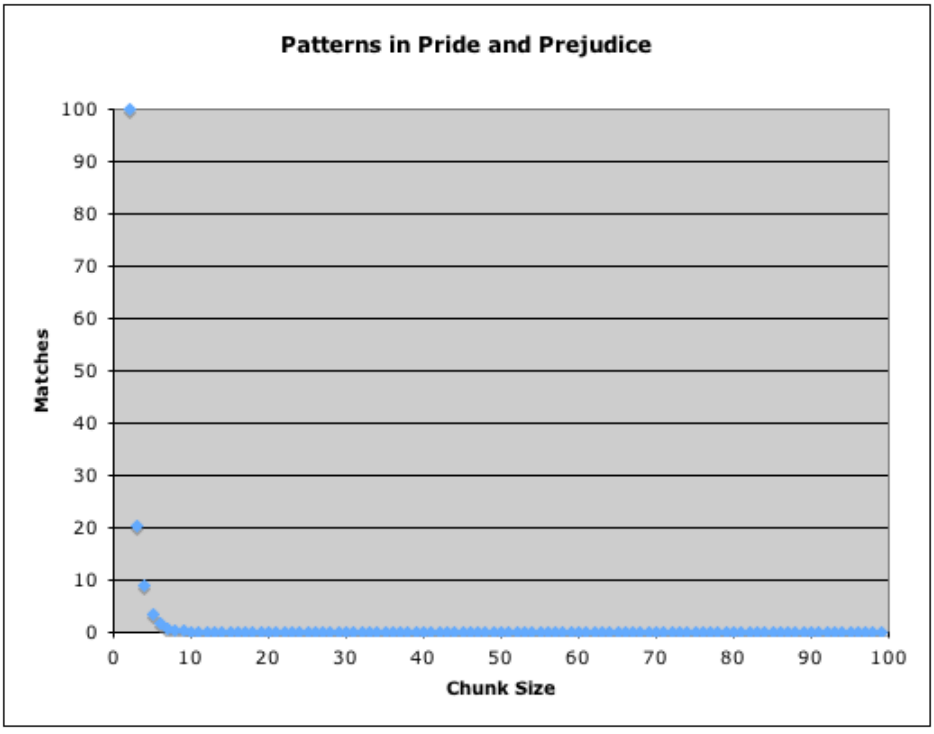
This simple strategy is relatively fragile. Minor changes between the two base documents can result in failure. In the above example if the function name had been changed from "maxbits" to "max_bits" the patch would have failed to match on the first pass, but would have matched on the second pass (once the first line of the prefix context was ignored). However, if the "b++;" had been changed to "b=b+1;", the patch would fail, despite the fact that all the other context lines match.

This simple strategy is also unrestricted by location. If there are two similar pieces of text, and the correct piece has been changed (or removed) causing it to fail to match, GNU's patch will happily target the other piece even if it is at the opposite end of the file.

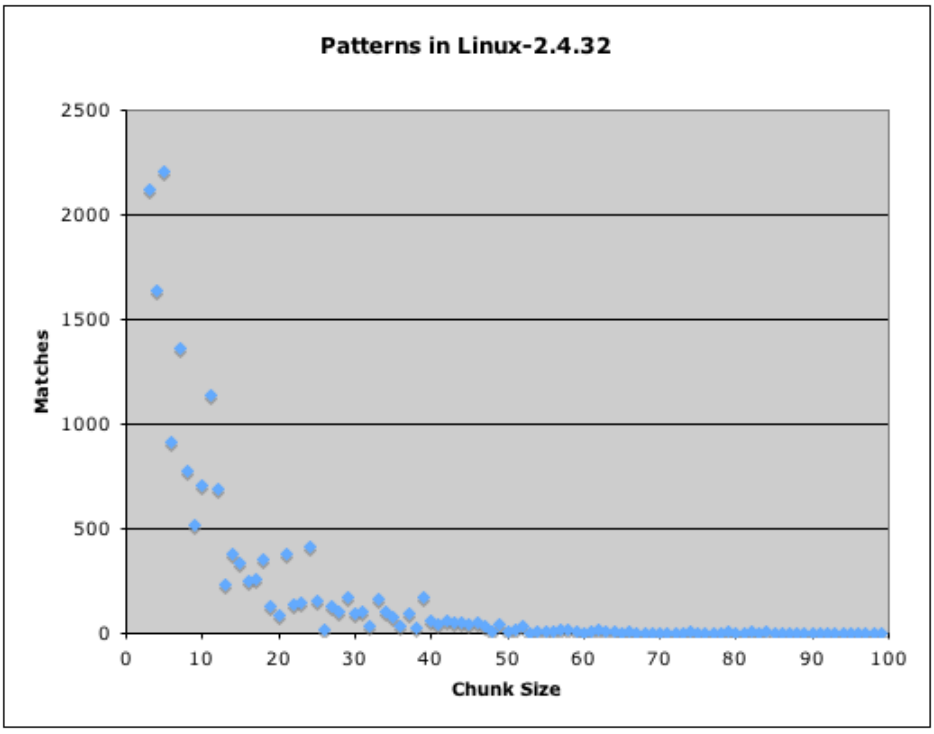
A more sophisticated approach would increase the number of successful matches (thereby avoiding unnecessary human intervention) and decrease the number of failed matches (thereby avoiding errors). A more sophisticated approach would be character-based instead of line-based (thereby allowing multiple edits to be merged onto the same line).

Context Size

GNU's patch expects three lines of context preceding and following each edit block. If this is insufficient context, the edit might be applied to the wrong place. If there is too much context, edits may fail due to unrelated changes which lie considerable distance from the target spot. Excessive context also increases the processing time for locating the best match; depending on the matching algorithm, potentially exponentially. Achieving the right balance is a challenge due to the nature of different texts.



English text follows a very clean and sharp exponential decline in matches as the pattern length increases. From the above analysis of Jane Austin's *Pride and Prejudice*, a fifteen-character pattern has less than a 2% chance of occurring twice in the same chapter.



By contrast, computer code contains far more repetitions. From the above analysis of the Linux Kernel's source code, a fifteen-character pattern is likely to occur 250 times in the same file. Interestingly, the overwhelming majority of these repetitions are not caused by code itself, but instead by decorative ASCII art. In particular, horizontal lines formed from repeated characters intended to separate one code block from another. The worst offender is the 1.7MB `include/asm-ia64/sn/sn2/shub_mmr.h` which includes 1644 identical copies of this line:

```
/* ===================================================================== */
```

Analysis Method

Both of these analyses selected a random chunk of characters from across the entire work (with longer chapters or files more likely to be selected than shorter ones). The originating chapter or file was then scanned for occurrences of the chunk, not counting the original location. This was repeated one thousand times for each chunk size, and the results averaged.

```
#!/usr/bin/python
# Scans a tarfile for frequency of repeated patterns.
# Usage:  ./pattern.py filename
# Expects 'filename.tar' to exist (used as an index) as well as a
# directory 'filename/' which contains all the expanded files.

import os, sys, random
from stat import *

projectname = sys.argv[1]
tarsize = os.stat(projectname+".tar")[ST_SIZE]
print 'Reading %s (%i bytes)' % (projectname+".tar", tarsize)
tarhandle = file(projectname+".tar", 'r')
tardata = tarhandle.read()
tarhandle.close()

def getfilename(tardata):
  # Jump to a random spot in the tar file.  Then backtrack to the filename.
  while 1:
    position = random.randint(0, len(tardata)-1)
    char = tardata[position]
    if (char < ' ' or '~' < char) and char != '\t' and char != '\n' and char != '\r':
      pass # Bad character
    else:
      position = tardata.rfind(projectname+"/", 0, position)
```

```
      if position != -1:
        filename = ''
        while ' ' <= tardata[position] and tardata[position] <= '~':
           filename = filename + tardata[position]
           position = position + 1
        # Ignore directory definitions
        if filename[len(filename)-1] != '/':
          return filename

# Start with two-character chunks, and grow.
print "Size\tAverage    \tMax    \tFile"
for chunksize in range(2, 100):
  score = 0
  verb = ''
  maxscore = 0
  minscore = -1
  # Do the test 1000 times then take the average.
  for i in range(1000):
    # Open a random file (weighted by size)
    filename = getfilename(tardata)
    filehandle = file(filename, 'r')
    filedata = filehandle.read()
    filehandle.close()

    if len(filedata) > chunksize:
      pointer = random.randint(0, len(filedata)-chunksize-1)
      chunk = filedata[pointer:pointer+chunksize]
      hits = -1 # Don't count the seed
      pointer = 0
      # Find out how many other instances of chunk are present
      while pointer < len(filedata):
        pointer = filedata.find(chunk, pointer)
        if pointer == -1:
          pointer = len(filedata)
        else:
          pointer = pointer + 1
          hits = hits + 1
      score = score + hits
      if maxscore < hits:
        maxscore = hits
        verb = filename
      if minscore == -1 or minscore > hits:
        minscore = hits
    else:
      # no ambiguity possible in this file at this chunksize
      minscore = 0;
  score = score / 1000.0
  print "%d\t%f\t%d\t%s" % (chunksize, score, maxscore, verb)

print 'Done.'
```

Hide Code

Based on the above, it would appear that not only is there no universally ideal context length, but there isn't even an ideal context length for each type of text. Therefore the context length must be tailored to each edit block. A good algorithm might be to start no context on either side, then increase both sides by four characters as many times as required until the block is unique to the particular file, and finish by adding another four characters to each side. Thus the minimum amount of context would be a total of eight, and the maximum would be the length of the file (assuming a file composed entirely of a single repeated character).

Levenshtein Distance

Instead of progressively truncating the context strings and seeking the nearest match (as GNU's patch does), it would be more forgiving and more accurate to look for the overall similarities between the entire set of strings (prefix context, deletions and suffix context) and the available base document. One simple

measure of similarity between two texts is the Levenshtein distance; the number of insertions, deletions and substitutions required to turn one text into another. The smaller this number, the better the match.

Much research has been done on algorithms which can calculate Levenshtein distances. One of the most efficient is the Bitap algorithm described in S. Wu and U. Manber's 1991 paper: Fast Text Searching With Errors. This algorithm searches for exact matches between a text and a pattern, then it searches for matches with one error, then with two errors, and so on.

Apples and Oranges

Several potential matches may be available as a result of the Levenshtein analysis. Some may be exact, others may be filled with errors. Some may be close to the expected area, others may be far away. These two dimensions of accuracy cause problems when trying to compare and select the best match. A formula is required to compute an overall score for each match based on the errors and the distance.

The exact formula is important to the behaviour of the patch algorithm, but it is not important to its operation. The following formula is proposed as a starting point for computing a score for each match:

```
s = (e / p) / c + (d / t) / (1 - c)

s -- overall score
e -- number of errors (the Levenshtein Distance)
p -- the length of the pattern
d -- the distance (in characters) that the match is offset from the expected location
t -- the length of the text
c -- multiplier to change the relative weighting of accuracy vs. distance (a constant)
```

In broad strokes this formula simply adds the number of errors in the match to the distance which the match has been shifted. The lower the result, the better the match. The constant 'c' determines whether to favour poor matches near the expected location as opposed to more accurate matches further afield. This allows the performance of patch to be tuned if it is performing poorly. Another variable which may be tuned is 't' which may be coerced to a predefined minimum for small files (where a distance of a few characters might otherwise represent a significant percentage of the file length) or coerced to a predefined maximum for large files (where a small percentage distance might otherwise represent an unreasonable distance). Evaluating the distance requires examining it from both a percentage and from an absolute perspective.

The tradeoff of favouring match location versus the match errors can be visualised using a four dimensional graph. The width and depth represent the two dimensions of accuracy, each within a range of [0..1]. The height represents the computed score, also within a range of [0..1]. As 'c' traverses between 0 and 1, the score for any given match becomes less influenced by one form of accuracy and more influenced by the other. Hover the mouse over the graph to the right to see this traversal.

Implementing this formula as a separate function allows the behaviour of patch to be tweaked and adjusted from one location. This function can control not only matching, but also cut-off points for stopping or restricting the matching algorithm. This separation works as long as the return score is deterministic and there is never a *reverse* correlation between score and distance or errors.

Inverse Pyramid

A simplistic matching algorithm would scan the whole text for the exact pattern, then scan for one error, then two, and so on until the number of errors equals one less than the length of the pattern. This results in

a rectangular grid containing possible matches. The scores for each match may be computed, and the best one returned.

```
  'Twas brillig, and the slithy toves¶Did gyre and gimble in the wabe.
0 00000000000000000001000000000000000000000000000000000000100000000
1 00000000000000000011100000100000000000000000000000000011100000000
2 00000000000000000111110001110011110000000111000000001110011111001110
3 11111111111111111111111111111111111111111111111111111111111111111111
```

In the above example the given string is being searched for the pattern "the". There are two matches with zero errors, seven matches with one error and twenty-six matches with two errors. Everything is a match at three errors (the length of the pattern), so there is no benefit to calculating this or subsequent lines. As a side note, the Bitap algorithm in this and subsequent examples is processed from right to left, so that the '1' signifying a match occurs at the start of the match.

This rectangular grid is inefficient if an an expected location is known. As the number of errors increases, the area diminishes in which a match can reside with a higher score than the best one found to date. In the example below, "was" is being sought near character 25.

```
  'Twas brillig, and the slithy toves¶Did gyre and gimble in the wabe.
0 --1000000000000000000000000000000000000000000000000000000000------
1 --------------000000000000000000000000----------------------------
2 ------------------------0000--------------------------------------
```

The termination points for each line can be calculated by solving the distance in the score formula for a given error level and a given score. However, if the score formula is a stand-alone function (where it may easily be adjusted), it may be easier to run a binary search for each termination point. Furthermore, when a match is found while scanning a line and the location is after the expected location, then no further scanning on that line is required since the score of further matches would always be lower. Alternately, when a match is found while scanning a line and the location is before the expected location, then the termination point can be advanced to the current distance mirrored on the other side of the expected location.

Locations

The match algorithm will locate the approximate start of the match (of if processed from left to right, the end of the match). By adding (or subtracting) the length of the pattern, the approximate start (or end) of the match can also be found. Unfortunately unless the match itself was exact, neither of these end points is likely to be exact. Furthermore, internal reference points are likely to be inexact as well. For example, consider this case where Alice changes a single word in her document and transmits a patch for Bob to apply to his version of the document.

```
Alice's side:
  V1:   The computer's old processor chip.
  V2:   The computer's new processor chip.
  Diff: puter's ~~old~~new process [starting near character 8]

Bob's side:
  Pattern: puter's old process [starting near character 8]
  V1:      The server's odd coprocessor chip.
  Match:   --------100000000000000000000000 [error level 6]
  Subject: er's odd coprocesso
```

Knowing the approximate start and end of the pattern allows the extraction of a subject string with which to work on. A prerequisite to making any insertions or deletions is the discovery of a translation matrix between indices in the pattern string and the subject string. One approach is to perform multiple Bitap matches between one string and all substrings of the other string. A less expensive approach is to diff the two strings:

```
Pattern: puter's old process
Subject: er's odd coprocesso
Diff:    ~~put~~er's o~~l~~d co processo
```

Diff algorithms can be complex but very accurate and efficient. For this application there is no need to invoke the line-based speedup, and it is important not to use any post-processing cleanup algorithms since one wants as much detail as possible. This diff allows indices from one string to be converted into indices on the other string.

Sample Code

```python
def diff_xindex(diff, loc):
  # loc is a location in text1, compute and return the equivalent location in text2.
  # e.g. "The cat" vs "The big cat", 1->1, 5->8
  chars1 = 0
  chars2 = 0
  last_chars1 = 0
  last_chars2 = 0
  for x in range(0, len(diff)):
    (op, txt) = diff[x]
    if op != 1: # Equality or deletion.
      chars1 += len(txt)
    if op != -1: # Equality or insertion.
      chars2 += len(txt)
    if chars1 > loc: # Overshot the location.
      break
    last_chars1 = chars1
    last_chars2 = chars2

  if len(diff) != x and diff[x][0] == -1: # The location was deleted.
    return last_chars2
  # Add the remaining len(character).
  return last_chars2 + (loc - last_chars1)
```

In this example, the internal format for representing a set of differences is an array of tuples. The first element of each tuple specifies if it is an insertion (1), a ~~deletion~~ (-1) or an equality (0). The second element specifies the affected text.

Show JavaScript Hide Python

Thus it becomes straightforward to delete "odd" and insert "new" it its place:

```
Bob's V2: The server's new coprocessor chip.
```

The final step is to note the difference between the expected location (8) and the found location (9). The next patch (if there is one) should have its expected location adjusted by the difference (+1).

Bite-sized Patches

The Bitap algorithm for fuzzy matching gains its speed by exploiting the computer's dexterity with bitwise operators. Unfortunately this technique fails when the number of characters in the pattern exceeds the number of bits in the register. This limit is typically 32. Since patches can come in larger sizes, they need to be broken up into smaller pieces. The most common situation is a large deletion:

```
@@ -1,45 +1,8 @@
 The
-quick brown fox jumped over the lazy
 dog.
```

Splitting a patch requires creating relevant context on each side of the new patches. Failure to provide some contextual padding at each end of a patch may result in dropped or extra characters due to an inability of the fuzzy matching to find the exact end points.

```
@@ -1,32 +1,8 @@
 The
-quick brown fox jumped o
 ver
@@ -1,21 +1,8 @@
 The
```

```
-ver the lazy
 dog.
```

The overlapping context strings of each patch necessarily specifies an order in which the patches must be applied. The context from one patch may be altered or deleted by another; in the above example "ver" is used as context by the first patch and deleted by the second patch. This contrasts with GNU's patch where each patch is separate from the others and may be installed in any order.

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