Simple Diff Algorithm
draft-cheney-simpliediff-00.txt

Abstract

This document describes a simplified algorithm for performing computational comparisons. The design goal is to achieve an approach that uses the smallest amount of logic. Necessary secondary goals include an approach that is language agnostic, fast, extensible, and more predictable than other similar algorithms.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

Comments

Comments are solicited and should be addressed to info@prettydiff.com.
Table of Contents

1. Introduction .................................................... 3
2. Understanding how to compare .................................... 3
   2.1. It is more about the equality than the differences ........ 3
   2.2. Algorithm quality ......................................... 4
   2.3. Speed ..................................................... 4
   2.4. Output format ............................................... 6
3. How the algorithm works at a high level ......................... 7
   3.1. Getting by with a hash map and some counts .............. 7
   3.2. The third and final pass through the data ............... 8
   3.3. The "fix" function ....................................... 11
   3.4. Done ..................................................... 16
4. Extensibility ..................................................... 23
5. Security considerations ........................................ 23
6. IANA considerations ............................................. 23
7. References ....................................................... 23
8. Author’s address ................................................. 23
1. Introduction

There are many comparison algorithms available, which are commonly referred to as "diff" algorithms after the similarly named Unix application. While there are many algorithms already available none achieve speed, precision, and simplicity in an easily predictable manner.

The most commonly fault of many diff algorithms is that they solve for the incorrect problem. Instead of accomplishing the goal, to find how two samples differ, they instead will try to solve for a common computer science problem called "Longest Common Subsequence (LCS)". The LCS problem seeks to find the longest blocks of equality in samples of comparison, which sounds ideal. Unfortunately, the execution of this approach is not predictable as it is bound to the commonality of the compared samples. Furthermore, it isn’t simple or fast. Worse still is that this approach is a precision failure in situations where differences are interspersed across frequent repetition.

2. Understanding how to compare

2.1. It is more about the equality than the differences

A good diff algorithm will attempt to identify as much equality as possible. Everything else qualifies as differences. The metric for quality and precision is a smaller count of captured differences. The smaller this number the better, presuming differences aren’t escaping undetected.

False negatives, which is allowing differences through without detection, is really bad. This is absolute failure in a diff algorithm. False positives, which is identifying more differences than there actually are is also bad, but a false positive is much better than a false negative. This means it is safer to report more differences than are actually present, which is why a higher number of reported differences is less precise. Maximum precision is reporting differences without any false negatives or false positives.

One way to achieve higher precision is to first capture as much equality as possible. For everything else that is left over prioritize unique items first and report these items as differences. Finally determine the next bit of equality or uniqueness and everything in between is either a change, an insertion, or a deletion.
2.2. Algorithm quality

The primary priorities when writing this kind of code are execution speed, precision (as previously described), and code simplicity. In most cases precision is the most important factor in code design, but in some cases speed is more important when processing large input or a batch of thousands of files. Simplicity is necessary so that other people can understand the code and modify it as necessary to improve upon the design and additional features. No algorithm is ever capable of meeting all needs for all use cases, so it is important that other people can understand the code with minimal effort.

2.3. Speed

Faster execution is the result of a couple of things. The most important consideration for making an algorithm faster is to reduce the number of passes through data where possible. After that eliminate all costly operations. In most programming languages simple arithmetic and static string comparisons are cheap to execute particularly if the examined strings aren’t changing. The theoretical minimum number of data passes is two as you have to read the contents of each submitted sample. This proposed algorithm achieves speed by only taking three complete passes through data and making all possible effort to never repeat a step or loop iteration.

This approach is linear and predictable where the number of iterations passing through data is computed from the sum of:

* number of iterations from the first sample
* number of iterations from the second sample
* the number of iterations from the smallest of those samples.

Performance of the mentioned approach is heavily based upon key access in a hash map, which will likely vary by programming language.

Until a way to perform a comparison without reading from the samples is discovered 2 data passes can be safely assumed as the fastest possible approach. Between that and the Simple Diff approach there are two possibilities for increased performance. The first possibility is to make decisions and write output immediately upon reading from the second sample so as to have only two data passes. The challenge with this approach is that analysis occurs in the moment of the current loop iteration without knowledge of what comes later in the second sample. A more practical second performance possibility is to write a smaller hash map. Writing a smaller hash map means moving some of the decision tree up front before a separate formal analysis phase. In order for this approach to be viable this early step in logic must be tiny, non-replicating, and a different means of iteration must be devised to account for a data store of unpredictable size.
The proposed algorithm could be faster than the popular Myers’ O(ND) approach. That may or may not be true and no evidence was provided either way (conjecture is not evidence). In terms of experimental criteria algorithms are themselves largely irrelevant. More important is the implementation of those algorithms. If exercising the Myers’ approach makes fewer decisions and has fewer total data passes, as described in the previous paragraph, then it likely is faster. Pronounced emphasis is applied to the word "total" because this makes all the difference.

Many prior diff applications don’t provide a complete third data pass. Instead they provide the minimum two complete data passes over the samples and various smaller passes as calculated by block moves and edit distances. If these various fractional passes are non-linear, which means starting and stopping in different places respectively, their performance is less predictable. If these fractional passes are non-linear and touch any given data index more than once they have repetitive logic, and likely are not as fast. To affirmatively guarantee superior performance over this Simple Diff approach there needs to be fewer passes over data, which means no repetition and a smaller number of iterations. Predictability ensures the performance of an application scales roughly proportionately to the size of the provided samples, where an unpredictable approach would scale disproportionately (slower over time).

The approach taken here is fast. It cannot be argued, scientifically, it is the fastest ever (or slowest) approach for its level of accuracy without also writing alternate algorithms into applications with identical application constraints. One could argue this approach is the fastest ever comparative algorithm for its level of predictability and precision.
2.4. Output format

This Simple Diff algorithm inherits its output format from the application jsdifflib, https://github.com/cemerick/jsdifflib. The output is a big array containing child arrays at each index. The child arrays each contain 5 indexes in the format:

- type
- start in first sample
- end in first sample
- start in second sample
- end in second sample

There are 4 types defined in the output:

- "equal" - both array index items are identical at the indexes compared
- "replace" - changes are present at indexes compared
- "insert" - the index of the second sample is unique to the second sample
- "delete" - the index of the first sample is unique to the first sample
3. How the algorithm works at a high level

3.1. Getting by with a hash map and some counts

This algorithm takes after the Paul Heckel algorithm. First thing is to transform the string input into arrays. The most primitive way to do this is to split the input by lines where each line of input is an index in an array. Once the two input samples are converted to arrays they will each have to be examined.

The first loop will walk through an array representing one of the code samples and make a decision for a hash map which is named "table" in the following code sample. The two arrays are simply named "one" and "two". Each index of the array, by default is a line of code, which will serve as a key in the hash map named "table". If this key does not exist then it will be added. The value for each key is an object with two properties named "one" and "two". These properties are simply a count indicating the number of times the key is encountered in the two arrays. If the key is already present then we will simply increase the value of properties "one" or "two" respective of the current array.

Here is an example of this step in code:

```javascript
do {
    if (table[two[b]] === undefined) {
        table[two[b]] = {
            one: 0,
            two: 1
        };
    } else {
        table[two[b]].two += 1;
    }
    b += 1;
} while (b < lenb);
```
3.2. The third and final pass through the data

Once the table is fully populated with all indexes from both arrays, "one" and "two", we need a third and final loop to walk across the data and make some simple decisions. Here is this complete step in code.

do {
  c = a;
  d = b;
  if (one[a] === two[b]) {
    equality();
  } else if (table[one[a]][1] < 1 && table[two[b]][0] < 1) {
    replaceUniques();
  } else if (table[one[a]][1] < 1 && one[a + 1] !== two[b + 2]) {
    deletion();
  } else if (table[two[b]][0] < 1 && one[a + 2] !== two[b + 1]) {
    insertion();
  } else if (table[one[a]][0] - table[one[a]][1] === 1 ||
             one[a + 1] !== two[b + 2]) {
    deletionStatic();
  } else if (table[two[b]][1] - table[two[b]][0] === 1 ||
             one[a + 2] !== two[b + 1]) {
    insertionStatic();
  } else if (one[a + 1] !== two[b]) {
    deletion();
  } else if (one[a] !== two[b + 1]) {
    insertion();
  } else {
    replacement();
  }
  a = a + 1;
  b = b + 1;
} while (a < lena && b < lenb);

Before moving any further please understand this logic is fast and simple, but it isn’t precise at all. We will fix this later with a child function named "fix". Corrections for precision are a secondary step so as to not disrupt performance and also isolate complexity into a separate single location.

In the above code references "a" and "b" are simply positive integer incrementors. The "a" reference is always associated with the "one" array while the "b" reference is always associated with the "two" array. This is necessary so that each array may increment independently without collision.
The "c" and "d" references are secondary incrementors used as closures in the child functions. These secondary incrementors allow for child loops to occur without mutation to the primary incrementors.

The first thing that happens in this loop is an attempt to discover equality. Everything else is less important and so happens later in the decision tree.

The second rule is to identify items that occur only one more time in each array. If the next item to occur in the arrays is not equal but occurs just once more then we know the item is a unique change.

The third rule is to determine if the current item is a deletion. If the current item, and possibly additional following items, is present in the first sample but no longer exists in the second sample then it is a dynamic deletion.

The fourth rule is the same as the third rule but in reverse to determine dynamic insertions.

The fifth rule is to determine if the current index occurs exactly one more time in the first sample than in the second sample and yet is not a match for the next item in the first sample with two items up in the second sample. This is a static deletion, or rather a deletion of a fixed number of items.

The sixth rule is the same as the fifth rule but in reverse to determine static insertions.

The seventh and final rule determines that the current items in the samples are non-unique changes.

The child functions are as follows:

```javascript
equality = function simpleDiff_equality() {
  do {
    table[one[c]].one -= 1;
    table[one[c]].two -= 1;
    c += 1;
    d += 1;
  } while (c < lena &&
  d < lenb &&
  one[c] === two[d])
  fix(["equal", a, c, b, d]);
  b = d - 1;
  a = c - 1;
}
```
deletion = function simpleDiff_deletion() {
  do {
    table[one[c]].one -= 1;
    c += 1;
  } while (
    c < lena &&
    table[one[c]].two < 1
  );
  fix(["delete", a, c, -1, -1]);
  a = c - 1;
  b = d - 1;
},
deletionStatic = function simpleDiff_deletionStatic() {
  table[one[a]].one -= 1;
  fix(["delete", a, a + 1, -1, -1]);
  a = c;
  b = d - 1;
},
insertion = function simpleDiff_insertion() {
  do {
    table[two[d]].two -= 1;
    d += 1;
  } while (
    d < lenb &&
    table[two[d]].one < 1
  );
  fix(["insert", -1, -1, b, d]);
  a = c - 1;
  b = d - 1;
},
insertionStatic = function simpleDiff_insertionStatic() {
  table[two[b]].two -= 1;
  fix(["insert", -1, -1, b, b + 1]);
  a = c - 1;
  b = d;
replacement = function simpleDiff_replacement() {
    do {
        table[one[c]].one -= 1;
        table[two[d]].two -= 1;
        c += 1;
        d += 1;
    } while (c < lena &&
        d < lenb &&
        table[one[c]].two > 0 &&
        table[two[d]].one > 0);
    fix(['replace', a, c, b, d]);
    a = c - 1;
    b = d - 1;
},
replaceUniques = function simpleDiff_replaceUniques() {
    do {
        table[one[c]].one -= 1;
        c += 1;
        d += 1;
    } while (c < lena &&
        d < lenb &&
        table[one[c]].two < 1 &&
        table[two[d]].one < 1);
    fix(['replace', a, c, b, d]);
    a = c - 1;
    b = d - 1;
}

It must be noted that most of the these child functions do contain their own loops, but these loops are not duplicates. The primary parent loop, the so called third and final loop, is adjusted forward by the distance these smaller loops increment so that there is no repetition or lose of execution speed.

3.3. The "fix" function

We can see from the child function code that arrays are generated which appear to be the format described as the child arrays of the output. Instead of pushing this data to the output array directly it will be passed through a function named "fix" which serves as a sort of quality control filter. This function checks for things like:
* A diff type immediately following the same diff type, which can be combined into a single child array for the output
* An insertion immediately following a deletion, or vice versa, which should likely be a replacement
* Whether false positives generated from replacements immediately following an insertion or deletion can be eliminated

The general idea is to only accept a child array of the output as an argument and make a decision after comparing it against the previous child array of the output. Aside from two narrow edge cases there is no analysis of the original data. Think of it like an appeals court in that no new evidence is allowed, because only the laws are under scrutiny.

The code for the fix function is as follows:

```javascript
fix = function simpleDiff_fix(code) {
  var prior = codes[codes.length - 1];
  if (prior !== undefined) {
    if (prior[0] === code[0]) {
      if (code[0] === "replace" || code[0] === "equal") {
        prior[2] = code[2];
        prior[4] = code[4];
      } else if (code[0] === "delete") {
        prior[2] = code[2];
      } else if (code[0] === "insert") {
        prior[4] = code[4];
      }
        if (code[0] === "replace") {
          prior[0] = "replace";
          prior[1] = code[1];
          prior[2] = code[2];
          code[0] = "insert";
          code[1] = -1;
          code[2] = -1;
        } else if (code[0] === "delete") {
          code[0] = "replace";
          code[3] = prior[3];
          code[4] = prior[4];
          codes.pop();
          prior = codes[codes.length - 1];
          if (prior[0] === "replace") {
            prior[2] = code[2];
            prior[4] = code[4];
            return;
          }
        }
      }
    }
  }
}
```
else if (code[0] === "delete") {
    prior[0] = "replace";
    prior[1] = code[1];
} else if (code[0] === "replace") {
    prior[0] = "replace";
    prior[1] = code[1];
    c = prior[2];
    d = prior[4];
    return;
}
else if (
    prior[0] === "insert" &&
    code[0] === "delete" &&
) {
    code[0] = "replace";
    code[3] = prior[4];
} else if (
    prior[0] === "delete" &&
) {
        if (code[0] === "replace") {
            prior[0] = "replace";
            prior[3] = code[3];
            prior[4] = code[4];
            code[0] = "delete";
            code[3] = -1;
            code[4] = -1;
        } else if (code[0] === "insert") {
            code[0] = "replace";
            code[1] = prior[1];
            code[2] = prior[2];
            codes.pop();
            prior = codes[codes.length - 1];
            if (prior[0] === "replace") {
                prior[2] = code[2];
                prior[4] = code[4];
                return;
            }
        }
    }
} else if (code[0] === "insert") {
    prior[0] = "replace";
    prior[3] = code[3];
) else if (code[0] === "replace") {
    prior[0] = "replace";
    prior[3] = code[3];
    c = prior[2];
    d = prior[4];
    return;
}
)
else if (prior[0] === "delete" &&
    code[0] === "insert" &&
{
    code[0] = "replace";
    code[1] = prior[2];
}
else if (prior[0] === "replace") {
    if (code[0] === "delete") {
            }
            c = c - 1;
            d = d - 1;
            return;
        }
            table[one[c - 1]][0] =
            table[two[d - 1]][0] - 1;
        }
    } else if (code[0] === "insert") {
            }
            c = c - 1;
            d = d - 1;
            return;
        }
        if (one[code[2] - 1] === two[prior[4]]) {
                table[two[d - 1]][1] =
                table[two[d - 1]][1] - 1;
            }
        }
    } else if (prior[0] === "delete" &&
        code[0] === "insert" &&
        code[0] = "replace";
        code[1] = prior[2];
        return;
    }

Cheney                      Informational                      [Page 14]
codes.push(code);
}
The following is the complete algorithm with all parts assembled:

```javascript
function simpleDiff () {
    var table = {},
        lines = function simpleDiff_lines(str) {
            str = str
                .replace(/\r\n/g, "\n")
                .replace(/\r/g, "\n");
            return str.split("\n");
        },
        one = (typeof options.source === "string")
            ? lines(options.source)
            : options.source,
        two = (typeof options.diff === "string")
            ? lines(options.diff)
            : options.diff,
        lena = one.length,
        lenb = two.length,
        a = 0,
        b = 0,
        c = 0,
        d = 0,
        codes = [],
        fix = function simpleDiff_fix(code) {
            var prior = codes[codes.length - 1];
            if (prior !== undefined) {
                if (prior[0] === code[0]) {
                    if (code[0] === "replace" ||
                        code[0] === "equal") {
                        prior[2] = code[2];
                        prior[4] = code[4];
                    } else if (code[0] === "delete") {
                        prior[2] = code[2];
                    } else if (code[0] === "insert") {
                        prior[4] = code[4];
                    }
                    return;
                } else if (prior[0] === "insert" &&
                    prior[0] = "replace";
                    prior[1] = code[1];
                    prior[2] = code[2];
                }
            } else {
                prior[0] = "insert" &&
                    if (code[0] === "replace") {
                        prior[0] = "replace";
                        prior[1] = code[1];
                        prior[2] = code[2];
                    }
                }
            }
        }
    return;
}
```
code[0] = "insert";
code[1] = -1;
code[2] = -1;
} else if (code[0] === "delete") {
  code[0] = "replace";
  code[3] = prior[3];
  code[4] = prior[4];
  codes.pop();
  prior = codes[codes.length - 1];
  if (prior[0] === "replace") {
    prior[2] = code[2];
    prior[4] = code[4];
    return;
  }
}

} else if (code[0] === "delete") {
  prior[0] = "replace";
  prior[1] = code[1];
} else if (code[0] === "replace") {
  prior[0] = "replace";
  prior[1] = code[1];
  c = prior[2];
  d = prior[4];
  return;
}

} else if (prior[0] === "insert" &&
  code[0] === "delete" &&
  code[0] = "replace";
  code[3] = prior[4];
} else if (prior[0] === "delete" &&
    if (code[0] === "replace") {
      prior[0] = "replace";
      prior[3] = code[3];
      prior[4] = code[4];
      code[0] = "delete";
      code[3] = -1;
      code[4] = -1;
    } else if (code[0] === "insert") {
      code[0] = "replace";
  
  }
code[1] = prior[1];
code[2] = prior[2];
codes.pop();
prior = codes[codes.length - 1];
if (prior[0] === "replace") {
prior[2] = code[2];
prior[4] = code[4];
return;
}
} else if (code[0] === "insert") {
prior[0] = "replace";
prior[3] = code[3];
}
} else if (code[0] === "replace") {
prior[0] = "replace";
prior[3] = code[3];
c = prior[2];
d = prior[4];
return;
}
} else if (prior[0] === "delete" &&
code[0] === "insert" &&
)
{
code[0] = "replace";
code[1] = prior[2];
}
else if (prior[0] === "replace") {
  if (code[0] === "delete") {
      }
      c = c - 1;
d = d - 1;
    return;
    }
  }
}
if (one[code[2]] === two[prior[4] - 1]) {
table[one[c - 1]][0] =
    table[one[c - 1]][0] - 1;
  }
}
} else if (code[0] === "insert") {

Cheney                  Informational                  [Page 18]
        c = c - 1;
        d = d - 1;
        return;
    }
    if (one[code[2] - 1] === two[prior[4]]) {
            table[two[d - 1]][1] =
                table[two[d - 1]][1] - 1;
        }
    }
}
codes.push(code);
}

equality = function simpleDiff_equality() {
    do {
        table[one[c]][0] = table[one[c]][0] - 1;
        table[one[c]][1] = table[one[c]][1] - 1;
        c = c + 1;
        d = d + 1;
    } while (c < lena && d < lenb && one[c] === two[d]);
    fix(["equal", a, c, b, d]);
    b = d - 1;
    a = c - 1;
},
deletion = function simpleDiff_deletion() {
    do {
        table[one[c]][0] = table[one[c]][0] - 1;
        c = c + 1;
    } while (c < lena && table[one[c]][1] < 1);
    fix(["delete", a, c, -1, -1]);
    a = c - 1;
    b = d - 1;
},
deletionStatic = function simpleDiff_deletionStatic() {
    table[one[a]][0] = table[one[a]][0] - 1;
    fix(["delete", a, a + 1,
        -1,
        -1
    ]);,
    a = c;
    b = d - 1;
insertion = function simpleDiff_insertion() {
    do {
        table[two[d]][1] = table[two[d]][1] - 1;
        d = d + 1;
    } while (d < lenb && table[two[d]][0] < 1);
    fix(["insert", -1, -1, b, d]);
    a = c - 1;
    b = d - 1;
},

insertionStatic = function simpleDiff_insertionStatic() {
    table[two[b]][1] = table[two[b]][1] - 1;
    fix(["insert", -1, -1, b, b + 1]);
    a = c - 1;
    b = d;
},

replacement = function simpleDiff_replacement() {
    do {
        table[one[c]][0] = table[one[c]][0] - 1;
        table[two[d]][1] = table[two[d]][1] - 1;
        c = c + 1;
        d = d + 1;
    } while (
        c < lena &&
        d < lenb &&
        table[one[c]][1] > 0 &&
        table[two[d]][0] > 0
    );
    fix(["replace", a, c, b, d]);
    a = c - 1;
    b = d - 1;
},

replaceUniques = function simpleDiff_replaceUniques() {
    do {
        table[one[c]][0] = table[one[c]][0] - 1;
        c = c + 1;
        d = d + 1;
    } while (
        c < lena &&
        d < lenb &&
        table[one[c]][1] < 1 &&
        table[two[d]][0] < 1
    );
    fix(["replace", a, c, b, d]);
    a = c - 1;
    b = d - 1;
};

// * First Pass, account for lines from first file
// * build the table from the second file
do {
    if (options.diffspaceignore === true) {
        two[b] = two[b].replace(/\s+/g, "");
    }
    if (table[two[b]] === undefined) {
        table[two[b]] = [0, 1];
    } else {
        table[two[b]][1] = table[two[b]][1] + 1;
    }
    b = b + 1;
} while (b < lenb);

// * Second Pass, account for lines from second file
// * build the table from the first file
lena = one.length;
a = 0;
do {
    if (options.diffspaceignore === true) {
        one[a] = one[a].replace(/\s+/g, "");
    }
    if (table[one[a]] === undefined) {
        table[one[a]] = [1, 0];
    } else {
        table[one[a]][0] = table[one[a]][0] + 1;
    }
    a = a + 1;
} while (a < lena);
a = 0;
b = 0;

// find all equality... differences are what’s left over solve
// only for the second set test removing reverse test removing
// undefined checks for table refs
do {
    c = a;
    d = b;
    if (one[a] === two[b]) {
        equality();
    } else if (table[one[a]][1] < 1 && table[two[b]][0] < 1) {
        replaceUniques();
    } else if {
        table[one[a]][1] < 1 &&
        one[a + 1] !== two[b + 2]
    } {
        deletion();
    } else if {
        table[two[b]][0] < 1 &&
        one[a + 2] !== two[b + 1]
    } {
        insertion();
    }
}
else if (table[one[a]][0] - table[one[a]][1] === 1 && one[a + 1] !== two[b + 2]) {
    deletionStatic();
} else if (table[two[b]][1] - table[two[b]][0] === 1 && one[a + 2] !== two[b + 1]) {
    insertionStatic();
} else if (one[a + 1] === two[b]) {
    deletion();
} else if (one[a] === two[b + 1]) {
    insertion();
} else {
    replacement();
}
}
while (a < lena && b < lenb);
if (lena - a === lenb - b) {
    if (one[a] === two[b]) {
        fix(["equal", a, lena, b, lenb]);
    } else {
        fix(["replace", a, lena, b, lenb]);
    }
} else if (a < lena) {
    fix(["delete", a, lena, -1, -1]);
} else if (b < lenb) {
    fix(["insert", -1, -1, b, lenb]);
}
return codes;
4. Extensibility

Since the code is simple and its execution is predictable the approach is open to modification. In different contexts a most precise comparison of text is not the most desirable objective. One example is to ignore certain particular differences that might be considered as unimportant. This degree of flexibility and extensibility allows for custom precision.

Custom flexibility allows for a variety of advanced concerns. Such concerns may include nuances in accounting for language specific criteria whether programming, spoken, or written. Due to flexible rules, whether colloquialisms or syntax variations, differences of literal text may commonly reside that are not represented as differences in the parsed, or interpreted, result. In artificially intelligent systems it is necessary to account for these variations as comparisons occur.

5. Security considerations

None.

6. IANA considerations

None.

7. References


8. Author’s address

Austin Cheney
http://prettydiff.com
info@prettydiff.com