XDP: XML Diff and Patch

Georgio Yammine, Rami Naffah

Abstract— This report focuses on implementing a tool in Java that compares two XML documents (structure and content) in the most efficient way in terms of both space and time complexities. After comparing the files, the tool provides the diff file represented in an XML format and that highlights the operations that must be executed in order to transform a file into the other. This diff can be made reversible to provide A->B patch or B->A patch. Moreover, this tool can do the patching having as inputs the XML file and its corresponding diff file and transforms it to the target XML document. This tool has many applications: i) data warehousing, ii) version control, iii) XML retrieval and much more.

CONTENTS

INTRODUCTION AND BACKGROUND 1
Pre-processing 2
Node Referencing Format 2
Tree Edit Distance 3
Contained In 3
Edit Distance of Text Nodes and Attributes 3
Reordering Attributes 4
Pruning technique: 4
Getting the Edit Script: 5
Patching Process: 6
Patch and Input file verification: 6
Experimental data: 7
Conclusion 7
References 7
Appendix A 8
Appendix B 11
Appendix C 12
Appendix D 13

INTRODUCTION AND BACKGROUND

In the previous years, W3C's XML has been one of the most important standards for efficient data management and manipulation. XML is mostly used in database information interchange and web services interaction. With the growth of the World Wide Web, there is an increasing need to automatically process Web documents for efficient data management, search applications and similarity clustering. With the constant increase of data publication on the Web and the users' interest in the evolution of documents (changes as information that can be consulted or queried), XML-based similarity/comparison becomes a central issue in the information retrieval

communities. Applications of XML comparison and diffs are numerous: i) version control, change management and data warehousing (finding, scoring and browsing changes between different versions of a document, support of temporal queries and index maintenance), ii) semi-structured data integration (identifying similar XML documents originating from different data sources, to be integrated so that the user can access more complete information), iii) classification/clustering of XML documents gathered from the web against a set of XML grammars declared in an XML database (just as schemas are necessary in traditional DBMS for the provision of efficient storage, retrieval and indexing facilities, the same is true for XML repositories), iv) as well as XML retrieval (finding and ranking results according to their similarity in order to retrieve the best results possible).
For this matter, Tree Edit Distance methods have been used in order to study similarities between XML documents and have been proven to be efficient and optimal. These methods produce a distance value which quantifies the resemblances/differences between the XML trees, as well as an edit script which describes the minimum number of modifications that transform one tree into the other.
In this study, we propose to exploit the well-known techniques for finding the edit distance between tree structures, to allow the comparison and differencing of semi-structures XML documents.

Tools Used

To implement our project we used Java, JavaFx with FXML and Jfoenix Library.

Pre-processing

The objective of this part is to transform input XML documents into rooted ordered labeled trees. We used the default XML DOM parser because it produces a tree that contains all of the elements of the input document. Also, it provides a variety of functions that can be used to examine the contents and structure of the document.
We used the Node.normalize() method which puts the specified node and all of its sub-trees into a "normalized" form. In a normalized sub-tree, no text nodes in the sub-tree are empty and there are no adjacent text nodes.
It also includes a clean method which role is the following:

- It removes leading and lagging spaces and replaces any number of spaces between words by a single space.
- It removes empty text nodes, which result from extra spaces or enter.
- It removes comments nodes.

Node Referencing Format

The format we used to refer to a node in a tree is the following: A.x.x.x... where A is the root name, the dot represents child nodes and x represents the position of the child node starting from 1 and not 0. For instance, A.1.3 represents the 3rd child of the first child of A. After further researches, we discovered that our format is similar to the Dewey Numbers format which we did not know when we implemented our algorithm.

Tree Edit Distance

The tree edit distance algorithm utilized in our study is an adaptation of Nierman and Jagadish's main edit distance algorithm [1]. However, in our case, we have to also take into consideration the content such as text nodes, attributes and more.

** The pseudocode of NNJ and our implementations can be found in Appendix A.*

The variations we implemented were:

1. Updating root node changes the root label and root attributes.
2. Update is only possible if the 2 nodes are of the same type.
3. Text nodes are tokenized on space (“ “) and are processed in EDWords.
4. Element Nodes are updated using a call of TED.

Pre-defined Costs:

The following are the defined costs for the smallest tokens of each category.

They all have as default a unity cost. These can be changed by the user to his own values.

```
static int updateRootName = 1;     // update root Label
static int insertContained = 1;    // insert node contained in A
static int deleteContained = 1;    // delete node contained in B
static int deleteOrInsertLeaf = 1; // Insert Leaf Node
static int attributeNameCost = 1;  // update attribute name
static int attributeValueCost = 1; // update attribute value
static int contentTokenCost = 1;   // inserting/update/delete word
```

Contained In

We implemented a containedIn method that determines if a node or sub-tree is contained-in the tree structure we are comparing our XML tree with. In this method, a sub-tree is contained in another if and only if it is exactly found in the other. The reason we chose to implement that way is that while inserting a node or a sub-tree, we have to insert it as a whole: we cannot insert a part of it because it defeats the purpose of the insert at cost 1.

Edit Distance of Text Nodes and Attributes

To get the edit distance of attributes and words between two xml files, we implemented algorithms that are an adaptation of the String Edit Distance algorithm. We compare strings and attribute values instead of characters.

The choice we made in our comparisons was to tokenize the text nodes into words separated by spaces (“ “) and do the ED on them, while for attributes, we considered the attribute value to be atomic.

Reordering Attributes

We implemented a reorder() method in which the similar attributes between the 2 input tree structures appear at the same positions in an ArrayList of Nodes.
**The pseudocode of the method can be found in the Appendix section A.*
The reason such a method is useful is the following:
In similarity and edit script algorithms, the order of attributes does not matter. However, in the algorithm we are using which is an adaptation of the String Edit Distance algorithm, the order is taken into consideration, which are by default ordered in an alphabetical order by the DOM parser. Hence, to eliminate the impact of the order on the edit distance, we reordered the attributes of both trees by mapping same keys to same indexes. In other words, same keys are placed at the same position. Note that the attributes with different keys are still ordered in alphabetical order.

Example:
XML Files:
File 1:

```
<a name="John" age="21" hobby="tennis" profession="Consultant" status="married"></a>
```

File 2:

```
<a major="COE" name="john" Married="No" hobby="Guitar" lastname= "Posner"></a>
```

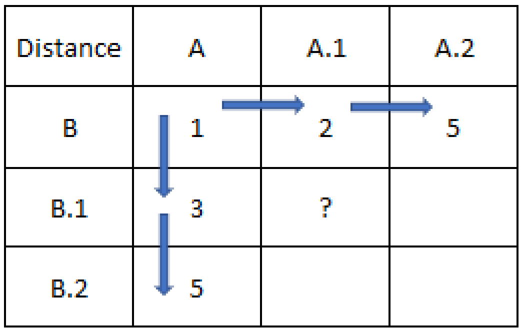
Before our reordering:

```
a {age: 21, hobby: tennis, name: John, profession: Consultant, status: married}
a {Married: No, hobby: Guitar, lastname: Posner, major: COE, name:john}
```

After reordering:
[hobby="tennis", name="John", age="21", profession="Consultant", status="married"]
[hobby="Guitar", name="John", Married="No", lastname="Posner", major="Computer Engineer"]

Pruning technique:

In the calculation of TED inside the matrix, we do not always need to compute update, delete, and insert cost before choosing the minimum. Sometimes computing only one is sufficient and thus saves time, especially if we can skip recursions in some situations. This is similar to alpha beta pruning in Minimax. An example of this is as follows.



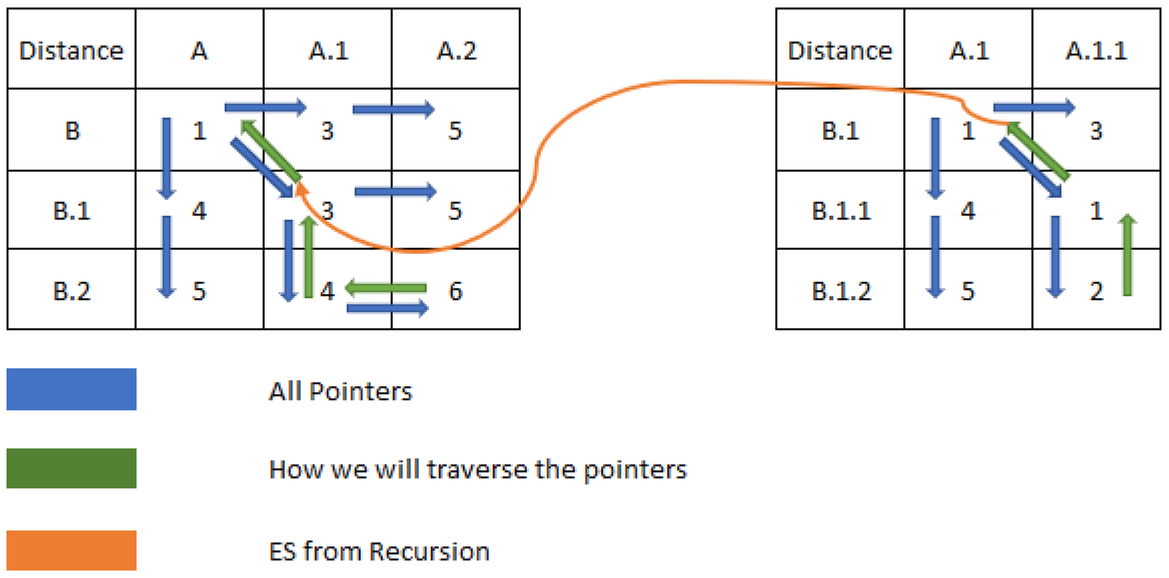
Distance	A	A.1	A.2
B	1	2	5
B.1	3	?	
B.2	5		

If A.1,B.1$_{update}$ = 1+0 = 1 < 2 (A.1,B) < 3 (A,B.1), we do not have to compute cost of insert/delete. We followed multiple heuristics to make the decision on which operation to start first.

Getting the Edit Script:

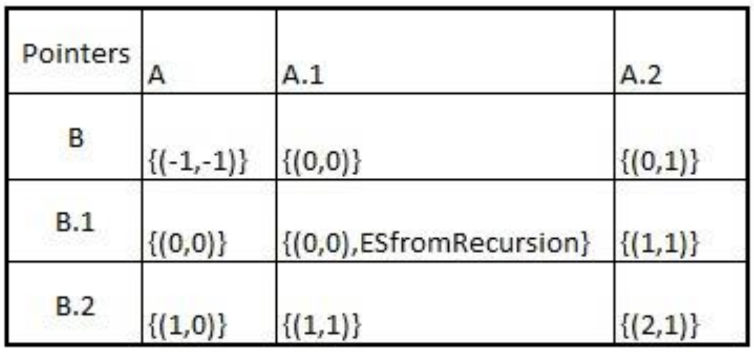
To get the edit script from the distance matrix, we created a pointers matrix of the same size of the distance matrix and stored pointers on each state pointing to the previous state. Then, we traversed the pointers matrix from the end to start. On any update operation the recursive call will store its own ES on the current state cell to be then used to form the full Edit Script.

Visualization of the Algorithm at Work:



Distance	A	A.1	A.2
B	1	3	5
B.1	4	3	5
B.2	5	4	6

Distance	A.1	A.1.1
B.1	1	3
B.1.1	4	1
B.1.2	5	2



Pointers	A	A.1	A.2
B	{(-1,-1)}	{(0,0)}	{(0,1)}
B.1	{(0,0)}	{(0,0),ESfromRecursion}	{(1,1)}
B.2	{(1,0)}	{(1,1)}	{(2,1)}

Figure 1: Simple respresentation of the pointers

After getting the pointers the operation can be easily parsed by comparing prev to next:

- prevI + 1 == newI and prevJ + 1 == newJ => update
- prevI == newI and prevJ + 1 == newJ => Insert
- prevI + 1 == newI and prevJ == newJ => Delete

A very similar but simpler process is applied to ESWords and ESAttributes since they do not have a recursive call.

Patching Process:

The patch format we implement contains the following operations:

- Distance, Similarity, XDP application version, and if the Patch is reversible.
- File A: name and hash.
- File B: name and hash.

The patch edit scipt contains the following operations:

- Update:
  - Element_Node:
    - Update Label
    - Attributes
      - Update Attribute (key, value, or both)
      - Delete Attribute
      - Insert Attribute
  - Text_Node:
    - Update Word
    - Delete Word
    - Insert Word
- Delete node
- Insert node

The patch format can be either reversable or no. A reversible file contains the changed information from both files. The reversible file size is about 15% larger.
**The patch file format can be seen in more detail in Appendix B and Appendix C.*

To process the patch, we followed the following order:

1. Update labels, attributes, text Nodes (update, delete, insert words)
2. Deleting Nodes starting from largest indexes first (Ex: A.6.4 -> A.6.1 -> A.4.7)
   This was used as if we deleted the smallest indexes first, the indexes of the largest nodes will be changed, and we will be deleting different nodes than intended.
3. Inserting Nodes starting from smallest indexes first (Ex: A.4.7 -> A.6.1 -> A.6.4)
   This was used because the insert position is the ultimate position in the tree.Ex if we need to insert A.2 and A.5, we cannot insert at A.5 if the tree currently only have 3 elements, we need first to Insert at A.3 and then we can Insert at A.5.

Patch and Input file verification:

To verify if the input XML file given for patch match the input file used to generate the patch, we used crc32 hash checksum (Ex: `crc32="7492dc6c"`) to verify if the 2 files are the same. Before the comparison a pre-processing step is done to write the input file in the format of the program (normalize, clean, and write in a predefined format).
Another checksum is included in the patch file for the output file which is used after a patching process to make sure that the generated file is correct and matches the pre-stored checksum.

Experimental data:

We tested our implementation on multiple small and large files:
SampleDoc (original) and SampleDoc (original) V1: (15KB, 16 first level child nodes, 400 lines)
get diff: 657ms | Patch 73ms | reverse diff70ms
afwikibooks-pages-articles: (01/2020 - 03/2020) (317KB, 180 first level child nodes, 7820 lines)
get diff: 23450ms | Patch 278ms | reverse diff 37ms
afwikibooks-20200201-pages-meta-current (01/2020 - 03/2020) (1263KB, 1038 first level child nodes, 30000 lines)
get diff: 147034ms | Patch 2967ms | reverse diff 126ms

We also compared our diff file size to the DeltaXML XML Compare diff size: our file size was about 50% to 75% smaller in size (Diff 1: our reversible diff was 67KB and 1897 lines, while Delta XML patch was 143KB and 2605 lines. Diff 2: our reversible diff was 212KB and 6391 lines, while Delta XML diff was 924KB and 20380 lines.). Also, a flaw we found in Delta XML is that Word by Word comparison cannot be done if the Diff file is changes-only and therefore must include all the unchaged information in both documents which results in a diff file larger that the initial file.

The Application User Interface can be seen in Appendix D.

Conclusion

In recent years, the proliferation of XML data sources on the Web has introduced the need of tools that help in information retrieval, data comparison and manipulation. In this paper, we propose a tool for XML content comparison, that highlights the similarity percentage between 2 XML tree structures, produces their corresponding diff file and patches one to transform it into the other. The diff file we created can be reversed to do the patch in both directions and was proven to be smaller in size that previously existing diffs such as DeltaXML.
This can be useful in data warehousing and version control where huge number of files need to be stored along with their previous versions, which requires such a tool to save space.

This tool can be further expanded to be used in a semi-structured data warehousing application tool, which can handle version control, XML retrieval, classification/clustering of XML documents, and much more.

References

[1] A. Nierman and H. V. Jagadish. Evaluating structural similarity in XML documents. In Proceedings of the 5th ACM SIGMOD International Workshop on the Web and Databases (WebDB), (2002) pp. 61-66.

Appendix A

Pseudocode of Nierman & Jagadish's main tree edit distance algorithm, 2002:

```
private int editDistance(Tree A, Tree B) {
int M = Degree(A);
int N = Degree(B);
int[][] dist = new int[0..M][0..N];
dist[0][0] = CostRelabel(λ(A), λ(B));

for (int j = 1; j ≤ N; j++)
dist[0][j] = dist[0][j-1] + CostGraft(Bj );
for (int i = 1; i ≤ M; i++)
 dist[i][0] = dist[i-1][0] + CostP rune(Ai);

 for (int i = 1; i ≤ M; i++)
 for (int j = 1; j ≤ N; j++)
 dist[i][j] = min{
 dist[i-1][j-1] + editDistance(Ai, Bj ),
 dist[i][j-1] + CostGraft(Bj ),
 dist[i-1][j] + CostP rune(Ai)
 };
 return dist[M][N];
 } //editDistance
```

Our own algorithm and implementation:

```
private int TED(Tree A, Tree B) {
    if(A.isEqualNode(B))
        return 0;

    int M = Degree(A);
    int N = Degree(B);
    int[][] dist = new int[0..M][0..N];
    dist[0][0] = CostUpdateRoot(A,B);

    for (int j = 1; j ≤ N; j++)
        dist[0][j] = dist[0][j-1] + CostInsert(Bj);
    for (int i = 1; i ≤ M; i++)
        dist[i][0] = dist[i-1][0] + CostDelete(Ai);

    for (int i = 1; i ≤ M; i++)
    for (int j = 1; j ≤ N; j++)
        dist[i][j] = min{
            if(Ai.type() == Bi.type()){
                if(type==Node.TextNode)
                    dist[i-1][j-1] + EDWords(Ai.TextContent,Bi.TextContent),
                else
                    dist[i-1][j-1] + TED(Ai,Bj),
            }
            dist[i][j-1] + CostInsert(Bj),
            dist[i-1][j] + CostDelete(Ai)
        }
};
return dist[M][N];
} //editDistance
```

```
private int CostUpdateRoot(A,B){
    int cost = 0;
    if(A.label()!=B.label())
        cost+= COST_CHANGE_LABEL;
    cost+= CostUpdateAttributes(A,B);
    return cost;
}
```

```
private int costDeleteOrInsertNode(node){
    if(containedInOtherTree)
        return INSERT/DELETE_CONTAINED

    if (node.isElementNode) {
        int cost = deleteOrInsertLeaf;
        NodeList childs = node.getChildNodes();
    for (int i = 0; i < childs.getLength(); i++)
         cost += CostDeleteOrInsertTree(childs.item(i));

    // cost of inserting attributes
    cost+=(attributeNameCost+attributeValueCost)*node.getAttributes().getLength();

     return cost;
     } else { // text Node
        int cost = contentTokenCost * node.getTextContent().split("\\s+").length;
        return cost;
     }
}
```

```
public int EDAttr(ArrayList<Node> attrA, ArratList<Node> attrB){

    reorder(attrA, attrB);

    ArrayList<Object> arl = new ArrayList<>();
    int[][] distance = new int[|attrA|+1][|attrB|+1];
    distance[0][0] = 0;

    for(i = 1; i<= |attrA|;i++){
        distance[i][0] = distance[i-1][0] + attributeNameCost + attributeValueCost;
    }

    for(j = 1; j <= |attrB|; j++){
        distance[0][j] = distance[0][j-1] + attributeNameCost + attributeValueCost;
    }

    for(i = 1; i <= |attrA|; i++){
      for(j = 1; j<=|attrB|;j++){
        dist[i][j] = min{
         dist[i-1][j-1] + CostUpdateAttr(attrA.get(i-1), attrB.get(j-1), attrA),
         dist[i][j-1] + attributeNameCost + attributeValueCost,
         dist[i-1][j] + attributeNameCost + attributeValueCost};
        }
    }}
    return distance[|attrA|][|attrB|];
}
```

```
private void reorderAttributes(ArrayList<Node> listA, ArrayList<Node> listB)    {
    ArrayList<Node> newListA = new ArrayList<Node>();
    ArrayList<Node> newListB = new ArrayList<Node>();
    for(i = 0; i<|listA|;i++){
        int j = Util.contains(listA.get(i), listB);
        if(j != -1){
            newListA.add(listA.get(i));
            newListB.add(listB.get(j));
                listA.remove(i);
                listB.remove(j);
            } else
                i++;
        }
        while (listA.size() > 0)
            newListA.add(listA.remove(0));

        while (listB.size() > 0)
            newListB.add(listB.remove(0));

        listA.addAll(newListA);
        listB.addAll(newListB);
}
```

```
public int EDWords(String[] rootAContent, String[] rootBContent){

    ArrayList<Object> arl = new ArrayList<>();
    int[][] distance = new int[|rootAContent|+1][|rootBContent|+1];
    distance[0][0] = 0;

    for(i = 1; i<= |attrA|;i++){
        distance[i][0] = distance[i-1][0] + contentTokenCost;
    }
    for(j = 1; j <= |attrB|; j++){
        distance[0][j] = distance[0][j-1] + contentTokenCost;
    }
    for(i = 1; i <= |attrA|; i++){
        for(j = 1; j<=|attrB|;j++){
            dist[i][j] = min{
                      dist[i-1][j-1] + CostUpdateContent(rootAContent[i - 1], root-
BContent[j - 1]),
                      dist[i][j-1] + contentTokenCost,
                      dist[i-1][j] + contentTokenCost
            };
    }}
    return distance[|rootAContent|][|rootBContent|];
}
```

Appendix B

Non reversible diff file format:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<XML_Patch distance="761" reversible="no" similarity="85.55%" version="1.0">
  <Original_File Hash="cb31d006" name="E2_A.xml"/>
  <Patched_File Hash="e232494f" name="E2_B.xml"/>
  <Edit_Script>
    <Update>
     <A.4>
        <Label>newLabel</Label>
        <Attributes>
            <update>
                <key change="both" newKey="newName" newValue="newValue">
                <key change="key" newKey="newName">
                <key change="value" newValue="newValue">
            </update>
            <delete>
                <key/>
            </delete>
            <Insert>
                <Key>value</Key>
            </Insert>
        </Attributes>
     <A.4/>
     <A.5.1>
        <update>
            <w5>newWord<w5/>
        </update>
        <delete>
            <w6/>
        </delete>
    </Update>
        <Insert>
            <w7>newWord<w7/>
        </Insert>
     <A.5.1/>
    <Delete>
        <A.1/>
     <A.6/>
    </Delete>
    <Insert>
      <A.1.1>
        <b attr1="val1" attr2="val2">
          <c attr="val"/>
          text data
          <d/>
        </b>
      </A.1.1>
      <A.2 containedAt="A.3.2"/>
    </Insert>
  </Edit_Script>
</XML_Patch>
```

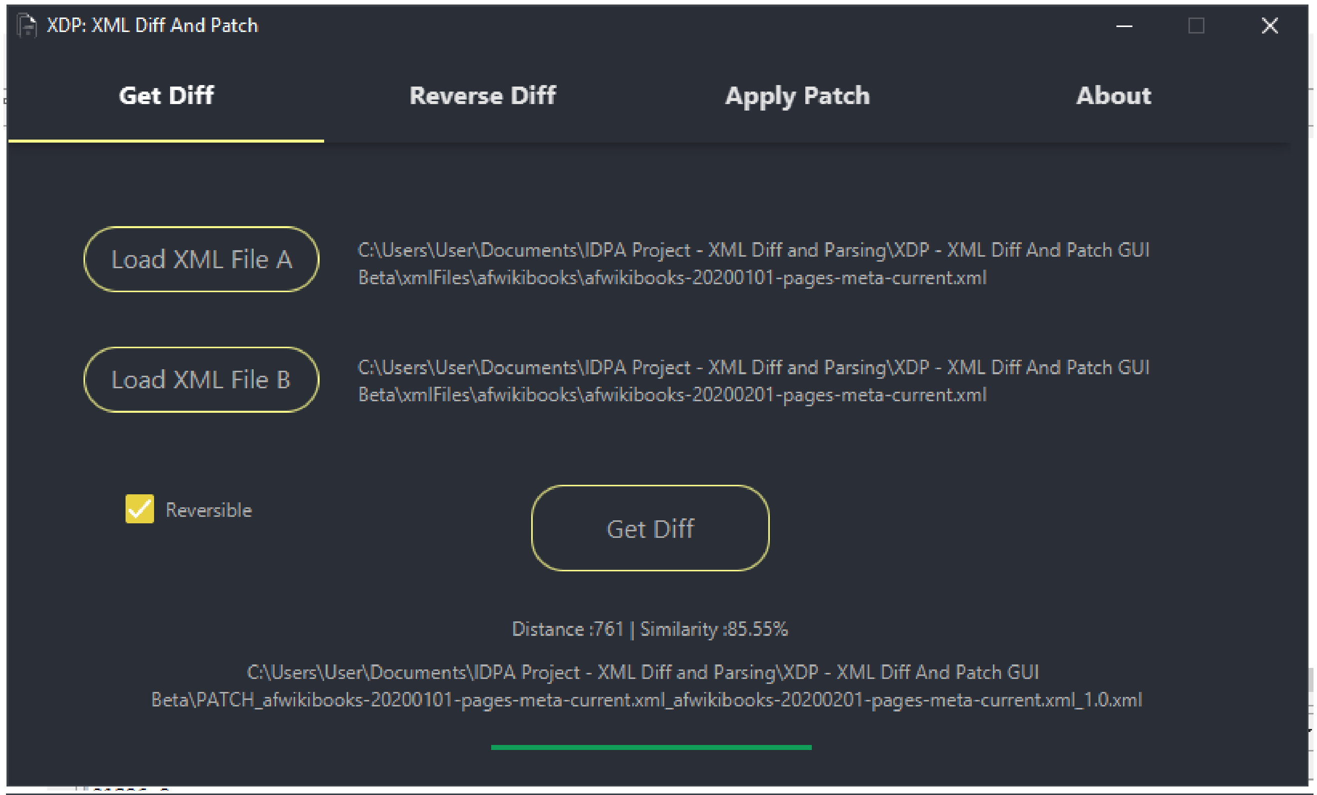
Appendix C

Reversible diff file format: (extra information from the non reversible are highlited in yellow)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<XML_Patch distance="761" reversible="yes" similarity="85.55%" version="1.0">
  <Original_File Hash="cb31d006" name="E2_A.xml"/>
  <Patched_File Hash="e232494f" name="E2_B.xml"/>
  <Edit_Script>
    <Update>
      <A.4 with="B.2">
        <Label oldVal="oldLabel">newLabel</Label>
        <Attributes>
            <update>
                <key change="both" oldKey="oldName" oldValue="oldValue"
newKey="newName" newValue="newValue">
                <key change="key" oldKey="oldName" newKey="newName">
                <key change="value" oldValue="oldValue" newValue="newValue">
            </update>
            <delete>
                <key>value</key>
            </delete>
            <Insert>
                <Key>value</Key>
            </Insert>
        </Attributes>
       <A.4/>
       <A.5.1 with="B.6.2">
        <update>
            <w5 oldVal="oldWord" with="w4">newWord<w5/>
        </update>
        <delete>
            <w6>oldWord</w6>
        </delete>
        <Insert>
            <w7>newWord<w7/>
        </Insert>
      <A.5.1/>
    </Update>
    <Delete>
      <A.1>
            <parent>
                  <child1>
                  Text
                  <child2 attr="val" >
            </parent>
      <A.1>
      <A.6 containedAt="B.7">
    </Delete>
    <Insert>
      <A.1.1>
        <b attr1="val1" attr2="val2">
          <c attr="val"/>
          text data
          <d/>
        </b>
      </A.1.1>
      <A.2 containedAt="A.3.2"/>
    </Insert>
  </Edit_Script>
</XML_Patch>
```

Appendix D

Our Application User Interface



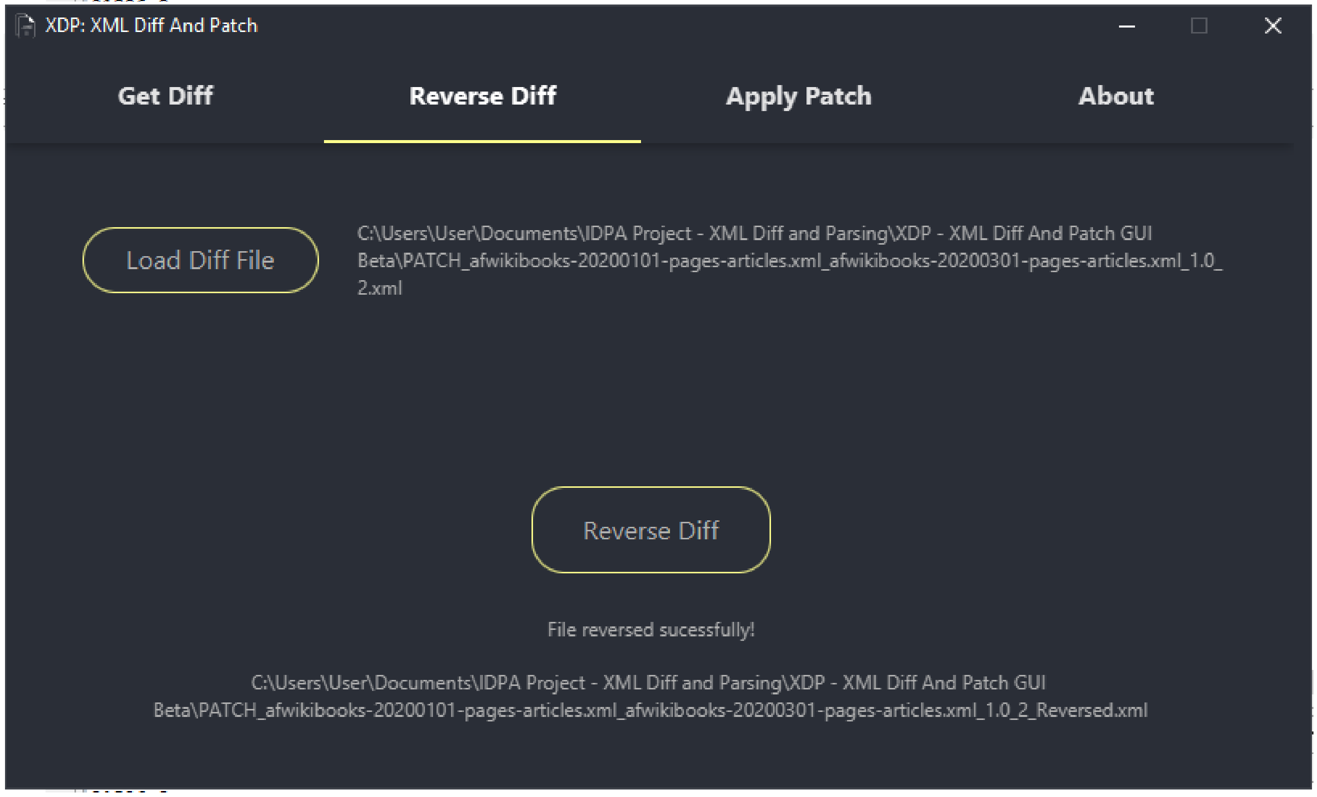
C:\Users\User\Documents\IDPA Project - XML Diff and Parsing\XDP - XML Diff And Patch GUI Beta\xmlFiles\afwikibooks\afwikibooks-20200101-pages-meta-current.xml

C:\Users\User\Documents\IDPA Project - XML Diff and Parsing\XDP - XML Diff And Patch GUI Beta\xmlFiles\afwikibooks\afwikibooks-20200201-pages-meta-current.xml

☑ Reversible

Distance :761 | Similarity :85.55%

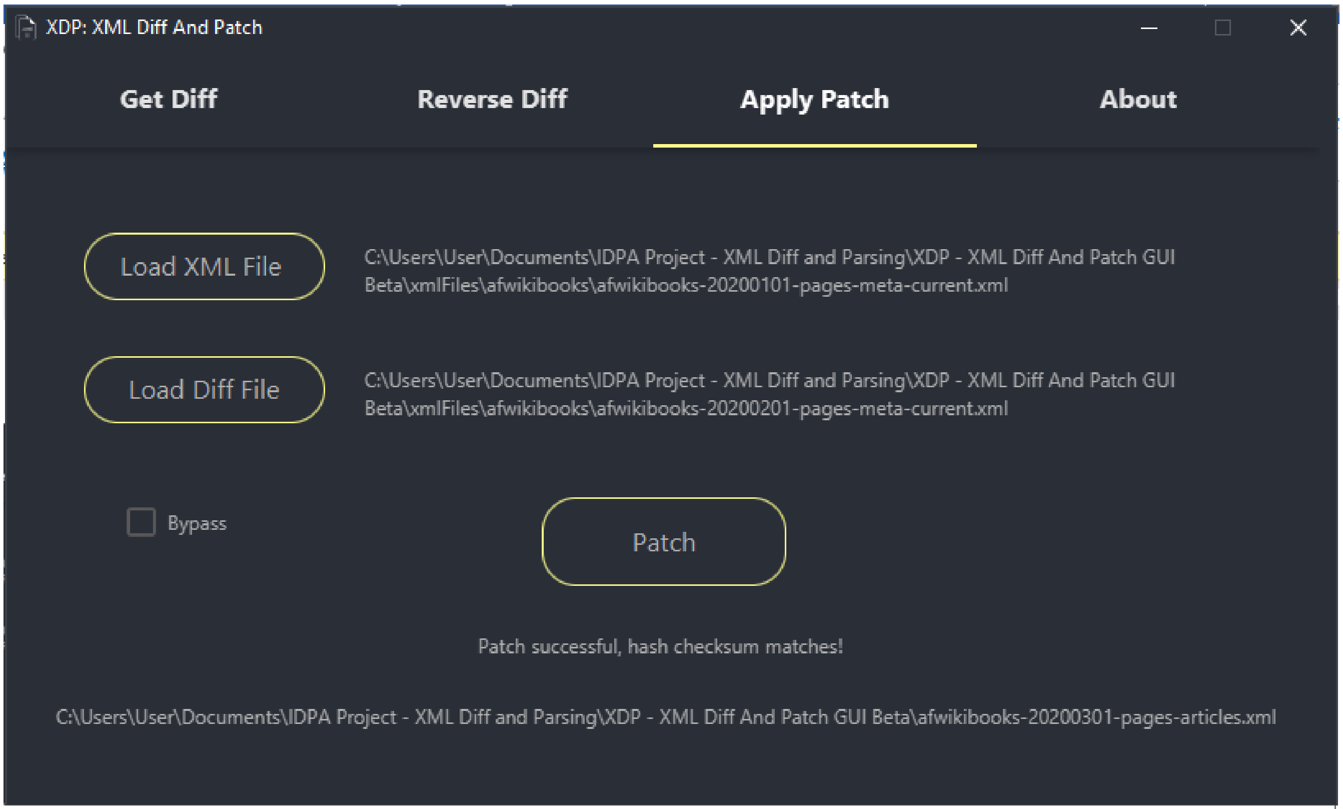
C:\Users\User\Documents\IDPA Project - XML Diff and Parsing\XDP - XML Diff And Patch GUI Beta\PATCH_afwikibooks-20200101-pages-meta-current.xml_afwikibooks-20200201-pages-meta-current.xml_1.0.xml



C:\Users\User\Documents\IDPA Project - XML Diff and Parsing\XDP - XML Diff And Patch GUI Beta\PATCH_afwikibooks-20200101-pages-articles.xml_afwikibooks-20200301-pages-articles.xml_1.0_2.xml

File reversed sucessfully!

C:\Users\User\Documents\IDPA Project - XML Diff and Parsing\XDP - XML Diff And Patch GUI Beta\PATCH_afwikibooks-20200101-pages-articles.xml_afwikibooks-20200301-pages-articles.xml_1.0_2_Reversed.xml

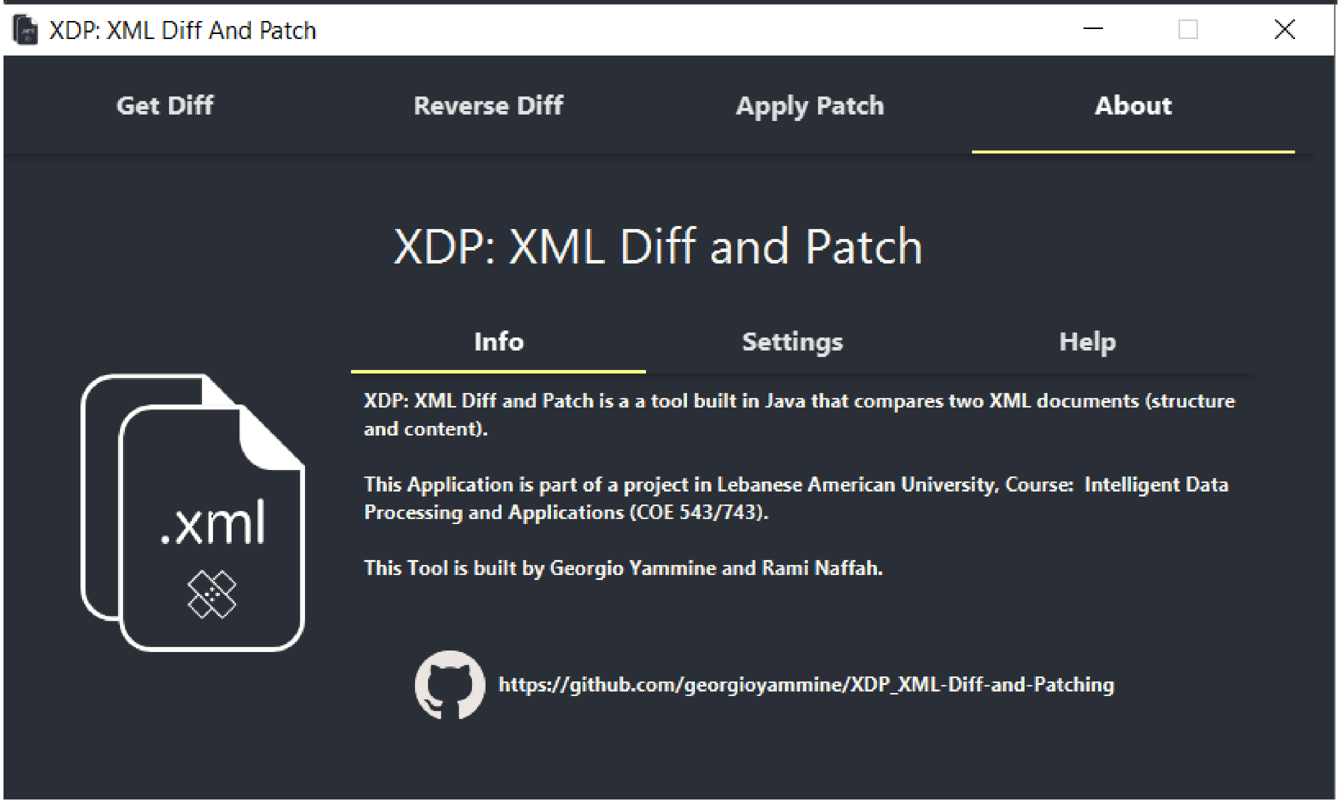


C:\Users\User\Documents\IDPA Project - XML Diff and Parsing\XDP - XML Diff And Patch GUI Beta\xmlFiles\afwikibooks\afwikibooks-20200101-pages-meta-current.xml

C:\Users\User\Documents\IDPA Project - XML Diff and Parsing\XDP - XML Diff And Patch GUI Beta\xmlFiles\afwikibooks\afwikibooks-20200201-pages-meta-current.xml

Patch successful, hash checksum matches!

C:\Users\User\Documents\IDPA Project - XML Diff and Parsing\XDP - XML Diff And Patch GUI Beta\afwikibooks-20200301-pages-articles.xml

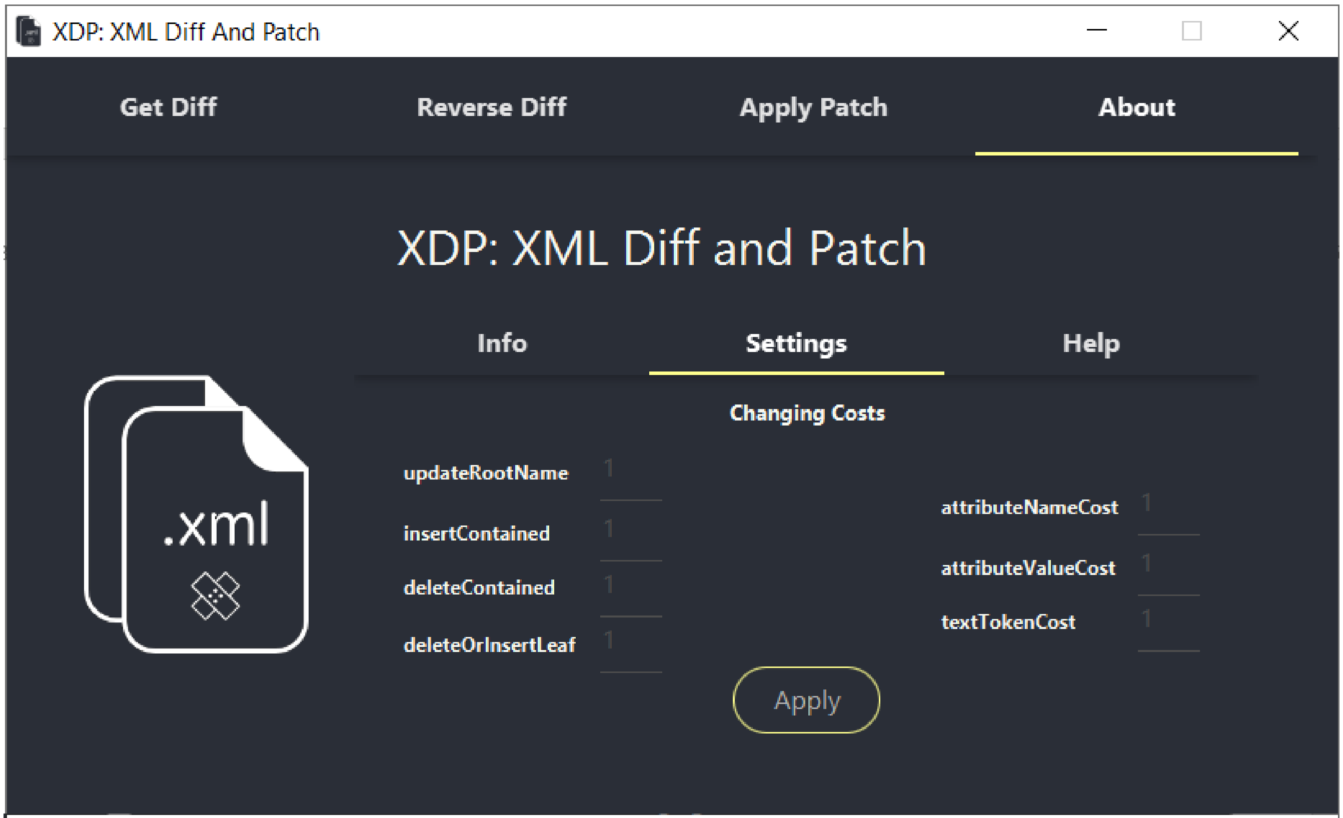


XDP: XML Diff and Patch is a a tool built in Java that compares two XML documents (structure and content).

This Application is part of a project in Lebanese American University, Course: Intelligent Data Processing and Applications (COE 543/743).

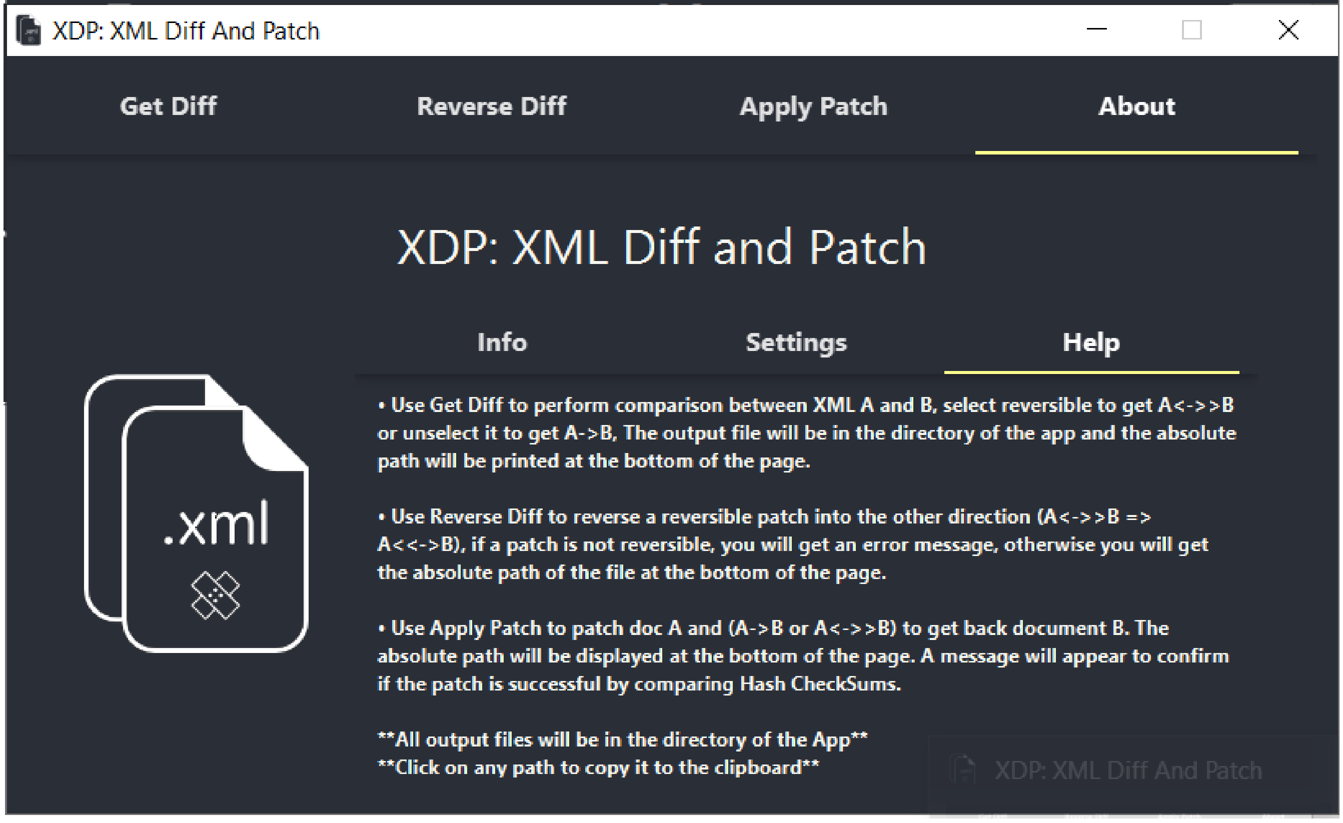
This Tool is built by Georgio Yammine and Rami Naffah.

https://github.com/georgioyammine/XDP_XML-Diff-and-Patching



Changing Costs

updateRootName	1	attributeNameCost	1
insertContained	1	attributeValueCost	1
deleteContained	1	textTokenCost	1
deleteOrInsertLeaf	1		



• Use Get Diff to perform comparison between XML A and B, select reversible to get A<->>B or unselect it to get A->B, The output file will be in the directory of the app and the absolute path will be printed at the bottom of the page.

• Use Reverse Diff to reverse a reversible patch into the other direction (A<->>B => A<<->B), if a patch is not reversible, you will get an error message, otherwise you will get the absolute path of the file at the bottom of the page.

• Use Apply Patch to patch doc A and (A->B or A<->>B) to get back document B. The absolute path will be displayed at the bottom of the page. A message will appear to confirm if the patch is successful by comparing Hash CheckSums.

All output files will be in the directory of the App
Click on any path to copy it to the clipboard