BioDIFF: An Effective Fast Change Detection Algorithm for Biological Annotations

Yang Song[1], Sourav S. Bhowmick[1,2], and C. Forbes Dewey Jr.[3]

[1] School of Computer Engineering, Nanyang Technological University, Singapore
[2] Singapore-MIT Alliance, Nanyang Technological University, Singapore
[3] Division of Biological Engineering, Massachusetts Institute of Technology, USA
assourav@ntu.edu.sg, cfdewey@mit.edu

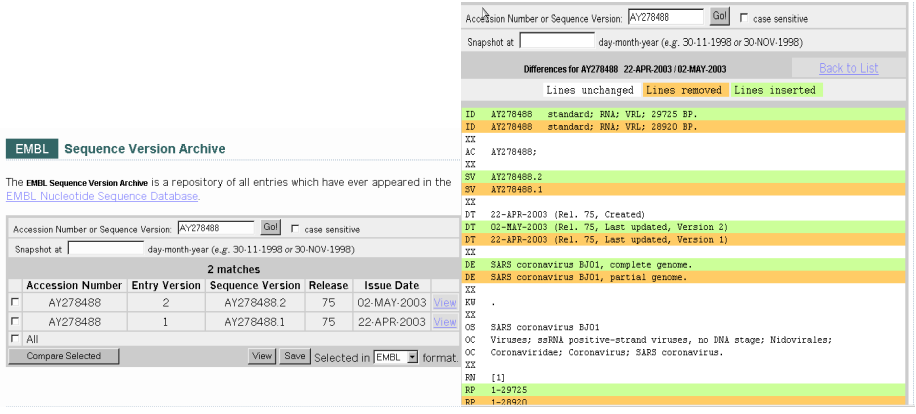
Abstract. Warehousing heterogeneous, dynamic biological data is a key technique for biological data integration as it greatly improves performance. However, it requires complex maintenance procedures to update the warehouse in light of the changes to the sources. Consequently, a key issue to address is how to detect changes to the underlying biological data sources. In this paper, we present an algorithm called BIODIFF for detecting exact changes to biological annotations. In our approach we transform heterogeneous biological data to XML format and then detect changes between two versions of XML representation of biological data. Our algorithm extends X-Diff, a published XML change detection algorithm. X-Diff, being designed for any type of XML data, does not exploit the semantics of biological data to reduce the data set of bipartite mapping. We have implemented BIODIFF in Java. We have conducted an extensive performance study using data from EMBL, GenBank, SwissProt and PDB. Our experimental results show that BIODIFF runs 1.5 to 6 times faster than X-Diff.

1 Introduction

Technological advances in high throughput screening coupled with the genomic revolution resulted in a large amount of life sciences data that are often stored in geographically distributed databases. Some of the key features of these databases are as follows [6]. (1) Many data sources are typically centered on one *primary* class of objects, such as genes, proteins, or DNA sequences. (2) The primary objects are furthered described by a set of nested fields, called *annotations*. Many of the annotations are text fields, such as description, functional annotation, source of biomaterial etc. (3) Databases heavily cross-reference each other. (4) Databases overlap in the objects they represent, storing sometimes redundant and sometimes conflicting data. As valuable information is scattered around over literally hundreds of these databases, a data integration system that can handle heterogeneous, complex, and geographically dispersed biological data sources is a key area of research in bioinformatics[8].

Biological data integration approaches can be broadly categorized into three types. The first approach provides a uniform Web interface to various databases and analysis tools[12]. These systems usually use CGI scripts or Java servlets to execute queries against databases, to call analysis programs, or to search file-based data repositories. The second approach focuses on formulating complex declarative queries that span

R. Kotagiri et al. (Eds.): DASFAA 2007, LNCS 4443, pp. 275–287, 2007.



(a) EMBL interface. (b) Change detection.

Fig. 1. Change detection in EMBL

multiple heterogenous databases[3]. These two approaches have several disadvantages. First, efficiency of the application may be choked by the slowest external data source or by communication latency in the execution of queries and programs. Second, we may not be able to run our applications at a time we wish because a needed external source is unavailable. Third, there is always the risk of unintended "denial of service" attacks on the original sources. Finally, as many biological sources have large number of errors, there is always the risk of running remote applications that are sensitive to certain errors that cannot be detected nor corrected on-the-fly.

The third approach addresses the above limitations by taking a *warehousing* approach [1,3,7]. The first step in this approach is to develop a unified data model that can accommodate all the information that is contained in the various source databases. The next step is to develop a mechanism that will fetch the data from the source databases, transform them to match the unified data model and then load them into the warehouse. The warehouse then can be used for answering any of the questions that the source databases can handle, as well as those that require integrated knowledge that the individual sources do not have.

The warehouse approach greatly improves performance [3]. However, as biological data is highly dynamic, it requires complex maintenance procedures to update the warehouse in light of the changes to the sources. This raises a number of practical problems [3]: (1) How can we detect that the underlying data sources have changed and what are these changes? (2) How can we automate the refresh process? (3) How can we track the origins or "provenance" of data? *In this paper, we focus on the first issue*.

An important aspect of any change detection problem is finding out exactly how the underlying data source has changed. In the case of biological databases, this is complicated by the fact that updates are typically propagated in one of the three ways [3,5]: (1) Producing periodic new versions that can be downloaded by the user community (2) Timestamping data entries so that users can infer what changes have occurred since

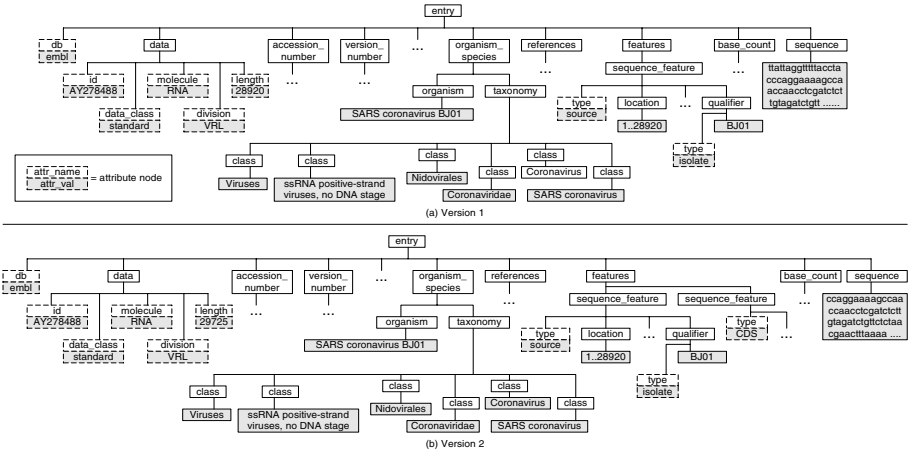


Fig. 2. XML representation of biological data (partial)

they last accessed the data (3) Keeping a list of additions and corrections; each element of this list is a complete entry. The list of additions can be downloaded by the user community. However, to the best of our knowledge, none of these methods precisely describe the minimal changes that have been made to the data. We illustrate this with an example.

Suppose that a warehouse stores a portion of EMBL data including data related to SARS (Severe Acute Respiratory Syndrome) virus. The EMBL data bank timestamps the data entries (Figure 1(a)) so that the warehouse maintainer can infer the latest version of the entry. The web site also provides a tool to compare the differences between two versions of SARS data by clicking the "Compare Selected" button in Figure 1(a). The differencing tool then highlights the changes by color coding the lines that are inserted, deleted, or remained unchanged during the transition as shown in Figure 1(b). The main drawback of this tool is that it does not exactly say how the entry has changed. The actual change may be very small. For example, consider the `ID` attribute in EMBL data. The general format of an `ID` line in EMBL is: `ID entryname dataclass; molecule; division; sequencelength BP`. From Figure 1(b) it is clear that only the sequence length is modified from "28920" to "29725". Values of the remaining attributes of the `ID` line are unchanged. However, the differencing tool in Figure 1(a) does not try to identify the exact change in the `ID` line. Rather, it represents the change as deletion and insertion of the `ID` line. Observe that the differencing tool represents an update of a line as a combination of deletion of the line followed by insertion of a new line (first two lines in Figure 1(b)). Assuming that the warehouse uses a relational database to store data, finding the exact change is important as it reduces the number of tables or tuples needed to be updated in the warehouse [3].

In this paper, we present an algorithm BIODIFF that can identify *exact* changes to the *annotations* associated with primary biological objects[1]. In the rest of the paper,

[1] A preliminary and shorter version of this paper appeared as a poster in [9].

we use the genomic and proteomic data sources as running examples to illustrate our change detection technique. However, as we shall see later, our approach is generic and can be applied to any biological annotations. Note that we do not discuss detecting changes to primary objects (e.g., protein sequences, nucleotide sequences) as the differences between two primary objects (say nucleotide sequences) can be computed using a modified sequence comparison algorithm instead of the *matching algorithm* used in BIODIFF for annotations. As there is a significant body of work on sequence comparison techniques, we do not focus our discussion on detecting changes to primary objects here.

In our approach, we first transform data (e.g., flat files) from various biological data sources to XML format using Bio2X [10]. Then, we can address the problem of detecting changes to biological data in the context of such XML documents. Consequently, BIODIFF takes as input two versions of XML representation of biological data and compute the changes. Since there are several recent efforts in the XML research community to develop change detection algorithm for XML documents [2,11], an obvious issue is the justification for designing a separate algorithm for detecting changes to the XML representation of biological data. In fact according to [3,5], XML change detection algorithms can be directly used to detect changes to XML representation of biological data. We argue that although such algorithms will clearly work for annotation data, they are not efficient as they do not exploit the *semantics* of biological data. For instance, the min-cost max-flow algorithm for computing the bipartite mapping between two XML trees is the most time consuming part in X-Diff [11]. Hence, it is desirable to reduce the size of data set during mapping. However, X-Diff fails to do so for biological data as it ignores the semantics of the XML elements. BIODIFF is developed to address this issue by extending X-Diff[11]. As we shall see later, it exploits the semantics of the XML elements to further reduce the data size for bipartite mapping. Consequently, our experimental results (Section 3) show that BIODIFF runs 1.5 to 6 times faster than X-Diff on genomic and proteomic annotations.

2 Algorithm

We have designed and implemented a wrapper called Bio2X that converts flat files into hierarchical XML form based on extraction rules[10]. We have observed that XML representation of data from many major biological sources can be considered as unordered. For example, consider the XML tree representation of two versions of EMBL data in Figure 2. Assume that the nodes <*class*>*Coronavirus*</*class*> and <*class*> *Nidovirales*</*class*> swap their positions in Figure 2. However, this change is not significant since the order does not influence the semantics of the biological data entry. Hence, in this paper we assume that an unordered XML model is more appropriate for representing biological data.

The pseudocode of BIODIFF algorithm is given in Figure 3(a) and can be best described by the following five phases: the *identifier checking phase*, the *type classification phase*, the *parsing and hashing phase*, the *matching phase*, and the *edit script generation phase*. The *identifier checking phase* takes as input two XML documents,

```
Input:  XML documents D1 and D2, DTD K
Output: Edit script E

     /* Identifier checking phase */
(1)  Set id1=identifier of D1;
(2)  Set id2=identifier of D2;
(3)  if (id1==id2) return E=no_change;
     /* Type classification phase */
(4)  if (TypeContainer C is empty)  C = ChooseType(K);
     /* Parsing and hashing phase */
(5)   parse D1 to XTree tree1;
(6)   XHash(T1);
(7)   parse D2 to XTree tree2;
(8)  XHash(T1);
     /* Matching & edit script generation phase */
(9)  for each node node1 in first-level elements of tree1  {
(10)   set name=tree1.GetName(node1);
(11)   set node2=tree2.GetNode(name1);
(12)   if (node2 == null)
(13)     add Delete(node1, tree1) to E;
(14)   else {
(15)     if type1(node1)
           E=OneToOne(node1, node2, E); /* Figure 3(c) */
(16)     else if type2(node1)
           E=IdenElemComp(node1, node2, E); /* Figure 4(a) */
(17)     else if type3(node1)
           E=IdenSignature(node1, node2, E); /* Figure 4(b) */
(18)     else E=bipartite(node1, node2, E);
(19)   }
(20) }
(21) for each node2 in tree2 but not in tree1
(22)   add Insert(node2, tree2) to E;
(23) return E;
```

(a) Main algorithm.

```
Input:  DTD K rooted at E
Output: TypeContainer C

(1)  if no sub-element of E
(2)    C={type_1};
(3)  for each sub-elements S of E {
(4)    add ChooseType(S) to C;
(5)    store <name, attributes> pair of S into M;
(6)  }
(7)  if M is not empty {
(8)    if each has distinct name
(9)    add type_1 into C;
(10) else {
(11)   if none has any attributes
(12)     add type_2 into C;
(13)   else if all has the same attributes
(14)     add type_2 into C;
(15)   else {
(16)     if the attribute is not an id
(17)       add type_3 into C;
(18)     else
(19)       add type_4 into C;
(20)     }
(21)   }
(22) }
(23) return C;
```

(b) Algorithm *ChooseType*.

Fig. 3. BIODIFF algorithm

new and old versions (denoted as D_1 and D_2), and determines whether they are identical. The equality of two XML representation of biological data can be concluded without parsing the entire XML documents. For instance, for genomic data sources, each biological data record has a *version identifier*. Whenever the data is changed, a new identifier will be assigned. So the identifiers of two data files can be extracted and compared first to determine whether the data files are identical. Similar identifiers can be identified for most of the important biological data sources. If the identifier checking phase detects that the two entries are not identical, then BIODIFF will parse the schema information of the documents (DTD/XML Schema) to classify the XML elements into four *types* depending on their structure. Such type classification information shall be used in the matching phase to minimize bipartite matching of the dataset. Note that if the DTD information is not available then it can be automatically generated from the XML documents using XTRACT[4]. We shall elaborate on this phase further later. Next, D_1 and D_2 are parsed into DOM Trees $tree1$ and $tree2$ in the *parsing and hashing phase*. The steps are similar to X-Diff[11] except for one key difference. Unlike X-Diff, when we parse the XML file, we encode the elements in the XML document with appropriate *type* of matching techniques based on the *type* information generated from the preceding phase. Note that if D_1 and D_2 contain primary objects (protein or nucleotide sequence) then they are excluded from parsing into nodes in the DOM trees. The goal of the *matching phase* is to compute the minimum cost matching between the DOM trees $tree1$ and $tree2$. We elaborate on this step later. In the *edit script generation phase*, we generate a minimum-cost edit script for changes to annotation data based on the minimum cost matching found in the matching phase. This step is similar to the one in X-Diff.

```
Input:  XTree tree1, XTree tree2, edit script E
Output: the edit script E

(1)   if hashcode(tree1)==hashcode(tree2) return E;
(2)   store the signatures of first-level elements of
         tree1 in HashMap1;
(3)   store the signatures of first-level elements of
         tree2 in HashMap2;
(4)   for each element in HashMap1 {
(5)      if (existing mapping element in HashMap2) {
(6)         if (it is an element node) {
(7)            if type1(node1)
                  E=OneToOne(node1, node2, E);
(8)            else if type2(node1)
                  E=IdenElemComp(node1, node2, E);
(9)            else if type3(node1)
                  E=IdenSignature(node1, node2, E);
(10)           else E=bipartite(node1, node2, E);
(11)        }
(12)        else {
(13)           compare their values;
(14)           if different
(15)              add Update(element1, element2) to E;
(16)        }
(17)     }
(18)     else
(19)        add Delete(element, tree1) to E;
(20)  }
(21)  for each element in HashMap2 {
(22)     if (no mapping element in HashMap1)
(23)        add Insert(element, tree2) to E;
(24)  }
(25)  return E;
```

(a) Function *OneToOne*.

```
Input: XTree tree1, XTree tree2, edit script E
Output: the edit script E

(1)  if hashcode(tree1)==hashcode(tree2) return E;
(2)  store the values of sub-elements of tree1
        in Vector1;
(3)  store the values of sub-elements of tree2
        in Vector2;
(4)  for each element value in Vector1 {
(5)     if (contained in Vector2)
(6)      mark unchanged;
(7)  }
(8)  for each element value in Vector2 {
(9)     if (changed) {
(10)       choose the next changed element in Vector1;
(11)       if (exist)
(12)          add Update(element1, element2) to E;
(13)       else
(14)          add Insert(element2, tree2) E;
(15)    }
(16) }
(17) for each element leaf in Vector1
(18)   add Delete(element1, tree1) to E;
(19) return E;
```

(b) Function *IdenElemComp*.

Fig. 4. BIODIFF algorithm (contd.)

2.1 Type Classification Phase

The min-cost max-flow algorithm for computing the bipartite mapping between two XML trees is the most time consuming part in X-Diff. Hence, a key goal of BIODIFF is to minimize bipartite mapping computation by exploiting the semantic relationship between various nodes in the XML tree. For example, the *data* element in Figure 2 contains only attribute values, whereas the *organism_species* element contains a list of subtree elements. Such differences in the structure of the subtrees are exploited in our approach to achieve this goal.

In this phase, we classify the XML elements into four different *types* based on their structures (we shall elaborate on these types later) by analyzing the DTD (or XML schema). As we shall see in the matching phase, instead of applying expensive bipartite matching for all cases, we apply four different matching techniques to the XML elements based on the types they belong to. Three of these matching techniques run in linear time in contrast to polynomial time complexity of bipartite matching.

The algorithm to classify the elements in the DTD to different types is shown in Figure 3(b). It takes as input the DTD of the XML representation of biological data and returns as output the *TypeContainer* C which contains information about different XML elements and corresponding types. The *ChooseType* function is invoked for each element in the DTD recursively and at each level, the names and attributes of all the subelements are examined to choose the type of the current element. Let us illustrate this with a simple example. Consider the subtree structure rooted at *organism_species* in Figure 2. *ChooseType* is first invoked for its subelement *organism* in the DTD, which is determined to be of *Type 1* since it has no subelements. *ChooseType* is then invoked for *taxonomy* and it is classified as *Type 2* since it contains a list of subelements having identical names (*class*). Finally, as the *organism_species* contains two distinct subelements, *Type 1* is chosen for this element.

```
Input:  XTree tree1, XTree tree2, edit script E
Output: the edit script E

(1)  set attrname=the names of key attributes;
(2)  for each first-level elements in tree1 and tree2
       include values of attrname into signature;
(3)  ta1=subtree containing nodes with distinct
       signatures in tree1;
(4)  ta2=subtree containing nodes with distinct
       signatures in tree2;
(5)  E=OneToOne(tb1, tb2, E);
(6)  tb1=subtree containing nodes without attributes
       in tree1;
(7)  tb2=subtree containing nodes with attributes
       in tree2;
(8)  E=IdenElemComp(tb1, tb2, E);
(9)  for each signature common to multiple elements {
(10)    tc1=subtree containing nodes with this
          signatures in tree1;
(11)    tc2=subtree containing nodes with this
          signatures in tree2;
(12)   E=IdenElemComp(tc1, tc2, E);
(13) }
(14) return E;
```

(a) Function *IdenSignature*.

```
<?xml version="1.0" encoding="UTF-8"?>
.....
        <coordinate section>
                <atom serial no="1" name="N" residue name="VAL"
                chain id="A" seq num="1" x-coord="-38.199"
                y-coord="-40.257" z-coord="97.510" occupancy="1.00"
                temp factor="22.53" />
                <atom serial no="2" name="CA" residue name="VAL"
                chain id="A" seq num="1" x-coord="-38.816"
                y-coord="-41.316" z-coord="96.669" occupancy="1.00"
                temp factor="18.15" />
        </coordinate section>
        ..............
```

(b) Element `coordinate_section` of PDB.

Fig. 5. BIODIFF algorithm (contd.)

2.2 Matching Phase

We now discuss the matching process in BIODIFF. Unless specified otherwise, in our following discussion we use the notion of signature as introduced by [11]. Observe that the first level element nodes (elements for brevity) in the tree representation of XML version of biological data have *distinct* structures (Figure 2). Each node has a unique name and hierarchy. Each node in the first level appears only once and mapping occurs only between nodes with the same signature. So the whole XML tree can be divided into a set of smaller subtrees rooted at each first-level node. Each smaller tree will be compared with another smaller tree from the second XML tree having the node with same name. For example, the subtree rooted at node labeled *organism_species* in Figure 2(i) will be compared with the subtree in Figure 2(ii) whose root is in the first level and has the same label. Note that this computation is independent from the remaining subtrees.

The above step alone does not provide performance improvement compared to X-Diff. X-Diff also achieves this with its signature definition. However, this step makes it possible to use *different* types of matching for *different* subtrees. Hence, we categorize the matching techniques into four basic types for both minimum-cost distance computation and minimum cost edit script generation. We discuss these techniques in detail now. Note that each subtree is treated independently for further matching. So any distance computation will be localized within each subtree.

Type 1: One-to-One Comparison. An element in the XML representation of biological data can be composed of a number of attributes, subelements or textual data. Some elements may exhibit unique signature within its subtree scope. For example, consider the *organism* element in Figure 2. Only one *organism* element can exist in the subtree rooted at *organism_species* element. So there will be at most one candidate-matching element in the newer version of the subtree. Hence, one-to-one comparison can be conducted between these two elements. Lines 1-2 in Figure 3(b) identifies these types while parsing the DTD.

Case 1: The element contains only text value: The corresponding element pair from the older and newer versions can be matched directly using one-to-one comparison. For example, the *version_number* element shown in Figure 2 contains only a string value representing the accession number and version number. So the two *version_number* elements in Figure 2 can be compared directly.

Case 2: The element contains distinct attributes: Each attribute should have a distinct name (signature). Each attribute can be matched with only a single attribute of the other XML file. The entire attribute list pair is then matched on a one-to-one basis. Take the *data* element in Figure 2(i) as an example. It contains the attributes *id*, *data_class*, *molecule*, *division*, and *length*. After obtaining the first attribute *id* in Figure 2(b)(i), the algorithm searches for attribute *id* in Figure 2(ii) in the element node *data* and compares their content. This is repeated for each attribute of *data*.

Case 3: The element contains a list of subelements with distinct names: One-to-one comparison is also sufficient to match the two lists. For example, consider the *dates* element in Figures 2(i) and 2(ii). Each *dates* element consists of two subelements: *created* and *updated*. These two subelements will appear at most once in this subtree rooted at *dates*. So given an element *created* in the older version of the subtree, there will be at the most one candidate-matching element in the newer version of the subtree. After matching the two *created* nodes, for example, further matching between the two subtrees of *created* is then computed recursively. The type of matching technique to be applied on the subelements depends on the specific structure again.

The pseudocode for one-to-one comparison is shown in Figure 4(a). It can be seen that with the utilization of a HashMap structure for storing elements, linear time complexity O($|T_1| + |T_2|$) is achieved, where $|T_1|$ and $|T_2|$ are the numbers of nodes in the subtrees T_1 and T_2 respectively. Note that although X-Diff only performs min-cost max-flow computation between the elements with the same signature, for a subtree with all unique signature subelements, the complexity will not reduce to O($|T_1| + |T_2|$). It still needs to enumerate all the pairs first to select the nodes with the same signature as it cannot know ahead of time that only one matching candidate actually exists.

Type 2: Identical Subelement Comparison. The first case discussed above is suitable for an element with a list of distinct subelements or attributes where each element or attribute must have a unique name within the scope of its subtree. Conversely, an element may contain a list of subelements having identical names. In this case, by following the X-Diff convention, if all the subelements have the same signature, then they will be compared using min-cost max-flow algorithm. However, as bipartite matching is an expensive procedure, we resolve this program by transforming a bipartite matching problem into linear-time matching whenever possible. Lines 10-14 in Figure 3(b) identifies the cases below while parsing the DTD.

Case 1: Elements without attributes: For example, the *taxonomy* element has a list of subelements with the same name *class*. Each of the *class* element does not contain any attribute. A *class* can be deleted, inserted, or updated. In our approach, we first find all the unchanged *class* pairs and then we assign all the unmatched *class* elements in the second subtree to unmatched elements in the first subtree sequentially and record

that they are changed. If a *class* element of the second subtree is left unpaired, then it is considered as newly inserted. Similarly, if a *class* element of the first subtree is left unpaired, then it is considered as deleted.

Case 2: Elements having identical attributes: Each element with identical name may have identical nonempty set of attribute name-value pairs. Consequently, this becomes identical to Case 1 where the elements are just identical by examining the titles of the nodes. Hence the same comparison technique will be applied to them.

Case 3: Elements of cases 1 and 2 containing subtree structures: The sequential matching becomes no longer valid since the matching pairs have to be computed by analyzing the entire hierarchy of the subtrees. A min-cost max-flow bipartite matching has to be carried out. However, such structure does not exist in our XML data formats generated by Bio2X [10]. Any element, which is not unique within the current scope and contains a subtree structure, will have at least one distinct attribute associated with it. Hence, such situation can be ignored.

The algorithm is shown in Figure 4(b). The complexity of this procedure is also linear in $O(|T_1| + |T_2|)$.

Type 3: Extended Signature Comparison. We now consider the case when identical subelements have different attributes (name or value). They can be text-value, attribute-value elements, or contain subelements. These two cases are defined by the Lines 16-17 in Figure 3(b).

Case 1: Each identical-name element has a distinct attribute: We can differentiate these elements by utilizing the attributes as identifiers. By incorporating the attribute name and value into the original signature, we generate unique signatures for each element within the current subtree. Hence, the matching problem will be reduced to a one-to-one comparison again as described earlier. For instance, the *qualifier* elements (Figure 2(i)) has a signature *entry/features/sequence_feature/qualifier/element* as defined by X-Diff. Observe that three of them have an attribute labeled *type*. Hence they can have the following distinguishable signatures: *entry/features/sequence_feature/ qualifier/ element/mol_type*, *entry/features/sequence_feature/ qualifier/element/isolate*, and *entry/features/sequence_feature/ qualifier/element/organism*. Consequently, the problem of matching an element in two versions of XML documents transforms into locating an element with the same refined signature. This is basically the one-to-one comparison algorithm described earlier. One *qualifier* element in Figure 2(i) has no *type* attribute. It will be matched with the other *qualifier* element without *type* (Figure 2(ii)). Multiple *qualifier* elements without attributes can exist in one subtree. Then they will be extracted and matched using the Type 2 technique. Also, if multiple *qualifier* elements have identical attributes, then they are also matched using Type 2 technique.

Case 2: Each identical-name element has multiple different attributes: The technique stated above is easily extensible to an element having multiple attributes. In this case, either any one of the attribute acts as the identifier or some of the attributes can be combined to act as the identifier. The attribute names are predefined for each XML element applying this matching algorithm. For example, consider the element <*database_reference*

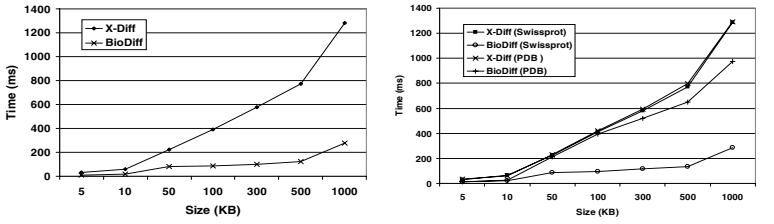
db="EMBL" primary_id="L09112" secondary_id="AAA96323.1" />, which has three difference attributes. The value of attribute *primary_id* or *secondary_id* is always unique within a database. Hence, attribute *primary_id* with *db* can be used as the identifier for this element. We do not choose *secondary_id* because it is an optional attribute. The choice of the key attribute or a combination of attributes depends on the biological context of the data, which is predefined in the biological databases.

The complete matching algorithm is outlined in Figure 5(a). After extending the signature, the elements will be matched using either Type 1 or 2 technique. Note that the complexity of the for loop (line (9) to (13)) is linear to number of nodes matched by Type 2 technique, which are subsets of T_1 and T_2. Hence, this matching technique also runs in linear time complexity: $O(|T_1| + |T_2|)$.

Type 4: Bipartite matching. The min-cost max-flow algorithm as used in *X-Diff* is used for elements that cannot be matched using the above three methods or a combination of them. This may occur when the attribute(s) is only an index for numbering the elements and the element pair with the same index may not match at all. For example, the *coordinate_section* of PDB database contains a list of *atom* elements (Figure 5(b)). The atom list records the names, locations and the atomic coordinates for standard residues. Each *atom* is identified by a *serial_number*, and the sequence of *atom* elements is determined by the order of the corresponding residue. Suppose in the newer version, an extra atom element is added for a newly inserted residue. The *atom* may be inserted in the middle of the atom list. When we compare the old and new version of *atom* list, the *atom* newly inserted will be matched to some *atom* in the old version, which is not correct. Hence, the alignment between the *atom* list of the old and new versions cannot be carried out reliably by matching the *atom* elements with the same *serial_number* (signature). Consequently, the only option left is to use min-cost max-flow bipartite matching. Note that this procedure requires $O(|T_1| \times |T_2| \times \max\{\deg(T_1),\deg(T_2)\} \times log_2(\max\{\deg(T_1), \deg(T_2)\}))$ complexity[11].

2.3 Complexity Analysis

Let $|T_1|$ and $|T_2|$ be the numbers of nodes in the two XML trees T_1 and T_2 respectively. Let n_1 and n_2 be the numbers of nodes requiring a min-cost max-flow matching algorithm. In the equality checking phase, the equality of two documents is determined by checking their identifier values. The time complexity to locate identifiers from two documents is $O(L_1 + L_2)$, where L_1 and L_2 are the numbers of nodes ahead of the identifiers. In the parsing and hashing phase, the time complexity to parse two documents and construct trees is $O(|T_1| + |T_2|)$. Hashing is performed during parsing. Since we need to sort child node XHash values before computing parent node XHash values, the upper bound of the complexity is $O(|T_1| \times \log|T_1| + |T_2| \times \log|T_2|)$[11]. In the matching phase different matching algorithm is applied to each subtree rooted at first-level node according to its node type. For Type 4 matching the time complexity is $O(n_1 \times n_2 \times max\{deg(n_1),deg(n_2)\} \times log_2(max\{deg(n_1), deg(n_2)\}))$. The time complexity for Type 1 to 3 matching techniques is $O(|T_1| + |T_2|)$. So the overall time complexity of the matching phase is $O(|T_1| - n_1 + |T_2| - n_2 + n_1 \times n_2 \times max\{deg(n_1),deg(n_2)\} \times log_2(max\{deg(n_1), deg(n_2)\}))$. It shows that the performance



(a) EMBL. (b) SwissProt and PDB.

Fig. 6. Performance study using genomic and proteomic data

primarily depends on the number of nodes in the files and the percentage of nodes that require Type 4 matching. When the changes between the two versions are not reflected in $|T_2|$ or n_2, the percentage of changes does not affect the performance as well, which is consistent with the experiments conducted in X-Diff[11]. Finally, for the minimum cost edit script generation phase, the minimum cost edit script is generated by traversing all nodes in the two trees. Hence, the complexity is $O(|T_1| + |T_2|)$.

Let us compare the time complexity of the matching process for nonsequence data in BIODIFF with X-Diff. As X-Diff requires about $O(|T_1| \times |T_2| \times max\{deg(T_1), deg(T_2)\} \times log_2(max\{deg(T_1), deg(T_2)\}))$ to match, the improvement of nonsequence element matching inBIODIFF over X-Diff can be estimated as$O(|T_1| \times |T_2| \times max\{deg(T_1), deg(T_2)\} \times log_2(max\{deg(T_1), deg(T_2)\})/(|T_1|+|T_2|-n_1-n_2+n_1 \times n_2 \times max\{deg(n_1), deg(n_2)\} \times log_2(max\{deg(n_1), deg(n_2)\})))$. Assuming $n_1 = n_2 = n$, $|T_1| = |T_2| = t$, and $\max\{\deg(T_1),\deg(T_2)\} \times log_2(\max\{\deg(T_1), \deg(T_2)\}$=x, the complexity comparisonbecomes $O(xt^2/(t - n + xn^2))$. For certain value of t, this percentage of speed up depends on the value of *n*. If $n = 0$ or $n = 1$, then the speed up is around $O(xt)$, which is the best-case performance. If $n = t$, then the speed up is equal to 1, which is the worst-case performance. If $n > 1$ and $n < t$, then the speed up is between 1 and $O(xt)$. In conclusion, the change detection performance of BIODIFF algorithm on annotations is always faster than X-Diff. However, the speed up depends on the percentage of elements utilizing the linear time matching methods.

3 Performance Study

BIODIFF is implemented in Java using J2SDK 1.4.0. Particularly, we investigate the impact of file size on response time of BIODIFF and X-Diff. We ran the experiments on a Pentium III 900MHz PC with 256 MB memory under MS Windows 2000 Professional. X-Diff is downloaded from http://www.cs.wisc.edu/∽yuanwan/xdiff.html.

The testing is done for GenBank, EMBL, Swiss-Prot, and PDB data separately. The size of the documents excluding the primary object (sequence data) ranges from 5 KB to 1 MB. As sequences occupy major chunk of space in the documents, we believe that 1 MB is large enough to represent annotations associated with primary biological objects.

Note that the execution time of X-Diff and BIODIFF algorithms only consists of the change detection time between the two XML trees. The preprocessing time, including the type generation and parsing of XML documents is not included in the performance evaluation. Note that preprocessing time depends only on the document and schema size and is not influenced by the algorithm efficiency. According to the complexity analysis, the proportion of nodes requiring Type 4 matching also affects the performance of matching phase. Hence, we choose data files with different proportion of tree structures that need Type 4 matching technique for each file size. The performance recorded is averaged from the testing results for each file size. For example, when testing BIODIFF on EMBL, Genbank, and SwissProt data, we took 10 files for each sample file size, with *references* elements occupying 1-10% of the entire file size. For PDB, we assumed 30-80% elements involved in Type 4 matching. Note that percentage of elements involved in Type 4 matching is chosen based on real life examples.

Figure 6 shows the results. We do not show the results on Genbank separately as its performance is similar to EMBL. It can be seen that BIODIFF outperforms X-Diff for all the four databases. As seen from the figures, X-Diff exhibits similar performance for different databases as it is a generic algorithm for all types of XML documents. Hence, its performance mainly depends on the number of nodes. BIODIFF, on the other hand, has different performance for each database since each database has different tree structures. If a database has more nodes that require min-cost max-flow matching algorithm, the improvement of BioDiff compared to X-Diff is less. For example, for Genbank, EMBL, and SwissProt, only the *references* element requires bipartite matching. Consequently, BIODIFF outperforms X-Diff 2 to 3 times for documents of size less than 100 KB. This increases to 6 times as the size of the documents increase to 1 MB. For PDB, *reference*, *coordinate_section*, and *secondary_structure* elements require bipartite matching. As number of nodes that requires min-cost max-flow matching algorithm for change detection is larger in PDB compared to Genbank, EMBL, and SwissProt, the execution time of BIODIFF on PDB is larger than the time taken for Genbank, EMBL, and SwissProt. Even then BIODIFF is almost 1.5 times faster than X-Diff for PDB dataset.

4 Conclusions

In this paper, we present an algorithm called BIODIFF for detecting exact changes to biological annotations. In our approach we transform heterogeneous biological data to XML format using Bio2X and then detect changes between two versions of XML representation of biological annotations. Our algorithm extends X-Diff [11], a published change detection algorithm for unordered XML. The min-cost max-flow algorithm for computing the bipartite mapping between two XML trees is the most time consuming part in X-Diff. BIODIFF addresses this limitation by exploiting the semantic relationship between various nodes in a subtree, attribute usage, presence or absence of optional elements, etc. Our experimental results show that BIODIFF runs 1.5 to 6 times faster than X-Diff.

References

1. Bahl,A. (2002) PlasmoDB: the Plasmodium genome resource. An integrated database that provides tools for accessing, analysing and mapping expression and sequence data (both finished and unfinished), *Nucleic Acids Res.*, **30**, 87-90.
2. Cobena,G., Abiteboul,S., Marian,A. (2002) Detecting Changes in XML Documents, *In Proc. of ICDE*, 41-52.
3. Davidson,S.,B., Crabtree,J., Brunk,B., *et al*. (2001) K2/Kleisli and GUS: Experiments in integrated Access to Genomic Data Sources, *IBM Systems Journal*, **40(2)**, 512-531.
4. Garofalakis,M.,N., Gionis,A., Rastogi,R., Seshadri,S., Shim,K.(2000) XTRACT: A System for Extracting Document Type Descriptors from XML Documents, *In Proc. of SIGMOD*, 165-176.
5. Hammer,J., Schneider,M. (2003) Genomics Algebra: A New, Integrating Data Model, Language, and Tool for Processing and Querying Genomic Information, *In Proc. of Conference on Innovative Data Systems Research (CIDR)*.
6. Leser,U., Naumann,F. (2005) (Almost) Hands-Off Information Integration for the Life Sciences, *In Proc. of CIDR*, 2005.
7. Ritter,O., Kocab,P., Senger,M., Wolf,D., Suhai,S. (1994) Prototype implementation of the integrated genomic database, *Comput. Biomed. Res.* **27**, 97115.
8. Stein,L.,D., (2003) Integrating Biological Databases, *Nature Rev Genet*, **4(5)**, 337-345.
9. Song,Y., Bhowmick,S.,S., (2004) BioDiff: An Effective Fast Change Detection Algorithm for Genomic and Proteomic Data. *In Proc. of ACM CIKM*(Poster), 146-147.
10. Song,Y., Bhowmick,S.,S., (2005) Bio2X: A Rule-based Approach for Semi-automatic Transformation of Semistructured Biological Data to XML, *Data and Knowledge Engineering Journal*, **52(2)**, 249-271.
11. Wang,Y., DeWitt,D., Cai,J-Y, (2003) X-Diff: A Fast Change Detection Algorithm for XML Documents, *In Proc. of IEEE ICDE* , 519-530.
12. Zdobnov,E.,M., Lopez,R., Apweiler,R., Etzold,T., (2002) The EBI SRS server-recent Developments, *Bioinformatics*, **18(2)**, 368-373.