Keywords: Wrapper Induction, XPath, Alignment, Data Extraction, DOM.

Abstract: Dealing with a huge quantity of semi-structured documents and the extraction of information therefrom is an important topic that is getting a lot of attention. Methods that allow to accurately define where the data can be found are then pivotal in constructing a robust solution, allowing for imperfections and structural changes in the source material. In this paper we investigate a wrapper induction method that revolves around aligning XPath elements (steps), allowing a user to generalise upon training examples he gives to the data extraction system. The alignment is based on a modification of the well known Levenshtein edit distance. When the training example XPaths have been aligned with each other they are subsequently merged into the path that generalises, as precise as possible, the examples, so it can be used to accurately fetch the required data from the given source material.

1 INTRODUCTION

Wrapper induction has been studied since the late nineties and has been an important topic ever since. It concerns creating a translation step between the semi-structured document and a data structure of choice, using manually annotated examples. The case we are focusing on is the extraction of information from web pages so it can be approached in a relational format.

There have already been numerous proposals as to how to approach this problem, ranging from semi-autonomous algorithms to completely manual approaches. In this paper we elaborate on a solution to the generation of the wrapper, which revolves around aligning the steps within the XPaths (see XPath syntax (xpa, 1999) and Section 3). Sample XPaths are aligned with each other, where the algorithm bases itself on the XPath specification as close as possible. Axis names, node tests and basic predicates are taken into account. Impurities in the source documents are properly dealt with, e.g., typo’s in the structure tags, structural differences ...

In this paper we limit our investigation to the child axis (with some exceptions for the introduction of, e.g., wildcards), basic node tests and one optional integer predicate, but as our approach is built around the actual XPath specification and not on the derived string based form, it will be expanded to accomodate more detailed and complex situations. After aligning the sample XPaths, they are merged with each other, leaving the user with a single merged XPath that encompasses the samples and generalises them in a meaningful way. This methodology was already put into practice in a solution which is currently used to crawl a large amount of websites comprising thousands of separate pageloads, resulting in a clean and comprehensible database.

The remainder of the paper is structured as fol-
flows. In Section 2, a brief overview is given of related work. Section 3 deals with some preliminary concepts that are utilised in the following sections. Our proposed technique is detailed in Section 4 and some results are given in Section 5, with future work listed in Section 6. Finishing the paper, conclusions are given in Section 7.

2 RELATED WORK

Wrapper generation can be categorised in two big trains of thought: the supervised algorithms (wrapper induction) that take user input to build the wrappers, and the unsupervised algorithms that do (mostly) everything on their own (automated approaches). Both have their merit, depending on the context, types of concerned documents and goals that need to be achieved.

(Semi-)automated approaches try to figure out the location of the data themselves, which makes them liable to make mistakes in the process. There are paths of research that focus on the analysis of visual features (Hao et al., 2011; Cai et al., 2004), try to identify repetitive patterns (Chang and Lui, 2001; Liu et al., 2003; Zhai and Liu, 2005) or take a set of pages out of which a template is distilled (Crescenzi et al., 2001; Arasu and Garcia-Molina, 2003). Some focus on specific features such as the HTML tag table (Wang and Hu, 2002) and table- and form-related tags (Liu et al., 2003). A further overview of automated extraction can be found in (Zhai and Liu, 2005) and others, as this is outside the scope of this paper.

The wrapper induction approach (Wong and Lam, 2010; Hsu and Dung, 1998; Kushmerick et al., 1997; Han et al., 2001; Sahuguet and Azavant, 1999; Myllymaki and Jackson, 2002; Anton, 2005; Cohen et al., 2002; Kushmerick, 2000; Muslea et al., 1999; Wang and Hu, 2002) is well extensively studied, with methods ranging from conditional random fields (Pinto et al., 2003) to tree-edit models (Hao et al., 2011).

Wrapper induction work that focuses especially on the use of item alignment techniques in the context of XPaths has been very sparse though, to our knowledge. A short reference to aligning XPaths by string alignment is given by (Sugibuchi and Tanaka, 2005), where it is used in a basic setting and does not take full advantage of what the XPath syntax has to offer. This reference was as well used by (Varun, 2011) in the system Silosear, again without any refinement added.

3 PRELIMINARIES

A couple of concepts need to be clear to be able to proceed to the core of the problem and our proposed solution. First of all, we briefly explain the string edit distance algorithm we base our alignment on, and secondly, we touch on the aspects of the XPath specification (xpa, 1999) that are used.

3.1 String Edit Distance & Alignment

Our method was originally inspired by the Levenshtein edit distance algorithm (Levenshtein, 1966) that allows calculating the minimal amount of operations (delete, insert, modification) that are needed to morph string $s_1$ into string $s_2$. Other algorithms that closely following the reasoning of Levenshtein exist, but for the sake of clarity and historical reasons we base our approach on Levenshtein. Briefly explained, the algorithm builds an edit distance matrix $d$ in which, at cell $d(i, j)$, the cost of transforming substring $s_1[i..j]$ into substring $s_2[i..j]$ is given.

For the base algorithm we refer to (Levenshtein, 1966). The cost of an insertion, deletion and substitution of a character are all set to 1. Jumps in the matrix can be seen as an edit operation: a vertical jump is an insertion character of string $s_2$, a horizontal jump is an insertion of the corresponding character of string $s_1$ and a diagonal jump is the substitution of a character of string $s_2$ into a character of string $s_1$. Each cell is calculated as the maximum of the cells above and aside of it, with the added cost of the corresponding edit operation.

A traceback can be calculated, based on $d$, indicating the path that leads to the calculated minimal edit distance. Multiple options are possible, it is up to the user to choose which he prefers.

Example 1 is given to illustrate the use of the edit distance matrix, as well as how a traceback might be constructed.

Example 1. To illustrate the Levenshtein edit distance calculation we use the following strings for $s_1$ and $s_2$:

- $s_1 := \text{wraptor}$
- $s_2 := \text{raptor}$

An example edit distance matrix $d$ is given below, with a basic cost 1 for each operation. In this case, the edit distance between $s_1$ and $s_2$ is two, as seen in the bottom-right cell (6,6) of the matrix. The example traceback (in bold font) indicates the insertion of the “w” of $s_1$ in the beginning and the removal of “t” of $s_2$, as is evident in the highlighted traceback.
3.2 XPath

XPath is a query language, defined as a standard by W3C, that allows to define expressions that look up data (text, atomic value, list of nodes ...) in an XML document. XPath 2.0 (xpa, 2010) is a superset of XPath 1.0 (xpa, 1999) (with some exceptions) and both give the user a powerful set of tools, some of which are necessary within the context of the paper and will be highlighted briefly in the following. XPath is widely spread and provides a solid tool for structured document investigation. We limit ourselves to examples to illustrate the language, for an in-depth explanation we refer to the W3C recommendation (xpa, 1999).

Model. An XPath expression consists of a number of steps, separated by a “/” character. Each expression can be evaluated within a given context (element of the document). If the expression starts with a “/” it indicates that it needs to be executed in the context of the root node of the XML document. A step consist of the axis it operates on, a node test and a number of predicates (e.g., `axisname::nodetest[predicate1][predicate2]`).

With regards to the axis, for our purposes, it suffices to know that it can be, amongst others, “child”, “self” and “descendant-or-self”.

The node test indicates what is selected according to the indicated axis. In our context this can be limited to `nodet()` (meaning: everything, because everything is a node) or the name of an HTML tag. In practice, we added additional predicates whenever a nodetest equalled to `nodet()`, to filter out text and other unwanted nodes, as we want to focus on the retrieval of elements.

Examples. It is easier to explain XPath expressions at a glance according to examples. The following should be sufficient to highlight some features of XPath that are useful to us and to make the rest of the paper clear as to what XPath is concerned.

- `/` selects the root node.
- `//child::div` selects all direct `div` children of the root node.
- `/child::node()` selects all children of the root node (nodetest wildcards), including comments, text nodes and processing-instructions.
- `/child::node()[1]` selects the first child of the root node.
- `/descendant-or-self::div` selects all `div` elements that are descendants of the root.
- `/descendant-or-self::tr/td[2]` selects every second `td` of every `tr` descendant of the root.

It is also possible to write abbreviations of the above syntax, which is what most people are familiar with. Again, we give some examples to show what is possible with abbreviations:

- `/*` is equivalent to `/child::node()` (with regards to our purposes anyway, the latter also selects `comment` and other nodes, but we remove those with additional predicates).
- `/child::node()/div` is equivalent to `//div`.
- `. ` indicates the current node, equivalent to `self::node()`.

As a final example for abbreviations we give both forms for the same XPath:


In this example the second `td` is selected that has a `tr` as a parent which has, as an ancestor, a `table`. That table is contained within a `div` that has, as an ancestor, a `body` which is contained in an `html` tag. That tag is contained within the root of the document (the XPath starts with `/`). Any `td` that is corresponding to these restrictions will be present in the resultset when the xpath is executed within a document.

4 PROPOSAL

We propose a method that takes a number of XPath examples that are manually annotated on a web page as input. These XPaths are then aligned, according to their steps, and merged with each other, resulting in a
single, generalised XPath. The main advantage of the step alignment approach is that we can handle errors and various structural differences in a flexible way, while staying within the syntax of the XPath standard.

For now we focus our attention on merging absolute paths and relative paths containing steps on the child axis, making it most useful for identifying recurring items on a web page (or, as said before, any other structured document fulfilling our requirements, but our recurring example use case remains a web page). This is typically useful for record detection (see, e.g., (Zhai and Liu, 2005; Zhu et al., 2005; Zhu et al., 2006)) and extraction of data from list pages in general: it can serve to extract the identifier for each record on a web page, but also to annotate recurring items within a record (subrecord).

4.1 Use Case

In Example 2 a list of people is shown on a web page. A typical use case would then be to highlight John and Frank, to serve as record identifiers. The XPaths of these elements (/div[1]/div[1]/div[1] and /div[1]/div[2]/div[1]) are subsequently aligned and merged (in this case, a trivial operation): /div/div/div[1]. Executing this XPath would retrieve John and Frank, but also the names of other persons defined in the div persons.

Example 2. In this example a snippet HTML structure is given for illustration purposes. It is a container with id persons, which encompasses a div for each person (record) that, in their turn, contain div elements (attributes of the record) with textual information.

```html
<div id="persons">
  <div class="person">
    <div class="name">John</div>
    <div class="telnr">0657/659012</div>
    <div class="hobbies">
      <div>Running</div>
      <div>Walking</div>
    </div>
  </div>
  <div class="person">
    <div class="name">Frank</div>
    <div class="telnr">3278/123278</div>
    <div class="hobbies">
      <div>Snooker</div>
      <div>Climbing</div>
    </div>
  </div>
  <div class="person">
    <div class="name">...</div>
  </div>
</div>
```

As can be seen in Example 2, it is also possible for records to contain lists of items, or other records in general. In this example, a list of hobbies is given for every person. When contained within a record, the XPaths of the annotations are created relative to the surrounding record’s identifier, in this case, the person’s name. The user could highlight two hobbies for John, resulting in the following XPaths: /div[1]/div[1]/div[3]/div[1] and /div[1]/div[1]/div[3]/div[2]. Merging them results in /div[1]/div[1]/div[3] which can be viewed in the context of the record’s identifier (name), creating a relative path ../div[3]/div. That path can be executed within the context of the identifier of Frank to retrieve his hobbies. We leave distilling the relative path (however trivial in this example) for another discussion, as this is heavily impacted by which elements of the XPath specification are allowed in the path.

In this instance we focus on merging absolute paths. If these paths should be made relative to another path (such as is the case with subrecords), the path is made relative after the merge.

4.2 Approach

Generalising XPaths can be used to describe the same elements as the example XPaths on which they are based, plus any element that exhibits the same structural similarities shared by the example XPaths. Dealing with the problem of generalising XPaths using alignment of steps allows for intuitive adjustments, according to the target use case. First of all, the XPaths are handled internally in non-abbreviated form, making the method robust, flexible and future-proof (other elements of the syntax can be considered analogously). In the following paragraphs the entire process is detailed, starting from the input XPaths to the result of the merged XPath.

4.2.1 Input XPaths

The input of the process is a set of n XPaths. These XPaths point to single examples (they resolve to one node) that need to be generalised. They are formed in an absolute way (they start at the root) and contain only child axis nodes (an assumption, as mentioned in Section 3.2, with exceptions to accommodate wildcards and alignment elements (see Section 4.2.2)). Each restriction as to which parts of the XPath syntax are used is put in place with the intention of investigating the process in a modular way, enabling us to focus on the basis.

4.2.2 Align Two XPaths

At the basis of the algorithm is the alignment of two XPaths (which is later generalised to n XPaths in
Section 4.2.3). Analogous to the Levenshtein algorithm (Levenshtein, 1966) (see Section 3), an edit distance matrix is built. A string character in the Levenshtein algorithm is being made equivalent to an XPath step. Custom costs for every operation are assigned, influencing the way the two XPaths are compared with each other.

As mentioned in the introduction, we consider basic steps (the XPath specification supports more complex elements) that contain axis name, node tests and optional single integer predicates. We limit this work to the child and descendant-or-self axis. The nodetest is typically a node name (e.g., div, table ...), where we also consider the nodetest node(), playing the role of a wildcard.

Building edit distance matrix \( d \) for XPaths \( x_1 \) (with \( s_1 \) steps) and \( x_2 \) (with \( s_2 \) steps) is done, analogously to Levenshtein, by initialising the matrix by setting the values in the first row and column to 0.

\[
d(i, 0) = 0, 0 \leq i \leq s_1 \\
d(0, j) = 0, 0 \leq j \leq s_2
\]

The rest of the indices is calculated using the following rule, for \( 0 < i < s_1 \) and \( 0 < j < s_2 \):

\[
d(i, j) = \min \begin{cases} 
  d(i-1, j) + tc(x_{i-1}, x_{2j}) \\
  d(i-1, j-1) + tc(x_{i-1}, x_{2j-1}) \\
  d(i, j-1) + tc(x_{i}, x_{2j-1}) 
\end{cases}
\]

where \( tc(y, z) \) is the transform cost of morphing step \( y \) into step \( z \), and \( x_{i,j} \) is the \( j \)th step of the \( i \)th XPath. This morph step is done according to the following set of rules, keeping in mind the restrictions we have built in ourselves, where the costs (shown as \( cost \)) can be chosen according to the application:

- **If** both steps are equal (equal axis, nodetest and predicate), use \( cost_e \).
- **Else**, if only the nodetest differs, use \( cost_n \).
- **Else**, if only the integer predicate differs, use \( cost_p \).
- **Else** use \( cost \), the two steps are treated as unequal.

After building \( d \), the traceback path through the matrix shows which operations led to the optimal result. Choices can be made in this step, whenever equivalent moves are encountered, that result in different merged XPaths. Currently, we make a naive assumption with regards to this choice: a diagonal, left and upwards moves in the matrix are preferred to each other respectively. In any case, all elements of the XPaths stay preserved: the traceback only serves to insert alignment elements to facilitate the alignment (the equivalent of the alignment character \( "*" \) described in Section 3). This ensures that both XPaths that served as input can be described, at minimum, by the eventual merged path. Inserted elements are neutral steps (self::node() or ., in abbreviated form), making sure the XPaths keep their semantic meaning after alignment.

### 4.2.3 Aligning \( n \) XPaths

In Section 3, the Levenshtein edit distance was briefly explained. This method is used to calculate the edit distance between two strings, whereas we need to be able to combine more than two. Typically (Gusfield, 1997), this is done in an iterative fashion, because of the exponential complexity of an \( n \)-dimensional implementation of the edit distance matrix. We employ, as well, an iterative approach.

For \( n \) XPaths as input, edit distances are calculated for each pair. The closest pair is aligned. Next, the on-average closest XPath to the aligned pair is taken and aligned in its turn. Alignment steps (insertions, deletions, substitutions) are performed for each of the XPaths simultaneously, resulting in three XPaths that have the same length and are aligned to each other. This process proceeds iteratively until all XPaths are aligned (see Example 3). Note that the algorithm manages to cope with a reasonable amount of error.

**Example 3.** In this example, the following three XPaths serve as input for the alignment algorithm:

\[
//body/div[1]/table[1]/td[1] \\
//body/table[2]/tr[2]/td[1]/a \\
//body/div[1]/table[1]/tr[2]/td[1]/a \\
\]

They were subsequently aligned with each other:

\[
\]

Missing and incorrectly named tags have been accounted for and steps for which only the predicates differ have been aligned with each other. These aligned XPaths are now ready to be merged into each other. All XPaths are shown in abbreviated form due to space constraints, although it should suffice for illustration purposes.
4.2.4 Merging XPaths

Merging aligned XPaths (that are, by definition, of equal length) in a coherent way can become a difficult process according to which items are allowed in the XPaths. In our case, we limited ourselves to 

\textit{child} (regular elements), \textit{descendant-or-self} (for wildcards resulting from inserted steps) and \textit{self} (inserted steps) axis steps, allowing us to implement a naive approach. As the aligned XPaths are of equal length \(n\) we can say that, for step index \(i, 0 \leq i \leq n\), all \(i^{th}\) steps can be considered together and merged into one step, according to the following rules:

- **Axis Name**
  - If all axes are the same, keep the axis-name.
  - Else, if at least one of the steps is // or if there is a mixture of neutral (.) and non-neutral steps, use descendant-or-self.
  - Else use descendant-or-self (a fallback, which is for the moment not being used, as each case falls within the top two cases due to assumptions made, but is mentioned for completeness).

- **Node Test**
  - If all node tests are equal, keep the same in the merged step.
  - Else, if at least one of the steps is // or if there is a mixture of neutral (.) and non-neutral steps, use node().
  - Else use node test node() in the merged step. This case handles unequal node tests and make sure that future, less naive, implementations have a fallback.

- **Predicate**
  - If all predicates for all steps are equal to each other, keep the original predicates in the merged step. This entails the most basic case: that each step has exactly one predicate and that those predicates are all integers. If those integers are equal, they are kept.
  - Else, if one of the steps has zero or more than one predicate, set no predicates in the merged step. Advanced predicate handling is kept for future work and is no trivial task.
  - Else, remove the predicates, which effectively introduces a predicate wildcard.

In Example 4, a couple of XPaths are merged with each other and the result is given as illustration. The example was kept as short as possible without losing accuracy and was chosen to illustrate the different errors that can occur when dealing with HTML source material.

*Example 4.* In this example, the aligned XPaths shown in Example 3, repeated below for convenience;

```
```

are merged with each other, using the rules set out in Section 4.2.4, resulting in the following XPath:

```
//body/table/*//
```

Missing steps have been replaced by descendant wildcard steps, tag typo’s have been corrected by introducing node test wildcards and changes in predicates have also been adequately corrected.

4.2.5 Fine-tuning

There are a number of things that can be fine-tuned to influence the execution of the algorithm and the way the XPaths are aligned and merged. In this paper we focus ourselves on what seems to be the most important of choices: the weights (or costs) that are used when building the edit distance matrix. We refer to Section 5 for tests that investigate these parameters.

5 RESULTS

In this section we briefly talk about the method we used to construct a rudimentary dataset to test our framework, after which we explain the methodology of the tests themselves. We conclude with some results pertaining to various properties of the algorithm.

5.1 Test Data

As our main use case revolves around extracting HTML data we focus our attention on the underlying data structure of HTML web pages: the DOM tree. A method was devised that generates random DOM documents. These DOM documents all have \textit{target} elements (annotated in the DOM with an attribute), representing the \textit{ground truth} that we wish to extract from the document.

There are two possible wrappers for these elements: either they are wrapped in a single element, or they are part of a list. We defined a couple of possibilities to list items: \textit{a div} containing a number of \textit{divs}, a \textit{table} containing rows and columns of data and \textit{ul} elements that represent unordered lists of \textit{li} items.

**Contents.** The contents of the DOM tree are determined in a random fashion. The tree is created by
generating a list $e$ of items and lists, as described above, where items have a higher chance of being generated (80%). An item is simply a container to contain the next element in $e$, whereas a list expands into multiple containers, thus branching the DOM tree, where each container contains the next element in $e$. The result of these operations is a DOM tree of varying depth, containing at least one list (this is enforced, otherwise the tree is a simple and singular path), with substeps represented by regular items. The leaf elements will be the targets (annotated by the attribute mentioned above), containing a text node.

Impurities. Impurities are introduced by chance. So far, two types have been used. Firstly, there is a certain chance (20% for each item, for the reported results) that, whenever an item is added to the DOM document, another element is wrapped around it. This makes the depth of each tree variable with regards to each separate leaf. Secondly, there is a certain chance (5% per character, for the reported results) that characters of each tag are morphed into another, characterizing typo-like errors.

About the Test Data. The way this test data is randomly generated represents a number of errors that can occur in real-life test sets. Due to the fact that a lot of possibilities are encountered it allows testing for combinations that would occur only rarely in real life, making the test a rather extensive one. The test data generation will be expanded in the future, for now it suffices for our purpose, which is highlighting the viability of our algorithm.

5.2 Methodology

For testing, we followed a couple of steps to investigate which settings had which effect. First of all, a random DOM document $doc$ was generated. This $doc$ contains target elements, whose XPaths were taken as ground truth. Following this, a large number of combinations of those XPaths were taken (combinations of 2, 3, 4 ... XPaths). Each of these combination was aligned and merged into an XPath which was evaluated against $doc$, resulting in a number of elements. In a perfect world, these elements should correspond exactly to the target elements, but as there are impurities in the dataset this was usually not the case: the more XPaths we merge into each other, the better the result we would expect to be.

5.3 Properties

Input XPath Amount. For the edit distance matrix, the following default costs (see Section 4.2.2) were used:

$\begin{align*}
&\text{cost}_n = 0 \\
&\text{cost}_p = 2 \\
&\text{cost}_\ne = 1 \\
&\text{cost}_n = 1
\end{align*}$

Random DOM documents were generated and combinations of size $2 \leq n \leq 10$ were allowed (2 is the lower limit for obvious reasons, and 10 seemed to be a reasonable upper limit out of experience with a practical application (mentioned in Section 1)). Results for default settings are shown in Table 1, which show a rise in average recall when we increase the combination amount, which was to be expected as explained above. Precision remains stable at $\pm 100\%$, indicating we do not introduce erroneous elements in our parse. It is clear that the amount of input XPaths has an appreciable impact on the quality of the resulting merged XPath.

Optimizing the recall remains a question of which settings give the best result for which type of document, which is investigated in the following paragraph.

Edit Distance Matrix Parameters. For the four costs ($\text{cost}_n$, $\text{cost}_p$, $\text{cost}_\ne$, $\text{cost}_n$) defined for the edit distance matrix (see Section 4.2.2) a new test was devised. With a high amount of repetition, random DOM documents were generated, as was the case in the previous paragraph. $\text{cost}_n$ was kept at 0 (for obvious semantic reasons and to reduce test complexity) and the other three parameters were varied between 0 and 4. For these settings, the latter three accounted for 54% of all variation in the recall value, when considering test cases of three example XPaths.

Best results were achieved with the following hierarchy between the three varied costs (and $\text{cost}_n = 0$):

$\text{cost}_p > \text{cost}_\ne > \text{cost}_n$
The spread between the best and the worst result in these tests was between 63% and 85%, for a fixed amount of example XPaths of 3, highlighting the importance of the edit distance matrix settings.

5.4 Performance

For this paper, there was no heavy focus on the performance of the algorithm. However, in this section, a couple of metrics can be found that illustrate the speed of the algorithm with regards to the different steps that need to be taken.

Figure 1 shows the average merge time taken, for different amounts of XPaths merged, and different lengths of resulting merged XPaths. In Figure 2 the average align time is shown. The merge operation takes appreciably less time than the align operation, which is to be expected. Each operation still completes under 3ms, a maximum only attained when dealing with 8 examples and a large XPath. Note that these tests were performed by a Java implementation on laptop hardware.

6 FUTURE WORK

A large amount of questions still arise when dealing with the use of XPath step alignment for wrapper induction. For one, the edit distance matrix can offer a lot more flexibility when considering other types of XPath syntax elements. Costs can be tweaked and expanded upon. The generated dataset that was used needs to be improved to more accurately reflect real-life situations, also with the goal in mind to make it freely available and to use this method to objectively compare with others. Other elements of the XPath syntax need to be incorporated, allowing for more complex align and merge operations, beyond what was introduced here.

7 CONCLUSIONS

In this paper we investigated how an in-depth application of XPath step alignment techniques could be used in the context of wrapper induction. A framework was constructed to facilitate testing (which was also used in production systems with satisfactory results). A couple of preliminary tests were run, showing dependencies of the correctness of the results on some of the parameters that are possible, as well as the amount of training examples (which was expected). We have shown that aligning XPaths on a semantic level with the goal of generalising examples is feasible and worthwhile, and we have introduced a rudimentary system that generates DOM documents to serve in algorithm testing environments, which is bound to be useful as a comparison tool in the nearby future.

REFERENCES


