Searching Semi-Structured Data Using Landmarks

Andrew Davison

Dept. of Computer Engineering
Prince of Songkla University
Hat Yai, Songkhla 90112,Thailand
Tel: +66 74 287 379

E-mail: ad@fivedots.coe.psu.ac.th

ABSTRACT

This paper introduces landmark search operators for extracting data from poorly formatted Web pages, plain text files, and XML/SGML documents lacking grammars. The emphasis is on ease of use, and a fast, simple implementation, which can be readily ported to a wide variety of host languages. There are two main operators: one using unique textual landmarks to divide text regions into smaller regions suitable for further search, and an operator that searches for XML/SGML tag pairs, and returns the matches as regions. An iterator class allows a search to be carried out repeatedly.

1. INTRODUCTION

Our aim is to create a simple, efficient set of methods for document search and information extraction based on finding *landmarks* in semi-structured data. Semi-structured data includes: Web pages marked up with (often incorrect) HTML, ASCII files, and XML documents lacking schema.

Although grammars exist for HTML (e.g. XHTML [7]), the sad reality is that most Web pages would fail a parsing test. Nevertheless, pages returned from the same service often exhibit similar formatting, (e.g. book pages from Amazon, sale item pages from eBay, departmental home pages). This is either because the pages are generated dynamically by server-side scripts using common templates, or because the host organization requires certain information to appear in certain places on a page. In other words, although the pages may not be grammatically correct, they do contain unique formatting information, such as titles, section heading, and indenting.

An ASCII file is often treated as a stream of characters (or bytes), as typified by UNIX tools. However, text files also contain formatting elements, such as titles, headings, and indenting. In common with Web pages, ASCII files developed for a single site (e.g. a library's book catalog, a school's class timetables) utilize common layout rules.

The general point is that semi-structured data which fails grammar or scheme validation, or even lacks a grammar, can still be searched by using the data's formatting elements. These elements may be recurring strings (e.g. "Section" at the start of each section), or patterns of white space (e.g. two newlines at the end of each 'paragraph'). We call these elements *landmarks*.

Although landmark search is aimed at data extraction from Web pages and text files, it's also useful for XML/SGML files. The markup tags can be treated as landmarks, allowing landmark search to be employed instead of grammar-based techniques.

In section two, we introduce the basic landmark operations. Section three contains examples showing how landmark search can be applied to Web pages, text files, and XML documents. Section four compares our work with region algebras, regular expressions, and XQuery. Section five draws some conclusions, and indicates future research directions.

A prototype implementation of landmark search in Java (together with examples) can be downloaded from http://fivedots.coe.pau.ac.th/~ad/landmarks.

2. LANDMARK SEARCH OPERATIONS

The two basic search operations are match3() and tagMatch4(). They both treat a document (be it a Web page, text file, or XML document) as a region. match3() utilizes landmarks patterns to search through the region, extracting smaller regions delimited by the matching landmarks. tagMatch4() performs a similar kind of search but using XML-style tag pairs. A support class, called RegionIterator (a Java iterator), can employ match3() or tagMatch4() to search repeatedly through a region for matches.

match3(), tagMatch4(), and RegionIterator are described in more detail below.

2.1 The match3() Operation

match3() carries out a linear search from the start of a region to find a landmark that matches the operation's start landmark pattern. The search then switches over

to looking for a landmark that matches the operation's end landmark pattern. The result is a region separated into three parts: the left, matching, and rest regions. The matching region is usually of most interest since it lies between the start and end landmarks.

The search result is shown graphically in Figure 1.

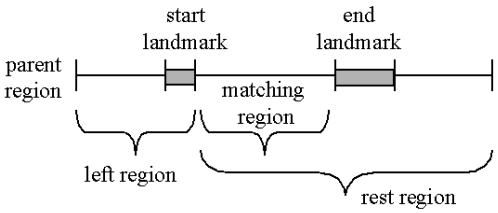

Figure 1. A Landmark Search using match3().

Landmark searches can be applied to the component regions, to 'zoom in' on areas of interest. When the information has been located, the region can be converted into a string, and manipulated using operations available in the host language.

match3() can be defined more formally:

pr.match3(slp, elp) returns {lr, mr, rr}

where
pr is $\{a_0..a_i, sl_0..sl_k, b_0..b_l, el_0..el_m, c_0..c_n\}$,
the parent region,

slp matches $sl_0..sl_k$, the start landmark,
and does not match anything in $a_0..a_i$,
slp is $slp_0..slp_k$, the start landmark pattern,

elp matches $el_0..el_m$, the end landmark,
and does not match anything in $b_0..b_l$,
elp is $elp_0..elp_m$, the end landmark pattern,

lr is $\{a_0..a_i, sl_0..sl_k\}$, the left region,
mr is $\{b_0..b_l\}$, the matching region,
rr is $\{b_0..b_l, el_0..el_m, c_0..c_n\}$, the rest region

Otherwise pr.match3(slp, elp) returns null.

A region is treated like a character sequence when being searched. The ".." symbol denotes a subsequence of characters.

The meaning of the "matches" operation will vary depending on the implementation; our Java prototype uses string equality: x..y matches s..t if the two subsequences contain the same text.

As the definition suggests, match3() can use string operations to perform a linear search over the parent region. The algorithmic complexity depends on the length of the sequence, and the cost of the "matches" operation. In general, if the landmark patterns are sufficiently small, then the algorithmic cost is on average O(*n*), where *n* is the length of the parent region.

Our implemented match3() operation is a little more general than the one outlined here. It is possible to specify that matching only succeeds when the m^{th} suitable start landmark is located in the parent region (rather than just the first, as here). Also the chosen end landmark can be set to be the n^{th} one found, counting from the start landmark's position.

2.2 The tagMatch4() Operation

tagMatch4() searches for tag pairs (e.g. <P> and </P>). On the face of it, this operation seems unnecessary since the tags could be represented by landmarks, and so found using match3(). However, there are two reasons for employing a separate method.

The first is that a start tag (e.g. <title>) may contain attributes, and we want to record them in an attribute region. The other reason is that tags may be nested (e.g. a <ul> list may appear as an item inside another <ul> list). The search ignores nesting, and finds the end tag which is at the same 'level' as the start tag.

tagMatch4() carries out a linear search from the start of a region to find a start tag that matches a supplied tag pattern. If the start tag contains attributes, these are stored in an attribute region. Then the search switches over to looking for an end tag that matches the tag pattern, with the proviso that the end tag must occur at the same level as the start tag.

The result is a parent region separated into four smaller regions: the left, matching, rest, and (potentially empty) attribute region.

Figure 2 shows the search graphically.

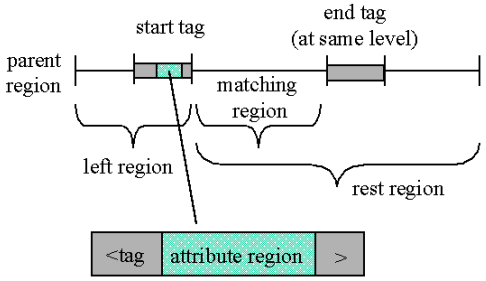

Figure 2. A Tag Search using tagMatch4().

Any of the four regions can be searched further by applying match3() or tagMatch4() to it.

tagMatch4() can be defined more formally:

pr.tagMatch4(tag) returns {lr, mr, ar, rr}

where
pr is $\{a_0..a_i, st_0..st_k, b_0..b_l, et_0..et_m, c_0..c_n\}$,
the parent region,

("<"+tag+">" matches $st_0..st_k$, the start tag,
and ar, the attribute region, is null) or
("<"+tag+" "+ $ar_0..ar_p$ +">" matches $st_0..st_k$,
and ar is $\{ar_0..ar_p\}$),
tag is $t_0..t_r$, the tag pattern,
the tag pattern does not match anything in $a_0..a_i$,
counter = 0,

startTag is "<"+tag+">" or "<"+tag+" " and
endTag is "</"+tag+">",

counter += count of startTag matches in $b_0..b_l$ –
count of endTag matches in $b_0..b_l$,

endTag matches $et_0..et_m$, the end tag,
and counter == 0

lr is $\{a_0..a_i, sl_0..sl_k\}$, the left region,
mr is $\{b_0..b_l\}$, the matching region,
rr is $\{b_0..b_l, el_0..el_m, c_0..c_n\}$, the rest region

Otherwise pr.tagMatch3(tag) returns null.

As in match3(), "matches" uses simple character comparisons between the tag pattern and the start and end tags.

The notion of level is captured with a counter which records the number of start and end tags encountered between the matching start tag and its corresponding end tag. The counter is incremented when another matching start tag is identified, and decremented when an end tag is encountered. The matching end tag is at the same level as the original start tag when the counter returns to 0.

The complexity of tagMatch4() depends on the length of the sequence, and the cost of the "matches" operation. The tag patterns are sufficiently small that the algorithmic cost is on average O(*n*), where *n* is the length of the parent region.

Our implemented tagMatch3() operation is more general than the one outlined here. It is possible to specify that successful tag matching only occurs when the m^{th} suitable starting tag is located in the parent region.

2.3 The Sequence Class

The implementation of the regions utilizes a Sequence class to reduce network and memory load.

The top-level region for the entire document is filled with text via a Sequence object which manages the downloading of the Web page or the reading of the local file. The Sequence object reads the text from the Web or local file a line at a time, and only reads a line when prompted by the region. This 'lazy' behaviour means that only as much of the Web page or file is loaded as is necessary to perform a search.

2.4 The RegionIterator Class

The landmarks operations are not intended to be a complete notation or language for text extraction. The host language is expected to have the usual control structures for looping, switching, and recursion, and (rudimentary) support for strings.

For example, repeated search through a parent region, looking for every matching region, can be coded using a while-loop and match3() (or tagMatch4()). A fragment of pseudo-code illustrates the idea:

```
Region r = /* region to be searched */
String slp, elp =
   /* start and end landmark patterns */

while (r.match3(slp, elp)  ==
    {left, matching, attribute, rest})  {
  // use matching region...
  r = rest;  // examine rest of region
}
```

However, searching for all the regions that match landmark (or tag) patterns is such a common task, that we have packaged it up inside a RegionIterator class. Several examples of its use appear in the next section.

3. SEARCH EXAMPLES

This section contains three Java examples using the landmark operations: details are displayed about a specified `Amazon.com` book, price information is extracted from a text file of airline fares, and statistics are collected from an XML version of *Macbeth*.

The complete code for all these examples, and some others, can be downloaded from `http://fivedots.coe.pau.ac.th/~ad/landmarks`.

3.1 Extracting Amazon Book Details

Given a book's ISBN number, details about its title, prices, reviews, and sales ranking are extracted from the `Amazon.com` page for the book, and printed to standard output. Typical output is:

```
Retrieving Amazon's page for 0596007302
Accessing URL:http://www.amazon.com/exec/
obidos/tg/detail/-/0596007302
Title: Amazon.com: Books: Killer Game
Programming in Java
List Price: $44.95; Amazon Price: $29.67
Star Rating: 4-5; No. of Reviews: 3
Sales Rank: 6,236
```

The top-level region for the book's Web page is created, then searched in various ways:

```
String AMAZON_URL =
 "http://www.amazon.com/exec/obidos/tg
  /detail/-/";

System.out.println("Retrieving
          Amazon's page for " + isbn);
Region topR =
     new Region(AMAZON_URL + isbn);

System.out.println("Title: " +
          topR.tagMatch("title"));
showPrices(topR);
```

```
showReviewInfo(topR); // see below
showRank(topR);
```

The title is obtained by searching for the tag pattern "title". This was determined by looking at the source code for several Amazon book pages, and noting that their titles were always wrapped in a "title" tag pair.

The version of tagMatch() used here only returns the matching region, which is cast to a string in println() by Region's toString() method.

Amazon summarizes review details with a star rating (out of 5), and the number of reviews. This part of a book's Web page has the format:

```
<img src="http://.../common/customer-
reviews/stars-4-5.gif"
height="12" border="0" width="64" />
based on 3 reviews.
```

The long URL always ends with "common/customer-reviews/" and a GIF file for the stars image. The surrounding text contains many tables, links, comments, fragments of JavaScript, and white space. The search can ignore all of this by focussing only on the unique landmarks in the source fragment:

```
private void showReviewInfo(Region r)
{
  Region reviewsRegion =
      r.match("common/customer-reviews/",
              "review");
  Region starsRegion =
    reviewsRegion.match("stars-",".gif");
  Region numReviewsRegion =
    reviewsRegion.match("based on "," ");

  System.out.println("Star Rating: " +
      starsRegion + "; No. of Reviews:" +
      numReviewsRegion);
}
```

reviewsRegion is created with match(), which returns the matching region between the two landmark patterns. For the example above, reviewsRegion will be:

```
{stars-4-5.gif"
height="12" border="0" width="64" />
based on 3 }
```

There is a space after the '3' at the end of the region.

starsRegion gets {4.5} from reviewsRegion, while numReviewsRegion extracts {3}.

The approach used by showReviewInfo() is quite common: first get *fairly* close to the required information with a region that cuts away *most* of the irrelevant data. This region should utilize landmarks which are unique across the entire document. The data is obtained in the second stage, using local landmarks next to the information, which only have to be unique within the region (e.g. in reviewsRegion).

3.2 Looking for Airfares

We want to retrieve the cost of a roundtrip flight between two cities from "airfares.txt". The information for a city is formatted like the following example:

```
Roundtrip Fares Departing From BOSTON, MA
To
----------------------------------------
       $209    INDIANAPOLIS, IN
       $189    PITTSBURGH, PA
```

The collection of roundtrip fares for a city start with the "Roundtrip Fares" heading, a dotted line, then multiple price lines. The lines end with two newlines before the next city collection.

The first step is to iterate through the city information until we find the desired 'from' city:

```
Region topR = new Region("airfares.txt");
showTripPrice(topR, "PHILADELPHIA",
                    "PITTSBURGH");
```

```
private void showTripPrice(Region topR,
                 String from, String to)
// show price of flight from-->to
{
  RegionIterator tripRegions =
       new RegionIterator(topR,
                 "Roundtrip", "\n\n");
  // iterate through the trip regions
  while (tripRegions.hasNext()) {
    Region tripRegion =
       (Region) tripRegions.next();
    Region fromRegion =
        tripRegion.match("From ", ", ");
    if (fromRegion.contains(from)) {
      // this trip is about <from>
      showToPrice(tripRegion, from, to);
      return;
    }
  }
  System.out.println("No fares found
                     from " + from);
}  // end of showTripPrice()
```

The RegionIterator repeatedly searches for the landmarks "Roundtrip" and "\n\n" which delimit the collection of roundtrip fares for a city. Each call to next() returns the next collection, storing it in tripRegion. The 'from' city is extracted by pulling the region between "From " and ", " from tripRegion,. This corresponds to {BOSTON} in the example above. If this is the desired city then showToPrice() looks at each price line to find the city we are interested in. This requires another RegionIterator:

```
RegionIterator priceRegions =
   new RegionIterator(tripRegion,
                "$", ", ");
```

This iterator extracts the price and 'to' city information from each price line. For the fares table above, toRegions will deliver `{209 INDIANAPOLIS}` and `{189 PITTSBURGH}`.

This example shows the utility of the RegionIterator for repeatedly applying a landmark pattern.

3.3 How Worried is Macbeth?

The "Cafe con Leche XML News and Resources" Web site (`http://www.ibiblio.org/xml/`) includes many plays by Shakespeare, formatted by Jon Bosak. We chose to examine *Macbeth* to discover just how many times Macbeth talks about "Birnam" and "Dunsinane" before his well-deserved end. This is admittedly rather silly, but it illustrates the ease of searching over a large XML file with complicated formatting, without employing a grammar/schema.

Landmark search means that most of the XML formatting can be ignored. The relevant parts for this task are the SPEECH tag pairs which wrap up speeches. A SPEECH block starts with a SPEAKER tag pair and one or more LINE tag pairs. For example:

```
<SPEECH>
<SPEAKER>MACBETH</SPEAKER>
<LINE>That will never be</LINE>
<LINE>Who can impress the forest,
      bid the tree</LINE>
   :
<LINE>Of Birnam rise, and our
      high-placed Macbeth</LINE>
   :
<LINE>Reign in this kingdom?</LINE>
</SPEECH>
```

The code iterates through each speech block, and, if the speaker is Macbeth, records the number of occurrences of the words "Birnam" and "Dunsinane":

```
Region topR = new Region("macbeth.xml");
Region speechRegion, speakerRegion;
String speechStr;
int numWords = 0;

RegionIterator speechIter = new
     RegionIterator(topR, "SPEECH");
while (speechIter.hasNext()) {
  // iterate through the SPEECH blocks
  speechRegion =
       (Region) speechIter.next();
  speakerRegion =
       speechRegion.tagMatch("SPEAKER");
  if (speakerRegion.contains("MACBETH")){
    // is the speaker Macbeth?
    speechStr = speechRegion.toString();
    numWords +=
      countString(speechStr, "Birnam") +
      countString(speechStr,"Dunsinane");
  }
}
System.out.println("No. words: " +
                                numWords);
```

The RegionIterator uses a "SPEECH" tag pattern to iterate through the speeches. The "SPEAKER" text is pulled from the speech and if it contains "MACBETH", then the number of times that "Birnam" and "Dunsinane" appear in the rest of the speech are counted. Incidentally, the count for the play is 10.

countString() is a simple method (written by us) that uses String.indexOf() to search over the supplied string and count the number of times a given substring is found.

4. COMPARISONS WITH OTHER APPROACHES

In this section we compare landmark search with region algebras, regular expressions, and the XQuery language.

4.1 Region Algebras

Region algebras include PAT expressions [6], overlapped lists [2], and nested region algebras [4]. They treat a region as a contiguous portion of text, delimited by landmarks (also called anchors and match points). The algebras typically allow relationships to be expressed between regions, including 'precedes', 'follows', and 'contains', and support operations for creating region sets using union, intersection, and exclusion.

Landmark search employs a similar underlying model, but without sets and most of the region operators; this simplifies the model considerably. Also, tag-based landmarks are singled out for extra support, due to their importance.

WebL is a Web page manipulation language, with region algebras underpinning its text search capabilities [5]. Its tag-based matching is similar to the version of tagMatch() that returns only a matching region. WebL employs regular expressions for matching against unstructured text.

4.2 Regular Expressions

Regular expressions have problems searching over structured text, the foremost being their default use of leftmost longest match [1]. That search mechanism is a good choice when the text is being tokenized into numbers or words, but consumes too much data when applied to repeating text patterns. Regular expressions are also unable to count, making it impossible for them to deal with arbitrarily nested elements such as tags.

Regex libraries, as found in Perl 5 and java.util.regex, have introduced lazy quantifiers [3], which can simulate simple forms of landmark search. However, the formulation also needs to employ other extensions such as a multiline mode, back references, word boundaries for repeated search, and numerical quantifiers. Even with these additions, regexs are still unable to correctly handle searches involving nested tags.

4.3 XQuery

XQuery is an extremely powerful SQL-like language for finding and extracting elements and attributes from XML documents [9]. XQuery utilizes XPath path expressions, which represent the XML document as a tree of nodes of various types, and offers a large number of operations for manipulating nodes and node sets [8].

Most XQuery queries are FLWOR expressions, such as the following one for printing a title of a book when its cover price is more than $30:

```
for $x in doc("books.xml")/bookstore/book
where $x/price > 30
return $x/title
```

The landmark version of this query is surprisingly similar, and not too much longer:

```
Region topR = new Region("books.xml");
RegionIterator ri =
    new RegionIterator(topR, "book");
while (ri.hasNext()) {
  // iterate through the books
  Region bookRegion = (Region) ri.next();
  Region priceRegion =
           bookRegion.tagMatch("price");
  double price = 0.0;
  try {  //convert price string to double
    price = Double.parseDouble(
                            priceRegion);
  }
  catch(NumberFormatException e){}
  if (price > 30.00)
    System.out.println(
        bookRegion.tagMatch("title"));
}
```

Landmark search is not meant to be a complete query language. It is restricted to data extraction, and relies on its host language for other features (such as parsing numbers).

5. CONCLUSIONS AND FUTURE WORK

Landmark search consists of two basic operations, match3() and tagMatch4(). match3() utilizes landmark patterns to identify regions within a parent region, while tagMatch4() uses tag pairs to find regions. Repeated application of these operations can be carried out using a RegionIterator class.

Although landmark search only employs a few operators, it is capable of extracting information from poorly formatted Web pages, plain text files, and XML documents lacking grammars or schemas.

The underlying aim of this work was to develop a small set of operations, that could be easily understood, readily added to a host language, and efficiently implemented. This contrasts with feature-rich alternatives, such as region algebras, regex packages, and XQuery.

The landmark operators currently pattern match via text comparison. It would be useful to add a (limited) set of regular expression capabilities. For example, case insensitivity, and meta-characters that match with the start and end of a region (i.e. the '^' and '$' symbols found in some regexs).

Supporting more regular expression power, perhaps multiplicity ('*') and selection ('|'), would allow landmark patterns to match against arbitrary numbers and words. However, this generality may not be required since landmarks and tag patterns tend to be unique strings.

A prototype Java implementation of the landmark operators (together with examples) can be downloaded from `http://fivedots.coe.pau.ac.th/~ad/landmarks`.

6. REFERENCES

[1] Clarke, C.L.A. and Cormack, G.V. On the use of regular expressions for searching text, *ACM Transactions on Programming Languages and Systems,* 19, 3, 413–426, May, 1997.

[2] Clarke, C.L.A., Cormack, G.V., Burkowski, F.J. An algebra for structured text search and a framework for its implementation, *The Computer Journal*, 38, 1, 43-56, 1995.

[3] Friedl, J.E.F. *Mastering Regular Expressions: Powerful Techniques for Perl and Other Tools,* O'Reilly and Associates, 2nd ed., 2002.

[4] Jaakkola, J. and Kilpeläinen, P. *Using sgrep for Querying Structured Text,* Department of Computer Science, University of Helsinki, Report C-1996-83, November, 1996.

[5] Kistler, T. and Marais, H. WebL - a programming language for the Web, In *Computer Networks and ISDN Systems, Proceedings of the WWW7 Conference*, 30, 259-270, April, 1998.

[6] Salminen, A. and Tompa, F.W. PAT expressions: an algebra for text search, *Acta Linguistica Hungarica,* 41, 1–4, 277–306, 1992.

[7] XHTML 1.0 The Extensible HyperText Markup Language (Second Edition), W3C, http://www.w3.org/TR/xhtml1/, August, 2005

[8] XPath: XML Path Language 2.0, W3C Working Draft, http://www.w3.org/TR/xpath20/, April, 2005

[9] XQuery 1.0: An XML Query Language, W3C Working Draft, http://www.w3.org/TR/xquery/, April, 2005