Examining Site Specific Data Extraction Algorithms for Their Value in General Data Extraction

Bryan Goldstein

Briarcliff High School

Briarcliff Manor, NY 10510

Introduction

The Internet has no lack of information. The future of the Internet is only limited by our ability to locate that information. Our only means of locating this information is through modern search engines. Although they have achieved a high standard, today's search engines have many limitations that will need to be improved to increase the efficiency of Internet use.

Among the limitations is the inability to extract structured meaning from pages that the search engine crawls and group it with other similar information. While there are many algorithms for taking information out of its context on the web, none of them are accurate enough to find both the very well and very poorly structured information. This research lays the ground work for a different type of algorithm that can analyze the format of a page that contains data such as calendar events and then creates a wrapper for extracting the event information. The benefit of this research comes by observing the relationships between the algorithms needed for specific sites, and the structure of the events on that site and drawing conclusions about the challenges involved. Even though HTML sites are considered unstructured, most web sites are built with their own unique structure from which information can be extracted with just a little more effort. The similarities in the algorithms of similarly structured sites can thus be used to make a generalization about parsing that ‘type’ of site. This study can be used in future work to create the algorithm that can write its own page wrappers for aggregating events just by analyzing the structure of a website.

The goal of this project is to create a tool that uses basic wrapper based structural and pattern matching methods to gather information in order to provide a further understanding of what is required to build a parser that automatically generates page wrappers as needed.

Review of Literature

In 1991 when Tim Berners-Lee created a hypertext interface that could be displayed over the Internet, he initiated the transformation of the internet from a bunch of secluded corporate servers that were able to connect to each-other, in to a popular phenomenon. This interface he created is known as HTML or hypertext markup language (Berners-Lee 1992). This creation allows us today to instantly access a world of information with the click of a button, from nearly anywhere on the globe.

The Internet is comprised mostly of unorganized data, mainly in the form of HTML files. Search engines have been around since before the creation of the massive link up of the world's networks: the World Wide Web. The first search engine was the Archie search engine and was followed by many others. After the World Wide Web was created search engines continued to index and allow people to search them. Whole web searching had been just computer science theory until 1995 when AltaVista launched (Khare 2004). In the late 1990's Yahoo and AltaVista became industry leaders in web search because they thoroughly searched the entire web. By 2001 though, a new search engine was on the rise, Google. Google was gaining publicity because of its new page ranking system. The Google PageRank system would keep a map of all the hyperlinks it indexed and increase the rank of results that are cited more often

(Brin 1998). Google would give you popular relevant results instead of a result matching your keyword on a website that gets no traffic.

Yahoo, Google, AltaVista, and other well-known Meta search engines have made significant progress in attempting to organize the data provided by the web and they all do a very good job of it. These big search engines have one problem, they return everything from everywhere that matches the search term you enter, and direct you to the website. Sometimes people do not want a website when they search, they want answers, and they want information. For that, there are specific search websites that search for a certain type of information (Sugiura 2000). In this case it will be calendar events.

The ideal way to get these events would be to have them all in a certain structured format on the web to make them easily understandable by a parser. Tim Berners-Lee, for this reason has re-envisioned the future of the web. His idea calls for a "Semantic Web," websites that have another layer of meta-data so that a search engine or other web tool can understand the context of the text it is indexing (Decker 2000). Web sites that are structured into special formats to give an interface for a computer, in addition to the current interface made only for the end user. This idea of a "Semantic Web," is sometimes referred to as the Web 3.0 (Web 1.0 being the original HTML web, and 2.0 being the modern web of Asynchronous JavaScript and XML or "AJAX") (Berners-Lee 1998). Looking deeper into these Web 3.0 formats, there are a few worth mentioning. First, there is RSS, or Really Simple Syndication. Although it is not one of the formats of the "Semantic Web," it uses an XML format that is meant to interface with the computer to tell it the news to display instead of telling the user directly, and it is a great example

of an innovative new web format that has recently made a swift entrance into the news and blogging industries. XML is a format that makes organized lists of information. The short amount of time it took RSS to take hold gives hope that something of this magnitude is indeed possible. The RDF or the Resource Description Framework format is the most common interface of the semantic web. The RDF format structures data on the web to make it recognizable by a web crawler. It allows the creator of a web page to tell the crawlers reading the web page the same information he/she is telling the people who are reading it. So instead of just knowing that the word "Bob" is near the term "bsmith@mysite.com," it knows that the latter is the email address of the former. While people who make websites could never be expected to put such superfluous meta-data into their work, the software that people use to make websites could be made to automatically insert this extra information (Kohler 2006). A "Semantic Web" is a very ambitious goal and is usually synonymous with artificial intelligence (Shadbolt 2006). This dream of giving the internet a database like structure is still a long way from reality and may or may not actually end up happening.

This leaves two other methods of extracting the target data. The first method is with computational linguistics. In this method of gathering data, part-of-speech tagging, word sense disambiguation, and other natural language context analysis algorithms are put to use the retain the meaning of the data and not just the words (Hearst 1999). Part-of-speech tagging can use statistical data or defined rules to determine the part of speech of each word in its context. Though this is extremely difficult due to the myriad of possible sentence arrangements, it is an essential step in gathering the meaning of a sentence (Brill 1992). Also, the second mentioned method, word sense disambiguation, uses statistical data to determine the meaning of a word in

context. Words can have many different meanings and some differences can be very subtle, so this task is harder than it seems (Sussna 1993). Natural Language Processing works well when you are trying to get information from data in paragraph form, but when data is scattered it is much more difficult to utilize this method.

The alternative is a structural wrapper based approach. This is where structural wrappers are created for individual sites. Though this is generally very accurate, it requires manual creation of wrappers, and is not practical when extracting data on a large scale. With another layer of pattern discovery and wrapper creation, a program could scale the web quickly and accurately (Chang 2003). This has proven a successful method and has been put to use on many different kinds of semi-structured data. The difficulty is creating this automatic pattern discovery/wrapper creation framework that works with any type of data structure (Knoblock 2000).

Other research has used a combination of ontological processing and structure analysis to obtain the target data. Their results were successful in processing data rich, paragraph written, obituaries (Embley 1999). This technique was also later successfully applied to tables with unknown, yet still some, structure (Embley 2005). However, when a table doesn't contain any discernible structure, this technique is not as easy.

This research is dealing with calendar data, and since calendar events are generally not structured in sentences, it is difficult to use natural language processing to obtain them. The best approach is one of individual structural wrappers. The question is: how can these structural wrappers be automatically created? In order to solve that problem, this research looks at individual site

specific structural wrappers. Site structures commonly used to list events, wrappers required to obtain these events, and the accuracy of results, were analyzed to determine the best framework for automatic wrapper generation. This framework has to be able to determine which parts of the page are dynamic and can be used as reference points, and which parts are static and must be pulled as data. It has to also use this extrapolated information and in this case determines which parts of the data go under which category, such as start date, end date, start time, end time, location, description, and title of a calendar event and in what format they are presented. Then finally it must generate a wrapper that instructs the rest of the program exactly how to obtain the information.

Research Objectives

Using conventional search and data mining techniques, it is impossible to find events, and separate all the details pertaining to those events. The first research objective was to create individual site specific parsers that find events and isolate out their details. This allows for analysis of the similarities between the code required to parse a page, and the structure of that page. The secondary objective was to put all of the events gathered by the individual web site parsers into one calendar that can be easily browsed and searched. Fortunately, Google Calendar has an API that can do just that. This secondary objective creates a valuable community resource that, for instance, puts all of a town's events in one place, but it is not essential to this research.

The next objective was to analyze the structure of the page, the code required to parse the page, and the accuracy of the results. A comparison of these three things, gives a great deal of insight into the possibility of generalizing the parsers and creating one parser for any page.

The final objective of this research was to devise a possible technique for gathering the details of events or other specific data on the web. The comparison of the data structure and parser code helped to create this new method of parsing data. The sum of my work on the individual parsers was to point future research in the direction of utilizing this new technique for gathering data.

Methods

The goal is to parse websites rich in calendar event data, and to aggregate them all to one calendar. Then to analyze the code written, the accuracy of the results, and the structure of the data in the website the code was written for, in order to find correlations that will allow for a method for analyzing any websites with code that is not specific to one particular site. To accomplish this task, site specific code had to be written for each of the sites used, then generalized until the site specific code had been trimmed to its bare minimum.

Two sites were chosen for the task. One of them is completely disorganized and has almost no discernible structure and one of them is a dynamic site with an organized list of events and links to pages with more information. After a base of general parsing and structuring scripts were written, individual scripts to fetch events from each site were created and trimmed by adding reusable methods to the base classes. Then a final script was created to gather the events and put them on to the calendar.

The data was then collected and analyzed along with the structure of the site it was taken from and the structure of the wrapper required to fetch it. The structure of the sites was also looked at in comparison to other event sites. The research is then summed up in this paper.

Process

The first order of business was to define an event, the class *Event* was created for use as a data type. The Event contains the four strings: title, location, description, and site. It also has two objects of the java Date class called *date*, and *endDate*. Those are the properties of an event that the parsers must retrieve information for. The next step was to choose the sites that to be used.

When choosing the sites to be used as samples in this research, the idea was to choose two sites that were as structurally opposite as possible. Thus by comparing a method with opposite extremes, if it works with one, and it works with the other, it should in theory work will everything in between. The informational sections of the resulting websites are shown in Figures 1, 2, and 3. Figure 1 shows a completely unstructured calendar from a restaurant venue and Figure 2 shows a very organized list of events generated by a dynamic event website, Figure 3 shows one of the pages containing the details of one of the events from Figure 2. The first thing that had to be done with these pages was to analyze their source.

On the first calendar page, the month and year are in the header of the calendar on a separate table from the events. The first row of the event table contains the days of the week and the second row contains redundant events, both will be ignored. Boxes do not all have days in them,

and when they do have days in them, they do not always have events. The title of the event and the time of the event are often but not always in separate paragraph tags, and when there are more then one event in a box they are always separated by paragraph tags.

After analysis of the first page, the initial parser was created. This required almost two-hundred lines of sloppy, do-whatever-it-takes code to extract the information. It had, at this point, a few key flaws, it could not decipher two events in one box, and it was so messy that absolutely no conclusions could be drawn from it.

The source of the second page is much neater than that of the first, but there is one other key difference, the details of the events are each on their own separate pages. On the main page with the list of events, the events are each in separate divider tags of class event. In each one of these dividers, there is a span tag of class *eventTitle* containing the title linked to the page with the details on the event. The source of the page containing the details of the event is just as organized. There is a divider tag called *eventDetail* containing a header tag with the title of the event directly underneath, followed by another divider underneath the header called details. Inside that divider tag, there are span tags of class *detailLabel* containing the text of the detail's label such as *DATE:*, *TIME:*, *VENUE:*, and *ADDRESS:*. Each one of the *detailLabel* span tags is directly followed by a *detailData* span tag with all the information on that aspect of the event.

After the analysis of this page, its initial parser was created. This parser was also about two-hundred lines of unorganized code. This code had one other flaw: The title of each event returned was placed on the wrong event details. This was a result of overlooked data in the detail page. In order to quickly switch from one page to the next, the site contains information on the previous and next events on the page along with the current event.

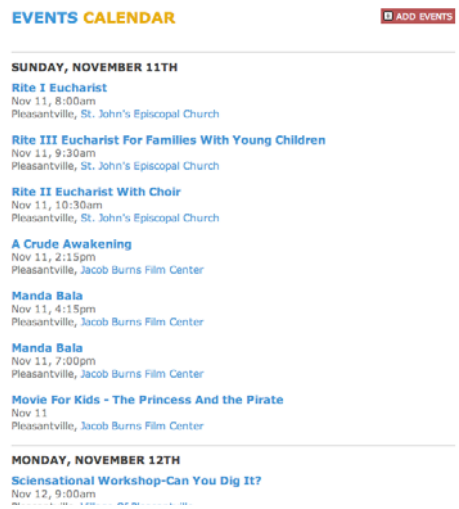
Figure 1: Jackson and Wheeler event calendar (http://jacksonandwheeler.com/EventCalendar.html)

OCTOBER 2007 EVENTS

Sun	Mon	Tue	Wed	Thu	Fri	Sat
Belly Dancing Dinner Special Evening Dinner Specials to Match Theme	Francine Tyler Psychic Consultant Every Other Thursday	Every Tuesday is the Geoff Hartwell Blues Jam. Chance to win 2 guitars Drawing 10/30	On Premise Bakery Featuring Planetary Pastries with gluten free, vegan and regular pastries	Every Thursday, The Westchester Jazz Jam starting at 8:30pm WINE DINNER OCTOBER 24TH FEATURING WOMEN OF THE VINE	Gluten Free Menu Available as well as gluten free beer and desserts	Off Premise Catering Available as well as Private Room for Events at Jackson & Wheeler
	1	2	3	4	5 Johanna 7:30pm	6
7 Champagne Jazz Brunch 11:30am-2:30pm	8	9 Geoff Hartwell Blues Jam 9:30pm	10	11 Francine Tyler Psychic Consultant Westchester Jazz Jam 8:30pm	12 Joe Noschese 7:30pm	13
14 Champagne Jazz Brunch 11:30am-2:30pm	15	16 Geoff Hartwell Blues Jam 9:30pm	17	18 Westchester Jazz Jam 8:30pm	19 Peter Carucci 7:30pm	20
21 Belly Dancing Evening 7pm Champagne Jazz Brunch 11:30am-2:30pm	22	23 Geoff Hartwell Blues Jam 9:30pm	24 Women of the Vine Wine Dinner 7pm	25 Francine Tyler Psychic Consultant Westchester Jazz Jam 8:30pm	26 Danny Russo 7:30pm	27
28 Champagne Jazz Brunch 11:30am-2:30pm	29	30 Geoff Hartwell Blues Jam 9:30pm Halloween Show Guitar Drawings	31			CLICK HERE FOR ANOTHER CALENDAR MONTH

These two parsers needed to be organized, and the first step was to identify what they both had in common. Both rough scripts had a huge mess of sloppy string-pattern-matching to navigate the tags, and both scripts had their own individual code to analyze the date and time and turn it in to a date object. To respond to these similarities, two classes were created as shown in Figure 4.

Figure 2: American Towns Pleasantville Event Calendar (http://www.americantowns.com/ny/pleasantville/events)



EVENTS CALENDAR

ADD EVENTS

SUNDAY, NOVEMBER 11TH

Rite I Eucharist
Nov 11, 8:00am
Pleasantville, St. John's Episcopal Church

Rite III Eucharist For Families With Young Children
Nov 11, 9:30am
Pleasantville, St. John's Episcopal Church

Rite II Eucharist With Choir
Nov 11, 10:30am
Pleasantville, St. John's Episcopal Church

A Crude Awakening
Nov 11, 2:15pm
Pleasantville, Jacob Burns Film Center

Manda Bala
Nov 11, 4:15pm
Pleasantville, Jacob Burns Film Center

Manda Bala
Nov 11, 7:00pm
Pleasantville, Jacob Burns Film Center

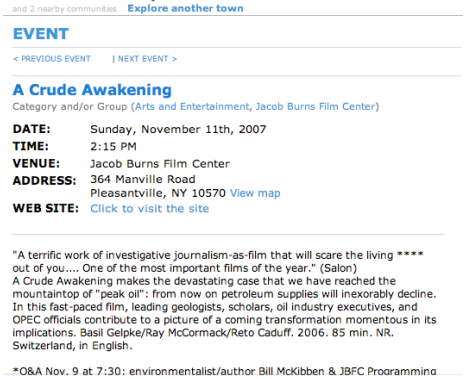
Movie For Kids - The Princess And the Pirate
Nov 11
Pleasantville, Jacob Burns Film Center

MONDAY, NOVEMBER 12TH

Sciensational Workshop-Can You Dig It?
Nov 12, 9:00am

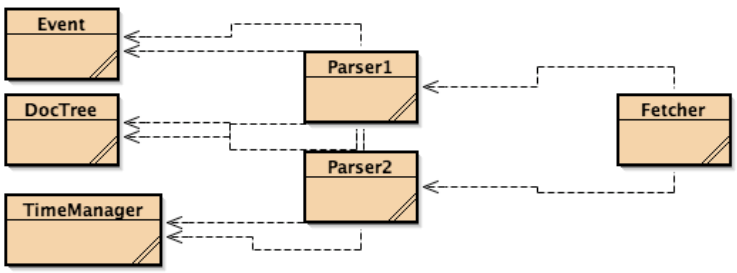
The first class created was the *DocTree* class. A *DocTree* object is constructed with a path to a web page. It uses an open source HTML parsing library called *HTMLParser*, and creates an *org.htmlparser.util.NodeList* object of the document at the path provided. *DocTree* has two

Figure 3: American Towns event listing (http://www.americantowns.com/ny/pleasantville/events/a_crude_awakening_2)



public methods besides its constructor. The first is *public Node nodeAtPath(String path)* where the string path similar to a file path, but it navigates the elements of an HTML document. The components of the path are the HTML tags in the file; although it is acceptable to go beyond the tag and into the attributes because the HTML tag is found by using the *startsWith()* method of the *java.lang.String* class to compare the path component with everything in between the carats. Putting a number *n* in square brackets after a path component returns the n^{th} match of that tag within the children of the previous path component. If there is no leading slash in the path, then the code is searched recursively until the first element is found, but a recursive search can be done at any point in a path by preceding a path component with a tilde. The second class is *public Node[] nodesAtPath(String path)*, this is the same as the first method except instead of returning only the node at index zero, or the specified index, it returns an array of all the nodes matching the path.

Figure 4: Class dependancy structure



The second class created was the *TimeManager* class. This class contains four simpler, self explanatory methods. These are *public static Boolean hasTime(String aString), public static int[] getTimeFromString(String theString), public static int[] getDateFromString(String theString), public static int convertMonthToInt(String month).*

The first method tests for a time within a string; the second gets a time from a string and returns an array of integers that can be used in conjunction with date information to create a time object; the third does the same as the second only with dates; and the fourth converts month names into integer representations.

These classes made it possible to strip the parsers down to the bare minimum required to parse the websites.

Results

The first portion of the results to be examined is the site specific wrappers that the parsers were boiled down to. While these wrappers are in the form of Java classes instead of another XML format, they are both pretty basic and are easily picked apart for the different steps required for each. The first is the wrapper for the web page which is a messy tag soup and is hand-made with no dependable structure other than the calendar table itself, it is shown in Figure 5.

Figure 5: Parser 1

```
/**
 * Parser 1 is a wrapper for jacksonandwheeler.com's calendar page.
 *
 * @author Bryan Goldstein
 */
import org.htmlparser.*;
import org.htmlparser.util.*;
import java.util.regex.Pattern;
import java.util.*;
import java.text.*;

public class Parser1
{
    public static Event[] getEvents() throws ParserException{
        Scanner scan;
        DocTree mainCalendar = new DocTree("http://www.jacksonandwheeler.com/EventCalendar.html");
        Pattern m = Pattern.compile("jan(uary)?+|feb(ruary)?+|march|april|may|june|july|aug(ust)?+|sept(ember)?+|oct(ober)?+|nov(ember)?+|dec(ember)?+",
                Pattern.CASE_INSENSITIVE);
        Pattern y = Pattern.compile("[2-9][0-9]{3}+");

        //Gets the first table containing the month and year
        String monthYearTable = new NodeList(mainCalendar.nodeAtPath("/html/body/table/tr[2]/td/table/tr[4]/td/table")).asString();
        scan = new Scanner(monthYearTable);
        final String MONTH = scan.findWithinHorizon(m,0).toUpperCase();
        final String YEAR = scan.findWithinHorizon(y,0);

        //Gets the second table containing the event information
        Node [] rowArray = mainCalendar.nodesAtPath("/html/body/table/tr[2]/td/table/tr[5]/td/table/tr");
        String dsvEventString = "",dsvDayString = "";//Uses delimiter seperated strings as placeholders for arrays
        for(int i1=2;i1<rowArray.length;i1++) {
            Node [] columnArray = mainCalendar.nodesAtPath("/html/body/table/tr[2]/td/table/tr[5]/td/table/tr["+i1+"]/td");
            for(int i2=0;i2<columnArray.length;i2++) {
                Node [] pArray = mainCalendar.nodesAtPath("/html/body/table/tr[2]/td/table/tr[5]/td/table/tr["+i1+"]/td["+i2+"]/p");
                String eventTitleWithoutTime = "";
                for(int i3=0;i3<pArray.length;i3++) {
                    String pText = new NodeList(pArray[i3]).asString().replaceAll("[ \t\n\f\r]+"," ").replaceAll(" ","");
                    if (!pText.trim().equals("")) {
                        if (i3!=0) {
```

```
                    if (TimeManager.hasTime(pText)) {
                        if (!dsvEventString.equals("")) {
                            dsvEventString += "-::event::-";
                        } if (!dsvEventString.equals("")) {
                            dsvDayString += "-::event::-";
                        }
                        if (!pText.equals(pText.replaceAll("[a-z]{3}+","")))
                            eventTitleWithoutTime = "";
                        dsvEventString += eventTitleWithoutTime + pText;
                        dsvDayString += new NodeList(pArray[0]).asString().replaceAll("[ \t\n\f\r]+"," ").replaceAll(" ","");
                    } else {
                        eventTitleWithoutTime = pText;
                    }
                }
            }
        }
    }

    //Determines and returns all the details of the information
    int i = 0;
    String [] eventInfo = dsvEventString.split("-::event::-");
    String [] eventDay = dsvDayString.split("-::event::-");
    Event [] results = new Event [eventInfo.length];
    while (i<eventInfo.length && eventInfo[i]!=null && eventDay[i]!=null) {
        int [] time = TimeManager.getTimeFromString(eventInfo[i]);
        Event result = new Event();
        result.title = eventInfo[i].substring(0,time[2])+eventInfo[i].substring(time[2]+time[3],eventInfo[i].length());
        GregorianCalendar date = new GregorianCalendar(Integer.parseInt(YEAR), TimeManager.convertMonthToInt(MONTH),
            Integer.parseInt(eventDay[i]),time[0],time[1]);
        result.date = date.getTime();
        if (time[5]!=-1) {
            GregorianCalendar endDate = new GregorianCalendar(Integer.parseInt(YEAR), TimeManager.convertMonthToInt(MONTH),
                Integer.parseInt(eventDay[i]),time[4],time[5]);
            result.endDate = endDate.getTime();
        }
        result.location = "Jackson & Wheeler, Pleasantville, NY";
        result.description = eventInfo[i];
        result.site = "http://jacksonandwheeler.com/";
        results[i] = result;
        i++;
    }
    return results;
  }

}
```

This parser must first find the month and year which is the same for each event because these events are listed on a monthly calendar. Then the parser must loop through each box of the calendar and then through each paragraph tag in each box, determining which time goes with which description. Then the results must be compiled into the right formats and returned as an Event array.

Figure 6: Parser 2

```
/**
 * Parser 2 is a wrapper for americantowns.com's events pages.
 *
 * @author Bryan Goldstein
 */

import org.htmlparser.*;
import org.htmlparser.util.*;
import java.util.regex.Pattern;
import java.util.*;
import java.text.*;

public class Parser2
{
  public static Event[] getEvents() throws ParserException
  {
    //get list of event description pages
    Scanner scan;
    DocTree eventListPage = new DocTree("http://www.americantowns.com/ny/pleasantville/events");
    Node [] eventPageLinks = eventListPage.nodesAtPath("/html/body[0]/div[0]/div[0]/div[1]/div[0]/div[0]/div class=\"event\"");
    Event [] results = new Event[eventPageLinks.length];
    for(int i = 0;i<eventPageLinks.length;i++) {
      String eventPageLinkNodeText = eventListPage.nodeAtPath("/html/body[0]/div[0]/div[0]/div[1]/div[0]/div[0]/div class=\"event\"["+i+"]/span/a").getText();
      String eventPageLink = "http://www.americantowns.com"+eventPageLinkNodeText.split("\"")[1];
      //call get event for each event description page
      results[i] = getEvent(eventPageLink);
    }
    return results;
  }
```

```
    protected static Event getEvent(String pageLink) throws ParserException
    {
        DocTree eventPage = new DocTree(pageLink);
        int [] timeInts = new int[7];
        int [] dateInts = new int[3];
        String website = "",location = "";
        Event result = new Event();
        //retrieve the event title and description
        result.title = new NodeList(eventPage.nodeAtPath("/html/body/div/div/div[1]/div/div/div[5]/div/div[1]/div/div/h1")).asString();
        result.description = new NodeList(eventPage.nodeAtPath("/html/body/div/div/div[1]/div/div/div[5]/div/div[1]/div/div/div[1]")).asString();
        //retrieve the details section of the page
        Node [] details = eventPage.nodesAtPath("/html/body/div/div/div[1]/div/div/div[5]/div/div[1]/div/div/div/span");
        //get each detail of the event
        for (int i=0;i<details.length;i+=2)
        {
            if (new NodeList(details[i]).asString().equals("DATE:"))
                dateInts = TimeManager.getDateFromString(new NodeList(details[i+1]).asString());
            else if (new NodeList(details[i]).asString().equals("TIME:"))
                timeInts = TimeManager.getTimeFromString(new NodeList(details[i+1]).asString());
            else if (new NodeList(details[i]).asString().equals("VENUE:"))
                location = new NodeList(details[i+1]).asString()+location;
            else if (new NodeList(details[i]).asString().equals("ADDRESS:"))
                location += new NodeList(details[i+1]).asString();
            else if (new NodeList(details[i]).asString().equals("WEB SITE:"))
                website = details[i+1].getChildren().elementAt(0).getText().split("\"")[1];
        }
        //return the details in the right format
        if (dateInts.length==3 && timeInts.length>2) {
            GregorianCalendar date = new GregorianCalendar(dateInts[0],dateInts[1],dateInts[2],timeInts[0],timeInts[1]);
            result.date = date.getTime();
            if (timeInts[5]!=-1) {
                GregorianCalendar endDate = new GregorianCalendar(dateInts[0],dateInts[1],dateInts[2],timeInts[4],timeInts[5]);
                result.endDate = endDate.getTime();
            }
        }
        result.location = location;
        result.site = website;
        return result;
    }
}
```

The second parser (Figure 6) is surprisingly similar, but instead of looping through the rows and columns of a table, it loops through pages and well-marked data tags. There is significance to how similar these two parsers are despite the large differences in the types of sites they are parsing.

Next portion of the results are the actual results printed by the program. Parser 1 returns sixteen results and Parser 2 returns forty results. Each and every result is displayed with the correct information so that is an accuracy of 56/56 or 100%. Wrapper based parsers should have a 100% accuracy, because the wrapper can be modified until the accuracy is perfect. The question is, what can we learn from these perfectly accurate bundles of code that take an impractical amount of time to create?

Conclusions

It has been shown in this research that although two web sites may structure their information very differently, the wrapper required to parse the two web sites is surprisingly similar. This can be used to create a program that automatically generates these wrappers. The differences in a page that determines how the wrapper is created can be separated into three categories: (1) the way the events are listed; (2) the way the details of the event are separated; and (3) the format of the details given, in this case mainly the date. These are all aspects that would need to be considered by any wrapper generating framework.

If the page is structured as a calendar, the events will be in boxes of a table, but if they are a list of links to another page, there may not be any details on the current page. The wrapper generating framework would need to systematically check the page to do a summary of where everything is. It will see how much information is given in headings and how much is in the individual listings. From this it will need to build a base of general information to fill in any missing information in the individual listings like month, year, website, and location. Then a pattern needs to be generated for how each event is separated. The framework will check for recurrences and when it finds the recurring list or calendar pattern, it will add that to the wrapper. A determination must then be made on whether the parser needs to follow links for more details. The framework will follow links to try and find more details relating to the event where the link was found. If the event pattern doesn't contain any links this can be skipped, but if details are found at the end of the links, then it will be written into the wrapper to follow the links. This section will also detail the section of the page the parser must go to find the event information.

The second step is to create a pattern of how the individual details are separated. If all the details are in a sentence, the parser will need to use natural language algorithms and locations from the base of information that the wrapper generating framework has gathered to fill in the details. If the details are listed neatly as in the site that Parser 2 parses, then it is a bit easier; the framework can just specify how each detail is marked and contained, and then the parser just needs to run through it like a shopping list. If the details are on a separate page they must be identified in the wrapper, along with which details (if any) should be obtained from the main page.

Finally, the wrapper generating framework needs to determine the format of the details. In this case, the details that require special formatting are the dates and times. Dates and times can be written in many ways, and the pattern for recognizing the dates and times must change each time to recognize the way they are written. A date can be written with words like *November 12, 2007* or with numbers like *11/12/2007* or with an abbreviated year. A date can also be written with dashes, periods, commas, and apostrophes. The list goes on and on. Then there are repeated dates and ranges of dates, and listings containing *every other Thursday*, the *first Monday of every month*, or *Mon.-Fri*. The wrapper generating framework will have to have all of those possibilities entered, and it will need to apply the appropriate date formatting patterns and algorithms to each particular wrapper.

In a massive event indexer, these wrappers can be cached for later use, or just regenerated every time. The benefit is that you can do either and any way you do it, you will have a parser that is many times more accurate than using any other individual method of data extraction. This

research shows how site specific wrappers could become an efficient means of extracting certain information from partially or totally unstructured websites on a large scale.

Acknowledgements: Many thanks to my mentor Jeanguy Dahan, for giving me his expert tutelage, my teachers Mr. Michael Inglis and Ms. Kim Dyer for there encouragement and guidance, and my parents for their support.

Work Cited

Berners-Lee, Tim. “Semantic Web Road Map.” (1998): 1-10. 6 Dec. 2006 <http://www.w3.org/DesignIssues/Semantic>.

Berners-Lee, Tim et al. “World-Wide Web: The Information Universe.” Electronic Networking: Research, Applications and Policy 1 (1992): 74-84. <http://citeseer.ist.psu.edu/berners-lee92worldwide.html>.

Brill, Eric. “A Simple Rule-Based Part Of Speech Tagger.” Proceedings of {ANLP}-92, 3rd Conference on Applied Natural Language Processing (1992): 152-155. <http://citeseer.ist.psu.edu/article/brill92simple.html>.

Brin, Sergey, and Page, Lawrence. “The Anatomy of a Large-Scale Hypertextual Web Search Engine.” (1998): 1-26. 27 Mar. 2007 <http://wx.bsjh.tcc.edu.tw/~t2003013/wiki/images/8/8f/ Anatomy.pdf>.

Chang, Chia-Hui et al. “Automatic information extraction from semi-structured Web pages by pattern discovery.” Decision Support Systems 34 (2003): 129-147. <http://www.sciencedirect.com/science/article/B6V8S-45XTR57-3/2/f8175b5989fba681063fee665bfa6d46>.

Decker, Stefan et al. “The Semantic Web: The Roles of XML and RDF.” IEEE Internet Computing (2000): 2-13.

Embley, David W. et al. “Automating the extraction of data from HTML tables with unknown structure.” Data & Knowledge Engineering 54 (2005): 3-28. <http://www.sciencedirect.com/science/article/B6TYX-4DVBDCD-1/2/9044732cd9f0cf17f10f135f24b03b38>.

Embley, David W. et al. “Conceptual-model-based data extraction from multiple-record Web pages.” Data & Knowledge Engineering 31 (1999): 227-251. <http://www.sciencedirect.com/science/article/B6TYX-3XJKBTJ-2/2/bf114bfa25560bf29d4c171bb2340af1>.

Hearst, M. “Untangling text data mining.” In Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (1999): 3–10.

Khare, Rohit. "Nutch: A Flexable and Scalable Open-Source Web Search Engine." CommerceNet Labs Technical Report 04-04 (2004): 1-15.

Knoblock, Craig et al. “Accurately and Reliably Extracting Data from the Web: A Machine Learning Approach.” IEEE Data Engineering Bulletin 23 (2000): 33-41. <http://citeseer.ist.psu.edu/knoblock00accurately.html>.

Kohler, Jacob et al. “Ontology based text indexing and querying for the semantic web.” Knowledge-Based Systems 19 (2006): 744-754.

Shadbolt, Nigel et al. “The Semantic Web Revisited.” IEEE Intelligent Systems 21 (2006): 96-101. <http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/mags/ex/&toc=comp/mags/ex/2006/03/x3toc.xml&DOI=10.1109/MIS.2006.62>.

Sussna, Michael. “Word sense disambiguation for free-text indexing using a massive semantic network.” Proceedings of the second international conference on Information and knowledge management (1993): 67-74. <http://portal.acm.org/citation.cfm?id=170106>.

Sugiura, Atsushi et al. “Query routing for Web search engines: architecture and experiments.” Computer Networks 33 (2000): 417-429. <http://portal.acm.org/citation.cfm?id=346329>.