IMPROVING YIOOP! USER SEARCH DATA USAGE

A Writing Report

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Tarun Pepira Ramaswamy

September 2012

SAN JOSÉ STATE UNIVERSITY

The Undersigned Project Committee Approves the Project Titled

IMPROVING YIOOP! USER SEARCH DATA USAGE

by

Tarun Pepira Ramaswamy

APPROVED FOR THE DEPARTMENT OF
COMPUTER SCIENCE

Dr. Chris Pollett, Department of Computer Science Date

Dr. Mark Stamp, Department of Computer Science Date

Dr. Soon Tee Teoh, Department of Computer Science Date

ABSTRACT

IMPROVING YIOOP! USER SEARCH DATA USAGE

User data in the form of visited URL's and search query provides a lot of information about the user. This data can be utilized to provide some value-added service to the end user. Commercial search engines like Google, Bing have started taking advantage of these data and provide features like trending/popular searches, personalized search results, etc.

This project aims to provide three user benefits using these user data. It provides a visualization tool of user's navigation history across the web. This would help the users to better understand their search history. Secondly, it improves the search results in Yioop by maintaining the user-searched queries across various popular search engines (Google, Bing, Yahoo) locally in the user's machine and re-ranking the Yioop result. Finally, it provides a related search feature using the local user search data.

ACKNOWLEDGEMENTS

I would like to thank my professor Dr. Chris Pollett for his help and guiding me throughout the project. He has constantly motivated me to work hard for the project. I would also like to thank my committee members Dr. Soon Tee Teoh and Dr. Mark Stamp for their valuable time and support. And special thanks to my friends for helping me with the testing and their feedback.

Table of Contents

1. Introduction 9
2. Preliminary Work 11
    2.1 Building Firefox Extension 11
    2.2 Firefox's History Data 13
3. User Data Visualization 15
    3.1 Force Directed Algorithm 15
    3.2 Implementation 18
4. Re-rank Yioop Result 21
    4.1 User Search Data 21
    4.2 Implementation 22
    4.3 Testing 28
5. Related Keywords in Yioop Result 33
    5.1 Relevance Calculation 33
    5.2 Implementation 35
    5.3 Testing 37
6. Conclusion 39
7. References 40

List of Figures

Figure 1: Firefox Extension Structure 12

Figure 2: install.rdf 12

Figure 3: chrome.manifest 13

Figure 3: main.xul 13

Figure 4: The Places Database 14

Figure 5: Pseudo code of Force Directed Algorithm 17

Figure 6: Force Directed Graph using Canvas 20

Figure 7: Database Schema of User Search data 21

Figure 8: 'result' hierarchy 24

Figure 9: Manipulating Yioop result 25

Figure 10: Yioop result without the re-rank feature 26

Figure 11: Re-ranked Yioop result 27

Figure 12: Recall comparison of Yioop and re-ranked Yioop result 30

Figure 13: Precision comparison of Yioop and re-ranked Yioop result 31

Figure 14: Adding relevant searches 35

Figure 15: Yioop result page without the related keyword 36

Figure 16: Yioop related keywords feature 36

Figure 17: Usability graph of Related Searches 38

List of Tables

Table 1: Recall calculation of Yioop result Page……29

Table 2: Precision calculation of Yioop result Page……31

Table 3: Usability score of related searches……37

1. Introduction

A search engine plays an important role in every user's web experience. Over the years, there has been a continuous effort in improving their capabilities. One among them is providing customized search results based on users past search activity. This is called as Personalized Search. In this, the search engine gives useful and most relevant search results by keeping a record of the users past searches. However, tracking of user's search results has lead to privacy concerns among the users. In this project, Yioop search engine gives a personalized search experience without any compromise in their privacy. Instead of storing the search history in server, it keeps the records in user's local machine. The project is written as a Firefox extension [1] where the implementation is transparent to the user thus providing user confidence.

One of Dr Pollett's previous student, Vijaya Pamidi[5] created a similar Firefox extension which captured user clicks but would send it to the Yioop Server. Although it improved the Yioop search results over a period of time, it didn't provide immediate benefit to the user. This led to the non usage of this extension. To make our extension user beneficial, the project provides a visualization tool which draws a directed graph of the user visited URL's. To draw this graph, it makes use of the browser's history data. In Firefox, the browser history is stored as an SQlite database called the Places [9]. It also provides special Storage API's to access these data. The extension uses this data and Canvas element of HTML5 to render the graph. And to make the

graph aesthetically pleasing, it makes use of Force Directed Algorithm [4] for drawing the graph.

The project also improves the search results of Yioop using the users past search data. The users past searches are stored as a separate file to provide faster read/ write access and avoid the complex architecture of the browser's history. Having a separate file also helps the user search data easy to maintain. Unlike other commercial search engines like Google, Bing, etc where the user has no control over the search data stored in the server, here the user has full control over these data for Yioop as the data is stored in the local machine.

The extension also tries to predict related searches based on the Yioop search query. This feature is similar to related searches in other search engines but uses only the user's local search data to calculate it.

The initial part of the report explains about the building blocks for creating a Firefox extension and also provides background information about the browser's History data. Then it describes in detail about the visualization tool and its implementation. The next part contains details about the re-rank feature and the experiments performed. The final part explains about the related searches in Yioop result page with its implementation details and experiments.

2. Preliminary Work

The project uses Firefox extension to build the visualization tool and to re-rank Yioop's search result. Implementing the project as an extension provides number of benefits. It helps to build user confidence as the user can view the code anytime. It can take advantage of Firefox's Storage API to access the browser history data. And the most important benefit is in capturing the user search results across multiple search engines in a transparent manner. Now, let us look in detail about how to build a Firefox extension and how the user history is stored in the browser.

2.1 Building Firefox Extension

The project uses Firefox extension to implement the features. Firefox extensions allow developers to add functionality to the browser and enhance the user interface in a way that is not directly related to the viewable content of Web pages. Extensions are distinct from plug-in which help the browser to display or play certain multimedia objects. In Firefox, the extensions are packaged and distributed in ZIP files or Bundles, with the XPI (pronounced "zippy") file extension. An extension's user interface is written using XUL (pronounced "zool"), CSS and JavaScript.

A typical xpi file would have the below folder structure

```
my_extension.xpi:                       //Equal to a folder named my_extension/
          /install.rdf                  //General information about your extension
          /chrome.manifest              //Registers you content with the Chrome engine
          /chrome/
          /chrome/content/              //Contents of your extension such as XUL and JavaScript files
          /chrome/icons/default/*       //Default Icons of the extension
          /chrome/locale/*              //Building an Extension# Localization
          /defaults/preferences/*.js    //Building an Extension# Defaults Files
          /plugins/*
          /components/*
          /components/cmdline.js
```

Figure 1: Firefox Extension Structure

The basic components of the extensions are three files,

- install.rdf
- chrome.manifest and
- main.xul

install.rdf contains the id and version of the extension. It also specifies required id, min and max version of the target application. Here, **{ec8030f7-c20a-464f-9b0e-13a3a9e97384}** is the application ID of Firefox.

```
<?xml version="1.0"?>

<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:em="http://www.mozilla.org/2004/em-rdf#">

<Description about="urn:mozilla:install-manifest">
    <em:id>yiooptoolbar@seekquarry.com</em:id>
    <em:version>1.0</em:version>
    <em:type>2</em:type>
    <em:optionsURL>chrome://yiooptoolbar/content/options.xul</em:optionsURL>
    
<!-- Target Application this extension can install into,
    with minimum and maximum supported versions. -->
<em:targetApplication>
    <Description>
    <em:id>{ec8030f7-c20a-464f-9b0e-13a3a9e97384}</em:id>
    <em:minVersion>1.5</em:minVersion>
    <em:maxVersion>15.0.*</em:maxVersion>
    </Description>
</em:targetApplication>

<!-- Front End MetaData -->
    <em:name>Yioop! Toolbar</em:name>
    <em:creator>Seekquarry.com</em:creator>
    <em:description>Yioop!</em:description>
    <em:homepageURL>http://www.yioop.com/</em:homepageURL>

</Description>
</RDF>
```

Figure 2: install.rdf

chrome.manifest contains the folder hierarchy, skin details and the main XUL file which needs to be overlaid on the browser.

```
content    yiooptoolbar    chrome/content/
overlay chrome://browser/content/browser.xul chrome://yiooptoolbar/content/main.xul
skin yiooptoolbar classic/1.0 chrome/skin/
```

Figure 3: chrome.manifest

main.xul has the functionality details of the extension. It includes the JavaScript file that helps to achieve the functionality and also contains details about how one wants to add the menu details.

```
<?xml version="1.0"?>

<overlay id="sample"
        xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul" >

<script type="application/x-javascript"
    src="chrome://yiooptoolbar/content/main.js" />

<menupopup id="menu_ToolsPopup">
    <menuitem id="yioop_menu" insertafter="javascriptConsole,devToolsSeparator"
              label="Yioop Search"
              oncommand="openUILinkIn('chrome://yiooptoolbar/content/main.html','current')" />
</menupopup>

</overlay>
```

Figure 3: main.xul

In this project, main.js would contain the logic to draw the graph and make use of the search result to re-rank the Yioop result and calculate the related keywords. A detailed description of this is discussed later.

2.2 Firefox History Data

Firefox uses a robust history management system called Places [9]. It stores the history and bookmark data in a flexible and easy to access manner. Internally, it is a SQlite database and uses the mozStorage interface. A partial database schema of this database is shown below.

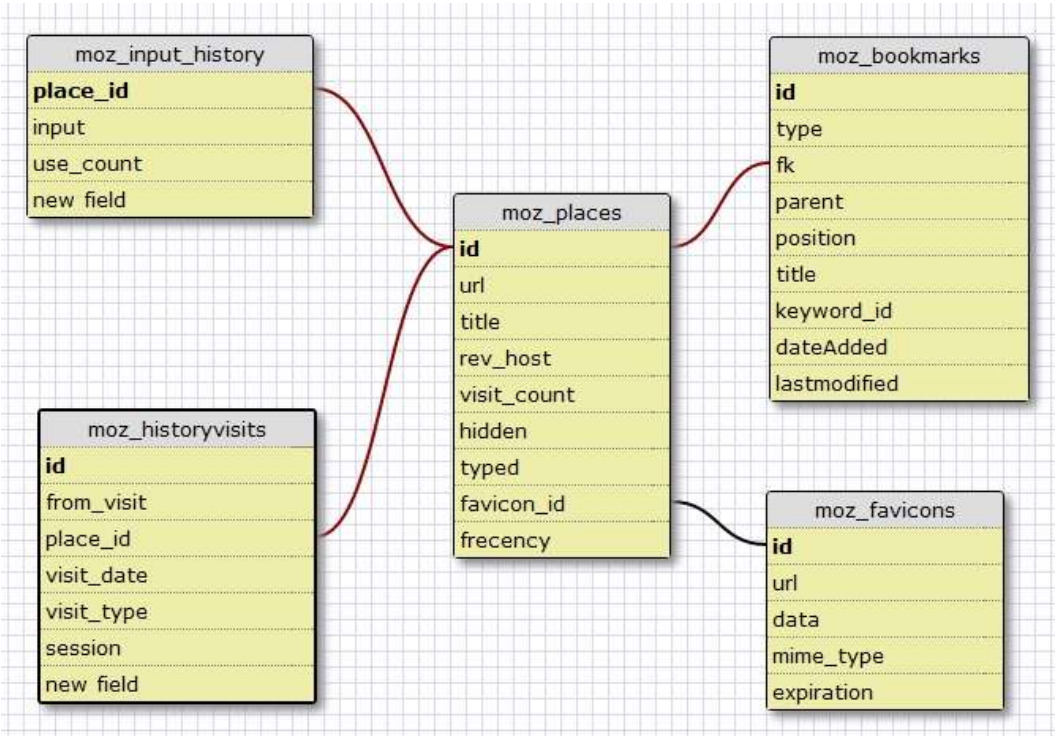

Figure 4: The Places Database

Although, the Places database has lots of information regarding bookmarks, favicons, we focus only on the URL's visited by the user and the navigation path of the usage. These details can be extracted from moz_places and moz_historyvisits table. From moz_historyvisits, we get the URL id of a page and the URL id of the page from which it came/visited. Using this id, we can get the actual URL from the moz_places table.

3. User Data Visualization

One of the main goals of the project was to show the user history in a graphical way. This would give the user a visual tool to view their visited URL's. The user history typically consists of the navigation history of users moving from one website to another in form of user clicks or user searches. Tabulating the list of the sites visited is one form of listing the visited websites. But, displaying it in the form of graph would be more easily readable by the users. For the graph, we can imagine the nodes as a particular website/web link and the edges as the navigation path between the two web links. There are different kinds of graph layouts available like Force-based layout, orthogonal layout, Spectral layout, Circular layout, Tree layout, etc. In this project, we use Force-based layout as it clearly represents the user navigation in an obvious way.

3.1 Force Directed Algorithm

Force based layouts uses the Force Directed algorithm [6] to draw a directed graph in an aesthetically pleasing way. The aesthetic criteria typically consist of having the vertices evenly distributed, fewer crossing edges, equal edge lengths and overall symmetry. In the Force directed model, the nodes of the graph are assumed as a metal ring and the edges are considered to be springs to form a mechanical system. The nodes are placed initially at some random positions and the spring's force will cause the nodes to move. The nodes will finally come to rest at equilibrium.

In Force directed algorithms designed by Fruchterman & Reingold [4], there are two forces that help to reach the state of equilibrium, Hooke's force of attraction and Coulomb's force of repulsion.

Hooke's Law : "*If the spring is compressed or extended and released, it returns to its original, or natural, length, provided the displacement is not too great. We see that for small Δx the force exerted by the spring is approximately proportional to Δ x. This result is known as Hooke's Law."[*7]

$$\mathcal{F}_x = -k(x - x_o) = -kx$$

Where k is the force of constant of the spring

Coulomb's Law: *"It states that the magnitude of the Electrostatics force of interaction between two point charges is directly proportional to the scalar multiplication of the magnitudes of charges and inversely proportional to the square of the distances between them."[*8]

$$|\mathcal{F}| = k_e \frac{|q_1 q_2|}{r_2}$$

Where K_e is the repulsion constant and q_1, q_2 are the two point charges.

In this algorithm, we randomly place the node in a given plane. Due to the random position, forces get applied to the node by either pulling them towards each other or by pushing them away from one other. This process continues till the system comes to an equilibrium state.

This can be explained in detailed by the below pseudo-code of the algorithm

```
//Place the nodes at random position and
//intialize their velocity to (0,0)
Loop

    //Initialize the total kinetic energy
    kinetic_energy = 0;

    for each node
        // Net force of this particular Node
        net_force = (0, 0);

        // Calculate the effect of Coulomb's Law
        for each other node
            net_force = net_force + Coulomb_repulsion (this_node, other_node);
         next node

        //Calculate the effect of Hooke's Law
        for each spring connected to this node
            net_force = net-force +  Hooke_attraction (this_node, spring);
        next spring

        //Update the velocity of the node using a damping constant(0 < d < 1)
        //Here, we are using the damping constant to be 0.5
        this_node.velocity = (this_node.velocity + timestep * net_force) * 0.5

        //Update the node's position
        this_node.position = this_node.position + timestep * this_node.velocity

        //Update the kinetic energy of the system
        kinetic_energy = kinetic_energy + this_node.mass * (this_node.velocity)^2

    next node

 until kinetic_energy < 0.01 //A small constant
```

Figure 5: Pseudo code of Force Directed Algorithm

After assigning random positions to the node, we are calculating the force on each node. For each node, we calculate the Coulomb's force of repulsion caused by all the other nodes in the system and the Hooke's force of attraction caused by the springs attached to this particular node. This net force is then used to update the velocity of the node which moves the node to the new

position. This process is continued till the kinetic energy of the system converges to zero or a really small value. Here, we chose the small value to be 0.01 to maintain a balance between the time taken to draw the graph and the quality of the final graph.

3.2 Implementation

The algorithm is implemented using JavaScript as it is easy to add JavaScript in the Firefox extension. For drawing the graph, we had the option of using SVG or Canvas. We choose Canvas due to the popularity of HTML5 and its wider support.

For drawing the graph, we are considering the unique URL's as the nodes and the spring to be the navigation across the URL's. As discussed earlier, Firefox provides the history data in the form a SQLite database. It also provides Storage API's to retrieve and manipulate the data.

The database can be opened by the below API,

```
var placeDb = Components.classes["@mozilla.org/browser/nav-history-service;1"]
      .getService(Components.interfaces.nsINavHistoryService)
      .QueryInterface(Components.interfaces.nsPIPlacesDatabase).DBConnection;
```

Once the connection is open, we can create sql statements and execute it as shown below

```
var statement = placeDb.createStatement("SELECT * FROM moz_historyvisits; ");
statement.executeAsync();
```

Here, we get all the nodes i.e. unique visited URL's from the tables moz_places and moz_historyvisits using the below statement.

```
   var statemt = placeDb.createStatement("SELECT distinct moz_places.url as URL
FROM moz_historyvisits, moz_places where moz_historyvisits.place_id =
moz_places.id;");
```

```
var point = new Point(Vector.random(), 1.0, url);
```

Once the nodes are drawn, we need to get the edges. Edges are calculated in a two step process. First, we find the “from_visit” attribute (which is the id of the source URL link) and the destination URL from the moz_historyvisits table.

```
var statement = placeDb.createStatement("SELECT moz_historyvisits.from_visit as fromURLId, moz_places.url as toURL FROM moz_historyvisits, moz_places where moz_historyvisits.place_id = moz_places.id;");
```

In the next step, for each of this source URL id’s we get their corresponding source URL.

```
var stmt = placeDb.createStatement("SELECT moz_places.url as fromURLS FROM moz_places WHERE moz_places.id IN (SELECT moz_historyvisits.place_id FROM moz_historyvisits where moz_historyvisits.id = :fromhistoryURL)");
```

For each of these source-destination pair, we draw the spring.

```
applyStrings(ColumnFromURL, ColumnToURL);
```

Once we have the nodes and their corresponding edges, we can use the Force directed algorithm to calculate and draw the graph. To show, the animation effect, we redraw the graph for each iteration.

In this implementation, we tried for different constant values that could be used as damping, repulsion and the timestep to update the velocity of the node. After some observation based on the time taken to draw and the overlap of the graph edges, we chose the damping constant to be 0.6, timestep to be 0.5 and the repulsion constant as 90 for drawing the graph.

In the graph, the edge tapers from the start node to the end node. The edge is thicker at the source node and becomes thinner towards the destination node. The below figure shows a sample graph.

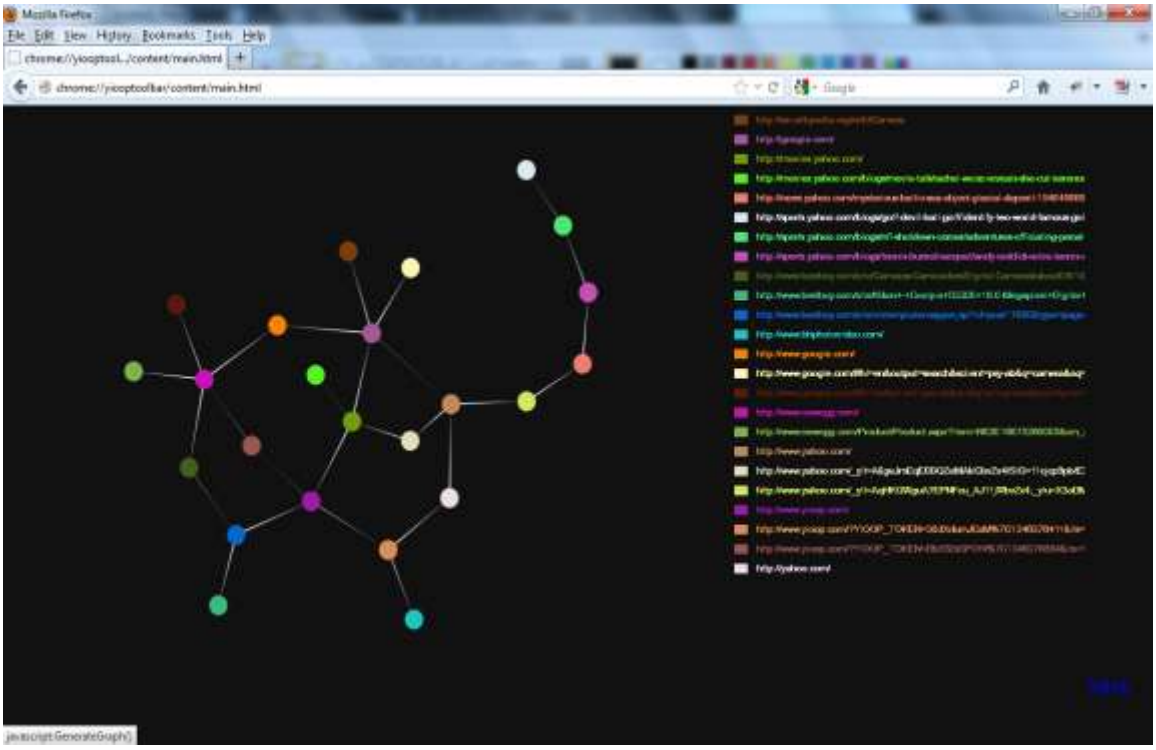

Figure 6: Force Directed Graph using Canvas

4. Re-ranking Yioop Result

The second aim of the project is to provide a better search result in Yioop using the user search data. For this, we need to filter the useful data that helps to identify the search keyword and the corresponding URL's the user clicked. To achieve this, we store the user-clicked searches from popular search engine like Google, Yahoo, Bing, Yioop etc. For this project, we are storing the user clicks for only these search engines. However, it can be extended to other search engines as well.

4.1 User Search Data

As explained in the previous section, Firefox provides a feature to store history data in a SQLite database. However, continuous retrieval of data in this normalized database could affect the performance of the query. Hence, we create a separate SQLite database called yioop_HistoryData.sqlite to capture the keywords, the url links, the visit count and source of the data.

Column ID	Name	Type	Not Null	Default Value	Primary Key
0	keyword	TEXT	0		1
1	url	TEXT	0		1
2	title	TEXT	0		0
3	visitcount	INTEGER	0		0
4	searchfrom	TEXT	0		0
5	timestamp	INTEGER	0		0

Figure 7: Database Schema of User Search data

As the name suggests, the "keyword" is the search query the user entered in the search engine page, "url" is the destination url the user reached by clicking on the search result, "title" represents the title of the result page,

“visitcount” represents the number of times this url got clicked for this particular search query, “searchfrom” simply represents the search engine from which this data got extracted and “timestamp” represents the time at which the result was captured.

For any given query on the Yioop Search engine, it returns the search result based on the data stored in the server. Since, these captured results are stored in the user machine and not sent to the Yioop Server, the results on the server is not influenced by these data. The extension now calculates the user clicked results based on the user query and manipulates the Yioop Search result page at runtime. The relevant search result can be selected based on the “timestamp” or their “visitcount”. In this project, we calculate the result based on the “visitcount” since, the more the user clicked on the particular link, the greater its value.

4.2 Implementation

To re-rank the Yioop result page on the fly, we need to manipulate the Document Object Model (DOM) of the page. There are three different methods to intercept the loading web pages and modify their contents. However, the use of these techniques depends solely on the requirement of the project. In this project, we want to modify the content of the Yioop result page based on the search query. For this, the use of Load Events is sufficient.

The other two techniques are HTTP Observers and WebProgess Listener. HTTP Observers intercepts for HTTP notification sent for the HTTP request and manipulates the content. If one needs a more controlled way of intercepting

and modifying the various stages of the web page load, WebProgessListener is the better choice. It can be used to keep track of all the progress listeners for each tab.

Let us discuss the LoadEvent technique used in the project in more detail. In this technique, we add an event listener for the load event. The function below adds an event listener on page load with the callback method myExtension.init().

```
window.addEventListener("load", function load(event) {
                    window.removeEventListener("load", load, false);
                    myExtension.init();
}, false);
```

Once the listener is added, the next step is to put in the functionality code to the myExtension function. The structure of myExtension would be something as shown below.

```
var myExtension = {
    init: function () {
        // The event can be DOMContentLoaded, pageshow, pagehide, load or unload.
        if (gBrowser)
            gBrowser.addEventListener("DOMContentLoaded", this.onPageLoad, false);

        //Initialization logic can be put here
    },
    onPageLoad: function (aEvent) {
       //Code that manipulates the web page
    }
};
```

Here, gBrowser is the global object that corresponds to each browser window. By attaching the “DOMContentLoaded” event handler, gBrowser allows to listen to all the tabs in a hassle free way.

In order to re-rank the results, we need to create new elements and update the Yioop result page with these new elements. The re-rank is applied only to

the first page of the Yioop result page. The next pages will be showing the actual results without the re-rank. In the Yioop result page, the results are listed as a ‘div’ element with a className ‘result’. So, we get the HTMLCollection having the className as ‘result’ and add the results as their siblings. The result has the following structure.

```
<div class="result">
  <h2>
      <a href="http://dictionary.reference.com/browse/fade" rel="nofollow">Fade | Define Fade at Dictionary.com</a>
  </h2>
  <p class="echolink">http://dictionary.reference.com/browse/fade</p>
  <p></p>
  <p class="serp-links-score">visit_count: 2 Source:yahoo</p>
</div>
```

Figure 8: ‘result’ hierarchy

To add the result, the exact HTML structure needs to be created and added to the web page. HTMLCollection objects support the following methods to modify the content of the webpage.

insertBefore(newElement, refElement) – This adds the newElement before the refElement. Since, there is no insertAfter() function, insertBefore() can be used as insertBefore(newElement, refElement.nextSibling) to get the same effect.

replaceChild(newElement, oldElement) – This replaces the oldElement with the newElement.

The below code creates such element and adds it to the Yioop result page.

```
//Get the element where we need to add new data
var divResult = content.document.getElementsByClassName('result');

//Code that creates the element
var divElement = content.document.createElement('div');
var aElement = content.document.createElement('a');
var pElement = content.document.createElement('p');
var h2Element = content.document.createElement('h2');
var pEcho = content.document.createElement('p');
var pEmpty = content.document.createElement('p');
var tElement = content.document.createTextNode(colValues[1]);
var echotext = content.document.createTextNode(colValues[0]);
var countElement = content.document.createTextNode("visit_count: " + colValues[2] +
 " Source:" + colValues[3]);

 //Code that combines these elements to form the 'result' div element
divElement.setAttribute('class', 'result');
aElement.setAttribute('href', colValues[0]);
aElement.setAttribute('rel', 'nofollow');
pEcho.setAttribute('class', 'echolink');
pElement.setAttribute('class', 'serp-links-score');
aElement.appendChild(tElement);
pEcho.appendChild(echotext);
pElement.appendChild(countElement);
h2Element.appendChild(aElement);
divElement.appendChild(h2Element);
divElement.appendChild(pEcho);
divElement.appendChild(pEmpty);
divElement.appendChild(pElement);

//Inserts the element to the webpage
divResult[0].parentNode.insertBefore(divElement, divResult[0]);
```

Figure 9: Manipulating Yioop result

Here, colValues[] array contains the keyword, title and the url of the result. This is used to create the ‘result’ div element and added to current web page. This code would come under the onPageLoad section of the above Load Event code. With this interception and manipulation of the result page, we can successfully add the search results captured from other search engine pages.

Below is the example of a search for the keyword “fade”. The first figure shows the current Yioop search result and the next figure shows the Yioop result with the re-rank function.

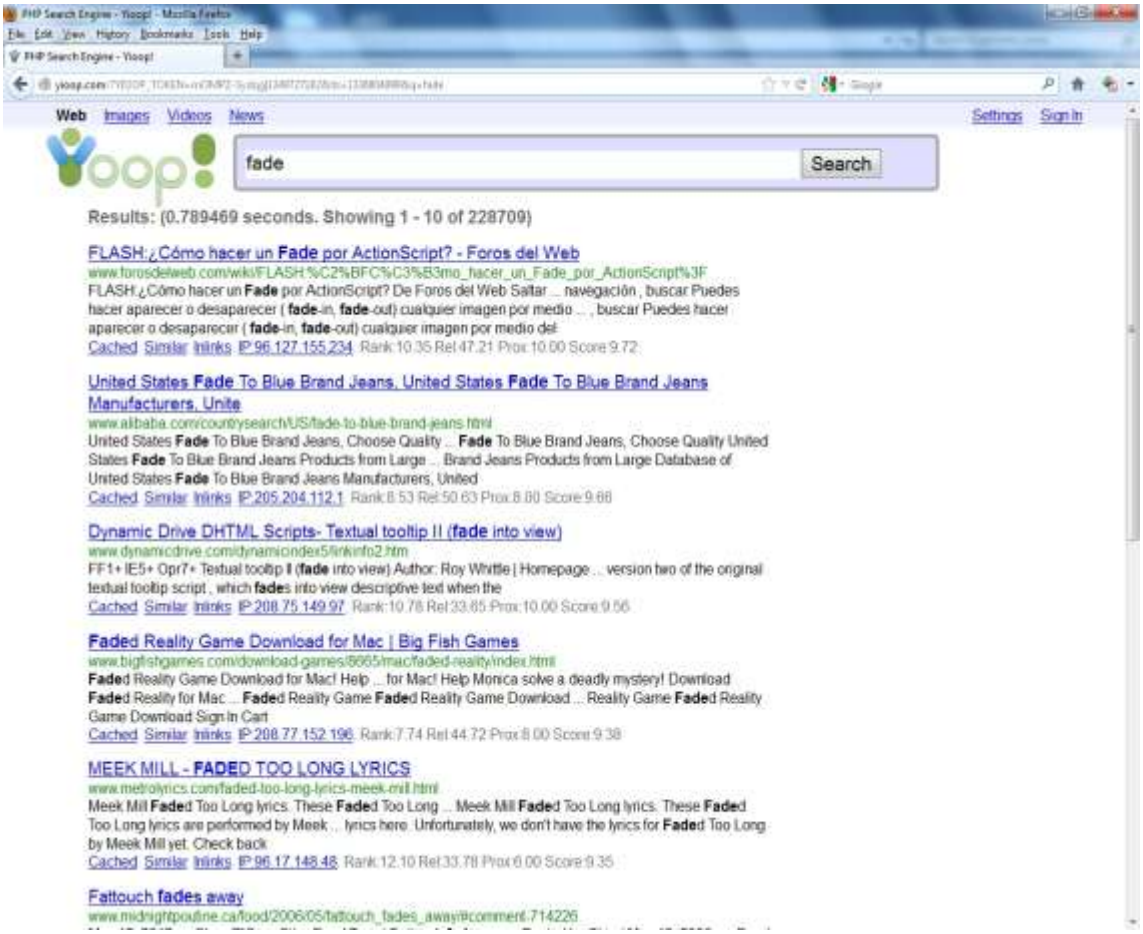

Figure 10: Yioop result without the re-rank feature

As shown in the below figure, the first result is being captured from the Bing search engine. It has been placed at the top due to its higher visit count. If there are many results coming from other search engine, we restrict to the top five results calculated from user search data. This gives a good balance

between the Yioop's actual result from the server and the results obtained from the user search data.

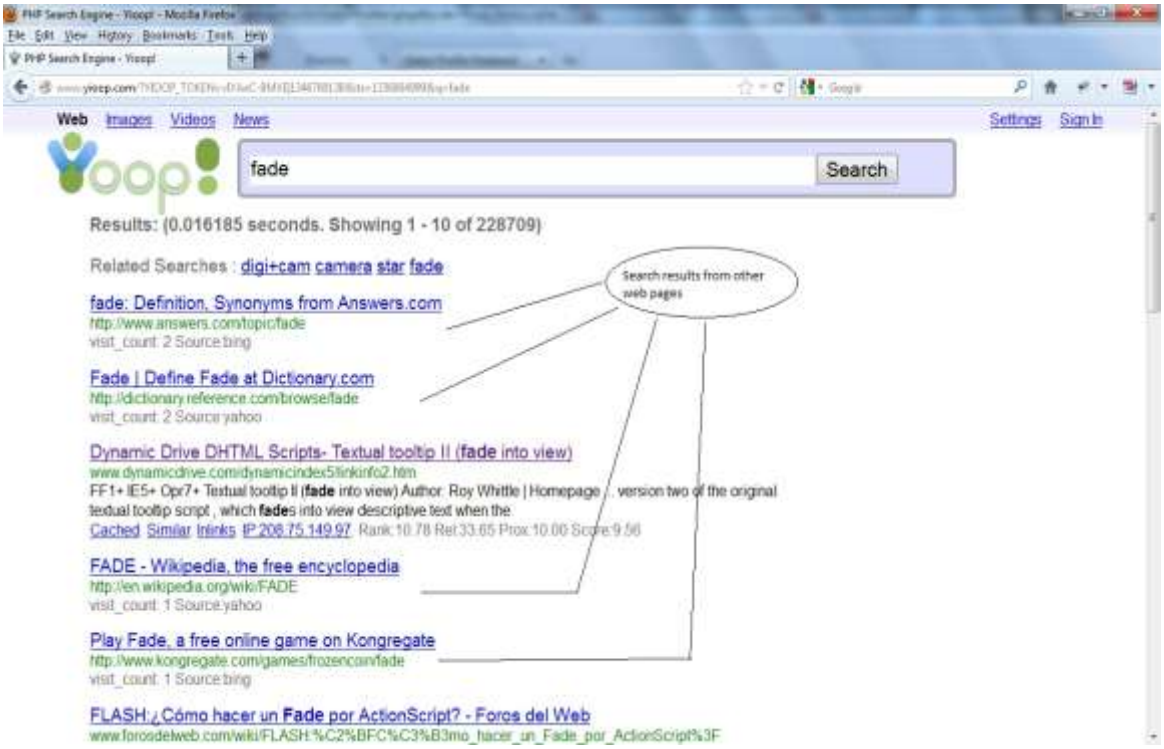

Figure 11: Re-ranked Yioop result

4.3 Testing

Here, we shall discuss some tests of the relevance of re-ranking the search results in the Yioop result page. In our experiments, we are using the feedback given by five volunteers who tested this feature and scored it based on how relevant the results were for the searches.

First, they were asked to search for a particular keyword in Yioop search engine and give a score of number of relevant results shown in the result page. Then, we added the re-rank feature and asked them to give a score of relevant search results. The effectiveness of the search result is calculated using two standard measures called precision and recall.

Precision [11] is the fraction of the result set that are relevant and Recall [11] is the fraction of relevant documents that appear in the result set.

Precision = $|Rel \cap Res|/|Res|$

Recall = $|Rel \cap Res|/|Rel|$

In this experiment, we have used some common terms like games, poetry, dogs and maps as the search term. It also shows three variants of the re-ranked Yioop result page, with three, five and eight results from the user search data.

On querying, the findings are as given in the below table.

Query	T_Y	T_{RY3}	T_{RY5}	T_{RY8}	$Recall_Y$	$Recall_{RY3}$	$Recall_{RY5}$	$Recall_{RY8}$
games	5	6.4	8.2	9.6	0.29	0.37	0.48	0.56
poetry	4.8	8	10.4	11.2	0.3	0.5	0.65	0.7
dogs	1.2	4.6	6	7.8	0.1	0.38	0.5	0.65
maps	2.4	5.6	6.2	6.8	0.24	0.56	0.62	0.68

Table 1: Recall calculation of Yioop result Page

Where,

T_y – Average relevant results from 5 users for Yioop

T_{RY3} – Average relevant results from 5 users for re-ranked Yioop with 3 user data

T_{RY5} – Average relevant results from 5 users for re-ranked Yioop with 5 user data

T_{RY8} – Average relevant results from 5 users for re-ranked Yioop with 8 user data

$Recall_y$ – Recall for Yioop

$Recall_{RY3}$ – Recall for re-ranked Yioop with 3 user data

$Recall_{RY5}$ – Recall for re-ranked Yioop with 5 user data

$Recall_{RY8}$ – Recall for re-ranked Yioop with 8 user data

The graphical representation of the recall values would like

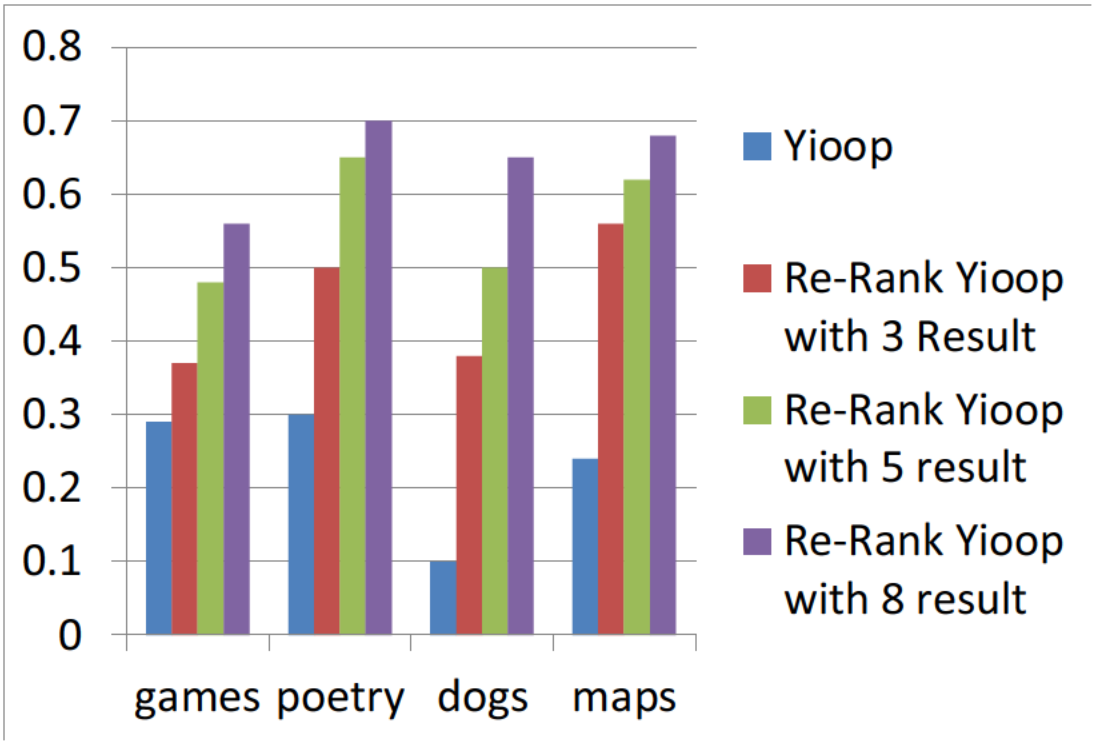

Figure 12 – Recall comparison of Yioop and re-ranked Yioop result

As we can see, the recall value improves with the addition of user data in the Yioop Search result page. The greater the number of user data included, the higher the recall value obtained.

Now, let's calculate the precision value for these data.

Here,

P_y – Precision for Yioop

P_{RY3} – Precision for re-ranked Yioop with 3 user data

P_{RY5} – Precision for re-ranked Yioop with 5 user data

P_{RY8} – Precision for re-ranked Yioop with 8 user data

Query	T_Y	T_{RY3}	T_{RY5}	T_{RY8}	P_Y	P_{RY3}	P_{RY5}	P_{RY8}
Term1	5	6.4	8.2	9.6	0.5	0.49	0.54	0.53
Term2	4.8	8	10.4	11.2	0.48	0.61	0.69	0.62
Term3	1.2	4.6	6	7.8	0.12	0.35	0.4	0.43
Term4	2.4	5.6	6.2	6.8	0.24	0.43	0.41	0.37

Table 2: Precision calculation of Yioop result Page

The graphical representation of the precision is as follows

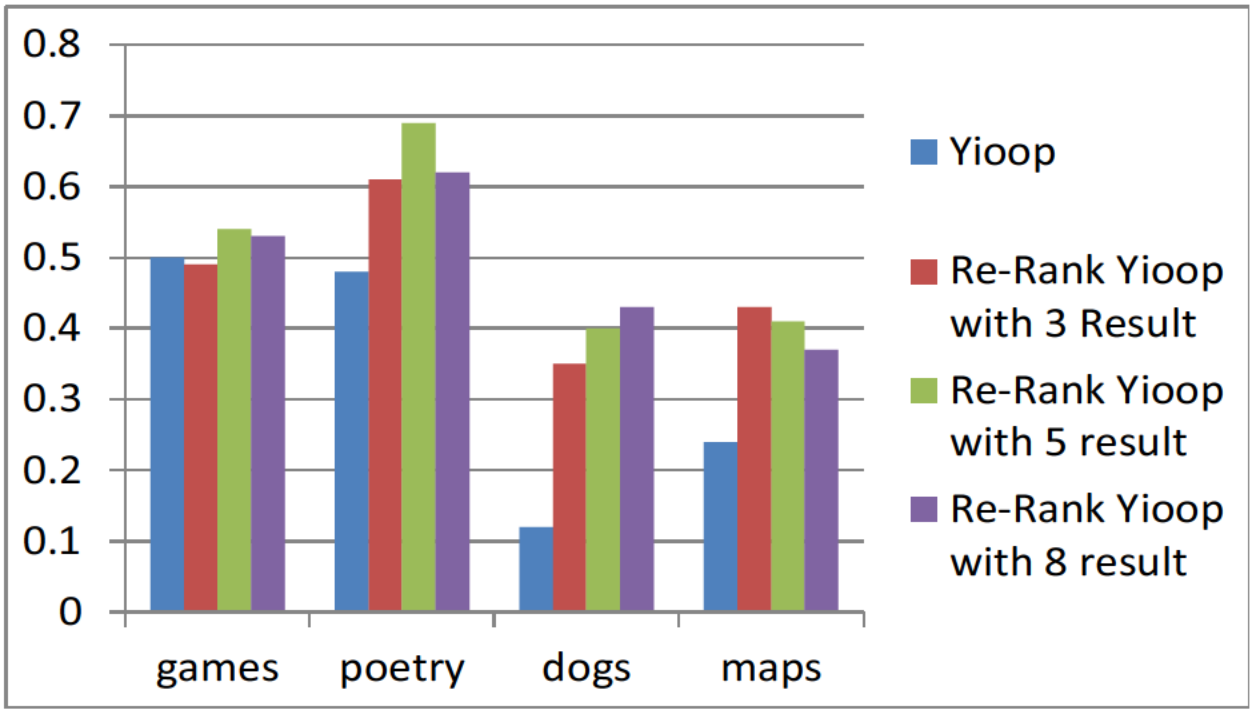

Figure 13 – Precision comparison of Yioop and re-ranked Yioop result

Here, as the number of user search result increases, the precision of the result page also increases except for the maps keyword. We can also see that the precision increases with the number of user search result being included in result page, but after a certain point the score improves slowly. In the above graph, the re-ranked Yioop result page with five user data added seems to be fine balance between the precision and the usability of the final result page.

5. Related Keywords in Yioop

The third aim of the project is to give related search queries for a given keyword. Since, we are already capturing the user search data to re-rank the result page; we make use of the same data to calculate the related keywords.

5.1 Relevance Calculation

There are different ways we can calculate the relevance of the keyword for a particular search query. We have different parameters like timestamp and visit count that helps in this calculation.

The simplest way is to calculate the result based on the timestamp of the user search data. In this, the keywords having the latest timestamp are shown as the related keywords. However, these keywords may not be related to given search query. If the user is lucky, they might get some useful related queries.

The second way is to calculate the result based on the visit count of the user search data. In this, the keywords having a higher visit count is given preference and shown in the result. It is highly probable that this gives a better result as these are results that user used most often.

In this project, we use Okapi BM25[10] ranking function to rank the matching search result based on its relevance to given search query. BM25 is a bag of words information retrieval function that ranks the set of documents based on the query.

The BM25 score is calculated by the below formula,

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})},$$

where, IDF(q_i) is the inverse document frequency,

f(qi,D) is the q_i's term frequency in the given document,

k_1 and b are free parameters with k_1 = [1.2, 2.0] and b = 0.75

|D| is the length of the document D and

avgdl is the average document length

Inverse Document Frequency(IDF) is computed as shown below

$$\text{IDF}(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5},$$

where, N is the total number of documents and

$n(q_i)$ is the number of documents containing q_i.

In this case, documents relate to the visited urls and query relates the search keywords.

5.2 Implementation

The process of adding elements to the document is similar to the one discussed earlier. We intercept the web page when the DOM content get loaded and insert the new elements into the document.

```
//Get the related search results
getRecentSearches();
var rsLength = recentSearch.length;

//Create a template to add the new values
var divHelement = content.document.createElement('div');
var textHelement = content.document.createTextNode("Related Searches :");
divHelement.setAttribute('class', 'serp-stats');
divHelement.appendChild(textHelement);

for(i = 0; i < rsLength; i++ ) {
    var spaceHelement = content.document.createTextNode("\t");
    divHelement.appendChild(spaceHelement);
    var aHelement = content.document.createElement('a');
    aHelement.setAttribute('href', "http://www.yioop.com/?q="+ recentSearch[i]);
    var dataHelement = content.document.createTextNode(recentSearch[i]);
    aHelement.appendChild(dataHelement);
    divHelement.appendChild(aHelement);
}

//Add the elements to the Yioop! result page
divResult[0].parentNode.insertBefore(divHelement, divResult[0]);
```

Figure 14- Adding relevant searches

Here, recentSearch is array containing the relevant keywords for the particular search. The code then iterates through this array and adds the element to the result page.

The below figure shows the current view of the Yioop result page and the next figure shows the Yioop result with the related keywords feature.

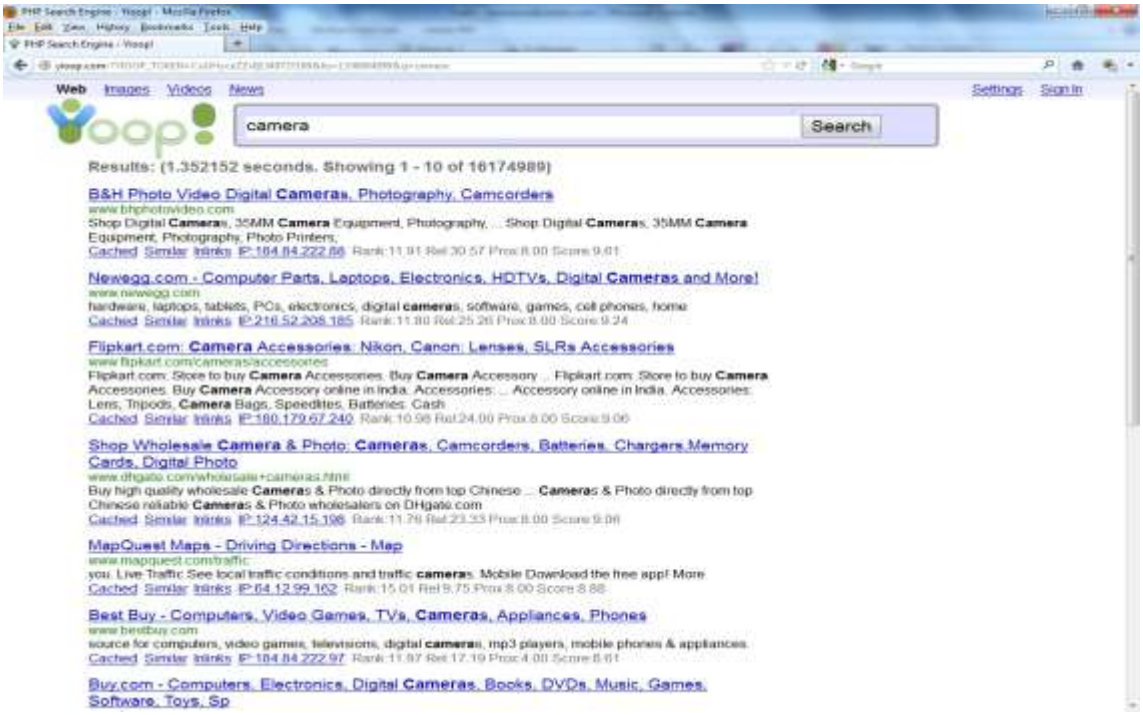

Figure 15- Yioop result page without the related keyword

The highlighted area in the below figure shows the related keywords for the given search query.

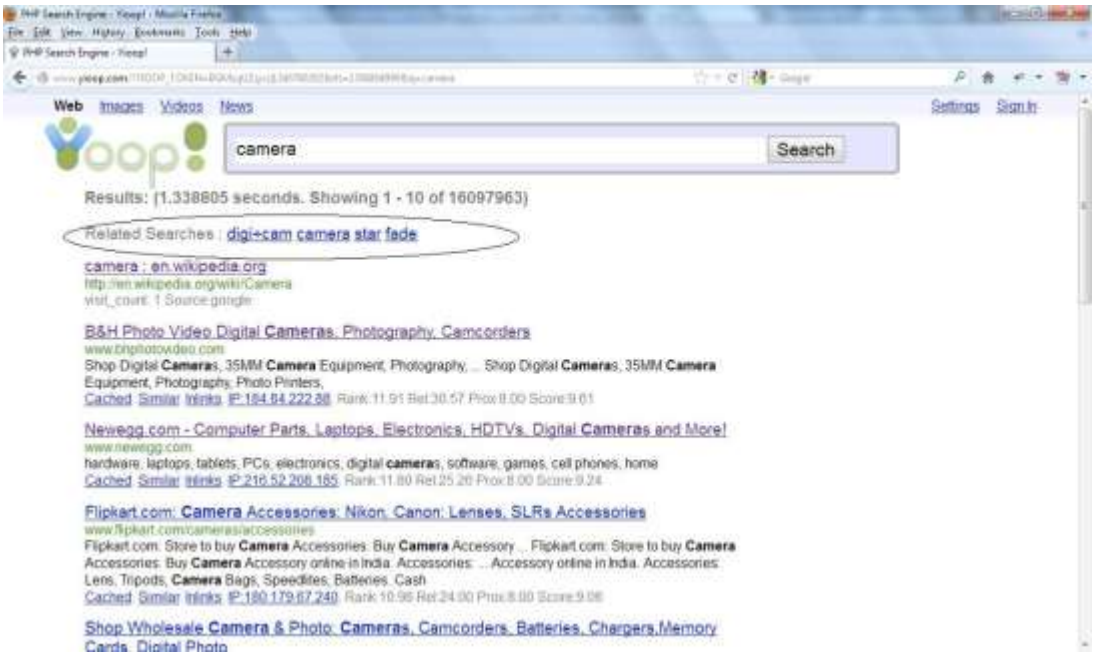

Figure 16- Yioop Related searches

5.3 Testing

Here, we tried to experiment with different number of related searches to be displayed in the search page and asked the user to vote for their best choice. For this, five users who have a fair knowledge of computer use and search engine use. The users were asked to rate in terms of effectiveness (how relevant the terms where) and satisfaction (how user friendly the results were displayed).

The below table represents the result based on the number of results being shown to the user.

Number of Related Search terms	Effectiveness	Satisfaction	Overall
1	34%	36%	33%
2	40%	44%	42%
3	48%	50%	49%
4	54%	56%	55%
5	60%	64%	62%
6	66%	64%	65%
7	68%	50%	59%
8	70%	44%	57%

Table 3: Recall calculation of Yioop result Page

The graphical representation of the overall score will be as shown below.

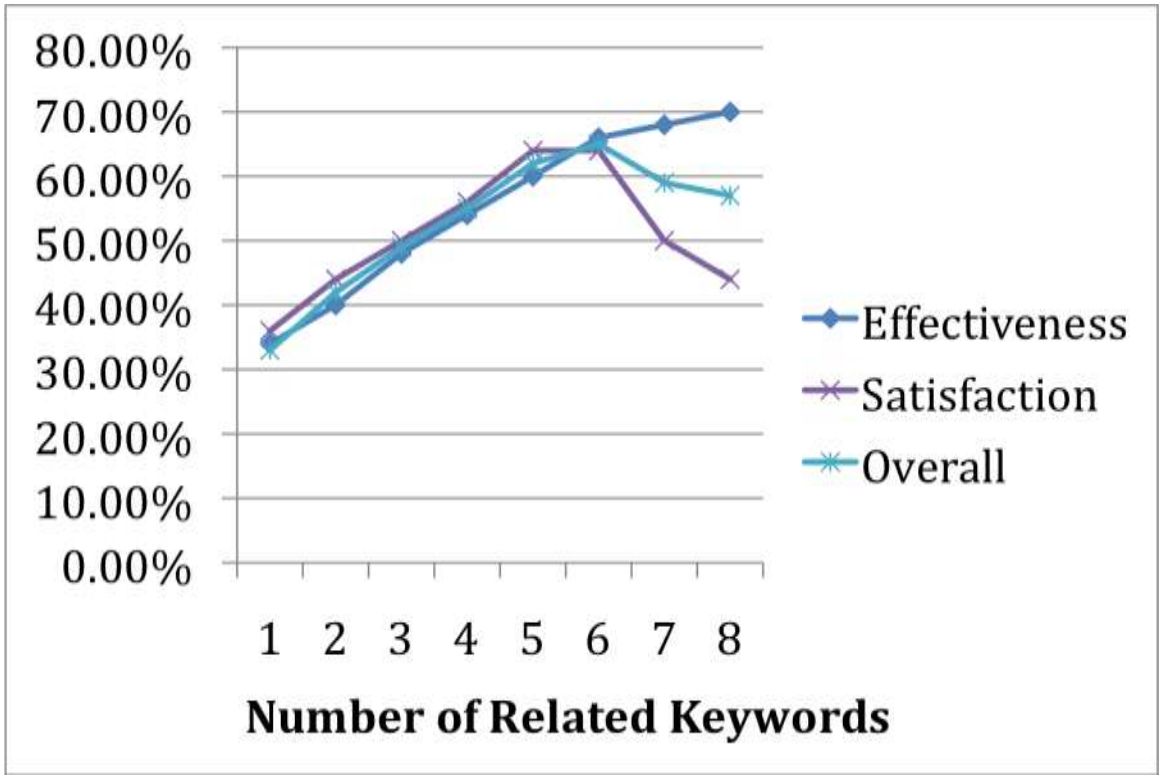

Figure 17- Usability graph of Related Searches

As seen in the above figure, the effectiveness of the related search increases with the increase in the number of keywords but the user satisfaction decreases because of the cluttering of the result page. It seems to have the highest overall score when number of search terms are six and score deteriorates with the increase in the keywords.

6. Conclusion

There are many ways we can improve the search experience of the Yioop user. In this project, we have focused on the user search history to improve Yioop. We have created a Firefox extension that provides user benefit by giving a graphical representation of the user-clicked pages. With this, user could be able to see the history instead of reading through the list of searches.

This extension also re-ranks the Yioop result based on the user-clicked links in other search engines. By using the top five results in the Yioop result, the average recall value has improved from 0.23 to 0.56 for these four search queries. This shows that the re-rank feature has improved the relevant results. Also, the average precision value has improved from 0.33 to 0.51. This shows that the re-rank feature has improved the count of more relevant results than the irrelevant ones. Both these measures show that the re-rank feature provides a better personalized search experience to the user.

The extension also provides a list of related keywords for a given search query. Based on the experiments performed, displaying six related keywords provide better user experience. By taking advantage of the past user searches, this project improves the overall search experience and search result of Yioop users.

7. References

[1] Pbb., Verruckt., Brettz9., Voronwe., Chee, Philip., Crookedfoot., . . . NikolayBot. (2012). Building an extension | MDN. Retrieved from https://developer.mozilla.org/en/Building_an_Extension

[2] Shen, Xuehua., Tan, Bin and Zhai, ChengXiang. (2005). Implicit User Modeling for personalized Search. ACM 1595931406/05/0010

[3] Skierpage., Imorchard., Takenbot., Jking3142., jswisher., Andismith., . . . Dria. (2012). Canvas Tutorial| MDN. Retrieved from https://developer.mozilla.org/en/Canvas_tutorial

[4] Fruchterman, M. J., and Reingold, M. (1991). Graph drawing by force directed placement. Software – Practice and Experience, 21, 1129-1164

[5] Vijaya Pamidi. (2010). Smart Search: A Firefox Add-On to Compute a Web Traffic Ranking. Retrieved from http://www.cs.sjsu.edu/faculty/pollett/masters/Semesters/Fall10/vijaya/CS%20298%20Project%20Report.pdf

[6] Force-based algorithms (graph drawing). (2012). Retrieved from http://en.wikipedia.org/wiki/Force-based_algorithms_(graph_drawing)

[7] P. A. Tipler.(1982). Physics, 2nd edn. New York, Worth Publishers

[8] Coulomb's law. (2012). Retrieved from http://en.wikipedia.org/wiki/Coulomb%27s_law

[9] Romano, Asaf., Deakin, Neil., Mills, Dan. and Spitzer, Seth. (2012). Places | MDN. Retrieved from https://wiki.mozilla.org/Places

[10] Okapi BM25. (2012). Retrieved from http://en.wikipedia.org/wiki/Okapi_BM25

[11] Pollett, Chris. (2011). CS297 Fall 2011. Retrieved from http://www.cs.sjsu.edu/faculty/pollett/267.1.11f/

[12] Skierpage., Takenbot., Taken., Dfrios., Mickiboy., Sheepy., . . . Amaltas. (2012). SVG|MDN. Retrieved from https://developer.mozilla.org/en/SVG