RECIPE SUGGESTION TOOL

A Writing Project

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 Computer Science

By

Sakuntala Gangaraju

May 2011

© 2011

Sakuntala Gangaraju

ALL RIGHTS RESERVED

SAN JOSÉ STATE UNIVERSITY

The Undersigned Writing Project Committee Approves the Writing Project Titled

RECIPE SUGGESTION TOOL

By

Sakuntala Gangaraju

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

____

Dr. Chris Pollett, Department of Computer Science 05/20/2011

____

Dr. Jon Pearce, Department of Computer Science 05/20/2011

____

Dr. Tsau Young Lin, Department of Computer Science 05/20/2011

ACKNOWLEDGEMENTS

I would like thank to Dr. Chris Pollett, for his encouragement, technical advice and guidance throughout this project. I also want to thank my committee members Dr. Jon Pearce and Dr. Tsau Young Lin for their suggestions, time and support. I also would like to thank my husband, Pavan Darvemula for his constant support throughout this project and Masters Program.

ABSTRACT

There is currently a great need for a tool to search cooking recipes based on ingredients. Current search engines do not provide this feature. Most of the recipe search results in current websites are not efficiently clustered based on relevance or categories resulting in a user getting lost in the huge search results presented.

Clustering in information retrieval is used for higher efficiency and better presentation of information to the user. Clustering puts similar documents in the same cluster. If a document is relevant to a query, then the documents in the same cluster are also relevant.

The goal of this project is to implement clustering on recipes. The user can search for recipes based on ingredient.

Table of Contents

1. Introduction 8

2. Clustering 10

2.1 Clustering in Information retrieval 10

2.2 Clustering based on Minimum Spanning Tree 11

2.3 Preliminary Work 12

3. Yioop! Search Engine 14

3.1 Introduction 14

3.2 System Architecture 14

3.3 Yioop Installation and Configuration 16

3.4 Managing Crawls 16

3.5 Yioop! User Interface 18

3.6 Modifications to Yioop! Architecture and User Interface 20

4. Design and Implementation 22

4.1 Design 22

4.2 Implementation 24

5. Testing and Results 31

5.1 EdgeWeight Boosting 31

5.2 Distance Measures 33

5.3 Performance Measure of distance measures 36

5.3 Performance Measure of distance measures 36

5.4 Scalability Test 37

6. Conclusion 39

References 40

List of Figures

Figure 1 : Applications of clustering in information retrieval 11

Figure 2 : Sample output for creating 3 clusters using Kruskal's algorithm 13

Figure 3 : Directory Structure of Yioop Search Engine 15

Figure 4 : Yioop Manage Crawl Page 17

Figure 5 : Yioop Crawl Options Page 18

Figure 6 : Yioop! User Interface Page 19

Figure 7 : Yioop! User Interface Page – Search Results 19

Figure 8 : Added Post Processors Options 20

Figure 9 : Sample output of result in Yioop! 30

Figure 10 : Error rates for Edge weight scenarios 32

Figure 11 : Error rates for distance measures 35

Figure 12 : Performance Comparison of Distance 36

Figure 13 : Performance Comparison of distance measures 36

Figure 14 : Scalability test results 37

Figure 15 : Graphical representation of Scalability test results 38

1. Introduction

Search engines have made access to information easy. One only needs to get connected to the internet to get the information one needs. When searching for cooking recipes, sometimes a user may prefer to search based on ingredients. The search results presented to the user are not relevant sometimes. The user might get lost in the huge sets of results presented. For example, a search for potatoes in Google displays information on potatoes rather than recipes of potatoes. If the search is made more specific, like recipes on potatoes, it displays links to websites which lists recipes on potatoes. There is a need to efficiently cluster the results based on relevance.

Clustering also known as cluster analysis involves grouping of data into clusters such that similar data belong to the same cluster. It is used in many fields which include data mining, information retrieval, and pattern recognition and image analysis. In particular in information retrieval, it is used for presenting the information efficiently and effectively to the user.

The goal of this project is to apply clustering on recipes. The user can search for recipes based on ingredient. The results presented are only recipes and are more relevant to the query. Yioop! Search Engine is used to perform the crawling and presenting the results to the user. Clustering of recipe pages is done after crawling is stopped. The recipes are represented as a tree and Kruskal's Minimum Spanning Tree algorithm is applied to find the minimum spanning tree. Clusters are formed by removing the most weighted edges from the minimum spanning tree. Breadth-first search is then implemented to group the recipes into clusters. Common ingredient for each cluster is determined. This ingredient uniquely identifies the cluster.

Yioop! is a GPLv3 open source search engine written in PHP. It is being developed by Dr. Chris Pollett. It has many features useful for implementing clustering of recipes. One of the features of Yioop! is that it allows users to add meta words to the documents. Meta words are the words that are not in the downloaded documents, but added to the index as

if they were in the document. This feature helped in retrieving the recipe pages for clustering and later for retrieval of search results for the query.

In Section 2 of this report Clustering and its applications in information retrieval are discussed. Implementation of the clustering algorithm used in this project is also discussed in this section. In Section 3, Yioop! Search Engine architecture and its features used in the project are discussed. Modifications made to the Yioop! Search Engine to fit for this project is also listed in this section. Design and Implementation of the clustering on recipes are discussed in Section 4. Three different distance measures are tested and the results are presented in Section 5. The Conclusion is presented in Section 6 followed by references.

2. Clustering

Clustering, also known as cluster analysis, involves grouping a set of observations into clusters so that similar observations fall into the same cluster. It is a common technique used for statistical data analysis and is used in many fields including machine learning, data mining, pattern recognition, image analysis, information retrieval and bioinformatics.

Different types of Clustering include:

- Hierarchical Clustering – These algorithms find successive clusters using previously established clusters. These can be agglomerative (“bottom-up”) or divisive (“top-down”). Agglomerative begin with each element as one cluster and merge them in to a larger clusters. Divisive begin with the whole set and divide it into smaller clusters.
- Partitional Clustering – These algorithms determine all clusters at once.
- Density based Clustering – These algorithms are devised to discover arbitrary-shaped clusters. In this approach, a cluster is regarded as a region in which the density of the data objects exceeds a threshold.
- Subspace Clustering – These algorithms look for clusters that can only be seen in a particular projection (subspace or manifold) of the data.

An Important step in clustering is to select a distant measure, which determines how the similarity of two elements is calculated. Common distance measures include Euclidean distance, Manhattan distance and Hamming distance.

2.1 Clustering in Information retrieval

The cluster hypothesis in information retrieval states that if a document from a cluster is relevant to a search request, it is likely that other documents from the same cluster are also relevant. This is because clustering puts similar documents in similar clusters with respect to relevance.

The following table lists some of the applications that use clustering in information retrieval.

Application	What is clustered	Benefit
Search result clustering	Search results	More effective information presentation to user
Scatter-Gather	Subsets of collection	Alternative user interface: “search without typing”
Collection Clustering	Collection	Effective information presentation for exploratory browsing
Language modeling	Collection	Increased precision and/or recall
Cluster based retrieval	Collection	Higher efficiency: faster search

Figure 1 : Applications of clustering in information retrieval

Source: http://nlp.stanford.edu/IR-book/html/htmledition/clustering-in-information-retrieval-1.html

2.2 Clustering based on Minimum Spanning Tree

In these algorithms, the data is represented in a minimum spanning tree where edge weight is the distance between two points. Clusters are obtained by strategically removing the edges. Different algorithms have been proposed for constructing the minimum spanning tree of which Prim’s algorithm and Kruskal’s algorithm are the most commonly used.

Kruskal’s algorithm is used for constructing the minimum spanning tree for this project. The data is represented as an undirected weighted graph. The implementation is done in two parts.

I. Kruskal’s algorithm is implemented to find the minimum spanning tree of the graph. It finds the subset of edges which includes all the vertices and the total weight of the edges are minimized. The algorithm includes steps:

1. Create a forest F (a set of trees), where each vertex in the graph is a separate tree.
2. Create a set S containing all the edges in the graph.
3. while S is nonempty and F is not yet spanning

    i. remove an edge with minimum weight from S

    ii. if that edge connects two different trees, then add it to the forest, combining two trees into a single tree

iii. Otherwise discard that edge.

II. Clustering is done by deleting the most expensive edge from the minimum spanning tree.

2.3 Preliminary Work

The first phase of this project involved implementing clustering based on minimum spanning tree constructed through Kruskal's algorithm. Implementation was done in PHP.

The edges of the graph with weights and the number of clusters to be formed were given as the input. Implementation includes creating the minimum spanning tree using Kruskal's algorithm.

Each vertex was considered as a separate tree. Each of the minimum edges was added to the minimum spanning tree if it did not create a cycle. The algorithm was implemented until all the vertices were covered. The minimum spanning tree obtained by applying Kruskal's algorithm was used to create the required number of clusters.

Clusters were formed by removing the most expensive edge from the minimum spanning tree. Removing an edge from the minimum spanning tree creates two clusters. The process was repeated until the required number of clusters was formed. Since removal of one edge creates two clusters, the number of edges to be removed was one less than the required number of clusters. Breadth-first search was implemented to group the vertices into clusters.

The output displayed the input tree, the minimum spanning tree edges and the clusters of vertices. Figure 2 shows a sample output for creating 3 clusters.

```
Data:
AB=>14
BC=>12
BD=>28
CD=>19
DE=>17
AF=>11
FC=>19
AE=>10
AD=>30
BE=>30

Minimum Edges:
AE=>10
AF=>11
BC=>12
AB=>14
DE=>17

Clusters:
Cluster 1={D}
Cluster 2={E,A,F}
Cluster 3={B,C}
```

Figure 2 : Sample output for creating 3 clusters using Kruskal's algorithm

This implementation of the algorithm is used in the project to cluster the recipes. The recipes are represented as the vertices of the tree. The edge weights are calculated by representing the ingredients as vectors and finding the Euclidean distance between the vectors. This implementation is later extended to find the common ingredient for each cluster of recipes.

3. Yioop! Search Engine

Yioop! Search Engine (Pollett, 2011) is a GPLv3, open source, PHP search engine developed by Dr. Chris Pollett. It was chosen to implement the clustering technique as it is based on open source licensing. The version used for this project is version 0.66.

3.1 Introduction

The Yioop! search engine is designed to allow users to produce indexes of a web-site or a collection of web-sites whose total number of pages are in the millions. The user can have control over the exact sites which are being indexed with Yioop! and over the kinds of results that a search will return.

The Yioop! uses its own web archive format and indexing technologies to handle the crawl data besides database to manage things like users and roles. Using web archives, crawls can be mirrored among several machines to speed up serving search results.

It has a web Interface to select seed sites for crawls. One can also configure it to crawl specific site or domain or collection of sites and domains. It has a GUI form that allows the users to specify the meta words to be injected into an index based on whether a downloaded document matches a url pattern. The web interface was used in this project to feed to seed sites of the recipe web sites. Meta words were to the recipe pages added during crawling.

3.2 System Architecture

Yioop Search Engine is based on the Model-View-Controller (MVC) design pattern. It is written in PHP. The system requirements for executing Yioop are:

1. A web server
2. PHP 5.3 or better
3. Curl libraries for downloading web pages

Figure 3 shows the directory structure of the Yioop! v0.66.

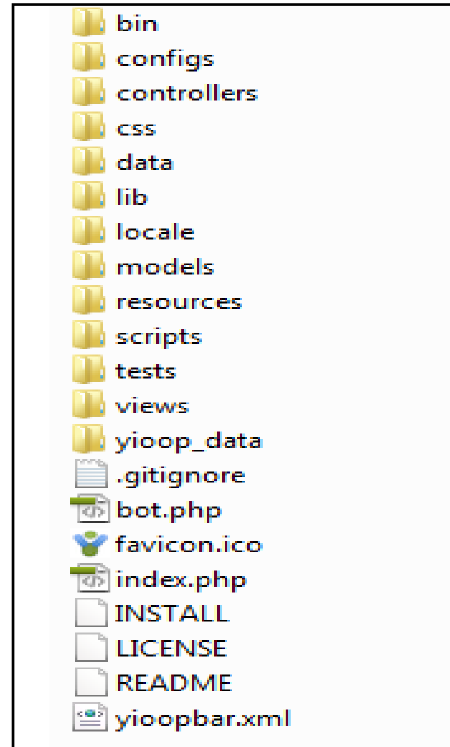


Figure 3 : Directory Structure of Yioop Search Engine

The bin folder contains the queue-server.php and fetcher.php. The queue-server generates the messages and schedules needed by fetcher. A schedule has list of urls to be crawled and the message has information about what kind of processing should be done. The fetcher crawls the urls listed in the schedules. It downloads the summaries of pages and sends the summaries back to queue-server which later adds to the index. The queue-server also keeps track what urls are seen and robots.txt file.

The index.php handles the queries coming to Yioop!. It has a query string with c and and arg as variables. The c= part says which controller to be used. The arg= part handles which data must be retrieved from which models and which view with what elements should be displayed back to the user.

The controller folder has the controller classes. The controller coordinates the communication between a model and view in a MVC design pattern. For example, a search_controller.php gets the search result for a query from phrase_model.php and sends the data to search_view.php.

The lib folder contains the classes used for processing the pages while crawling and subsequent indexing of the data. The processors folder inside the lib folder has the processors used for processing a page. The mime type of the page is determined by the fetcher and based on the mime type the fetcher calls the appropriate processor. Lib folder has another subfolder which has the iterators used while displaying the search results for the query.

The views folder has the subclasses of the views used for displaying various activities of Yioop! Search Engine. It has other sub folders like elements, layouts and helpers which have common functions used across views to output a web page.

The models folder has the subclasses of the models used for access to the database and archive folders used for storing the summaries of the pages. It has a datasources subfolder which has the sql queries to access different databases used for Yioop!.

Yioop! creates a Work Directory during installation. This folder contains a cache sub folder which contains cache data. It maintains two sub folders inside it which maintains the time stamped archive and index data used by the Yioop! Search Engine.

3.3 Yioop Installation and Configuration

The Yioop! application can be downloaded from download page at seekquarry.com. The set up environment used for this project was:

Operating System: Windows Vista

Web Server Package: XAMPP

XAMPP is an open source cross platform web server solution stack package consisting Apache, PHP and MySQL that are needed for the execution of Yioop!.

3.4 Managing Crawls

To start a crawl, the queue server and fetcher should be running. The following commands are used on a command prompt:

to start a queue-server:

```
php queue_server.php terminal
```

to start a fetcher:

```
php fetcher.php terminal
```

Create a Crawl

Description: [ ] Start New Crawl Options

Currently Processing

Description: CS Dept SJSU Stop Crawl
Time started: Fri, 17 Sep 2010 11:08:06 -0700
Server Peak Memory: 505443376
Fetcher Peak Memory: 129869568
WebApp Peak Memory: 7707872
Visited Urls Count: 1364
Total Urls Extracted: 7548
Most Recent Fetcher: ::1

Most Recent Urls

http://www.cs.sjsu.edu/CRC/rules/lab_rules.html
http://www.cs.sjsu.edu/CRC/rules/statement.html
http://www.cs.sjsu.edu/CRC/images/rules-title.gif
http://www.cs.sjsu.edu/100w/html/style_guide.html
http://www.cs.sjsu.edu/CRC/rules/images/dept_logo.gif
http://www.cs.sjsu.edu/CRC/rules/images/lab_logo.gif
http://www.cs.sjsu.edu/CRC/rules/images/state_logo.gif
http://www.cs.sjsu.edu/100w/assets/images/autogen/Course_Policies_Hbuts1.gif
http://www.cs.sjsu.edu/%7Erchun/cadence/index.htm
http://www.cs.sjsu.edu/~ravila/_derived/up_cmp_aftrnoon010_vbtn.gif

Previous Crawls

Description:	Timestamp:	Visited/Extracted Urls:	Actions:		
test	**1284746704** Fri, 17 Sep 2010 11:05:04 -0700	32/493	Resume	Search Index	Delete

Figure 4 : Yioop Manage Crawl Page

Source: "SeekQuarry" (2011)
http://www.seekquarry.com/

The Manage Crawl activity looks similar to the Figure 4. While crawling, statistics of the crawl are shown along with a 'Stop Crawl' button. Clicking on this button stops the crawl. During crawling, a sequence of index shards, which keep track of which word occurs in which document, is written. An IndexDictionary of which word appears in which shard is also written.

Previous crawls are listed at the bottom of the page. They provide information on timestamp and the urls visited. They also list actions that can be performed on as particular crawl. Figure 4 shows information about the crawl previously made. The resume action lets the corresponding crawl to resume crawling. The second action, set as index, lets the user to set index point to this crawl. Once it is set, the action is changed to 'Search Index'. The third action 'Delete' deletes the crawl information from the cache.

Figure 5 : Yioop Crawl Options Page

Source: "SeekQuarry" (2011)
http://www.seekquarry.com/

In the Crawl options page, there is an 'options' link besides the 'Start New Crawl' button. Clicking on the options should display a page similar to Figure 4.

Two kinds of crawls can be performed by Yioop! – web crawl and archive crawl. Web crawl crawls the sites on the web. Figure 5 shows the different options a user can set for the web crawl. The user can set the crawl options, crawl order, allowed and disallowed sites to crawl and seed sites. The user also has an option to add meta words to documents. Here a meta-word is a word which wasn't in a downloaded document, but which is added to the inverted-index as if it had been in the document. Archive crawl performs a crawl of data that has been previously stored in a supported archive format.

3.5 Yioop! User Interface

The main search form for Yioop! is shown in Figure 6.

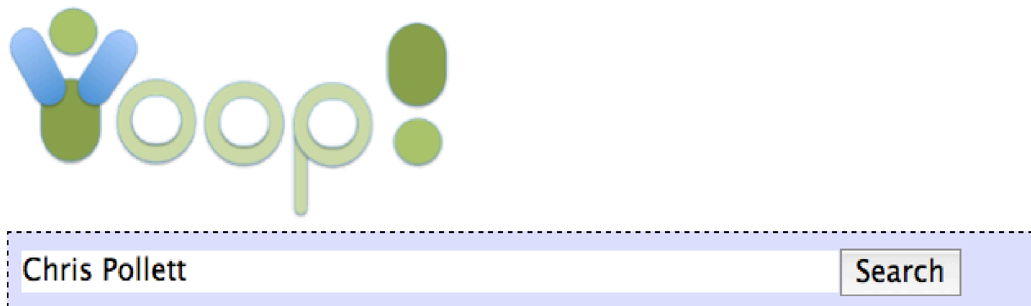
Developed at SeekQuarry

Figure 6 : Yioop! User Interface Page

Source: "SeekQuarry" (2011)

http://www.seekquarry.com/

One can perform a search by typing a query in the search form field and clicking the search button. Figure 7 shows a results search query – "Chris Pollett".



Query Results: (Calculated in 0.04003 seconds. Showing results 0 - 10 of 62)

Chris Pollett's Homepage

Chris Pollett Homepage contains... **Chris Pollett** Homepage contains information
http://www.cs.sjsu.edu/faculty/pollett/ Rank: 417.38 Rel: 12.52 Score 2906 Cached. Similar. Inlinks.
IP:130.65.86.46.

About **Chris Pollett**'s Web Site

About **Chris Pollett**'s Web Site Standards,... About **Chris Pollett**'s Web Site Standards, browser... and links to archived versions. **Chris Pollett** > About this site... and links to archived versions. **Chris Pollett** > About this site , , .
http://www.cs.sjsu.edu/faculty/pollett/about.html Rank: 37.99 Rel: 12.44 Score 339. Cached. Similar. Inlinks.
IP:130.65.86.46.

Figure 7 : Yioop! User Interface Page – Search Results

Source: "SeekQuarry" (2011)

http://www.seekquarry.com/

Each of the results displayed will have a title which is a link to the page that matches the query term. A brief summary of the page is displayed below the title. The document rank, relevance and overall score are displayed.

Yioop! supports a variety of search box commands and query types. Meta-words added to the documents can be used as keywords to search for specific documents. For example, filetype: extension returns documents found with the extension. So a search SJSU fieltype:pdf will return pdf documents containing the words SJSU.

3.6 Modifications to Yioop! Architecture and User Interface

For this project the following modifications were made:

1. **Addition of new folder 'components'**: A new folder 'components' was added to the 'lib' folder. This folder includes subclass recipe component which performs processing of the recipe pages while crawling and clustering the recipes after crawling. While crawling the recipe component detects a recipe page and extracts the ingredients from the page. Once the crawl is stopped, the recipe component then extracts the recipes to perform the clustering. Clustering based on Kruskal's Minimum Spanning Tree implementation is included in the recipe component.

2. **Modification to the Crawl options user interface**: A checkbox labeled 'Recipe Processor' was added to the Yioop! crawl options interface. Clustering of recipes is performed when this option is selected. The following figure shows the addition to the user interface:

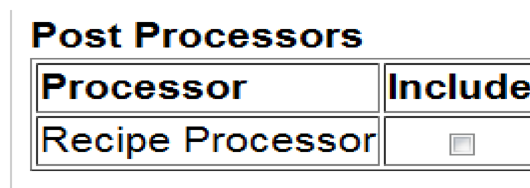


Figure 8 : Added Post Processors Options

3. **Modification to Html processor of Yioop!:** Html processor of Yioop! was modified to detect Recipe pages and extract only the ingredients. The Html processor processes a recipe page twice. Once using the regular process method of the Yioop! and second time using the process method of the recipe component. The summary extracted using the recipe components process method is marked as 'recipe'. This later helps to add meta words to the recipe pages.

4. **Modifications to fetcher:** The fetcher was modified to detect the documents marked as 'recipe' and add the meta word, 'recipe' to the document while building the mini inverted index. This meta word was used in the recipe component to extract the recipes for clustering.

5. **Modifications to queue-server**: The queue-server was also modified to call the recipe component's post processing method when the crawling is stopped. The index archive to be used is sent as an argument during the function call.

6. **Addition of meta words to recipe pages:** A meta word 'ingredient' was added to the recipes to cluster them based on the ingredient. This meta word is used to query the Yioop!. One needs to type 'ingredient: query term' to search for recipes based on the ingredient. For example, one needs to type 'ingredient:shrimp' to get recipes with shrimp as ingredient.

7. **Addition of new helper**: A new helper, displayResults, was added to the helpers used to automate the task of outputting certain kinds of web tags. This helper displays the ingredients as a list when the search view hits a recipe page while displaying.

4. Design and Implementation

The goal of this project is to display recipes based on the ingredient typed by the user in the Yioop Search Engine. For this, the recipe pages should be crawled, clustered and indexed. The study of Yioop! Search Engine helped in understanding how the crawler works, how the data is indexed for later retrieval.

4.1 Design

The Yioop! Search Engine v0.66 was thoroughly analyzed. The following are the observations made:

The queue-server is the coordinator for the crawls. The fetcher crawls and downloads the summaries of pages. The queue server initially generates messages and schedules for the fetcher. A schedule is the data to process and a message has control information about what kind of processing should be done. The fetcher processes the schedule and sends the summaries of pages to the queue-server which later merges the result into the index.

To build a database of recipes for this project, recipe websites should be given as seed sites for the crawl. As indexing is done while crawling, the clustering algorithm can be applied as soon as the crawl is stopped. The retrieval of pages for clustering is also faster.

Yioop! has different processors to extract page summaries of different mime types. The fetcher while processing the pages determines the mime type of the pages. Based on the mime type, it determines which processor to use.

To cluster recipe pages based on the ingredient, extraction of the ingredients list from the pages is sufficient. Since recipe pages are text, the mime type of these pages is text/html. The html processor will be used by the fetcher to extract ingredients from the recipe pages. This processor should be modified so that when the fetcher hits a recipe page, it should extract only ingredients as description.

Yioop! has a feature to add meta-words for the documents. Here, meta-word is a word which was not in the downloaded document but which is added to the index as if it has been in the document. The fetcher adds a list of meta-words to the documents while building a mini-inverted index for the downloaded pages. The same meta-word can be prefixed while querying Yioop!.

This feature of Yioop helps in adding meta-words for recipe pages. This can be used in two stages. Initially while crawling, the fetcher can add a meta-word 'recipe' for recipe pages. This meta word can be used to retrieve recipe pages for clustering. After clustering a meta word 'ingredient' can be added to pages. This meta word can be used while querying Yioop! for recipes based on ingredient.

The study of how the fetcher and the queue-server create the index helped in understanding how to add the clusters of recipes to the index. The fetcher builds a partial inverted index called index shard for the current batch of downloaded pages. The fetcher also adds the meta words while doing this. The queue-server merges this partial inverted index to the index of the current generation of crawl. The complete inverted index for the whole crawl is built out of these inverted indexes of generations.

Since the clustering of recipes is done after the crawl is over, these steps are to be repeated to add the results of clustering to the index. Adding the meta-word 'ingredient' should also be done at this point.

To summarize, the clustering algorithm should be implemented after the crawling is stopped. While crawling the fetcher should be able to identify the recipe page and download only the ingredients of the recipe. The fetcher should also add a meta-word 'recipe:all' to the pages which is later used for retrieving the recipe pages for clustering. After clustering the recipes, common ingredient should be determined for each cluster. A meta-word 'ingredient:<common ingredient>'for each of the recipes in a cluster needs to added while adding the pages to the index.

4.2 Implementation

For this project the following sites are given as seed sites:

http://allrecipes.com/

http://www.food.com/

http://www.betterrecipes.com/

http://www.foodnetwork.com/

http://www.bettycrocker.com/

The html processor was modified to extract the ingredients of the recipe when the fetcher hits a recipe page while crawling. XPath expressions were constructed to detect recipe pages from the above seed sites. If the XPath evaluates to true for a page, the fetcher determines that it is a recipe page and extracts the ingredients.

The following code snippet shows the XPath expressions for extracting the title and ingredients of the recipe pages of the seed sites.

```
$titles = $xpath->evaluate(
            "/html//div[@class='rectitle'] |
             /html//h1[@class = 'fn'] |
             /html//div[@class = 'pod about-recipe clrfix']/p |
             /html//h1[@class = 'recipeTitle' ");

$ingredients = $xpath->evaluate(
                    "/html//div[@class = 'ingredients']/ul/li |
                    /html//div[@class='body-text']/ul/li[@class= 'ingredient'] |
                    /html//ul[@class = 'clr']/li |
                    /html//div[@class ='recipeDetails']/ul[@class='ingredient_list']/li |
```

The Html processor was modified to add an attribute ‘subdoctype:recipe’ to mark the pages as recipe pages. This was later used in the fetcher while adding meta words to the documents. The fetcher while building the partial inverted index adds a meta word ‘recipe:all’ if the document has subdoctype:recipe set.

The next step in the implementation was done after the crawling is stopped. The recipe pages downloaded need to be retrieved for clustering. The meta word “recipe:all”

was used to extract the recipes. The following code snippet uses a function of Yioop! Search Engine to retrieve recipe pages.

```
$raw_recipes = $this->phraseModel->getPhrasePageResults(

                              "recipe:all",0, 100, false);
```

The function 'getPhrasePageResults' retrieves the formatted document summaries of documents that match the phrase – "recipe:all".

The ingredients list of the recipes retrieved had measurement and other adjectives along with the main ingredient. For instance, consider the ingredient, "1 lb Potatoes, boiled and mashed'. The main ingredient potato needs to be extracted from the ingredients listed. This was one of the challenging steps in the implementation. The ingredient list is unstructured data. Regular expressions were used to remove the measurement and special characters. The following code snippet shows extraction of the main ingredient.

```
if(!in_array($word,$measurements) && !in_array($word,$sizes)
   && !in_array($word,$prepositions) && !in_array($word,$misc)
                                                   && !in_array($word,$attributes))
{
        $ending = substr($word, -2);
        $ending2 = substr($word, -3);
        if($ending != 'ly' && $ending != 'ed' && $ending2 != 'ing')
        {
                $nouns[] = $word;

        }
```

Each recipe extracted will have only the main ingredient after this step.

The next step in the implementation was to construct the weight matrix for the recipes. These weights are needed to construct the graph initially and later apply Kruskal's algorithm to find the minimum spanning tree.

To construct the distance matrix, for each two recipes, an ingredients vector is constructed which includes ingredients of both the recipes. Ingredients which appear in both the recipes were considered only once. Then a 0-1 ingredient vector was constructed for each recipe. If the ingredient in the ingredients vector was present in the recipe it was marked 1 otherwise 0.

One of the important steps in clustering is choosing the distance measure. Three distance measures were considered – Dot product, Manhattan distance and Euclidean distance. Tests were conducted to find the error rate, which is the measure for number of recipes falling into the wrong clusters. Test case 5.2 in Section 5 of this report shows the results of the tests. It was observed that the Manhattan distance and Euclidean distance have better error rate than Dot product. Test case 5.3 in Section 5 shows the results of performance of each of the distance measure. Euclidean distance was selected as the distance measure for this project.

The Euclidean distance between the ingredient vectors of the recipes was calculated. This represents the distance between the recipes in the distance matrix. For instance consider the following four recipes after extracting the main ingredient from the ingredient list:

1. Whipped Potatoes - potatoes, cheese, cream, butter, garlic, pepper, paprika
2. Skewered Potatoes – potatoes, water, mayonnaise, chicken, rosemary, garlic
3. White Chocolate Cookies – butter, sugar, egg, cocoapowder, flour, chocolate
4. Chocolate Cake - chocolate, butter, salt, flour, sugar, vanillaextract

The ingredients vector for recipes, Whipped Potatoes and Skewered Potatoes will be – [butter, cheese, chicken, cream, garlic, mayonnaise, paprika, pepper, potatoes, rosemary, water].

The 0-1 ingredient vector for each recipe will be -

Whipped Potatoes - 1 1 0 1 1 0 1 1 1 0 0

Skewered Potatoes – 0 0 1 0 1 1 0 0 1 1 1

The Euclidean distance between the two recipes is calculated as:

Distance = $\sqrt{(1+1+1+1+0+1+1+1+0+1+1}$ = $\sqrt{9}$ = 3

The distances between all the recipes are calculated and are stored in the distance matrix. The distance matrix for all of the four recipes is as follows.

Whipped Potatoes -- Skewered Potatoes : 3

Whipped Potatoes -- White Chocolate Cookies : 3.3166247903554

Whipped Potatoes -- Chocolate Cake : 3.3166247903554

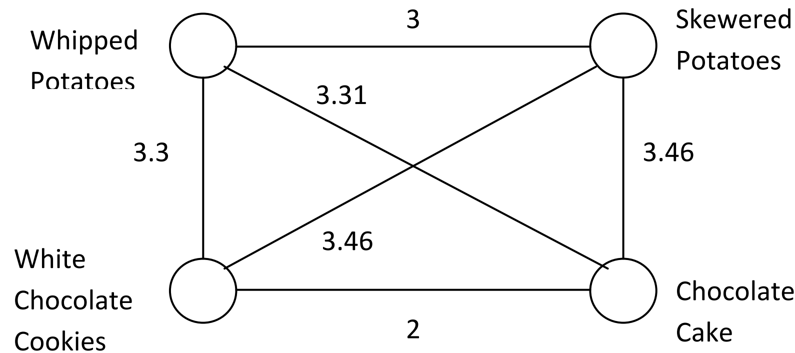
Skewered Potatoes -- White Chocolate Cookies :3.4641016151378

Skewered Potatoes -- Chocolate Cake : 3.4641016151378

White Chocolate Cookies -- Chocolate Cake : 2

The next step in the implementation was the application of clustering using Kruskal's Minimum Spanning Tree algorithm. This is done in three steps:

First, a graph was constructed using recipes as the vertices and the distances between them as the edge weights. The graph constructed is undirected graph. The following diagram represents the graph constructed for the four recipes with the edge weights calculated:



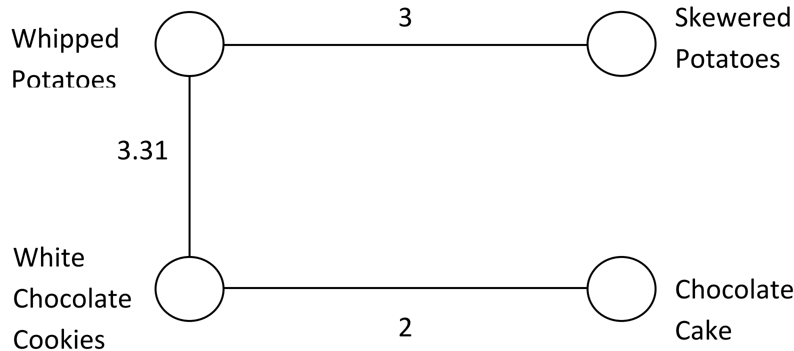
Kruskal's Minimum Spanning Tree algorithm is then applied on the tree. The result after applying the algorithm will be:

White Chocolate Cookies -- Chocolate Butter Cream : 2

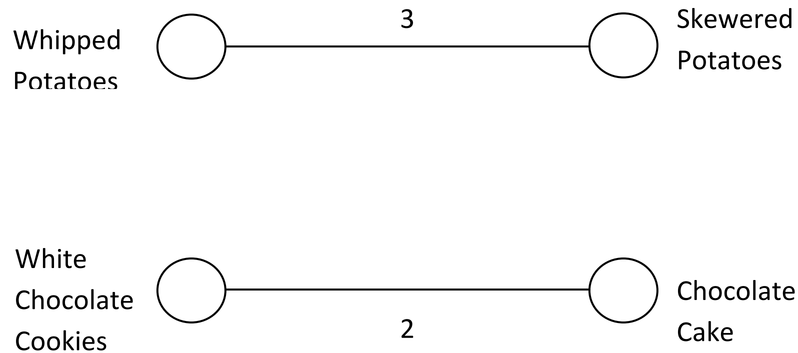
Whipped Potatoes -- Skewered Potatoes : 3

Whipped Potatoes -- White Chocolate Cookies : 3.3166247903554

The following shows the resultant minimum spanning tree:



Clustering is done by removing the most weighted edge from the minimum spanning tree constructed. The number of edges to be deleted is one less than the desired number of clusters. In the example, if two clusters are needed, the most weighted edge, Whipped Potatoes – White Chocolate Cookies, is removed from the minimum spanning tree. This results in two clusters as shown below:



Breadth-first search is implemented on each cluster to group the recipes in each cluster. In the example, the resultant clusters of recipes are:

Cluster 1 = {Whipped Potatoes, Skewered Potatoes}

Cluster 2 = {White Chocolate Cookies, Chocolate Cake}

The next step in the implementation was to find the most common ingredient of the recipes in each cluster. This was done by finding the number of occurrences of each ingredient of the recipes in the cluster. The ingredient which has the maximum number of occurrences will be the common ingredient of all of the recipes of the cluster. This ingredient should uniquely identify the cluster. A meta-word ‘ingredient: <common ingredient’> is added to all of the recipes of the cluster while adding the cluster results to the index.

In the example mentioned above, for Cluster 1, two ingredients appear in both the recipes – potatoes, garlic. Potato is given higher priority. Certain ingredients like garlic, salt, oil, pepper etc are basic ingredients and appear in almost all of the recipes. Such ingredients are given less priority. Similarly, for Cluster 2, chocolate is given more priority than butter.

Final step in the implementation is to add back the results of clustering to the index with the meta words. For this, a partial index, index shard, is created which stores one generation worth of document index. The following code snippet adds a new document to the index shard with the given summary offset.

```
$result = $index_shard->addDocumentWords($doc_key,  self::NEEDS_OFFSET,
            $word_counts, $meta_id, true, false);
```

Here, $doc_key is the document Id of the recipe page. $meta_id is an array of meta-words to be added to the document. The meta-id for the recipe pages in each cluster will be ‘ingredient:<common-ingredient of the cluster>’.

The next step was to adding the recipe pages summary to the WebArchiveBundle and the mini inverted index created to the IndexArchiveBundle. The current shard is added to the Dictionary and then merged.

The following code snippet shows the function calls made:

```
$index_archive->addIndexData($index_shard);
$index_archive->saveAndAddCurrentShardDictionary();
$index_archive->dictionary->mergeAllTiers();
```

This finishes the implementation. The results can be tested by querying the Yioop! Search Engine. To retrieve the recipes based on ingredient, one needs to type in 'ingredient:' followed by the ingredient name.

The following figure shows a sample output of the search made by ingredient – chicken.

ingredient:chicken Search

Query Results: (Calculated in 0.661279 seconds. Showing results 50 - 60 of 83)

Chicken "Bruschetta"

4 skinless, boneless chicken breast halves
2 T. olive oil
2 to 4 T. minced garlic
1 lg. tomato, cored & chopped
6 basil leaves, cut into small pieces
1/4 cup balsamic vinegar
1/2 cup shredded mozzarella cheese

http://chicken.betterrecipes.com/chicken-bruschetta.html Rank: 3.90 Rel: 0.00 Score 1.95 Cached. Similar. Inlinks.

Figure 9 : Sample output of result in Yioop!

5. Testing and Results

The following tests are conducted for a sample of 100 recipes. The formula used to calculate the error rates:

Error rate = (False Positives/total number of recipes)

Here False Positives refer to the number of recipes which appear in the wrong cluster.

5.1 Edge Weight Boosting

With Euclidean distance as distance measure and number of clusters as 16, it was observed that 84 recipes all fall into one cluster. One cluster has one false positive. The remaining 14 recipes form one cluster each. Error rate in this case is calculated as:

Error rate = 85/100 = 0.85

It was observed that recipes have lower edge weights with recipes having fewer ingredients and fewer matches when compared to recipes having more ingredients and more matches. Preference should be given to number of matches between the two recipe vectors. The edge weight should be boosted with the number of matches.

Boosting was done by counting the number of matches. Once the edge weight was calculated using a distance measure, the edge weight was multiplied with the number of non-matches. This will increase the edge weights between recipes will less number of ingredients and fewer matches, when compared with recipes with more number of ingredients but a better match.

After boosting the edge weights with number of non-matches, it was observed that 76 recipes fall into one cluster. The remaining 24 recipes still form one cluster each.

Error rate in this case is calculated as:

Error rate = 76/100 = 0.76

To improve the error rate, boosting of the edge weight is done if the main ingredients do match. It was observed that most of the recipes have the main ingredient in the title of the recipe. The title was used to extract the main ingredient of the recipe. This was done by comparing each ingredient with the title of the recipe, if there was a match, and then the ingredient was considered as the main ingredient. The main ingredients of two recipes were compared while calculating the edge weights. If the main ingredients do not match, then the edge weight is boosted again.

After boosting the edge weights by number of non-matches and by 10 if the main ingredients do not match, it was observed that the 7 recipes out of 100 were not in the right clusters. The error rate in this case is:

Error rate = 7/100 = 0.07

Boosting the edge weights reduced the error rate significantly. A comparison table lists the error rates for the above cases.

Edge weight scenarios	Error rate
Edge weights with no boosting	0.85
Edge weights with boosting by number of non-matches in the recipe vector	0.76
Edge weights with boosting by number of non-matches and non-match of the main ingredient	0.07

Figure 10 : Error rates for Edge weight scenarios

5.2 Distance Measures

An important step in clustering of data is the selection of distance measure. It calculates the similarity between the two elements of data. In this project, the distance between two recipes is calculated and it is represented as the weights of the edges in the minimum spanning tree. The following distance measures were used for testing the distance measure calculation between the recipes.

1. Dot product – Dot product of two vectors is obtained by multiplying the corresponding entries of the vector and summing the product.

   The dot product of two vectors $a = [a_1, a_2, ..., a_n]$ and $b = [b_1, b_2, ..., b_n]$ is defined as:

   $$a.b = \sum a_i b_i = a_1 b_1 + a_2 b_2 + ... + a_n b_n$$

2. Manhattan distance – Manhattan distance (d_1) is obtained by subtracting the corresponding entries of the vector and summing the absolute value of the difference.

   The Manhattan distance of two vectors $a = [a_1, a_2, ..., a_n]$ and $b = [b_1, b_2, ..., b_n]$ is defined as:

   $$d_1 (a, b) = \sum |a_i - b_i| = |a_1 - b_1| + |a_2 - b_2| + ... + |a_n - b_n|$$

3. Euclidean distance – Euclidean distance is the square root of the sum of the squares of the difference of the between the corresponding entries of the vector.

   The Euclidean distance of two vectors $a = [a_1, a_2, ..., a_n]$ and $b = [b_1, b_2, ..., b_n]$ is defined as:

   $$d (a, b) = \sqrt{\sum (b_i - a_1)^2} = \sqrt{[(b_1 - a_1)^2 + (b_2 - a_2)^2 + ... + (b_n - a_n)^2]}$$

The three distance measures were tested on 100 recipes and the error rate was calculated for each distance measure. The edge weights were boosted by a factor of number of non-matches and by a factor of 10 if the main ingredients of the recipes do not match.

The following observations were made for each distance measure:

Dot Product – The dot product between two vectors assigns 1 if the ingredients match and a 0 if they do not match. The edge weight calculated will be greater for similar recipes when compared with edge weight for non-similar recipes. Since the clustering algorithm cuts the maximum edge, for the algorithm to give right results the edge weights are negated and divided by the boost factors.

For example, consider the following three recipes:

1. Whipped Potatoes - potatoes, cheese, cream, butter, garlic, pepper, paprika
2. Skewered Potatoes – potatoes, water, mayonnaise, chicken, rosemary, garlic
3. White Chocolate Cookies – butter, sugar, egg, cocoapowder, flour, chocolate

The recipe vector for Whipped Potatoes and Skewered Potatoes will be – [butter, cheese, chicken, cream, garlic, mayonnaise, paprika, pepper, potatoes, rosemary, water].

The 0-1 ingredient vector for each recipe will be -

Whipped Potatoes (wp) - 1 1 0 1 1 0 1 1 1 0 0

Skewered Potatoes (sp) – 0 0 1 0 1 1 0 0 1 1 1

The dot product between the two recipes is calculated as:

wp . sp = 1*0+1*0+0*1+1*0+1*1+0*1+1*0+1*1+0*1+0*1 = 2

The recipe vector for Whipped Potatoes and White Chocolate Cookies will be – [butter, cheese, chocolate, cocoapowder, cream, egg, flour, garlic, paprika, pepper, potatoes, sugar].

The 0/1 ingredient vector for each recipe will be –

Whipped Potatoes (wp) – 1 1 0 0 1 0 0 1 1 1 1 0

White Chocolate Cookies (wcc) – 1 0 1 1 0 1 1 0 0 0 0 1

The dot product between the two recipes is calculated as:

wp . wcc = 1*1+1*0+0*1+0*1+1*0+0*1+0*1+1*0+1*0+1*0+1*0+0*1 = 1

The edge weight between similar recipes, Whipped Potatoes and Skewered Potatoes, is more than the edge weight between non-similar recipes, Whipped Potatoes

and White Chocolate Cookies. As the clustering removes the maximum weight edge, with these weights the algorithm will not give the right results. So, the edge weights need to be negated. Similarly, edge weight boosting factors need to reduce the edge weight by dividing.

With dot product as the distance measure, it was observed that the number of false positives counted to 10 out of 100. The error rate will be:

Error rate = 10/100 = 0.1

With Manhattan distance as the distance measure, it was observed that the number of false positives were 7 recipes out of 100. The error rate will be:

Error rate = 7/100 = 0.07

With Euclidean distance as the distance measure, the error rate as calculated in the first test case was 0.07.

Distance Measures	Error rate
Dot product	0.1
Manhattan distance	0.07
Euclidean distance	0.07

Figure 11 : Error rates for distance measures

It was observed that Manhattan distance and Euclidean distance gave better results for a sample of 100 recipes when compared to Dot product. So either Euclidean distance or Manhattan distance can be used as a distance measure.

5.3 Performance Measure of distance measures

The following table lists the time taken by each distance measure considered for a sample of 400 recipes.

Distance Measure	Time taken (in sec for 400 recipes)
Dot product	11
Manhattan distance	13
Euclidean distance	13

Figure 12 : Performance Comparison of Distance

It was observed that the Dot product takes less time to cluster the recipes when compared to Manhattan and Euclidean distances. The following graph shows the performance comparison of the three distance measures.

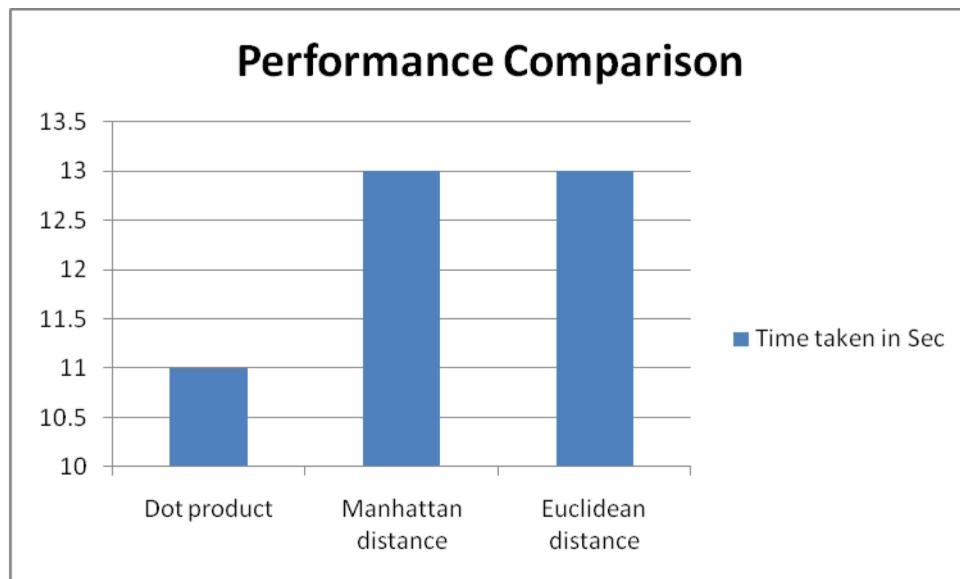


Figure 13 : Performance Comparison of distance measures (in sec for 400 recipes)

As the number of recipes increases, the difference in the time taken would be more significant. But in the previous test case, it was observed that the Manhattan distance and the Euclidean distance show a better error rate when compared to Dot product. So, either Manhattan distance or Euclidean distance can be used as a distance measure.

For this project, Euclidean distance was considered as the distance measure as it is more commonly used.

5.4 Scalability Test

Ingredient vectors are calculated between recipes and the distance between the vectors was calculated using Euclidean distance. Each recipe will have an edge with every other recipe. The distance between the recipes was considered as the edge weights of the graph constructed for clustering. As each recipe has an edge with every other recipe, scalability test was conducted to determine how the implementation scales as the number of recipes increases. The following observations were made:

Recipes Size	**Number of edges generated**	**Edges = n(n-1)/2**
10	45	(10*9)/2
20	190	(20*19)/2
30	435	(30*29)/2
40	780	(40*39)/2
50	1225	(50*49)/2
100	4950	(100*99)/2

Figure 14 : Scalability test results

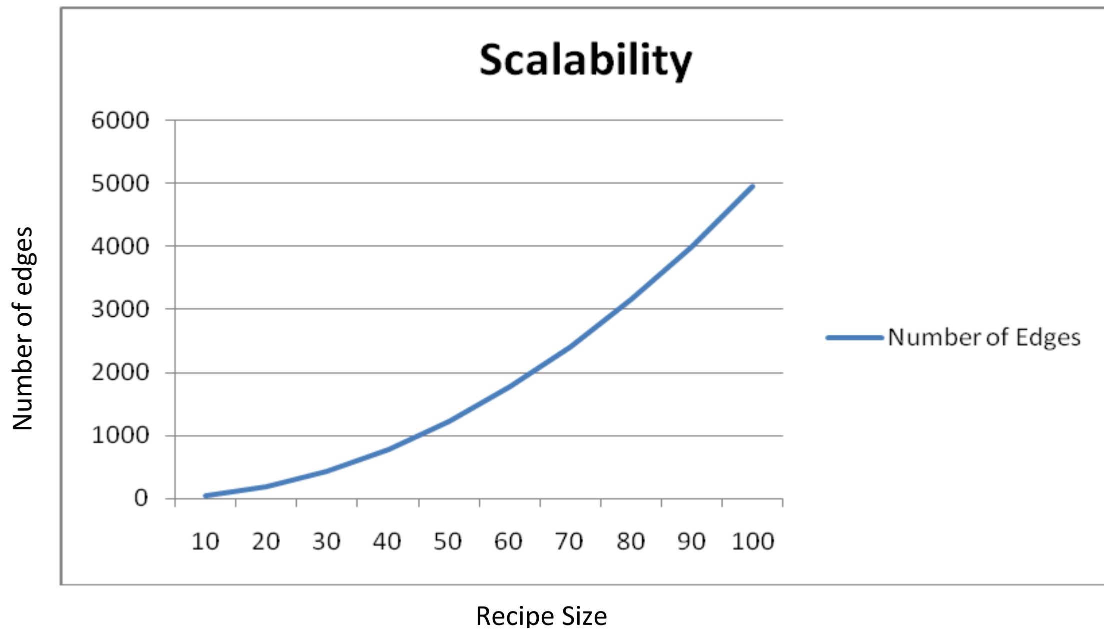


Figure 15 : Graphical representation of Scalability test results

It was observed that the number of edges created depends on the size of the recipes and is calculated as n(n-1)/2 where n is the number of recipes. As the number of recipes increases, the number of edges increases quadratically.

The clustering algorithm can be modified to reduce the number of edges generated. Clustering can be implemented in parts. Clustering can be done for every 1000 recipes. Similar recipes will have the same common ingredient representing the cluster and the same meta word is added to the recipe pages. So when a search is done, all the clusters of recipes with the common ingredient will be displayed. Instead of having a single cluster of similar recipes, clustering incrementally results in multiple clusters having the same ingredient identifying the clusters.

6. Conclusion

With information growing day by day, the results presented by the search engines are less relevant to the query typed in. The user gets lost in the huge sets of results displayed. There is a need for a domain based search engine. My project aims to provide a recipe based search engine. The user can search for recipes based on ingredient. The results presented are only recipes and are more relevant to the query.

In this project, a recipe based search is implemented by applying clustering on recipe pages. Clustering technique is being used in information retrieval for presenting information more efficiently and effectively. Clustering groups similar documents into one cluster. In this project, clustering is used to group the recipes into clusters based on the ingredients present in the recipes. Clustering based on Kruskal's Minimum Spanning Tree was used to cluster recipes. Common ingredient is determined for each cluster which identifies that cluster. When the user searches for recipes for this ingredient, all the recipes in the cluster are displayed. As the documents in one cluster are similar, the results presented are more relevant to the query. Yioop! Search Engine v0.66 is used for crawling the recipe pages and presenting the results to the user.

Yioop! v0.66 provided the necessary framework to implement clustering. Adding meta-words to the documents is one of the features Yioop! provides which is used effectively to cluster the recipes based on ingredient. We plan to include search for recipes based on ingredient in the future versions of Yioop! as well.

References

[1] *Yioop Documentation*. (n.d). Retrieved April 27, 2011 from SeekQuarry web page: http://seekquarry.com/?c=main&p=documentation#interface

[2] *Kruskal's algorithm - Wikipedia*. (n.d). Retrieved April 27, 2011 from Wikipedia web page : http://en.wikipedia.org/wiki/Kruskal%27s_algorithm

[3] *Cluster Analysis – Wikipedia*. (n.d). Retrieved April 27 2011 from Wikipedia web page: http://en.wikipedia.org/wiki/Cluster_analysis

[4] Manning, C. D., Raghavan, P., & Schutze, H. (2008). Clustering in information retrieval. In *Introduction to Information Retrieval* (16). Retrieved from http://nlp.stanford.edu/IR-book/html/htmledition/clustering-in-information-retrieval-1.html

[5] *Minimum Spanning Tree – Wikipedia*. (n.d). Retrieved April 27 2011 from Wikipedia web page: http://en.wikipedia.org/wiki/Minimum_spanning_tree

[6] *Euclidean distance – Wikipedia.* (n.d). Retrieved May 6 2011 from Wikipedia web page: http://en.wikipedia.org/wiki/Euclidean_distance

[7] *Dot product – Wikipedia.* (n.d). Retrieved May 6 2011 from Wikipedia web page: http://en.wikipedia.org/wiki/Dot_product

[8] *Taxicab geometry – Wikipedia.* (n.d). Retrieved May 6 2011 from Wikipedia web page: http://en.wikipedia.org/wiki/Manhattan_distance