Build a search engine it is not an easy thing to do. There are many interesting challenges that software engineers must solve in a way that millions of users must be satisfied every single day. Even the small components in a search engine must be optimized to satisfy a lot of requirements: quality, performance, accessibility and so on. For example, did you ever thought about how that small snippets of search results are built? Despite the importance of that small text to the end user, what serves as a "hint" of what is more relevant before the user click, people may ignore the clever solutions behind that snippet.

Example of text snippet that emphasizes the query terms

That "small" component of a search engine is executed, at least, for each result on the first page every query processed. That snippet is built from much larger texts found in webpages and you still get it all in milliseconds. That must be really fast...

This post will give you algorithms to build that snippets in a clever way, giving to the end-user a relevant part of the original text found in webpages/documents and doing that fast.

**Problem Definition:**

Suppose that you have two inputs: the user query and one text. Also you have one constraint: The limit of characters to show. Your task is select the most relevant parts of the original text (giving the user query terms and one constraint of the snippet length) to show to the user. "Most relevant" can be defined in different ways depending on your audience, which documents you are indexing and others things I can't imagine right now... But one thing that we can assume is that people may expect snippets that include some terms she/he used in query time. So, I'll break the problem in two parts:

- First, select the minimum text window from the original text that includes ALL query terms (I'll call that as \textit{MINWINDOW} algorithm).
- Second, select the most relevant parts from that window, subject to the length constraint. (I'll call that as \textit{CUT} algorithm).

Breaking that in two parts helps to abstract the concept of "most relevant" and left that to be solved further, also, it helps we use different strategies on the second part (what depends on many factors stated before). In that post I'll give you one strategy, but I'm planning a second post to give you other.
Suppose you have a search engine API that returns a list of positions on the original text that marks where each query term has matched. For example, suppose the text below:


And the query: lorem sed massa.

Your API may return something like that:

token (lorem) found at positions: {0, 89, 130}
token (sed) found at positions: {95, 123, 177, 199}
token (massa) found at positions: {70, 105, 117}

ps.: In the final of this post you can download a simple API I designed to validate the algorithms, which is based on the inverted indices. If you want a professional API that do that and it is also free, you may consider Apache Lucene/Solr. Both Lucene and Solr have mechanisms to retrieve highlighted snippets, check it out at Solr Highlighting documentation, in special, hl.snippets and hl.fragsize. Also, If you want to get an idea of how inverted indices API can be implemented in a scalable way, you may consider this link "Batman e a Escalabilidade" (in Portuguese).

Consider the positions lists as an input of the MINWINDOW algorithm. In the case above:

List 1 = { 0, 89, 130 };
List 2 = { 95, 123, 177, 199 };
List 3 = { 70, 105, 117 }.

The problem can be stated as: select one element from each list so that the difference between the minimum and the maximum of the selected elements is minimized.

If you are not worrying about the minimization, then you may get several solutions, for instance:

1. "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras id erat massa. Ullamcorper Lorem Sed". (At positions 0, 95 and 70)
2. "massa. Ullamcorper Lorem Sed". (At positions 89, 95 and 70)
3. "Lorem Sed ipsum massa". (At positions 89, 95 and 105)
4. "massa sed id Lorem". (At positions 130, 123 and 117)

The fourth snippet is the minimum window that includes all query terms, so the solution of MINWINDOW algorithm.

Now, I think we can start design the algorithm: One quick and dirty algorithm is the brute force one. In other words, verify all combinations. Of course that works, but you don’t want that! If you have a constant number of terms in the query, this algorithm is polynomial on the average length of lists. Although most of search engines limits the number of tokens in query, say K, this is still impracticable: O(avg^k). But this algorithm can be still useful as an Oracle for your unit tests.
It is possible design $O(\text{avg(list length)} \times K \times \log(K))$: Select the first element from each list as an initial solution. To improve the current solution, i.e. decrease the current window length, you must change one element selected from any of the lists in a way that you increase the current minimum element or decrease the maximum element selected so far. If the lists are sorted (ascendent), you can’t decrease the maximum element, then the only choice you have is try to increase the minimum. If you can’t increase the minimum then the current best solution cannot be improved and the algorithm finishes.

The current minimum can be selected using a minimum heap, which gives the log($K$) factor - $O(1)$ to select and $O(\log(K))$ to update the datastructure. The figure below shows the algorithms steps:

Here you have a Java implementation of the algorithm described above, but instead using a heap it’s implemented a linear search to find the current minimum, so the final complexity is $O(\text{avg(list length)} \times K^2)$:

```java
    //to avoid special cases, it is assumed that
    //the last element of each list is the Integer.MAX_VALUE
    //the algorithm returns the position of each term of
    //the minimum window
    static int[] solve(int[][] lists) {
        int m = lists.length;
        //the current selected element from each list
        int[] pos = new int[m];
        //the current best solution positions
        int[] sol = new int[m];
        //the score (window length) of current solution
        int currSol = Integer.MAX_VALUE;
        while(true) {
            //select the list that has the increasing minimum element
            int minList = argmin(pos,lists);
            //if you can't increase the minimum, stop
            if (minList == -1) break;
            //calculate the window size
            int minValue = lists[minList][pos[minList]];
            int maxValue = max(pos,lists);
            int nextSol = maxValue - minValue;
            //update the solution if necessary
            if(nextSol < currSol) {
                currSol = nextSol;
                System.arraycopy(pos, 0, sol, 0, m);
            }
            //update the current minimum element
            pos[minList]++;
        }
        return sol;
    }

    private static int argmin(int[] pos, int[][] v) {
        int min = Integer.MAX_VALUE;
        int arg = -1;
        for(int i = 0 ; i < v.length; ++i) {
            if(v[i][pos[i]] < min) {
                min = v[i][pos[i]];
                arg = i;
            }
        } return arg;
    }
```

The CUT Algorithm

After running the MINWINDOW algorithm you obtain the smallest window text that contains all query terms. But that window can be still larger than what you want to show to the users. In that case, you must cut the smallest window. A valid strategy is cut that window in several positions, giving to the end-user a sequence of fragments which keep as much queried terms as possible. It looks like the majors search engines do that... I'll left that strategy, referred here as FRAGMENTCUT algorithm, to the next post. Now, I'll try a different approach: Cut only the edges.

The EDGECUT Algorithm

The idea of this algorithm is cut words from the edges to preserve the text flow. The heuristic is that users may like that because they can understand better the snippet compared to the FRAGMENTCUT strategy.

There are several different cuts on the edges that produces snippets that fits the length constraint. So, what is the best one? One may prefer the cut that keep the maximum number of terms, other may prefer the cut that preserves the query terms that maximize the tf-idf. To abstract that, suppose you have an objective function F that encodes what you prefer. In that case, let's design an efficient algorithm:

First, the number of all edge cuts is \(O(M^2)\), where \(M\) is the number of terms in the query. So the brute force algorithm is \(O(M^2)\). If we consider \(M\) a constant, which is true in many search engines, that algorithm is \(O(1)\). But I'll ignore that and show that EDGECUT can be done in \(O(M + M^2 \log(M))\).

For this algorithm consider as input a list of positions of terms matched selected by the MINWINDOW algorithm. Also, the length of each term. Using the previous example:

```java
//max: the constraint of snippet length
//pos: the positions of each term and lengths
//returns: the first and last position
public static int[] cut(TokenInfo[] pos, int[][] v) {
    int n = pos.length;
    int arg = argmax(pos, v);
    return v[arg][pos[arg]];
}
```

Java implementation of EDGECUT algorithm

First sort that list of positions (keep track of the lengths). Next it's very simple, you just need evaluate the objective function in every window size equals the constraint length. You start moving that window from the first position to the last one.
You can download all sourcecode here. This file includes the algorithms described here plus simple tokenization, inverted indice search and text terms emphasize algorithm.

Like 13 people like this. Sign Up to see what your friends like.

POSTED BY RODRIGO REZENDE AT 2:57 AM

olde