Patrick Dengler
String Search Algorithms
CSEP-521
Winter 2007

This paper will address the various different ways to approach searching for strings within strings. It will analyze the different algorithms that have been developed over time. Searching for strings is almost ubiquitous in its applications. From simple text search in a document, to structured text searching in a database, to unstructured searching of large amounts of text (i.e. the internet), string searching is a very common function of many end-user applications which involve data.

From Bad

One thing we learned was to start somewhere; in this case, we will start with brute force. Assume we have a reasonably small string (of size m) which we want to find if it exists at least once in a larger string (of size n). The brute force approach would be to simply examine each character of the string for a match:

```
Let S = larger string (m= length)
Let F = string to find (n= length)
for i = 1 to m
      for j = 1 to n
            if S[i+j] != F[i+j]  exit
            Return true
```

While this will definitely find the string, the resulting running time is $O(m^n)$ which is unreasonable for almost all scenarios. We can improve this slightly by using the idea of "bad character shift" which would increase i to the position where it first found a bad character.

```
      for j = 1 to n
            if S[i+j] != F[i+j]
                  i=j  <<<<
                  exit
            Return true
```

In the real world, this may help a little, our true upper bound is still $O(m^n)$.

A final improvement on this brute force method that could improve performance in real world data is to identify uncommon characters in the pattern we are searching for, and seek those in the larger string first. This would prevent falling into the comparison loop less frequently and in general would improve performance.

```
Let S = larger string (m=length)
Let F = string to find (n= length)
Let C = most uncommon character in the search string (e.g. q,x,z)
```

```
Let Cpos = the position in the search string
for i = 1 to m
      if S[i]=C
            for j = 1 to n
                  if S[i+j-Cpos] != F[i+j-Cpos]  exit
                  Return true
```

To better

One way to improve string searching is to ignore the string itself and change the search pattern to a hash. This was first introduced by Karp-Rabin. The idea is to find if a current search buffer looks enough like the search string. In order to truly increase processing time, the hash needs to be efficient and designed to discriminate well for strings.

```
Define Hash(f) as an efficient and string discriminate hash function
Let f_hash = the initial hash of the string to find
For I = 1 to S.Length
      If Hash(S[i]..S[I +f.length]) == f_hash
            Check Each Character
```

While it still does a lot of work, the efficiency of the algorithm has improved. The hash searching phase of the algorithm is done in O(mn) whereas the expected number of text comparisons is O(m+n).

When I initially started writing this paper, I thought I would expand upon some applying concepts we learned in class to them. That was until I ran into Boyer-Moore.

To best: Finding strings in linear time

The Boyer-Moore introduced an algorithm to search for strings in linear time. The Algorithm preprocesses the pattern to search for, and then shifts accordingly based upon that pattern and patterns discovered in the search strings. It essentially searches from right to left which allows for two different potential shifts in the comparison to occur : GoodCharacter and BadCharacter. Let's walk through an example:

```
Let S = String to Search: "An example string to search sample for you"
Let F = String to Find: "sample"
```

The first thing the algorithm does is create a "shift table" for the word sample to be able to identify, if a match is found, but not aligned, how far should the search buffer be shifted. The table built for this example would be:

```
sample
65431
```

The algorithm aligns the search buffer at the beginning of the search string and works from right to left

```
sample
An example string to search sample for you
```

It compares the last characters first and in the case of finding a mismatch decides how far to shift the buffer. In this case we have a mismatch: a != e; however, a is in our GoodCharacterShiftTable with an index of 5, so we do a GoodCharacterShift of 5 and realign the comparison buffers as follows:

```
    sample
An example string to search sample for you
```

Here we again compare the last letters and we find a match. Next we work forward, character by character until we find a mismatch or a complete match. And in this case, the comparison works backwards. It is at this time where s != x and x is not in our GoodCharacterShiftTable. Thus, it is bad character and we use the BadCharacterShiftTable to find out how far to shift the buffer. Since we are working backward, we will shift the buffer all of the way to the next available window:

```
          sample
An example string to search sample for you
```

In this case our first check of the last letter finds a value that is not in the GoodCharacterShiftTable and thus we can shift it out a full buffer window again.

```
                sample
An example string to search sample for you
```

To finish this to the end, we find mismatch but with a good character and shift the word accordingly

```
                      sample
An example string to search sample for you
```

Followed by a mismatch which requires a bad character shift:

```
                           sample
An example string to search sample for you
```

Followed finally by a mismatch and a good character shift which finds our match:

```
                           sample
An example string to search sample for you
```

That which is extremely amazing about this innovative approach with real world data is that it finds its results in linear time (m/n) and gets better of course as the size of the search pattern increases.[1]

[1] I was so impressed with this that I created a C# implementation (in the appendix)

Current works

Depending upon the scenario, you may want to shift processing time. For example, if it was absolutely critical to find the string in the most efficient time, but you had knowledge of the data upfront and that data was fairly static, you might build a cache of the data and index it by words.

```
FindIndexOf(searchstring)
      return CacheTable(searchString)
```

This is not unlike what is done in databases; databases will maintain defined and learned indexes of data in row based queries; a database can build indexes on all or portions of strings (usually the former as the strings are often coded).

This use of indexes has been applied in full-text search engines as well, only better devised for unstructured or semi-structured data. When Yahoo was first introduced, it was a category based search engine. It used its own tables (indexes) to produce search results. However, WebCrawler and Lycos among others were known to crawl the web to supposedly provide full text search across the entire web. And though they may store a lot of information, they usually only work from frequently used caches and apply search algorithms such as nearest neighbor and reverse indexing to create efficiencies. Search algorithms eventually became intellectual property as the competition for searching heated.

But as noted even in early work by Chakrabarti, Dom, Gibson, Kleinberg, Kumar, Raghavan, Rajagopalan and Tomkins (http://www.cs.cornell.edu/home/kleinber/sciam99.html) web page authors began to adapt their content to rise to the top of different search engines.
In this paper, the authors suggest using hyperlinks as the main source of searching, filtering and then separating them into special categories of “authorities” and “hubs”. This seriously decreases the amount of data that needs to be searched and can still provide an reasonably accurate search result in the vast majority of the cases.

What’s Next

Clearly, the world of searching for a string in a buffer while interesting academically is not where most of the future research is being done. Most is being done on the web, which is the largest text “buffer” around. While the work by the Clever team still stands out as a viable search algorithm, Google’s page ranking, historical data, further filtering based upon relevance and many other algorithms are going to stretch string searching into far more than buffer analysis.

Appendix

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class search
    {
```

```
void BoyerMoore(char[] x, int m, char[] y, int n)
{
    int i, j;
    int[] bcst; int[] gcst;
    int tmpSizeGood, tmpSizeBad;
    bcst = BadCharacterShiftTable(x, m);
    gcst = GoodCharacterShiftTable(x, m);
    j = 0;

    while (j <= n - m)
    {
        for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i) ;
        if (i < 0)
        {
            Console.Writeline(j.ToString());
            j += gcst[0];
        }
        else
        {
            tmpSizeBad = bcst[y[i + j]] - m + 1 + i;
            tmpSizeGood = gcst[i];
            if (tmpSizeGood > tmpSizeBad)
                j += tmpSizeGood;
            else
                j += tmpSizeBad;
        }
    }

}
int[] BadCharacterShiftTable(char[] x, int m)
{
    int i;
    int[] bcst = new int[x.Length];
    for (i = 0; i < x.Length; ++i)
        bcst[i] = m;
    for (i = 0; i < m - 1; ++i)
        bcst[x[i]] = m - i - 1;
    return bcst;
}




int[] GoodCharacterShiftTable(char[] x, int m)
{

    int i, j;
    int[] suff = new int[x.Length];
    int[] gcst = new int[x.Length];
    suff = Suffixes(x, m);

    for (i = 0; i < m; ++i)
        gcst[i] = m;
    j = 0;
    for (i = m - 1; i >= -1; --i)
        if (i == -1 || suff[i] == i + 1)
```

```
                for (; j < m - 1 - i; ++j)
                    if (gcst[j] == m)
                        gcst[j] = m - 1 - i;
            for (i = 0; i <= m - 2; ++i)
                gcst[m - 1 - suff[i]] = m - 1 - i;

            return gcst;
        }

        int[] Suffixes(char[] x, int m)
        {
            int f, g, i;
            f = 0;
            int[] suff = new int[m];
            suff[m - 1] = m;
            g = m - 1;
            for (i = m - 2; i >= 0; i--)
            {
                if (i > g && suff[i + m - 1 - f] < i - g)
                    suff[i] = suff[i + m - 1 - f];
                else
                {
                    if (i < g)
                        g = i;
                    f = i;
                    while (g >= 0 && x[g] == x[g + m - 1 - f])
                        g--;
                    suff[i] = f - g;
                }

            }
            return suff;
        }


    }
}
```