Knuth-Morris-Pratt Algorithm

Meeshaan Shah
CPSC 406
May 6th, 2015
TR 5:30 – 6:45pm

ABSTRACT

The purpose of this paper is to look at the implementation, empirical and mathematical analysis of the Knuth-Morris-Pratt Algorithm. Using c++ as the programming language of choice we analyze the efficiency of the KMP algorithm with a significantly large target string to try to see how efficient we can return a pattern match within said target string. The implementation was tested on multiple machines with varying processing speeds and memory sizes and differing text files of varying sizes.

1. INTRODUCTION

Like most string matching algorithms the KMP algorithm works best in matching/finding occurrences of a pattern string in a text file or similar target string. These algorithms are best utilized in equivalent database and text processing applications. Benefits that the KMP algorithm has over naïve string matching implementation is it reduces the total number of comparisons of the pattern against the target string. And since the algorithm was the first linear time string matching algorithm general efficiency is improved over older naïve implementations. Another use of the KMP algorithm is used in the string prefix-matching problem. This problem refers to the matching of prefixes of the pattern and text, and also for checking for the longest prefix of some given text or sequence.

2. HISTORY

Donald Knuth, James Morris, and Vaughan Pratt developed the KMP algorithm in 1974, although Morris developed it independently of Knuth and Pratt. The three then published the algorithm together in 1977. Knuth discovered linear time nature of the algorithm, Pratt made the run time independent of the alphabet, and Morris discovered the algorithm while attempting to implement a text editor. Using analysis of the naïve algorithm they were able to deduce the first linear time string matching algorithm. By doing this they were able to avoid the comparisons of already compared data to more efficiently traverse the target string.

3. ALGORITHMIC OUTLINE

Essentially the KMP problem can be broken into two parts, as one is needed to help the other to search. The first part is the computation of the partial match table, or failure function, and the other being the KMP algorithm itself.

First let's talk about he partial match table. Essentially the partial match function gains knowledge about how the pattern matches against itself. In this function the only information that is needed is information about the pattern you wish to search for. We can take advantage of this information to avoid comparing useless information in the KMP algorithm function. By doing this we know exactly at which position a new match could begin prior to the current position. Essentially we are pre-computing the pattern itself and compiling a list of all possible positions to fallback to that skips the maximum number of unmatched characters whilst not missing any potential matches as well. The general idea for how to compute the partial match table is by looking at the length of the longest proper prefix in the sub-pattern that matches a proper suffix in the same sub-pattern. So essentially if we are at PMT[i], the number that fills this position is the length of the longest prefix that matches the longest suffix in the pattern up to pattern[i]. The run time for this commutation is O(m) where m is the length of the pattern as we need to loop through the pattern string at most once to build the partial match table.

With the partial match table computed we could then use it to help search the target string. The comparison of the characters in the pattern and target string are similar to the naïve implementation however when we receive a mismatch we can then use the partial match table to skip ahead in the target string to avoid comparisons of unnecessary characters. The runtime of the KMP algorithm is O(n) where n is the length of the target as we have to loop through the target only once thanks to the use of the partial match table.

By combining the runtime of commuting the partial match table and the KMP algorithm itself we get a runtime of O(m+n).

4. DATA

The data that was employed to test the KMP algorithm was two text files of differing sizes. One roughly 130,000 lines long and the other being around 5 million lines long. The smaller file consisted of lines of strings that represented half the stories of the Sherlock Holmes books. The large file consisted of the collection of Amazon food reviews as of 2004 received from the Stanford snap data set website. In addition, the algorithm was tested on two machines, one being a laptop with 4 gigabytes of ram with a 2.4 GHz processor and the other a desktop with 12 gigabytes of ram and a 3.6 GHz processor. The process for which the file was used to test the algorithm was concatenating each line read from file into a string target variable.

5. RESULTS

Running the algorithm on these text files proved to provide positive results. The clock test to find a match in both files seemed relatively quick.

On the smaller file, matches were generally found in less than 0.2 seconds on the laptop and less than 0.1 seconds on the desktop. On the worst case for the smaller file, which generally meant no match was found, the clock test was roughly greater than 0.2 seconds on the laptop and 0.1 seconds on the desktop.

On the larger file clock tests were generally a little longer depending on where the match occurred in the string. On average the worst case, meaning the match was nearer to the end or no match was found, is around 15 seconds on the laptop and 11 seconds on the desktop.

One thing to note is for the implementation for this algorithm required reading of the date from the text file, which took a significant amount of time especially with the larger file. Majority of the computation time was used in the processing of the text file. However, the clock test was only applied to the algorithm itself. Meaning the clock started when the file was read and user had entered a pattern string.

6. CONCLUSION

Altogether the implementation and analysis of the KMP algorithm was a success. This exercise allowed for better understanding of the algorithm which in turn led to a greater appreciation for what Knuth, Morris, and Pratt were able to implement. How the three were able to come up with this algorithm that is very intuitive with minimal code required is impressive.

Perhaps for further research we can look at file much larger than already provided and perhaps with more repetition in the string representation to see how the algorithm would react in more substantial conditions.

7. REFERENCES

[1] Wikipedia contributors. "Knuth–Morris–Pratt algorithm." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 27 Apr. 2015. Web. 6 May. 2015.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

8. APPENDIX

```
#include <iostream>
```

```
#include <string>
#include <fstream>
#include <time.h>
using namespace std;
//creates partial match table base on pattern
void partialMatchTable(string p, int pi[])
{
    int m = p.length();
    int q;
    pi[0] = -1;
    for (int i = 1; i < m; ++i)
    {
        q = pi[i-1];
        while (q >= 0)
        {
        if (p[q] == p[i-1])
                break;
        else
                q = pi[q];
        }
        pi[i] = q + 1;
    }
    //print partial match table
    cout << "PMT: ";
    for (int i = 0; i < m; ++i)
    {
        cout << pi[i] <<  " ";
    }
    cout << endl;
}
//runs KMP algorithm pattern p on target t, returns a boolean if or not in target
bool KMP(string p, string t)
{
    int n = t.length();
    int m = p.length();
    int pi[m];
    partialMatchTable(p, pi);
    int i = 0;
    int q = 0;
    while (i < n)
    {
    if (q == -1)
    {
        ++i;
        q = 0;
    }
    else if (t[i] == p[q])
    {
        ++i;
        ++q;
        if (q == m)
                return true;
    }
    else
        q = pi[q];
    }
    return false;
}
int main(int argc, char *argv[])
{
//shows how program needs to run if command line entry was wrong
if (argc != 2)
{
cout << "usage: "<< argv[0] << "</../nameOfText.txt>" << endl;
}
//actual program
else
{
string fileName = argv[1];
ifstream file(fileName);
string line;
char answer;
int lineCount = 0;
int start;
int end;
string target = "";
string pattern;
//check if did not open
if (!file.is_open())
{
    cout << "Could not open file" << endl;
}
else
{
//read target from file
while (!file.eof())
{
    getline(file, line);
    target += line;
    ++lineCount;
    cout << lineCount << endl;
    }
    file.close();
    //kmp continue to ask until user says no
    cout << "Run kmp on target? (y/n) "; cin >>
answer;
    cin.ignore();
    while (answer == 'y' || answer == 'Y')
    {
    cout << "Enter pattern:" << endl;
    getline(cin, pattern);
    //start clock
    start = clock();
    //KMP
    if (KMP(pattern, target))
        cout << "Match" << endl;
    else
        cout << "No match" << endl;
    //stop clock
    end = clock();
    //prints clock test
    cout <<  "Clock Test: " << ((float)end -
start)/CLOCKS_PER_SEC << " seconds" << endl;
    cout << "--------------------------------" << endl;
    cout << "Run again? (y/n) "; cin >> answer;
    cin.ignore();
    }
}
}
return 0;
}
```