Assignment 3 - Report

Devid Duma
dduma19@epoka.edu.al
Epoka University - Theory of Computation (CEN 350)

Abstract

Knuth-Morris-Pratt is an important online string searching algorithm published jointly by Donald E. Knuth, James H. Morris Jr. and Vaughan R. Pratt in 1977.[1]. In this assignment I am going to present a use case of Knuth-Morris-Pratt at the heart of a simple, keyword based recommender system.

1 Introduction

Repository The source code of the project is written in **C++17**. I did not upload the repository to repl.it, since this website did not recognize the header file $< filesystem >$, which is used in the source code. That being said, I chose to rather upload the source code via a zip file at Google Classroom.

Procedure As a first step I downloaded 11 different websites of companies specializing in the area of Internet of Things. I experimented with the topic of Internet of Things, but the recommender system can be used with any other topic.

I specified 16 keywords that the algorithm should focus on when ranking the pages, based on what I was most interested in. You can use any number of keywords that you like. The chosen keywords are: *"device", "internet", "thing", "node", "network", "protocol", "data", "comput", "process", "distributed", "smart", "automation", "system", "robot", "autonomous", "intelligent"*.

2 Algorithm

2.1 Knuth-Morris-Pratt

Implementation I implemented the algorithm as was described in the class CEN350, with some tweaks. Instead of outputting or saving the indices where the match happens, it simply counts the matches, to save space. It also supports case insensitive comparison of characters. Let's have a look at the code:

C++17 code

```
class KMP {
private:
    int* kmpNext;
    string pattern;

    void preKmp() {
        int i = 0;
        int j = kmpNext[0] = -1;

        while (i < pattern.length()) {
            while (j > -1 && pattern[i] != pattern[j])
                j = kmpNext[j];
            i++;
            j++;
            if (pattern[i] == pattern[j])
                kmpNext[i] = kmpNext[j];
            else
                kmpNext[i] = j;
        }
    }

public:
    explicit KMP(const string& pattern) {
        this->pattern = pattern;
        kmpNext = new int[pattern.length()];

        // Preprocessing
        preKmp();
    };

    long execute(string searchIn, bool caseSensitive = false){
        int i, j;

        // Searching
        i = j = 0;
        // remember the indices that match
        long count = 0;

        while (j < searchIn.length()) {
            while (i > -1 && !compare(pattern[i], searchIn[j],
        caseSensitive))
                i = kmpNext[i];
            i++;
            j++;
            if (i >= pattern.length()) {
                count++;
                i = kmpNext[i];
            }
        }

        return count;
    }

    static bool compare(char x, char y, bool caseSensitive =
        false) {
        if( !caseSensitive && ((x >= 'a' && x <= 'z') || (x
        >= 'A' && x <= 'Z')) &&
                ((y >= 'a' && y <= 'z') || (y >= 'A' && y
        <= 'Z'))) {
            return abs(x - y) == 0 || abs(x - y) == abs('A'
        - 'a');
        } else {
            return x == y;
        }
    }
};
```

Online preprocessing The most important aspect of the Knuth-Morris-Pratt algorithm is that it is an online algorithm. This means that it first analyzes (preprocesses) a pattern that will be searched for and after that can accept any string where we will search for the pattern. The part of the code where the online preprocessing happens can be seen in the function *preKmp()*.

Case insensitive comparison The algorithm also supports case insensitive comparison, if that is desired. The character comparison is done by the method *compare()*. By default, case insensitive comparison is enabled, but it can be disabled by simply setting the parameter *caseSensitive* to $true$.

2.2 Complexity

Let **P** be the pattern with length **m**, **S** be the string where we are searching in with length **n**.

The preprocessing phase is always on the order of **m** comparisons, so the time complexity is **O(m)**. KMP spends a little time precomputing a table, and then it uses that table to do an efficient search of pattern **P** in the string **S**.That being said, the searching phase is always on the order of at most **2n** comparisons, so the time complexity is **O(n)** for the searching phase.

That gives us a total time complexity of **O(n + m)**. Taking into consideration that the table produced by the preprocessing phase also contains **m** entries, we can say that the total space complexity of the algorithm is **O(m)**.

2.3 Web Score algorithm

Implementation Scoring how good a webpage fits to our interests is very important to this project. Let's first look at the implementation:

C++17 code

```
class IO {
public:
    static string readFile(const string& path) {
        ifstream fin(path);

        string content((istreambuf_iterator<char>(fin)),
                istreambuf_iterator<char>());

        return content;
    }
};

class ScoreWeb {
private:
    vector<KMP> kmpVector;
    double maxScore = 0;
    vector<tuple<string, double>> scores;

public:

    explicit ScoreWeb(const vector<string>& keywords) {
        for(const string& keyword: keywords) {
            kmpVector.emplace_back(KMP(keyword));
        }
    }

    double scoreWebsite(const string& file_path) {
        string content = IO::readFile(file_path);

        double score = 0;
        for(KMP kmp : kmpVector) {
            long match_count = kmp.execute(content);

            // most important part: the score calculation
            score += log(double(match_count + 0.75));
        }

        if(score > maxScore)
            maxScore = score;

        if(score < 0)
            score = 0;

        scores.emplace_back(tuple(file_path, score));

        // sorts the scores from highest to lowest
        sort(scores.begin(), scores.end(), [](const tuple<string, double>& element1, const tuple<string, double>& element2){
            return get<1>(element1) > get<1>(element2);
        });

        return score;
    }

    vector<tuple<string, double>> normalizedScores() {
        vector<tuple<string, double>> normalizedScores;

        for(tuple pair: this->scores) {
            double score = get<1>(pair);

            normalizedScores.emplace_back(tuple<string, double>(get<0>(pair), score/this->maxScore * 100.0));
        }

        return normalizedScores;
    }
};
```

IO streams Reading the content of webpages is provided in the IO class. Key to notice here is that after it is declared, the *content* string is not assigned a value directly from *fin.read()* function, since this would occupy too much of the heap space. The content variable is rather fed by an *istreambuffer* on demand, which in turn is pointing to the *ifstream* instance.

Scoring The score calculation is provided in the *scoreWebsite()* function. It uses the logarithmic function. This is done to penalize when no match for a keyword was found in the website and to prevent exploding scores by spam. Mathematically, the score S of a website can be written as follows: $S = \sum_i^n log(m_i + 0.75)$, where n is the number of keywords and m_i is the number of matches for a given keyword in that website.

Since $m_i >= 0$, then $log(m_i + 0.75) >= log(0.75) \approx -0.1249$. This is done to penalize the score of a website if no match for that keyword was found in the website.

The scoring function also sets the overall score to 0 if the overall score turns out negative. It also sorts the scores present at the object of type *ScoreWeb* from highest to lowest before returning the score of current website.

Normaizing the scores Scoring is one thing, but normalizing the scores to a user friendly output is also important. The function *normalizedScores()* serves that purpose. It simply normalizes the scores to the range $[0, 100]$ and returns the vector.

3 Experiment and Results

Experiment I experimented by feeding as an input 11 different webpages of companies in the Internet of Things arena to the recommender system. I also set the keywords to be the following, as I mentioned in the "Introduction" section:

"device", "internet", "thing", "node", "network", "protocol", "data", "comput", "process", "distributed", "smart",

"automation", "system", "robot", "autonomous", "intelligent".

Results The results seem very good. The top 3 companies shown as best matches were Microsoft Azure with 100%, Amazon Web Services with 89% and IOTA with 81% match, which was expected. You can see the full results below:

File	Score
azure.htm	100%
aws.htm	89.35%
iota.htm	81.3657%
eta.htm	71.632%
hologram.htm	56.7635%
influx.htm	53.0488%
spanio.htm	36.0636%
inspiritiot.htm	32.5196%
pfp.htm	31.2282%
swim.htm	28.1689%
karambasecurity.htm	19.6697%

4 Conclusion

The idea for a simple recommender system using Knuth-Morris-Pratt string searching under the hood is a good idea, especially for fast prototyping or whenever simple tasks like keyword based recommendations have to be solved.

Recommender systems nowadays have become too advanced, with the development of artificial intelligence areas like deep learning. At the same time though, the time and resources needed to build a good recommender system, like the ones Google, Amazon or other moguls of the industry use has increased dramatically. Nevertheless, not every business needs such complicated recommender systems.

In this assignment we showed that a simple, keyword based recommender system can be very easy to build and can produce good results while also being relatively fast and very efficient in resources.

References

[1] Donald E. Knuth, James H. Morris, Jr., Vaughan R. Pratt, *Fast Pattern Matching in Strings*. 10.1137/0206024.

[2] Donald E. Knuth, *The Dangers of Computer-Science Theory*. 10.1016/S0049-237X(09)70357-X.

[3] Mireille Regnier, *Knuth-Morris-Pratt algorithm: An analysis*. 10.1007/3-540-51486-4_90.

[4] Robbi Rahim, Iskandar Zulkarnain, Hendra Jaya, *A review: search visualization with Knuth Morris Pratt algorithm*. 10.1088/1757-899X.