UDC 004.02
IRSTI 20.53.15

ALGORITHM FOR OCR TEXT SEARCHING

Mirzakhmet Syzdykov[1], Santanu Kumar Patro[2]

[1]al-Farabi Kazakh National University, Almaty, Kazakhstan
[2]Berhampur University, Odisha, India
[1]mspmail598@gmail.com, [2]ksantanupatro@gmail.com
[1]ORCID ID: https://orcid.org/0000-0002-8086-775X
[2]ORCID ID: https://orcid.org/0000-0002-7917-4841

Abstract. The optical character recognition (OCR) is the modern global trend in the digital world. However, due to the errors in character recognition device, this could be almost impossible to obtain user friendly and readable version of the obtained data. For this reason, in this article we present the algorithm for effective text searching in extracted OCR thread. The novel technique is simply used based upon the sliding algorithm with error corrections, which could be also applied to the dictionary data: here this technique is based on the usage of window to search the matching and the Tesseract OCR is used as an OCR engine. In this article we also show that this algorithm is applicable only when the change character error is present. The overview of the past work is also given with respect to the classically known algorithms like Knuth-Morris-Pratt (KMP) or Boyer-Moore (BM). We also show that our algorithm works efficiently in linear polynomial time.

Keywords: Optical Character Recognition, Linear Algorithm, Mathematical Probability.

Introduction

The most known algorithms to the present time for pattern string matching, i.e. the problem of finding the occurrences of pattern in the matching string, like Knuth-Morriss-Pratt [1] and Boyer-Moore [2] were adapted for the digital patterns, when pattern can be taken from the scanned image [3, 4]. We present the linear error-prune algorithm for OCR text matching.

The matching cost can be computed using variety of techniques, for example, Hamming distance [5]. For experimental purpose as it was stated in the abstract Tesseract OCR [6] engine is used. However, the Hamming distance is good for low-cost computing, while the approach described in this paper can be used for large data. To handle the large data thread, we have used the windowing technique, so by this method the necessary data for the pre-defined time interval can be obtained as:

Window [1….t] = Data * Time[1..t] (1).

From the obtained data the penalty can be computed. This penalty is the measure of equivalence between searching text and scrambled data, possibly by OCR engine. So that the following holds true:

Penalty [Data [1…t]] == A ~ B (2),

where:

‘A’ is the searching text and
‘B’ is scrambled data.

Now we would like to present a short introduction to the naive approach and, in overall, saying, this paper is organized in such a way that after presenting a short introduction, we give the Java code for this method. After which we present the scalable window approach, with its codes. In the very next section, we present its application part, i.e the experimental results.

Naive Approach

The naive approach is nothing but to calculate the distance between two words. This method has a good application when number of symbols (or the weight of word) in text to be searched is

small. Obviously by this method we get the magnitude equal to the weight of text. To compute this distance the bi-linear search is executed at every position in scrambled text. The cost here is, thus, a cumulative sum of penalties at every step when symbols coincide. This can be written as:

Cost (A, B) = SUM {1.0 / Distance[X[1], X[2]]} (3),

where,

A, B are words and

X[1], X[2] are positions of two consecutive matchings.

It's to be noted that the algorithm experimentally produces good results for low-cost pattern (word A). Even the java code can be used for this purpose: so now we would like to present its coding part in Java.

The java code for naive approach:

```
public static SearchResult query_exact (String text, String query_text, String text_0, String query_text_0) {
        float max_scal_match = 0.0f;
        int best_match_start = -1;
        int best_match_end = -1;
        int best_page_line = -1;
        int best_page_offset = -1;
        int best_page = -1;
        int page_line = 0;
        int page_offset = 0;
        int page = 0;

        for (int i = 0; i < text.length(); ++i) {
                float scal_match = 0.0f;
                int last_match_pos = -1;
                for (int j = 0, k = i; j < query_text.length() && k < text.length(); ++j, ++k) {
                        if (text.charAt(k) == '\n') {
                                ++page_line;
                                page_offset = 0;
                        } else if (((int)text.charAt(k)) == 12) {
                                ++page;
                                page_line = page_offset = 0;
                        }

                        if (text.charAt(k) == query_text.charAt(j)) {
                                int gap = 1;
                                if (last_match_pos != -1)
                                        gap = j - last_match_pos;

                                last_match_pos = j;

                                scal_match += 1.0f / ((float) gap);

                                if (scal_match > max_scal_match) {
                                        max_scal_match = scal_match;
                                        best_match_start = i;
                                        best_match_end = k;
                                        best_page_line = page_line;
                                        best_page_offset = page_offset;
```

```
                                        best_page = page;
                                    }
                                }

                                ++page_offset;
                            }
                        }

                        float ratio = max_scal_match / ((float) query_text.length());

                        if (ratio >= GOLDEN_RATIO) {
                            SearchResult result = new SearchResult ();
                            result.ratio = ratio;
                            result.offset = best_match_start;
                            result.length = best_match_end - best_match_start + 1;
                            result.text = text_0;
                            result.query_text = query_text_0;
                            result.page = best_page;
                            result.page_line = best_page_line;
                            result.page_offset = best_page_offset;
                            return result;
                        }

                        return null;
                    }
```

After successful presentation of the Java code, we would like to present the scalable window approach and its code.

Scalable window approach

The scalable window approach opposite to the naive method uses flowing window event when the mismatching occurs: in other words, it slowly checks for occurrences of match in the appearing window, thus the possible result of excess symbols in scrambled text is reduced. The sub-algorithm decides what symbol will be gap's left border, while the right border lays on the mismatched position [Y[1], Y[2]] (in pattern and scrambled text).

Example 1. Sample of scrambled text retrieved by Tesseract OCR

```
"дн     щ\n"
"У Х гасшрда Улш Моравия мемлекегйНЕ\n"
"цьшган мемлекег\n"
"мемлекетй\n"
"кен тараган славян таипалар в\n"
"Ь1с жерйнде Киев Русй аггьх\n"
```

This is to be better illustrated by the java code where the simple windowing algorithm is realized (code is below).

Java code for Scalable window approach:

```
public static SearchResult query (String text, String query_text, int cur_page) {
        if (text == null || query_text == null) return null;

        String text_0 = text;
```

```
		String query_text_0 = query_text;

		text = convert (text);
		query_text = removeSpaces (convert (query_text));

		if (query_text.length() <= MIN_LETTERS) return query_exact (text, query_text, text_0, query_text_0, cur_page);
		if (true) return query_large (text, query_text, text_0, query_text_0, cur_page);

	int pos_text = 0, pos_query = 0;
	int matched = 0;
	int last_match_pos = -1;
	int last_match_pos2 = -1;
	int max_matched = 0;
	int cons_match = 0;
	float scal_match = 0.0f;
	float max_scal_match = 0.0f;

	int start_match_pos = -1;
		int end_match_pos = -1;

		int best_pos_start = -1;
		int best_pos_end = -1;

		int last_pos_query = -1;

		int best_page = -1;
		int best_page_line = -1;
		int best_page_offset = -1;
		int page = 0;
		int page_line = 0;
		int page_offset = 0;


		int match_gap = MATCH_GAP;

		if (query_text.length() >= match_gap)
			match_gap = query_text.length() + 2;

		float max_ratio = 0;

	for (; pos_text < text.length() && pos_query < query_text.length(); ) {


	 char c1 = text.charAt(pos_text);
	   char c2 = query_text.charAt(pos_query);

	   if ((scal_match > max_scal_match) && matched > 1)
	   {
		max_scal_match = scal_match;
		best_pos_start = start_match_pos;
		best_pos_end = end_match_pos;
		best_page = page;
```

```
            best_page_line = page_line;
            best_page_offset = page_offset;
            max_matched = matched;
            max_ratio = (scal_match/((float)matched));
}

  if (c1 == c2) {
     int max_gap = 1;
     int a_gap = last_match_pos == -1 ? 0 : (pos_query - last_match_pos);
     int b_gap = last_match_pos2 == -1 ? 0 : (pos_text - last_match_pos2);
     int c_gap = a_gap > b_gap ? a_gap : b_gap;

     int p = 0;

     switch (cons_match) {
        case 1: max_gap = 2; break;
        case 2: max_gap = 3; break;
        case 3: max_gap = 4; break;
        default: if (cons_match > 3) max_gap = 5;
     }


     if (c_gap <= 1)
        ++cons_match;
     else
        cons_match = 0;

     if (c_gap == 0) c_gap = 1;
     if (c1 == c2) p = 1;

     if ((last_match_pos == -1 || a_gap <= max_gap)
           && (last_match_pos2 == -1 || b_gap <= max_gap)) {
        last_match_pos = pos_query;
        last_match_pos2 = pos_text;
        scal_match += ((float) p) / ((float)c_gap);
        matched += p;
        end_match_pos = pos_text;


     } else {

        if ((scal_match > max_scal_match) && matched > 1)
         {
                  max_scal_match = scal_match;
                  best_pos_start = start_match_pos;
                  best_pos_end = end_match_pos;
                  best_page = page;
                  best_page_line = page_line;
                  best_page_offset = page_offset;
                  max_matched = matched;
                  max_ratio = (scal_match/((float)matched));
         }
```

```
                matched = 1;
                cons_match = 1;
                scal_match = 1.0f;
                last_match_pos = -1;
                last_match_pos2 = -1;
                pos_text = -1;

                start_match_pos = -1;
                end_match_pos = -1;

                page = 0;
                page_line = 0;
                page_offset = 0;
                c1 = '\0';
            }

            if (start_match_pos == -1) {
                start_match_pos = pos_text;
                end_match_pos = pos_text;
            }

            if (c1 == '\n') {
                        page_line++;
                        page_offset = 0;
                } else if (((int) c1) == 12) {
                        ++page;
                        page_line = page_offset = 0;
                } else
                        ++page_offset;

            ++pos_text;
            ++pos_query;


        } else {
            boolean matched1 = false;
            for (int i = pos_query + 1; i < query_text.length() && (i < (pos_query +
match_gap)); ++i)
                if (c1 == query_text.charAt(i)) {
                    pos_query = i;
                    matched1 = true;
                    break;
                }
            if (!matched1) {
                int v_page = page;
                int v_page_line = page_line;
                int v_page_offset = page_offset;

                for (int i = pos_text + 1; i < text.length() && (i < (pos_text + match_gap)); ++i)
                {
                    if (c2 == text.charAt(i)) {
                            page = v_page;
```

```
                    page_line = v_page_line;
                    page_offset = v_page_offset;

                pos_text = i;
                matched1 = true;
                break;
              }

              if (text.charAt(i) == '\n') {
                    v_page_line++;
                    v_page_offset = 0;
             } else if (((int) text.charAt(i)) == 12) {
                    ++v_page;
                    v_page_line = v_page_offset = 0;
             } else
                    ++v_page_offset;
            }

            if (!matched1) {
               ++pos_query;

            }
        }


    }
}

if (pos_text == text.length()) {
   last_pos_query = pos_query;
} else {
   last_pos_query = -1;
}

if (text.length() == 0) return null;

if ((scal_match > max_scal_match) && matched > 1)
{
 max_scal_match = scal_match;
           best_pos_start = start_match_pos;
           best_pos_end = end_match_pos;
           best_page = page;
           best_page_line = page_line;
           best_page_offset = page_offset;
           max_matched = matched;
           max_ratio = scal_match/((float)matched);
 }

float retf = 0;
if (max_matched > 0)
 retf = Math.max(retf, (max_scal_match / ((float) max_matched)));
else
 retf = Math.max(retf, (max_scal_match / ((float)query_text.length())));
```

}

Obviously, the algorithm uses input parameters for windowing algorithm. This set can be extended according to the, for instance, textual model of the text. In this example the textual model is a scrambled text with excess symbols produced by the engine which are incorrect and are to be removed or replaced. Of course, the final reduction leads to the exact text extraction.

Now we are going for the application part, i.e the experimentation part. We have presented an Android Program, which was developed by authors, lunched by Google Inc.

Experimentation

Actually, the method described in this article were realized in the application, published by Google Inc. The program simply searches for extracted text in file system of the mobile device (Android program). Two different views are given for reference.

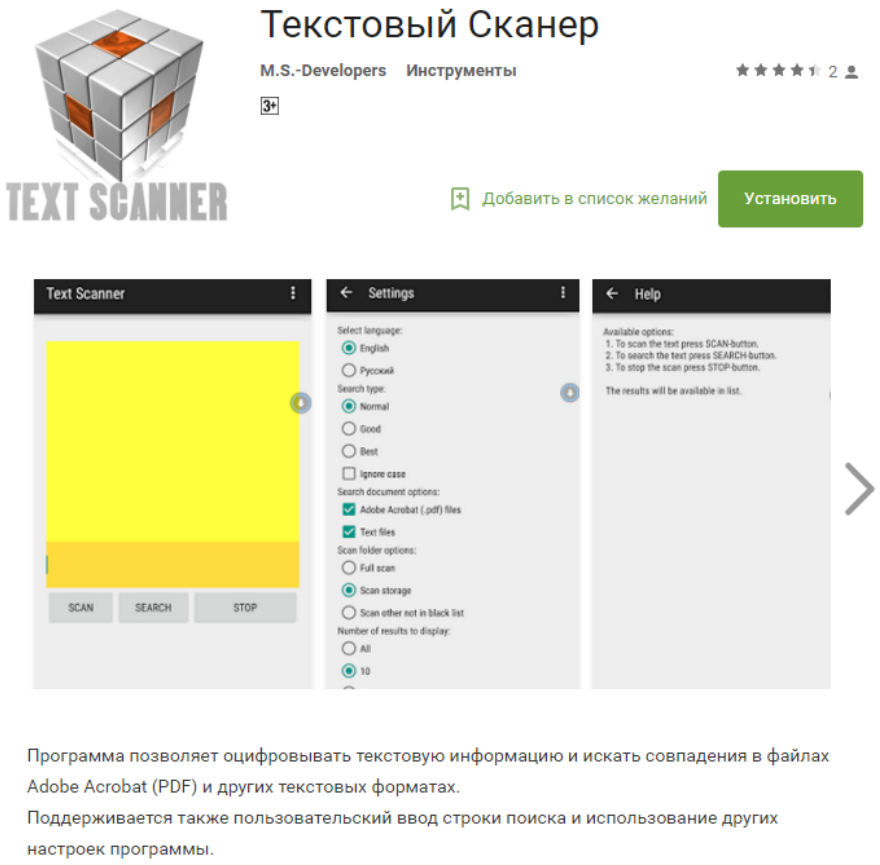

Figure 1 – The experimental program on Google Play

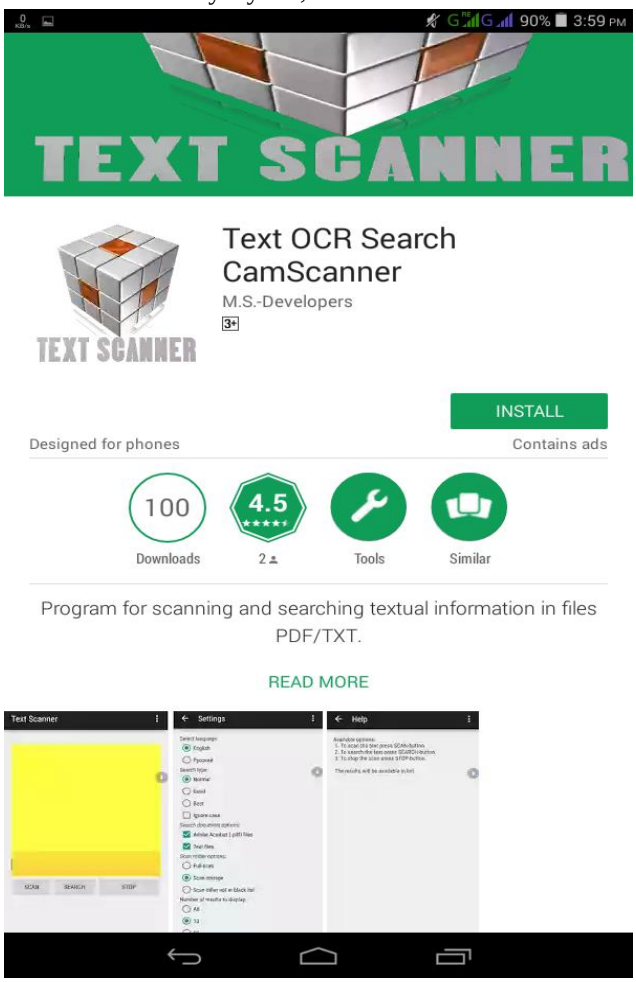

Figure 2 – A view of Android program on Google play store

Conclusion

So we have used the above algorithm for the extraction of text patterns. However the gap penalties as well as OCR engines were well studied in the past, for example, the problem of extracting the text pattern still remains open. This is mainly due to the complexity of the task when we need the fast pattern extractor on the partially extracted set of string data. So we think this paper gives a short development to this problem. And the interested researcher may put their focus on these issues.

Acknowledgements

Authors owe his special debt of gratitude towards his family members, for their keen interest and cooperation.

Funding

This project was partially supported by an educational grant of the Ministry of Education and Science of Republic Kazakhstan during 2006-2009.

References

[1] Régnier, Mireille. Knuth-Morris-Pratt algorithm: an analysis. International Symposium on Mathematical Foundations of Computer Science. Springer, Berlin, Heidelberg, 1989.

[2] Boyer, Robert S., and J. Strother Moore. A fast string searching algorithm. Communications of the ACM 20.10. 1977. 762-772.

[3] Prasetia, Yoga, Ghulam Asrofi Buntoro, and Dwiyono Ariyadi. Application of the Knuth-Morris-Pratt Algorithm on Android-based Money Recognition Applications for the Blind. Journal Teknik Informatika CIT Medicom. 2021. 13(2). 82-93.

[4] Wankhede, Poonam A., and Sudhir W. Mohod. A different image content-based retrievals using OCR techniques. IEEE Int. conf. of electronics, communication and aerospace technology (ICECA). 2017. 2.

[5] Hamming, Richard W. Error detecting and error correcting codes. The Bell system technical journal. 1950. 29(2). 147-160.

[6] Tesseract OCR, GitHub // https://github.com/tesseract-ocr (accessed Oct 11, 2021).