Improved Greedy Algorithm for Computing Approximate Median Strings

Ferenc Kruzslicz *

Abstract

The distance of a string from a set of strings is defined by the sum of distances to the strings of the given set. A string that is closest to the set is called the median of the set. To find a median string is an NP-Hard problem in general, so it is useful to develop fast heuristic algorithms that give a good approximation of the median string. These methods significally depend on the type of distance used to measure the dissimilarity between strings. The present algorithm is based on edit distance of strings, and constructing the approximate median in a letter by letter manner.

1 Introduction

If the solution of the optical character recognition (OCR) problem is considered as a "black box" process where images are mapped to character strings, then we usually use a certain kind of off-line approach. In this way the efficiency of some OCR processes could be increased in an OCR software and language independent manner. Suppose we have a set of strings as the result of several OCR processes of the same input bitmap. When the same OCR software was used to produce this set, with different paper orientation, changed resolution or simply repeated OCR processes we can eliminate the effects of noise (fingerprints on the glass etc.). While in case of different OCR software their efficiency can be compared to each other [7].

2 String distance

Finding a median string that is minimal in sum of distances form a given input set of strings, is known to be an NP-hard problem [8]. Therefore it is interesting to find fast algorithms, that give us good approximations. One of the latest algorithms can be found in [3]. It is called greedy algorithm, because it builds up the approximate median string letter by letter, by always choosing the best possible continuation. In this paper an improvement of this algorithm is described.

*Department of Business Informatics, Janus Pannonius University, Rákóczi út 80, 7622 Pécs, Hungary. Email: kruzslic@ktk.jpte.hu

Suppose that all the strings are defined over the same fixed alphabet Σ (for European countries Σ is usually a certain kind of extended ASCII). The most widely used edit distances are similar to the Levenshtein distance. The improved greedy algorithm is based on the dynamic programming approach [4], therefore it is suitable for all $d : (\Sigma^*)^2 \to R$ distances that satisfies the following properties.

$d(t,s) \geq 0$
$d(t,s) = 0 \Longleftrightarrow t = s$
$d(s,t) = d(t,s)$
$d(s,r) + d(r,t) \geq d(s,t)$
for all $r, s, t \in \Sigma^*$.

In case of $c \in \Sigma$ let $c_{ins}(c,r)$, $c_{del}(c,r)$, $c_{sub}(c,r)$ denote the cost of insertion, deletion and substitution of letter c in string r. The costs of edit operations do not depend on letter c and on the place of operations in r.

The Levenshtein distance is derived from this class of distances by choosing the following values: $c_{ins}(c,r) = c_{del}(c,r) = c_{sub}(c,r) = 1$. To establish the Levenshtein distance between two strings, the dynamic programming approach can be used with $O(nm)$ time and $O(n)$ space complexity. The general algorithm to compute the minimal edit distance, using the dynamic programming technique is given in the paper of Kruskal [5]. With the aid of this method, we get the following in the case of two strings (s and t):

```
Let D[i,0] = i and D[0,j] = j for i=0..|s| and j=0..|t|.
For i=1..|s| and j=1..|t| calculate the next elements of matrix D
   D[i,j] = min (D[i-1,j]+ c_ins, D[i,j-1]+ c_del, D[i-1,j-1]+δ([i,j]), where
          δ([i,j] = c_sub if s[i]≠t[j], and 0 otherwise.
```

It is clear that the distance is $d(s,t) = D[|s|,|t|]$.
Much space can be saved if the matrix D is computed in a row by row manner.

3 Approximate median

This dynamic programming technique is suitable for a large number of heuristics. Almost all of the "natural" heuristics can be described by the following informal scheme, where $|r|$ denotes the length of r, and λ is the empty string.

```
function ApproximateMedian(s1,s2 ... sn): string;
   preprocess(s1, s2, ... sn);
   median = λ;
   do
      c_best = arg best (weight(median, c, s1, s2, ..., sn) : c ∈ Σ);
      median = median + c_best;
   while ( it was worth to append c_best );
return( best prefix of median ).
```

Basically, an ApproximateMedian algorithm of this type builds the median string letter by letter, and in case of each letter it uses a weight decision function to select the next letter for median string to continue with. It makes judgements on the base of input strings $s_1, s_2, ..., s_n$ and the prebuilt median appended with letter c. The previous loop has to be continued, until a stopping condition holds. In the last step we can select the best prefix of median to return.

The time and space complexity of these algorithms is determined by the complexity of the preprocessing phase and the weight function. The previous scheme is general enough, because any type of algorithm can be written in this high level form. In case of greedy algorithms no preprocessing phase is allowed, and the weight function must be linear.

4 The Improved Greedy Algorithm

The earlier scheme of algorithms gives a large variety of heuristics. We have freedom to choose the weight functions, the stopping condition, and the last prefix correction.

A fairly good greedy heuristic can be obtained if we use the method in [3], i.e.

- The weight function is the sum of minimal elements in the last rows containing letter c in the dynamic programming matrix, computing $d(median + c, s_i)$. The next letter to be appended is the letter with the minimal weight.
- The main loop is stopped if the length of median reaches the length of the longest input string.
- The prefix of median is returned, that minimise the sum of distances from the input strings.

The greedy algorithm computes the whole dynamic programming matrixes, but stores only the last rows of them, and it loses a lot of information, because it uses only the minimal element of this vector. Let the algorithm improve by gaining more information from this vector.

If we sum these vectors, we get information on what would happen if we stop the algorithm immediately. The values of the summed vector show the sum of distances of median from the input strings, and the sum distances of median from the input strings without their last letter, etc. For example strings *aabb*, *ab*, *bbb* and median string *ab* will be examined:

b	2	1	1	1	2
a	1	0	1	2	3
λ	0	1	2	3	4
	λ	a	a	b	b

b	2	1	0
a	1	0	1
λ	0	1	2
	λ	a	b

b	2	1	1	2
a	1	1	2	3
λ	0	1	2	3
	λ	b	b	b

The sum of the last rows of matrixes D is defined as follows:

aabb	2	1	1	1	2
ab			2	1	0
bbb		2	1	1	2
S	2	3	4	3	4

or more precisely, let m denote the length of median string, and $k = max(|s_1|, |s_2|, ..., |s_n|)$. Moreover the last row of the ith matrix is denoted by $V_i =< D[|m|, 0], D[|m|, 1], ..., D[|m|, s_i|] >$. For convenience, we also assume that the co-ordinate $V_i[t] = 0$, whenever t is not in the $0...|s_i|$ interval. The summarised vector S is defined with the following expression

$$S[i] = \sum_{j=1}^{n} V_i[i - k + |s_i|], \text{ for } i = 0, ..., k.$$

With these notations the weight function in the greedy algorithm can be formulated in a simple way:

$$weight(median, c, s_1, s_2, ..., s_n) = \sum_{j=1}^{n} min(V_j[0],\ V_j[1],\ ...,\ V_j[|s_j|])$$

and the letter with the least weight will be appended to the median string.

Unfortunately this weight function frequently gives the same value for different letters, and in such a case the next letter is selected arbitrary. The weight function behaves better if we use the whole V vector to pick the best continuation of the median. Let us choose the letter in case of draw, that is minimal in lexicographic order of the reversed sum vectors $< S[k]; S[k-1]; ...; S[0] >$. Clearly the choice of next letter tries to minimise the expected sum of distances, furthermore the time and space complexity of the algorithm remains the same.

The improved algorithm runs in $O(k^2n|\Sigma|)$ time, and it is given in the following pseudocode.

```
function ImprovedApproximateMedian(s1, s2, ..., sn) : string;
   constants
      k = max( |s1|, |s2|, ..., |sn|);
      c_ins, c_del, c_sub;     /* Cost of edit operations */
   variables
      Vi : array [0..|si|] of integer; /* for i=1..n */
      Dist, S, S_best, tmp : array [0..k] of integer;
      c : char;
      min_best, min_sum, i, j : integer;
      median : array [1..k] of char;
   algorithm
      median = λ;              /* Initialization */
      Dist[0] = 0;
      for i=1 to n do
```

```
        for j=0 to |s_i| do V_i[j] = j; od
        Dist[0] = Dist[0] + |s_i|;
    od
    for i=1 to k do          /* Building the median letter by letter */
        S_best := [0,0,...,0];
        for j=1 to n do S_best := add_vect(S_best, V_j, k - |s_j|); od
        for each c ∈ Σ do        /* Selecting the best letter */
            min_sum := 0;
            S := [0,0,...,0];
            for j=1 to n do test_letter(c, j, FALSE); od
            min_sum := min_sum + min_best;
            S := add_vect( S, tmp, k - |s_i| )
            if weight(min_sum, S, min_best, S_best) < 0 then
                S_best := S;
                min_best := min_sum;
                median[i] := c;
                Dist[i] := S_best[k];
            fi
        od
        for j=1 to n do test_letter(median[i], j, TRUE); od
    od
    i := 0;
    for j=1 to k do
        if Dist[j] < Dist[i] then i = j; fi
    od
return median[1..i]

function test_letter(c, i, update) : integer;
    local variables
        j : integer;
    procedure                  /* Calculating the edit distance */
        min_best := +∞;
        tmp[0] := i;
        for j=1 to |s_i| do
            tmp[j] := min( V_i[j-1]+c_ins, V_{i-1}[j]+c_del, V_{i-1}[j-1]+c_sub );
            if median[j] = c then
                tmp[j] = min( tmp[j], V_{i-1}[j-1] );
            fi
            if tmp[j] > min_best then
                min_best = tmp[j];
            fi
        od
        if update then          /* Updating vectors when a */
            V_i := tmp[0..|s_i|];   /* new letter was appended. */
        fi
return min_best;

function add_vect(S,V,offset) : array [0..k] of integer;
    local variables
        i : integer
    procedure        /* Vector addition with offset */
        for i=0 to k - offset do
            S[i] := S[i] + V[i-offset];
return S[0..k];
```

```
function weight(min, S, min_best, S_best) : boolean;
   local variables
      diff, i : integer   /* Negative value is returned if the new */
   procedure             /* character is better than the old one. */
      diff := min - min_best;              /* Greedy heuristic */
      i := k;
      while ( i ≥ 0 and diff = 0 ) do      /* Lexicographic order */
         diff := S[i] - S_best[i];
         i := i-1;
       od
return diff
```

To illustrate how the algorithm works and to show the improvement, let us examine the following example:

The alphabet contains only two letters $\Sigma = \{a, b\}$, and the input strings are $s_1 = ab$, $s_2 = bab$.

a	1	$\underline{0}$	1
λ	0	1	2
	λ	a	b

a	1	$\underline{1}$	1	2
λ	0	1	2	3
	λ	b	a	b

b	1	$\underline{1}$	1
λ	0	1	2
	λ	a	b

b	1	$\underline{0}$	1	2
λ	0	1	2	3
	λ	b	a	b

It is easy to see that we are in the draw situation, since for

median = a, min_sum = 1, S = ¡1,2,1,3¿, and for

median = b, min_sum = 1, S = ¡1,1,2,3¿.

By the rule of the improved greedy algorithm letter a will be selected as the first letter of the median string.

5 Experimental Results

The improved approximate algorithm was tested on the same garbled strings as the greedy algorithm. In the test sets the string were deformed with equally probable delete, insert and substitute operations, with probability of 1/4.

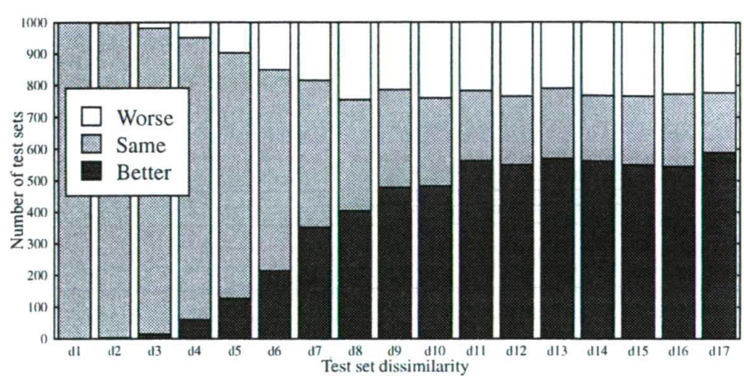

Figure 1: Efficiency of the improved versus the greedy algorithm

Original words:

hector	helsinki	iapr	ojo	pepermint	recognition	sim patica

Garbled strings:

erth	eksh	arr	jj	etepi	etorgon	icpsa
ttohr	ielkhnnki	rpp	oo	petpnrmin	tricogntionr	sic static
ectoo	hlsinhki	iaria	j	pepeprmiimtn	recggginiiong	simpsatiapat
heceor	hislnsiki	iaprr	ojjo	peermmint	receniicion	sipatpica
htoor	hselsekni	iapri	jjo	epneemine	egcoogeieion	imtpitici
ecttor	eelseskli	iappp	oj	merpeement	regtoggniitocn	pimmpitaca
hetroe	hlliinki	irap	ojo	pepitrmminnt	recortoit	siaatpta
hecetrc	hiklssinnksl	iappr	oojo	mrpermimm	ecgnittin	satica
heeter	elsinss	iai	oooj	eentin	reoritoc	pppttca
hectter	esnkki	iraar	oj	pepterintm	enoeniiion	smpactia
hector	helsinki	iapr	ojo	pepermint	recognition	simptatica

Greedy approximate medians:

hector	helsinki	iapr	ojo	pepermint	recognition	simp tatica
[26]	[39]	[19]	[12]	[39]	[50]	[45]

Improved approximate medians:

hector	helsinki	iapr	ojo	pepermint	recogniion	simptatica
[26]	[39]	[19]	[12]	[39]	[47]	[45]

When we used the new algorithm for the second test sets published in [6], there were no improvements at all.

Original words:

hector	helsinki	iapr	recognition

Garbled strings:

hetcr	cheinni	cianr	rgfkfgnition
heptor	hlelsiki	iap	recoxsniimoi
hector	hesenkc	iapi	riecoxgnifon
hevor	velskki	lapr	jeognitigqn
hetuor	ceeltsinkmi	ilp	resonigior
hscor	elnsgxnki	riapr	reoinitiggn
htuctor	gbheklsink	ialr	rciorgnitvihn
fjhecto	htosini	iar	recognin
getoqr	hxlsiky	iapd	ecotnritiin
hetofr	heklusnkk	iuar	grecpoginitko

Greedy and improved approximate medians:

hector	helsinki	iapr	recognition
[13]	[34]	[13]	[42]

The real advantage of the improved algorithm appeared when the probability of the edit operations has been increased. The Fihure 1 is obtained by the following test sets. The string *recognition* was garbled with delete, insert and substitute edit operations. For substituting and inserting only the letters *r, e, c, g, t, i, o, n, s, p, a* were used, and each of the operations and its place was uniformly distributed.

Every test consists of 10 garbled strings, and the index of a test means how many operations was performed in the garbling procedure. All column of the diagrams represents the results of 1000 tests of the same type. The greedy and the improved algorithms were compared, the bars show in how many cases which one was the better. In some cases the greedy algorithm proved to be superior to the improved one. The reason for this is, that in a case of draw in the greedy algorithm the next letter was chosen randomly, that could result in a better performance.

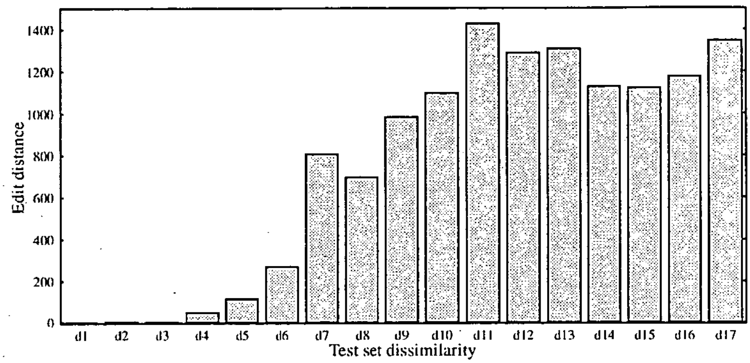

Figure 2: Improvements measured in edit distance.

In Figure 2 the total distances were summed (i.e. the distances of the approximate median from the test set). The same garbled sets were used as in Figure 1 and values of diagram are the difference between the totals for the improved and the greedy algorithm. We see that a slight modification in the greedy algorithm results in computing better medians whenever the problem becomes more difficult. Since the total sum of distances is bounded from above by the total length of strings from the test set, the results remain stable when we choose the dissimilarity value higher than the length of the distorted string.

6 Conclusions

The improved approximate median algorithm is a simple refinement of the greedy algorithm [3]. It has the same time complexity $O(k^2n|\Sigma|)$ as the previous one. The space complexity was a bit reduced by the help of storing only the last rows of the distance matrixes. This idea is based on [9], in this way the new algorithm runs in $O(kn)$ space. The closer the garbled strings are to each other the improvement is less significant. Therefore the improved algorithm presented in this paper is more suitable for searching approximate median of highly dissimilar strings.

Acknowledgement

I am grateful to the anonymous reviewers for their helpful comments which helped me improve the quality of the paper. In particular, I thank the anonymous referee who provided an improved English version of my manuscript, and Dr. János Csirik for calling my attention to the median string problem.

References

[1] D. Lopestri, J. Zhou: Using Consensus Voting to Correct OCR Errors. Series in Machine Perception and Artifical Intelligence Vol. 14 pages 157-168, 1995

[2] A. Juan, E. Vidal: An Algorithm For Fast Median Search. Pattern Recognition and Image Analysis Vol. 1 pages 187-192, 1996

[3] F. Casacuberta, M. D. Antonio: A greedy algorithm for computing approximate Median Strings. Pattern Recognition and Image Analysis Vol. 1 pages 193-198, 1996

[4] M. Crochemore, W. Rytter: Text Algorithms. Oxford University Press, 1994

[5] J.B. Krushkal: An overview of sequence comparison: Time warps, string edits, and macromolecules. SIAM Review Vol. 25 pages 201-237, 1983.

[6] H. Rulot: Un Algoritmo de Inferencia Gramatical mediante Correccin de Errores. Tesis Doctoral. Universitat de Valencia, 1992.

[7] S. V. Rice, J. Kanai, T. A. Nartker: A difference algorithm for OCR-generated text. Proceeding of the IAPR Workshop on Structural and Syntactic Pattern Recognition, Bern, 1992.

[8] T. Kohonen: "Median strings". Pattern Recognition Letters Vol. 3 pages 309-313, 1985.

[9] L. Allison, T. I. Dix: A bit-string longest-common-subsequence algorithm. Information Processing Letters Vol. 23 pages 305-310, 1986