Efficient Multi-word Parameterized Matching on Compressed Text

Rajesh Prasad
Department of Computer Science
Yobe State University
Damaturu, Nigeria
rajesh_ucer@yahoo.com

Rama Garg
Department of Computer Science & Engineering
Ajay Kumar Garg Engineering College
Ghaziabad, India
ramagargit3@gmail.com

***Abstract*—Searching set of patterns {P_1, P_2, P_3, ….P_r}, r≥1, inside body of a text T[1...n] is called multi-pattern matching problem. This matching is said to be parameterized match (p-match), if one can be transformed into the other via some bijective mapping. It is mainly used in software maintenance, plagiarism detection and detecting isomorphism in a graph. In the compressed parameterized matching problem, our task is to find all the parameterized occurrences of a pattern (set of patterns) in the compressed text, without decompressing it. Compressing the text before matching reduces the size and minimizes the matching time also. In this paper, we develop an efficient algorithm for parameterized multi-word matching problem on the compressed text, where both patterns and text are compressed before actual matching is performed and pattern is treated as word. For compressing the pattern and text, we use efficient compression code: Word Based Tagged Code (WBTC) and bit-parallel algorithm is used for searching purpose. Experimental results show that our algorithm is up to three times faster than the search on the uncompressed text.**

Keywords—String matching, Compressed parameterized matching, compressed pattern matching, information retrieval, multiple matching and word based tagged code.

I. INTRODUCTION

Finding all the occurrences of a pattern P of length m in a text T of length n (m ≤ n) is called string matching problem [1, 5, 10, 11]. Searching set of patterns P_1, P_2, P_3, ….P_r, r≥1, each of different length, inside body of a text T[1...n] is called multi-pattern matching problem. We assume that pattern and text are drawn from some finite alphabet set Σ. This problem has important applications in information retrieval, computational biology, text editor, plagiarism detection etc. One variation of string matching is parameterized matching [2, 3, 6, 15, 16, 17], generally used in plagiarism detection and software maintenance. It refers to finding all the parameterized occurrences of a pattern in the text. In this type of matching, a text and a pattern is given and we find all substrings of the text that can be transformed into the pattern by using a bijective mapping on the alphabet. Bijective mapping is a renaming of a set of distinct characters into another set of characters. Two strings S and R drawn from an alphabet Σ are parameterized match (p-match), if ∃ a bijection on the alphabet Σ. For example, strings “abae” and “fcfg” are p-match because there exist a bijection mapping {a->f, b->c, e->g} which transforms “abae” into “fcfg”.

Parameterized matching was first introduced by Baker [2] as a tool for software duplication detection in software maintenance. Fredriksson [6] suggested algorithm based on shift- or and backward DAWG matching (BDM). Baker in [3] worked on improving the performance of parameterized matching on average case. Her algorithm finds parameterized matches using suffix tree and pre-process both the text and pattern. Fast Parameterized Boyer-Moore-Horspool (FPBMH) algorithm has been developed by L.Salmela in [15]. This algorithm is extension of Horspool variant of Boyer-Moore algorithm into a parameterized string matching domain. This algorithm is fast in practice, uses only parameterized alphabet and is basically suited for color image processing [15].

In the compressed parameterized matching problem [4], our task is to find all the parameterized occurrences (p-match) of a pattern in the compressed text, without decompressing it. The pattern may or may not be compressed. By compressing the text before matching, it reduces the size and minimizes the matching time also. Richard and Donald in [4] developed an algorithm for the same problem but they are not compressing the pattern but only the text. They use the LZ-family of algorithm for compression.

Since words are treated as a basic unit in Information Retrieval (IR) system, therefore, in this paper, we mainly focus on the parameterized word matching on the compressed text. Here both word and text are compressed before actual matching is performed. For compression, we use efficient compression code: Word Based Tagged Code (WBTC) [8, 9]. This code allows direct searching of a pattern inside the text and also allows partial compression and decompression of arbitrary portion of the text.

In this paper, we develop an efficient algorithm for parameterized multi-word matching problem on the compressed text, where both patterns and text are compressed before actual matching is performed and pattern is treated as word. For compressing the pattern and text, we use efficient compression code: Word Based Tagged Code (WBTC) and bit-parallel algorithm is used for searching purpose. Experimental results show that our algorithm is up to three times faster than the search on the uncompressed text. Experimental results

978-1-4799-4998-4/14/$31.00 ©2014 IEEE

show that our algorithm is up to three times faster than the search on uncompressed text (FPBMH, 2008) [15]. To the best of my knowledge, currently, there is no work to support p-matching between compressed multiple word and compressed text.

The paper is organized as follows. In Sec. II terminologies and concepts related to the compression and parameterized matching is introduced. In Sec. III, proposed algorithm: Multi-word Parameterized Matching (MPM) is introduced. In Sec. IV, we discuss the experimental results. Finally, we conclude in Sec.V.

II. RELATED CONCEPTS

A. Word Based Tagged Code

Gupta and Agarwal in [8, 9] have developed an efficient compression technique: Word based Tagged Code (WBTC) where word is treated as basic unit of compression. WBTC is a semi-static statistical word-based model. The WBTC shows improvement in compression time while keeping all the good features of Tagged Sub-optimal Code like:

- It is a prefix code.
- It enables fast decompression of arbitrary portions of the text by using a flag bit.
- It permits searching directly on the compressed text.

The code generated in WBTC always ends with either **01** or **10**. This implies that the bit combination 01 or 10 act as a flag to indicate the end of code.

The coding procedure of WBTC is simple. First, source text is parsed and all the statistics of vocabulary in the text is gathered. The vocabulary is sorted with non increasing frequency. Each codeword in WBTC will be generated with the help of 2 bit patterns (00, 11, 01, 10) [18]. Following procedure has been used for the assignment of codes to the vocabulary:

1. The first 2^l words (rank: 0 to 2^0) of the vocabulary are assigned codes as 01 and 10 respectively. Here l=1.
2. Next 2^l words in the positions from $2^1 + 0$ to $2^2 + 2^0$ are encoded using four bits by adding 00 and 11 as prefix to all the codes of previous level. Here l=2.
3. In general, for any value of l, next 2^l words present in the positions from $2^l - 1 + (2^l - 2 + ... + 0)$ to $2^l + (2^l - 1 + ... + 2^0)$ of vocabulary are assigned codes using $2 \times l$ bits, by adding 00 and 11 as prefix to all the codes generated at preceding step.
4. Steps 1-3 above for code assignment to vocabulary words are repeated until the entire N words in the original sequence are encoded.

Example 1 given below illustrates the concept of coding in WBTC.

Example 1

Let the sequence is T = AATT CCGG GATA CGAC AATT GATA GATA ACAC.

The content of vocabulary is given in the Table I. The content of vocabulary after sorting is given in the Table II and assignment of the code words is shown in the Table III.

TABLE I. CONTENT OF VOCAB FOR EXAMPLE 1

Index	Word	Frequency
0	AATT	2
1	CCGG	1
2	GATA	3
3	CGAC	1
4	ACAC	1

TABLE II. CONTENT OF VOCAB AFTER SORTING FOR EXAMPLE 1

Index	Word	Frequency
0	GATA	3
1	AATT	2
2	CCGG	1
3	CGAC	1
4	ACAC	1

TABLE III. CODE ASSIGNMENT FOR EXAMPLE 1

Index	Word	Code
0	GATA	01
1	AATT	10
2	CCGG	0001
3	CGAC	0010
4	ACAC	1101

B. Parameterized String Matching Problem

Parameterized matching problem [2, 3] refers to finding parameterized match (p-match) occurrences of all the substrings of text with the symbols of the pattern. In this matching, generally we use two different alphabets: Σ: for fixed alphabet and Π: for parameterized alphabet. Symbols from fixed alphabet remain the same whereas symbols from parameterized alphabet are consistently renamed. Any one set can be empty also. In this matching a text and a pattern is given and we find all substrings of the text that can be transformed into the pattern by using a bijective mapping on the alphabet. Bijective mapping is a renaming of set of distinct characters into another set of characters. Two strings S and R drawn from an alphabet Σ are a parameterized match (p-match) if, there exist a bijection on the alphabet Σ.

Strings “abae” and “fcfg” are a p-match because there exist a bijection mapping {a->f, b->c, e->g} which transforms “abae” into “fcfg”. On the other hand, strings “eebb” and “acbb” are not a p-match because a bijection cannot map both ’a’ and ’c’ to ’e’, and thus there is no bijection that can transform “acbb” to “eebb”. A natural way to determine whether two strings are

parameterized match or not is to consider their predecessor strings.

Predecessor String

For a string S, the predecessor string is denoted by pred(S) and is defined as: If a character at position i has an earlier occurrence in the position j, the predecessor string contains i-j at position i. Otherwise, the predecessor string contains 0. First occurrence of each character is denoted by 0. Predecessor value for the symbol of Σ remains the symbol of Σ (In fact predecessor value is assigned to symbol of Π only). For example: for the string S = "aabac" over Σ=ϕ and Π={a, b, c}, the predecessor string of S is pred(S)=01020 and for the string S="aabac" over Σ={b} and Π={a, c}, the predecessor string of S is pred(S)=01b20. If two strings p-match then their predecessor strings match exactly. Parameterized matching (p-matching) can be performed by only considering predecessor strings using the fact that two equal length strings S and S' p-match iff their predecessor strings matches exactly. In the further discussion, we assume Σ = ϕ (i.e. strings are having symbols from Π only.

Index Calculation

In this method, we first transform the given string to a predecessor string. Now the predecessor string is "00033". After obtaining predecessor string, we convert it into an index [15]. For first character, no bits are reserved because it is always the same so it is not used in the index calculation. For second character one bit is reserved, so first bit in the index will be 0. For third character two bits are reserved, so we transform the 0 into bits 00 and so on. Therefore the string "acgac" will be converted into index $(00011011)_2$ in binary which represents the number 27.

C. Bit Parallel Method of Matching (Shift-or)

First we define the following terms:

- $b_{m-1}b_{m-2}\ldots b_0$ denotes the bits of a computer word of length m. Here bits are numbered from right to left.

- Exponentiation is used to denote bit repetition (e.g. $0^21=001$)

- C-like syntax is used for operations on the bits of computer words: "|" is for bit-wise or, "&" is for bit-wise and, "~" complements of all the bits. The shift left operation "<<r", moves all the bits to the left by r bits and enters 'r' zeros in the right.

Now we present the first bit-parallel algorithm (shift-or) [1] for exact string matching. In this algorithm, an automaton for the pattern P[0…m-1] is constructed as follows: The automaton has states 0, 1, 2…m. The state 0 is the initial state, state m is the final state and ∀ i = 0…m-1, there is a transition from state i to state i+1 for the character P[i]. In addition, for every c∈Σ, there is a self loop on the initial state. For example, for the pattern P = 10, with Σ = {0, 1}, the automaton is shown in Fig. 1.

In the preprocessing phase, algorithm builds a table B of size |Σ|× m. ∀c ∈ Σ, the mask B[c] has i^{th} bit (from right to left) equal to 0 iff P[i] = c otherwise it is 1, for 0 ≤ i ≤ m-1. These correspond to the transitions of the implicit automaton. That is, if the bit *i* in *B*[*c*] is 0, then there is a transition from the state *i* to the state *i* + 1 with character *c*. For example, for P = 10, B[0] = 01, B[1] = 10.

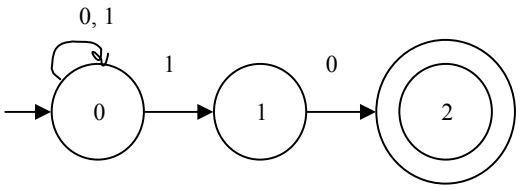

Fig. 1. A shift-or automaton for P = 10

In the searching phase, the algorithm uses a bit vector D of size m for the states of the automaton. The i^{th} bit of D is set to zero iff state i is active. Initially each bit of D is set to 1. After reading the i^{th} text character T[i], 0≤i≤n-1, the vector D is updated by: D ← (D << 1) | B[T[i]]. This simulates all the possible transitions of the nondeterministic automaton in a single step. Whenever after reading T[i], the $(m-1)^{th}$ bit of D becomes zero, it shows occurrence of P in T with shift i-m. If m ≤ w, then the running time of the algorithm is *O*(n).

III. PROPOSED ALGORITHM (MPM)

This section presents our proposed Multi-word Parameterized Matching (MPM) algorithm on the compressed text. Algorithm works in two phases: Preprocessing and Searching.

A. Preprocessing Phase

Let the words to be searched in the compressed text are P_1, P_2, P_3,...P_r, r ≥1. Suppose that the minimum length of a word among all the word is m. Preprocessing is done on both the text and patterns as follows:

Preprocessing of Text

Algorithm reads a source text file and extracts all the words of length m. If any word's length is greater than m, then it is splitted into m-gram (sequence of m-characters). For example, suppose m = 4 and any word on alphabet {A, B, C, D} of a text is ABCCBA. The different 4-grams are: ABCC, BCCB and CCBA. The reason behind splitting the text into m-gram is that the word to be searched may parameterize match (p-match) in the substring also. For example, let the searched word is P = DAAD. It p-matches the substring "BCCB" of ABCCBA using the mapping: D↔B and A↔C.

After obtaining words of m-grams, we perform prev-encoding (ref. Sec. II (B)) on each word. After prev-encoding, we get the equivalent index using the method discussed in Sec. II (B). Now, the prev-encoded words are stored in the vocabulary table along with its frequency and index. Every time a word repeats, its frequency is incremented by 1. Now we sort the content of vocabulary. Working of the phase is illustrated with the help of following example:

Let the original sequence on alphabet set {A, B, C, D, E} is: T = ABAA BABAA BCDD ACBB DEDEBB. Further, let the minimum length of set of words is 4. We divide each words having length 4 as follows to get the new set of word:

T = ABAA {BABA ABAA} BCDD ACBB {DEDE EDEB DEBB}, where {…} denotes that a particular word of length greater than 4 is splitted into 4-grams. This is only for convenience. We may remove it without affecting the answer.

The prev-encoded text is: T = 0021 {0022 0021} 0021 0021 {0022 0020 0021}.

Let the vocabulary be ***vocab*** and its content are shown in Table IV.

TABLE IV. CONTENT OF VOCAB

S/N	Word	Index (Numeric value of Word)	Frequency
0	0021	9	5
1	0020	8	1
2	0022	10	2

Now we sort the content of **vocab** with respect to non-increasing frequency. The contents of vocab after sorting are shown in Table V.

TABLE V. SORTED CONTENT OF VOCAB

S/N	Word	Index (Numeric value of Word)	Frequency
0	0021	9	5
2	0022	10	2
1	0020	8	1

After getting sorted the vocab, the method actually does encoding as explained in Sec. II(A). The codes are shown in the Table VI.

TABLE VI. CODE ASSIGNMENT

S/N	Word	Index (Numeric value of Word)	Codes
0	0021	9	01
2	0022	10	10
1	0020	8	0001

Hence, the compressed text T′ of the above text T is: T′ = 01 10 01 01 01 10 0001 01.

Preprocessing the Set of Words

Suppose that the set of words are P_1, P_2, P_3,…..P_r, r≥1 and minimum length of a word among all the words is m. In this phase, we consider only m-length word and remove the other characters from a word having length > m (because text is having only m-length word). We take prev-encoding of each word and calculate its index (*numeric value*) also. For example, suppose we are having words: ABAA, BABAA and BBAAB. Minimum length is=4. Now 4-lentgh words are: ABAA, BABA, and BBAA. Prev-encoded words along with index (in parenthesis) are: 0021(9), 0022(10) and 0101 (17).

Now we search the index, numeric value (*searching index, requires less comparison as compared to searching prev-encoded word)* of the set of word in the table obtained above. If any index (numeric value) is not present in table, then the corresponding word is removed from the set of words for further processing as this word will never be present in the text. For example, the word BBAA (17, BBAAB) will be removed from the set of words stating that it can't be present in the compressed text.

Now, we apply the bit-parallel algorithm for further encoding the set of words. We find bit-table for the corresponding words by considering their code from the above tables. We take logical AND of individual bits of each bit table to find the final bits of each characters. Following Table VII illustrates the concept:

TABLE VII. CONTENT OF BIT TABLE B

Word	Prev-encoding	Code	B [Code]
ABAA	0021	01	$B_1[0]$=1110 $B_1[1]$=1101
BABA	0022	10	$B_2[0]$=1101 $B_2[1]$=1110
ABAD	0020	0001	$B_3[0]$=1000 $B_3[1]$=0111
Final Bit Table			B[0]=1000, B[1]=0100

The length of B is equal to maximum of length of the codes. For example, the length of B in Table VII is 4.

B. Searching Phase

Searching for set of words inside the text can be carried out using the bit-parallel algorithm (Sec. II (C)) with little modification: after each reporting of a word inside the compressed text, vector D is initialized as ~0. If any one of the *l* bit (where *l* is described in Sec. II (A)) is zero, word may occur in the text which is confirmed after verification. For example, consider the above set of words: ABAA, BABAA and BBAAB inside the text T = ABAA BABAA BCDD ACBB DEDEBB. As discussed, the code of corresponding words is: 01, 10, and 0001 respectively.

Compressed text is T′ = 01 10 01 01 01 10 0001 01. Initially D=1111

1. D= 1110 | B[0]
   =1110 | 1000 = 1110
2. D=1100 | B[1]
   =1100 | 0100 = 1100

Now, 2nd bit (from right to left) is zero, so word having code 01 (ABAA) may match inside the compressed text, which is confirmed after verification. Now D is re-initialized as ~0.

3. D=1110 | B[1]
   =1110 | 0100 = 1110
4. D=1100 | B[0]
   =1100 | 1000 = 1100

Again 2nd bit (from right to left) is zero, so word having code 10 (BABA) may match inside the compressed text, which is confirmed after verification. This process continues until end of the text.

So, the method speeds up the searching process and also saved the space.

IV. EXPERIMENTAL RESULTS

In order to evaluate the efficiency of our algorithm (MPM), we performed several experiments on three kinds of texts:

- Alphabet of size 4 characters: Genome is a DNA sequence composed of the four nucleotides also known as base pairs: Adenine, Cytosine, Guanine and Thymine. The size of file 16KB is taken from file ftp://ftp.ncbi.nih.gov/genomes.
- Alphabet of size 2 characters: text is composed of two characters 0 and 1 that are randomly created. The size of file is equal to 12 KB.
- Alphabet of size 256 characters: Bible files of size 4 KB is taken from www.BibleTexts.com.

To evaluate algorithms thoroughly, we run them with different pattern-set-length and different size of pattern set. In each run, the pattern set X is randomly created.

We have implemented our proposed algorithm (MPM) in C, compiled with turboC++ 3.0 on the windows 7, 2.53GHZ processor with 4GB RAM. Table V shows the execution time of PMM algorithm on varying alphabet sizes (Binary alphabet = {0, 1}, DNA alphabet = {A, C, G, T} and ASCII alphabet). Table VI shows the Comparison of Execution time(s) of (MPM) algorithm and existing FPBMH [15] algorithm (For fixed alphabet size = 256). Table VII shows the execution time of algorithm against multiple pattern of different length on alphabet size = 256.

V. CONCLUSION & FUTURE SCOPE

In this paper, we have applied the parameterized word matching on the compressed text, where both patterns and text are compressed before actual matching is performed. For compressing the pattern and text, we use efficient compression technique: Word Based Tagged Code (WBTC). From Table VIII, we see that the execution time of algorithm decreases after increasing the alphabet size and pattern length. From Table IX, we see that the execution time of compressed algorithm (MPM) is approximately one third to that of FPBMH [15] algorithm, when pattern length increases. From Table X, it is clear that the execution time is increasing with increasing number of patterns for fixed size and length of pattern, but it is better than the time for individual searching of a pattern. Limitation of algorithm is that it can't handle the words if code length > w (word length of computer used).

REFERENCES

[1] R. Baeza-Yates, "Efficient Text Searching," Ph. D. Thesis, Department of Computer Science, University of Waterloo, 1989.

[2] B. S. Baker, "A theory of parameterized pattern matching: Algorithms and applications," Proceedings of the 25th ACM Symposium on the Theory of Computation, ACM Press, New York, 1993, pp. 71–80.

[3] B. S. Baker, "Parameterized diff.," Proceedings of the 10th Annual ACM–SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1999, pp. 854–855.

[4] R. Beal, and D. A. Adjeroh, "Compressed Parameterized Pattern Matching," 2013 IEEE Data Compression Conference, March 20-22, 2013, pp. 461-470.

[5] R. S. Boyer, and J. S. Moore, "A fast string-searching algorithm," Communication of ACM, 20(10), 1977, pp. 762-772.

[6] K. Fredriksson, and M. Mozgovoy, "Efficient parameterized string matching," Information Processing Letters 100 (3), 2006, pp. 91–96.

[7] A. Goel, and R. Prasad, "Efficient Indexing Techniques for Record Matching and Deduplication," International Journal of Computer Vision and Robotics (Inderscience)-Vol. 4, No. (1/2), 2014, pp. 75-85.

[8] A. Gupta, and S. Agarwal, "A Scheme That Facilitates Searching And Partial Decompression of Textual Documents," Intl. Journal of Advanced Computer Engineering, 1(2), 2008.

[9] A. Gupta, and S. Agarwal, "A Novel Approach of Data Compression for Dynamic Data," In proc. of IEEE third International Conference on System of Systems Engineering, California, USA, 2008.

[10] R. N. Horspool, "Practical fast searching in strings," Software-Practice & Experience 10(6), 1980, pp. 501-506.

[11] T. Lecroq, "Fast Exact String Matching Algorithms," Information Processing letters, 102(6), 2007, 229-235.

[12] E. S. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates, "Fast searching on compressed text allowing errors," In Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Melbourne, Australia, Aug. 24–28, 1998, pp. 298–306.

[13] E. S. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates, "Direct pattern matching on compressed text," In Proceedings of 5th International Symposium on String Processing and Information Retrieval (SPIRE'98), IEEE Computer Society, Washington, pp. 90–95, 1998.

[14] E. S. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates, "Fast and flexible word searching on compressed text," ACM Transaction on Information Systems, 18(2), 2000, pp.113-139.

[15] L. Salmela, and J. Tarhio, "Fast Parameterized Matching with q-grams," Journal of Discrete Algorithm 6(3), 2008, pp. 408-419.

[16] S. Tevatia, and R. Prasad, "Multi-patterns Parameterized Matching with Application to Computational Biology," International Journal of Information and Communication Technology (Inderscience)-Accepted (in press), 2014.

[17] S. Tevatia, R. Prasad, and D. Rai, "An offensive algorithm for multi-pattern parameterized string matching," proc. of IEEE International conference (ICCCCM-13), Allahabad, India, August 3-4, 2013, pp.1-5.

[18] A. Gupta, V. Rishiwal and S. Agarwal, "Efficient Storage of Massive Biological Sequences in Compact Form," proc. of Contemporary Computing - Third International Conference, IC3 2010, Noida, India, August 9-11, 2010, pp. 13-22.

TABLE VIII. EXECUTION TIME(S) OF (MPM) ALGORITHM ON DIFFERENT ALPHABET (NO. OF PATTERN =1)

Length of Pattern	Execution Time (alphabet size=2)	Execution Time (alphabet size=4)	Execution Time (alphabet size=256)
4	0.641026	0.641026	0.659341
6	0.516483	0.439560	0.494505
8	0.512820	0.439560	0.238095
10	0.659341	0.421245	0.201465
12	0.586081	0.366300	0.183150

TABLE IX. COMPARISON OF EXECUTION TIME(S) OF (MPM) AND FPBMH (2008) ALGORITHM

(FOR FIXED ALPHABET SIZE = 256, NO.OF PATTERN = 1)

Length of Pattern	Execution Time (FPBMH)	Execution Time (MPM)
4	1.714280	0.659340
6	1.325260	0.494500
8	0.666650	0.238090
10	0.604380	0.201460
12	0.549450	0.183150
14	0.494500	0.164830

TABLE X. EXECUTION TIME(S) OF (MPM) FOR VARYING NO. OF PATTERNS (FOR FIXED ALPHABET SIZE = 256)

Min Length	No. of Patterns					
	2	3	4	5	6	7
4	1.153846	1.648350	2.472530	3.241758	3.296703	4.010989
	1.043956	1.758240	2.417580	3.186813	3.241758	3.956044
	1.000000	1.428570	2.362640	3.076923	3.131868	3.681319
5	1.043956	1.208790	1.703300	2.157802	2.692308	2.692308
	0.934066	1.153850	1.648350	2.127802	2.197802	2.582418
	0.879121	1.043960	1.373630	2.087912	2.142857	2.472527
6	1.000000	0.879120	1.428570	1.538462	2.197802	2.472527
	0.714286	0.769230	1.263740	1.483516	2.142857	2.417582
	0.659341	0.659340	1.000000	1.373626	1.868232	2.362637
7	0.934066	0.604400	0.824180	1.000000	1.208791	1.318681
	0.329670	0.549450	0.769230	0.824176	1.098901	1.208791
	0.274725	0.494510	0.659340	0.769231	1.043956	1.153846
8	0.384615	0.659340	0.769230	0.934066	0.989011	1.263736
	0.329670	0.604400	0.714290	0.879121	0.879121	1.208791
	0.274725	0.549450	0.494510	0.714286	0.824176	0.989011
9	0.384615	0.439560	0.532560	0.769231	0.934066	1.098901
	0.274725	0.384620	0.494510	0.714286	0.824176	0.989011
	0.100000	0.329670	0.439560	0.604396	0.769231	0.934066