ABSTRACT

FAST PROGRAM FOR SEQUENCE ALIGNMENT USING PARTITION FUNCTION POSTERIOR PROBABILITIES

by
Meera Prasad

The key requirements of a good sequence alignment tool are high accuracy and fast execution. The existing Probalign program is a highly accurate tool for sequence alignment of both proteins and nucleotides. However, the time for execution is fairly high. The focus is therefore, to reduce the running time of the existing version of Probalign, maintaining its current accuracy level.

The thesis conducts a detail analysis of the performance of Probalign to bring down the running time of the existing code. A modified version of Probalign, Version 1.4 is released. A new program for sequence alignment with faster computation is also introduced.

FAST PROGRAM FOR SEQUENCE ALIGNMENT USING PARTITION FUNCTION POSTERIOR PROBABILITIES

by
Meera Prasad

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Bioinformatics

Department of Computer Science

May 2011

APPROVAL PAGE

FAST PROGRAM FOR SEQUENCE ALIGNMENT USING PARTITION FUNCTION POSTERIOR PROBABILITIES

Meera Prasad

Dr. Usman Roshan, Thesis Advisor Associate Professor of Computer Science, NJIT	Date
Dr. Zhi Wei, Committee Member Associate Professor of Bioinformatics, NJIT	Date
Dr. Jason T.L. Wang, Committee Member Professor of Bioinformatics and Computer Science, NJIT	Date

BIOGRAPHICAL SKETCH

Author:	Meera Prasad
Degree:	Master of Science
Date:	May 2011

Undergraduate and Graduate Education:

- Master of Science in Bioinformatics,
  New Jersey Institute of Technology, Newark, USA, 2011

- Bachelor of Science in Computer Science,
  MG University, Kerala, India, 2007

Major:	Bioinformatics

This thesis is a dedication to my beloved family.

To my loving husband, without whose constant support and guidance, it would not have been possible.

To my parents and brother, for their unconditional affection, encouragement and patience.

To God Almighty for his unbound love and blessings.

ACKNOWLEDGMENT

I am grateful to my advisor, Dr. Usman Roshan, for his constant guidance and support throughout my master's study at the New Jersey Institute of Technology. It has been an invaluable opportunity for me to work at the Bioinformatics Lab under his direction. I believe that the experience and the exposure I gained will significantly benefit my future career. Special thanks to my dissertation committee members, Dr Jason Wang and Dr Zhi Wei for their support and guidance. I also thank the department chair, Dr. Narain Gehani, and all other faculty members for their encouragement.

This project would not have been a success without the help of System administrators. I would like to thank Kevin Walsh, Douglas Eadline, Gedaliah Wolosh and all other people at system administrator for providing service for Kong, AFS and OSL systems.

I would like to thank my good colleagues and friends for their support in my studies. I am also grateful to my TA, Wei Wang for his constant support and guidance from time to time.

Most of all, I am grateful for the constant support, understanding, patience, and trust of my husband, my parents and my brother without whom none of this work would have been possible.

Thanks to all for helping me complete this thesis.

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Objective

The focus of this thesis is to improve the run time performance of Probalign which constructs maximal expected accuracy sequence alignments. This thesis also presents a new program for high speed alignment of a short sequence to an entire genome.

Initial focus was to improve the performance of Probalign. The existing program was considerably optimized to bring down the running time. On achieving faster performance, the author then focused on developing a new application which performs the sequence alignment of nucleotides much faster than Probalign Version 1.3 using Smith – Waterman model. The run time of the modified version of Probalign and the newly introduced program were compared against the original Probalign code.

1.2 Background Information

Protein and nucleotide sequence alignment is a widely employed task in bioinformatics (Notredame et al., 2002) that helps in detecting functional regions and evolutionary histories (Durbin et al., 1998). There are many existing alignment tools like ClustalW (Thompson et al., 1994), Probcons (Do et al., 2005), MAFFT (Katoh et al., 2005). In terms of accuracy, recent studies show that MAFFT and Probcons are among the top performing alignment tools (Do et al., 2005; Katoh et al., 2005). Probalign bridges two important bioinformatic techniques in an effort to produce more accurate multiple sequence alignment (Roshan and Livesay, 2006). The first approach estimates posterior probabilities from the partition function of alignments and the second approach computes maximal expected accuracy alignment after applying the probability consistency transformation of Probcons (Do et al., 2005; Roshan and Livesay, 2006). Probalign was found to produce statistically significantly better alignments on BaliBASE 3.0, HOMSTRAD and OXBENCH benchmarks (Roshan and Livesay, 2006).

1.3 Overview of Probalign

Probalign is a sequencing tool that can be used to sequence multiple alignments of proteins and nucleotides. The program performs global multiple alignment in which each residue of each sequence needs to be aligned. Partition function methodology is used to estimate the pair wise posterior probabilities of the residues to align the sequences. This in turn is used to compute the maximum expected accuracy optimization for multiple sequences [as discussed in Roshan and Livesay, 2006]. Probalign does not look at all the alignments but only a subset of suboptimal alignments. This is determined by the thermodynamic temperature. It is of interest that these suboptimal alignments are biologically more accurate. In most of the experiments, the suboptimal alignments proved to be biologically more significant than the most optimal alignment. This was the underlying motivation for the Probalign approach (Roshan and Livesay, 2006).

Pros and Cons of Probalign

In terms of accuracy, Probalign is found to outperform other existing tools such as Probcons, MAFFT, MUSCLE and so on. Probalign works better than other existing methods when it comes to aligning heterogeneous datasets which are extremely long.

Also, another advantage is, Probalign does not have to learn model parameters from training data.

However, in terms of speed, Probalign runs impractically slower than these existing tools. The larger the length of the sequences that has to be aligned, the slower is the performance. This leads to the need of a competitive execution time of alignment computation. The requirement is thus a tool which can align short sequences to an entire genome accurately with high computation speed.

CHAPTER 2

PROBALIGN V.1.4

2.1 Time Profile Analysis

In order to bring down the running time of the existing version of Probalign (Roshan et al., 2008), it was essential to study the time taken for computation. A time profile of the various blocks and functions in the code was conducted. This was then executed under different scenarios. A detailed analysis of this time profile helped to analyze the running time of the various blocks of code.

2.2 Methodology

The existing code was broken down into several blocks. To begin with, the main function, Main.cc, was divided into blocks, each consisting of a function that is invoked for computing the alignment. On time profiling, it was found that one of the functions DoAlign was taking the largest running time of 309 seconds. This in turn invokes ComputePostProbs function which is defined in the PostProbs.cc file. For further analysis, PostProbs.cc file was divided into several blocks. These blocks were then time profiled. This returned the time taken for memory allocation, initialization of Z matrices, computation of posterior probabilities for each residue of the sequences and freeing of the allocated memory. The block that includes the computation of the Z matrices using the exponentiation function took the largest running time. On further analysis, it was observed that these exponentiation functions were invoked in within loops. The next step was to set these exponentiation values to variables and thus move them outside from within these loops. The existing code was then modified to induce this change. The modified code brought down the running time considerably.

The variable numIterativeRefinementReps determines the number of iterations required for accurately determining the alignments. The number of iterations could vary from one to hundreds. The value of the variable was originally specified as a command line argument. However, the time for the execution of the program was large due to the number of times the computation was repeated without much improvement in the accuracy. On analyzing it was found that this variable can be set to one. This brought down the running time further. The time taken by DoAlign was brought down to 108 seconds.

The scorez_matrix in the file Matrix.h was updated to the values of +5/-4 match-mismatch score. This matrix was then used in the ReadMatrix.cc file. Thus, the total run time of Probalign was reduced to 15 seconds. The values for temperature, gap open and gap extend was set to 7, 26 and 5 respectively as command line arguments while executing the code. This is the new version introduced; Probalign V.1.4.

Table 2.1 Run Time Comparison of Version1.3 and Version 1.4

VERSION	MAIN	DOALIGN	POSTERIOR PROBABILITY	CONSISTENCY TRANSFORMATION	FINAL ALIGNMENT	ITERATIONS
PROBALIGN 1.3	290	290	186	26	5	1
PROBALIGN 1.4	15	15	8	3	2	1
PROBALIGN 1.3	493	493	278	25	190	100
PROBALIGN 1.4	115	115	10	2	103	100

Table 2.1 provides the running time for each of the function block. The run time under two different scenarios; when numIterativeRefinementReps =1 and later when set to 100,

is also specified in the table. The running time, in seconds, of the various functions that are invoked in the Main are compared. It can be seen that Probalign Version 1.4 has an execution speed much faster than Version 1.3. Version 1.4 of Probalign can align sequences of both proteins and nucleotides using the pair wise posterior probability methodology. The accuracy of the alignment was not compromised and still remains the same.

CHAPTER 3

PROBALIGN V.1.5

3.1 Time Profile Analysis

The focus of this module was to further bring down the run time of the newly modified version of Probalign. This program is quite fast when compared to its previous version. However, a faster code to align sequences of nucleotides was of interest. The goal is to use the same pairwise posterior probability methodology for calculating the suboptimal alignments. Time profiling of Version 1.4 gave a clear picture about the functions which took maximum amount of time for execution. This helped in determining the key functions which were required for sequence alignments. The other functions where then omitted.

3.2 Method and Modifications

The new code is written in C instead of C++ to bring down the run time overhead of invoking objects of different classes. There is considerable improvement in running time when coding in C. The objective of the new code is to align nucleotide sequences; hence the requirement of invoking different classes is omitted.

The main function is SRAlign.c which contains one function invoke to ComputePostProbs(). PostProbs.c which contains ComputePostProbs has the same functionality as the C++ file in Version 1.4. The functionality of the main file is divided into input, fragmentation, probability computation, recurrence, traceback and output blocks.

The inputs to the main function are two files; the first input file is a read file and the second one is a large genome file. The genome can be of varying length. It could vary from 5 million to 50 million or more. The next block is fragmentation, where the entire genome is broken into 1 million long fragments and the calls to ComputePostProbs(), recurrences, tracebacks, alignments of each fragment with the short read and mean posterior probabilities are calculated for each of these fragments.

The focus now moves onto the posterior probability computation block. A function call is made to PostProbs.c that computes the probability matrix used for alignment. On time profiling, it was found that the memory allocation and de-allocation consisted of the major running time. In order to decrease this time, the Z matrices, used in PostProbs.c, are allocated in the main function before the fragmentation process takes place. This way the memory for Z matrices are allocated only once and the allocated memory is freed only one. This saves a lot of computation time. The same memory space is being reused for every fragment.

Next, the recurrence and traceback of each of these fragment with the short sequence is computed. A simple dynamic program, the Smith – Waterman model, is used for recurrence and traceback. The mean posterior probability of these alignments is then calculated. The alignment with the largest mean posterior probability is returned as the output. Along with this output, the position of the alignment of the short sequence to the entire genome and the largest mean posterior probability are also returned.

Table 3.1 Run Time Comparison of New Program for Varying Genome Sizes

GENOME LENGTH	INITIALIZATION	MEMORY ALLOCATION	COMPUTATION	MEMORY DEALLOCATION	MAIN
1 Million	0	4	0	2	7
2 Million	0	8	0	4	13
5 Million	0	14	0	14	35

Table 3.1 provides the running time of the blocks in seconds. The table shows the run time of the blocks in PostProbs.c, for both the reverse partition functions and forward partition function. The table clearly shows that the new program is much faster than the existing programs with the same level of accuracy in the alignments.

CHAPTER 4

CONCLUSION

4.1 Results

From the run time values, we can conclude that the new code is faster than all the existing versions of Probalign. This program is two times faster than the previous version of Probalign; Version 1.4.

4.2 Discussion

The program uses most of the techniques for reducing the run time. Coding in C instead of C++ reduced the run time over head considerably. The genome is fragmented into 1 million sized fragments. ComputePostProb function along with the other funcationalities is then invoked for each of these fragments. PostProbs.c is called as many times as the number of these fragments. If Z matrices were allocated memory in this file, then malloc is invoked a number of times, which increases the execution speed. Hence, these memory allocations were moved to main function. Malloc in invoked for each of these Z matrices just once and the same memory space is then reused for alignment calculation for each of the fragments. This brings down the run time further. Also, the use of the simple Smith – Waterman model further reduces the run time.

This program is a fast and accurate program for aligning short sequences to a long genome. This thesis work can further be extended to aligning protein sequences and that would be the focus for expanding this research work.

APPENDIX

PROGRAM CODE FOR THE MAIN FUNCTION

This is the code for the main function that invokes the function, ComputePostProbs, in the PostProbs.c file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include "ctype.h"
#include "PostProbs.c"

typedef struct
{
  char monomers[26];  /* amino or nucleic acid order */
  float matrix[676];    /* entries of the score matix, 26*26=676 */
} score_matrix;

float TEMPERATURE = 7;
float GAPOPEN = 26;
float GAPEXT = 5;
argument_decl argument;

char bases[26]= "ABCDGHKMNRSTUVWXY";
int subst_index[26];
float sub_matrix[26][26];
float scorez_matrix[26][26];

//  Specifies scoring matrices and their structure
/////////////////////////////////////////////////////////////////
score_matrix nuc_simple =
{
        "ABCDGHKMNRSTUVWXY",
        {
               5,
               0,   0,
               -4,   0,   5,
               0,   0,   0,   0,
               -4,   0,   -4,   0,   5,
               0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               -4,   0,   -4,   0,   -4,   0,   0,   0,   0,   0,   0,   5,
               -4,   0,   -4,   0,   -4,   0,   0,   0,   0,   0,   0,   5,   5,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,  0
        }
};
```

```
inline void read_matrix(score_matrix matrx)
{
    int i, j, basecount,position=0;
       basecount = 17;

    for (i = 0; i < basecount; i++)
              subst_index[i] = -1;

    for (i = 0; i < basecount; i++)
              subst_index[bases[i] - 'A'] = i;

    if (TRACE == 1)
              printf("\nbases read: %d\n", basecount);


    for (i = 0; i < basecount; i++)
              for (j = 0; j <= i; j++)
              {
                      sub_matrix[i][j]=matrx.matrix[position++];
                      sub_matrix[j][i] = sub_matrix[i][j];
                      scorez_matrix[i][j]= exp(sub_matrix[i][j]*argument.beta);
                      scorez_matrix[j][i] = exp(sub_matrix[j][i]*argument.beta);
              }


}

//
//intialize the arguments (default values)
/////////////////////////////////////////////////////////////////////////////////
void init_arguments()
{
    float gap_open = -4, gap_ext = -0.25;
    int le;

       le =4;
    argument.N = 1;
    strcpy(argument.input, "tempin");
    argument.matrix = le;
    argument.gapopen = GAPOPEN;
    argument.gapext = GAPEXT;
    argument.T = TEMPERATURE;
    argument.beta = 1.0 / TEMPERATURE;
    argument.opt = 'P';

    read_matrix(nuc_simple);
     //now override the gapopen and gapext
    if (argument.gapopen != 0.0 || argument.gapext != 0.00)

    {
        gap_open = -argument.gapopen;
        gap_ext = -argument.gapext;
    }

    argument.gapopen = gap_open;
    argument.gapext = gap_ext;
    argument.opt = 'P';
}                               //end of init
/////////////////////////////////////////////////////////////////////////////////
//main  :  Code written by Meera Prasad, September 2010
/////////////////////////////////////////////////////////////////////////////////
```

```
int main ( int argc, char* argv[])
{
        FILE *input1, *input2;
        char *rFile = "";
        char *gFile = "";
        long rFileLen, gFileLen;
        char *rThisPtr, *gThisPtr;
        char *sequence1 = "";
        char *sequence2 = "";
        char *temp = "";
        int i = 0;
        int j = 0;
        float *postptr = 0;
        int seq1len, seq2len;
        int firstcounter;
        int secondcounter;
        char *firstalign = "", *secondalign = "", *finalFirstAlign="",
*finalSecondAlign="";
        int lencounter1, lencounter2;
        float **V;
        int **T;
        float p1, p2, p3;
        long double **Ze = NULL;
        long double **Zf = NULL;
        long double **Zm = NULL;
        long double **Zm_rev = NULL;
        int count, fragment = 0;
        int seqcount = 0;
        int templen = 1000000;
        int size, tracker = -1, point = 0, startpos = 0;
        float calculatedProb = 0.0;
        float largestCalcProb = -1.0;

        if( argc != 3 )
        {
                printf("Please verify input file, program requires input file as
parameter.\n");
                return 1;
        }
        input1 = fopen(argv[1], "r");
        input2 = fopen(argv[2], "r");
        if(input1 == NULL || input2 == NULL)
        {
                 perror("Error reading file");
        }
        else
        {
                printf( "\nReading file %s, %s ", argv[1], argv[2]);
                fseek(input1, 0L, SEEK_END);
                fseek(input2, 0L, SEEK_END);
                rFileLen = ftell(input1);
                gFileLen = ftell(input2);
                printf("\n");
                rewind(input1);
                rewind(input2);
                rFile = malloc((rFileLen+1) * sizeof(char));
                gFile = malloc((gFileLen+1) * sizeof(char));
                sequence1 = malloc((rFileLen+1) * sizeof(char));
                sequence2 = malloc((gFileLen+1) * sizeof(char));

                        if( rFile == NULL || gFile == NULL)
```

```
			{
				printf("\n Insufficient memory to read file.\n");
				return 0;
			}
		fread(rFile, rFileLen, 1, input1);
		fread(gFile, gFileLen, 1, input2);
		fclose(input1);
		fclose(input2);
	}

	//processing read File
	rThisPtr = rFile;
	while (*rThisPtr)
	{
		if( *rThisPtr == '>')
		{
			while(*rThisPtr != '\n')
			{
				*rThisPtr++;
			}
		}
		else if( *rThisPtr == '\n')
		{

		}
		else
		{
			sequence1[i] = *rThisPtr;
			i++;
		}
		*rThisPtr++;
	}
	free(rFile);
	gThisPtr = gFile;
	while (*gThisPtr)
	{
		if( *gThisPtr == '>')
		{
			while(*gThisPtr != '\n')
			{
				*gThisPtr++;
			}
		}
		else if( *gThisPtr == '\n')
		{

		}
		else
		{
			sequence2[j] = *gThisPtr;
			j++;

		}
		*gThisPtr++;
	}
	free(gFile);

	sequence1[i] = '\0';
	sequence2[j] = '\0';

	seq1len = strlen(sequence1);
```

```
        seq2len = strlen(sequence2);

        //allocation
        T=(int **)malloc((seq1len+1) * sizeof(int*) );
        V=(float **)malloc((seq1len+1) * sizeof(float*) );

        for(i=0;i<(seq1len+1); i++)
        {
                T[i] = malloc((templen+1) * sizeof(int));
                V[i] = malloc((templen+1) * sizeof(float));
        }

        //initializing arguments
        init_arguments();
        // allocation
        firstalign = malloc((templen+1) * sizeof(char));
        secondalign = malloc((templen+1) * sizeof(char));
        finalFirstAlign = malloc((templen+1) * sizeof(char));
        finalSecondAlign = malloc((templen+1) * sizeof(char));

        //Define Zm, Ze, and Zf here and allocate space for them
        Ze = (long double **)malloc(2 * sizeof(long double));
        Zf = (long double **)malloc(2 * sizeof(long double));
        Zm = (long double **)malloc((templen+1) * sizeof(long double));
        Zm_rev = (long double **)malloc(2 * sizeof(long double));

        for(i=0; i<=1; i++)
        {
                Ze[i] = malloc((seq1len + 1) * sizeof(long double));
                Zf[i] = malloc((seq1len + 1) * sizeof(long double));
                Zm_rev[i] = malloc((seq1len + 1) * sizeof(long double));
        }
        for(i=0; i<=templen; i++)
        {
                Zm[i] = malloc((seq1len + 1) * sizeof(long double));
        }

        temp = malloc((templen + 1) * sizeof(char));

        // split the genome into fragments and pass each fragment
        while(seqcount < seq2len)
        {
                fragment++;
                printf("\nfragment %d\n",fragment);
                count = 0;
                while(count < templen && seqcount < seq2len)
                {
                        temp[count++] = sequence2[seqcount++];
                }
                temp[count++]='\0';
                size = strlen(temp);
                init_arguments();
                postptr = ComputePostProbs(&argument, sequence1, seq1len, temp, size,
Ze, Zf, Zm, Zm_rev);
                assert(postptr);

                // initialization
                calculatedProb = 0; startpos = 0;
         for(i=0;i < (seq1len+1); i++)
         {
                 V[i][0] = 0;
                T[i][0] = 2;
        }
        for(j=0; j < (size+1); j++ )
        {
                V[0][j] = 0;
                        T[0][j] = 0;
        }
                //recurrence
        for( i=1; i<=seq1len; i++)
                {
                        for( j=1; j<=size; j++)
                        {

                                p1 = V[i-1][j-1] + postptr[(i) * (size + 1) + (j)];
                                p2 = V[i-1][j];
                                p3 = V[i][j-1];

                                if(p1>=p2 && p1>=p3)
                                {
                                        V[i][j]=p1;
                                        T[i][j]=1; startpos = j;
                                        //assign current probability
                                        calculatedProb = calculatedProb + (postptr[(i) *
(size + 1) + (j)]);
                                }
                                else if(p2>=p1 && p2>=p3)
                                {
                                        V[i][j]=p2;
                                        T[i][j]=2;
                                }
                                else
                                {
                                        V[i][j]=p3;
                                        T[i][j]=0;
                                }
                        }
                }

                //traceback
                calculatedProb = calculatedProb/seq1len;
                firstcounter = seq1len;
                secondcounter = size;
                lencounter1 = size;
                lencounter2 = size;
                while( firstcounter != 0 || secondcounter != 0)
                {

                        if( T[firstcounter][secondcounter] == 1)
                        {
                                firstalign[lencounter1-1] = sequence1[firstcounter-1];
                                secondalign[lencounter2-1] = temp[secondcounter-1];
                                firstcounter--;
                                secondcounter--;
                        }
                        else if(T[firstcounter][secondcounter] == 2 )
                        {
                                firstalign[lencounter1-1] = sequence1[firstcounter-1];
                                secondalign[lencounter2-1] = '-';
                                firstcounter--;
                        }
                        else
                {
                        firstalign[lencounter1-1] = '-';
                        secondalign[lencounter2-1] = temp[secondcounter-1];
                        secondcounter--;
                }
                lencounter1--;
                lencounter2--;
        }
        firstalign[size+1] = '\0';
        secondalign[size+1] = '\0';

        //among my fragments, calculate larger mean prob
        if( largestCalcProb <= calculatedProb )
        {
                tracker = (fragment-1) * size + startpos - seq1len;
                point = fragment;
                largestCalcProb = calculatedProb;
                strcpy(finalFirstAlign, firstalign);
                strcpy(finalSecondAlign, secondalign);

        }

    }  // while ends here

    printf("\nFragment number %d\n", point);
    printf("Position of alignment %d\n", tracker);
    printf("Largest Mean Posterior Prob # %f\n", largestCalcProb);

    //deallocate all memory space
    free(Zf[0]);
    free(Ze[0]);
    free(Zf[1]);
    free(Ze[1]);
    free(Zm_rev[0]);
    free(Zm_rev[1]);
    free(Zm);
    free(sequence1);
    free(sequence2);
    free(temp);

    for( i=0; i<(seq1len+1);i++)
    {
            free(V[i]);
    }

    for( i=0; i<(seq1len+1);i++)
    {
            free(T[i]);
    }

    // freeing memory space
    if( postptr)
            free(postptr);

    free(firstalign);
    free(secondalign);

    free(finalFirstAlign);
    free(finalSecondAlign);
    return(0);
}
```

REFERENCES

Bahr,A. et al. (2001) BAliBASE (Benchmark Alignment dataBASE): enhancements for repeats, transmembrane sequences and circular permutations. *Nucleic Acids Res*, **29**, 323-326.

Do,C.B. et al. (2005) ProbCons: Probabilistic consistency-based multiple sequence alignment. *Genome Res*, **15**, 330-340.

Durbin,R. et al. (1998) *Biological sequence analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge, United Kingdom.

Edgar,R.C. (2004) MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Res*, **32**, 1792-1797.

Karlin,S. and Altschul,S.F. (1990) Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proc. Natl. Acad. Sci. U.S.A*, **87**, 2264-2268.

Katoh,K. et al. (2005) MAFFT version 5: improvement in accuracy of multiple sequence alignment. *Nucleic Acids Research*, **33**, 511 -518.

Notredame,C. (2002) Recent progress in multiple sequence alignment: a survey. *Pharmacogenomics*, **3**, 131-144.

Roshan,U. et al. Searching for evolutionary distant RNA homologs within genomic sequences using partition function posterior probabilities. *BMC Bioinformatics*, **9**, 61-61.

Roshan,U. and Livesay,D.R. (2006) Probalign: multiple sequence alignment using partition function posterior probabilities. *Bioinformatics*, **22**, 2715 -2721.

Thompson,J.D. et al. (1994) CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res*, **22**, 4673-4680.

Thompson,J.D. et al. (1999) BAliBASE: a benchmark alignment database for the evaluation of multiple alignment programs. *Bioinformatics*, **15**, 87 -88.