EAST WEST UNIVERSITY

FALL - 2018

Project Report

Project Title: Sequence Alignment Problem

Project Number: 09

Course Code: CSE 245

Course Title: Algorithm

Section: 03

Submitted To:
Md. Shamsujjoha

Submitted By:
1. Gazi Mahfuzur Rahman, ID: 2014-3-60-027.
2. Umma Jahan, ID: 2017-1-60-053.
3. Md. Ridwan Sarker Turza, ID: 2017-2-60-135.

Date of Submission: **November 26, 2018.**

Sequence Alignment Algorithm

Gazi Mahfuzur Rahman, Umma Jahan and Md. Ridwan Sarker Turza

November 26, 2018

Abstract

In bioinformatics, a sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. Aligned sequences of nucleotide or amino acid residues are typically represented as rows within a matrix. Gaps are inserted between the residues so that identical or similar characters are aligned in successive columns. In this report, We discussed about Sequence alignments which are also used for non-biological sequences, such as calculating the edit distance cost between strings in a natural language or in financial data. It is a fundamental problem in Biological Science.

Keywords— complexity, dynamic programming, sequence alignment, computational biology.

1 Introduction

Dynamic programming algorithms are recursive algorithms modified to store intermediate results, which improves efficiency for certain problems like Sequence Alignment Problem. The technique of dynamic programming is theoretically applicable to any number of sequences; however, because it is computationally expensive in both time and memory, it is rarely used for more than three or four sequences in its most basic form. This method requires constructing the *n*-dimensional equivalent of the sequence matrix formed from two sequences, where *n* is the number of sequences in the query. Standard dynamic programming is first used on all pairs of query sequences and then the "alignment space" is filled in by considering possible matches or gaps at intermediate positions, eventually constructing an alignment essentially between each two-sequence alignment. Although this technique is computationally expensive, its guarantee of a global optimum solution is useful in cases where only a few sequences need to be aligned accurately. One method for reducing the computational demands of dynamic programming, which relies on the "sum of pairs" objective function, has been implemented in the MSA software package.

2 History of Dynamic Programming

Dynamic programming is both a mathematical optimization method and a computer programming method. The method was developed by Richard Bellman in the 1950s and has found applications in numerous fields, from aerospace engineering to economics. In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner. While some decision problems cannot be taken apart this way, decisions that span several points in time do often break apart recursively. Likewise, in computer science, if a problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have optimal substructure.

If sub-problems can be nested recursively inside larger problems, so that dynamic programming methods are applicable, then there is a relation between the value of the larger problem and the values of the sub-problems. In the optimization literature this relationship is called the Bellman equation.

3 Optimal Solution

In terms of mathematical optimization, dynamic programming usually refers to simplifying a decision by breaking it down into a sequence of decision steps over time. This is done by defining a sequence of **value functions** V_1, V_2, ..., V_n taking y as an argument representing the **state** of the system at times i from 1 to n. The definition of $V_n(y)$ is the value obtained in state y at the last time n. The values V_i at earlier times $i = n-1, n-2, ..., 2, 1$ can be found by working backwards, using a recursive relationship called the Bellman equation. For $i = 2, ..., n$, V_{i-1} at any state y is calculated from V_i by maximizing a simple function (usually the sum) of the gain from a decision at time $i-1$ and the function V_i at the new state of the system if this decision is made. Since V_i has already been calculated for the needed states, the above operation yields V_{i-1} for those states. Finally, V_1 at the initial state of the system is the value of the optimal solution. The optimal values of the decision variables can be recovered, one by one, by tracking back the calculations already performed.

4 Fibonacci Sequence

Here is a naïve implementation of a function finding the nth member of the Fibonacci sequence, based directly on the mathematical definition:

```
function fib(n)
if n <= 1 return n
return fib(n − 1) + fib(n − 2)
```

Notice that if we call, say, fib(5), we produce a call tree that calls the function on the same value many different times:

1. fib(5)
2. fib(4) + fib(3)
3. (fib(3) + fib(2)) + (fib(2) + fib(1))
4. ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
5. (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

In particular, fib(2) was calculated three times from scratch. In larger examples, many more values of fib, or *subproblems*, are recalculated, leading to an exponential time algorithm.

Now, suppose we have a simple map object, *m*, which maps each value of fib that has already been calculated to its result, and we modify our function to use it and update it. The resulting function requires only O(*n*) time instead of exponential time (but requires O(*n*) space):

```
var m := map(0 → 0, 1 → 1)
function fib(n)
if key n is not in map m
m[n] := fib(n − 1) + fib(n − 2)
return m[n]
```

This technique of saving values that have already been calculated is called *memoization*; this is the top-down approach, since we first break the problem into subproblems and then calculate and store values.

In the **bottom-up** approach, we calculate the smaller values of fib first, then build larger values from them. This method also uses O(*n*) time since it contains a loop that repeats n − 1 times, but it only takes constant (O(1)) space, in contrast to the top-down approach which requires O(*n*) space to store the map.

```
function fib(n)
if n = 0
```

```
return 0
else
var previousFib := 0, currentFib := 1
repeat n − 1 times // loop is skipped if n = 1
var newFib := previousFib + currentFib
previousFib := currentFib
currentFib  := newFib
return currentFib
```

In both examples, we only calculate fib(2) one time, and then use it to calculate both fib(4) and fib(3), instead of computing it every time either of them is evaluated.

Note that the above method actually takes $\Omega(n^2)$ time for large n because addition of two integers with $\Omega(n)$ bits each takes $\Omega(n)$ time. (The n^{th} fibonacci number has $\Omega(n)$ bits.) Also, there is a closed form for the Fibonacci sequence, known as Binet's formula, from which the n^{th} term can be computed in $O(n(\log n)^2)$ approximately time, which is more efficient than the above dynamic programming technique. However, the simple recurrence directly gives the matrix form that leads to an approximately $O(n(\log n)^2)$ algorithm by fast matrix exponentiation.

5 The Source code of Sequence Alignment Algorithm

```
#include <bits/stdc++.h>

using namespace std;

void getMinimumPenalty(string x, string y, int pxy, int pgap)

{      int i, j;

       int m = x.length();

       int n = y.length();

       int dp[n+m+1][n+m+1] = {0};

       for (i = 0; i <= (n+m); i++)

       {

              dp[i][0] = i * pgap;

              dp[0][i] = i * pgap;

       }

       for (i = 1; i <= m; i++)

       {

              for (j = 1; j <= n; j++)

              {

              if (x[i - 1] == y[j - 1])

              {
```

```
	dp[i][j] = dp[i - 1][j - 1];
	}
	else
		{
		dp[i][j] = min(dp[i - 1][j] + pgap ,
		dp[i][j - 1] + pgap );
			dp[i][j] = min(dp[i - 1][j - 1] + pxy ,  dp[i][j]);
		}
	}
}
int l = n + m;
i = m; j = n;
int xpos = l;
int ypos = l;

int xans[l+1], yans[l+1];
while ( !(i == 0 || j == 0))
{
	if (x[i - 1] == y[j - 1])
	{
		xans[xpos--] = (int)x[i - 1];
		yans[ypos--] = (int)y[j - 1];
		i--; j--;
	}
	else if (dp[i - 1][j - 1] + pxy == dp[i][j])
	{
		xans[xpos--] = (int)x[i - 1];
		yans[ypos--] = (int)y[j - 1];
		i--; j--;
	}
	else if (dp[i - 1][j] + pgap == dp[i][j])
	{
```

```
                xans[xpos--] = (int)x[i - 1];
                yans[ypos--] = (int)'_';
                i--;
        }
        else if (dp[i][j - 1] + pgap == dp[i][j])
        {
                xans[xpos--] = (int)'_';
                yans[ypos--] = (int)y[j - 1];
                j--;
        }
}
while (xpos > 0)
{
        if (i > 0) xans[xpos--] = (int)x[--i];
        else xans[xpos--] = (int)'_';
}
while (ypos > 0)
{
        if (j > 0) yans[ypos--] = (int)y[--j];
        else yans[ypos--] = (int)'_';
}
int id = 1;
for (i = l; i >= 1; i--)
{
        if ((char)yans[i] == '_' && (char)xans[i] == '_')
        {
                id = i + 1;
                break;
        }
}
      cout << "Minimum Penalty in aligning the Sequence = ";
```

```
	cout << dp[m][n] << "\n";
	cout << "The aligned sequence are :\n";
	for (i = id; i <= l; i++)
	{
		cout<<(char)xans[i];
	}
	cout << "\n";
	for (i = id; i <= l; i++)
	{
		cout << (char)yans[i];
	}
	return;
}
int main(){

	string gene1;
	string gene2;
	cout<<"Sequence 1: ";
	cin>>gene1;
	cout<<"Sequence 2: ";
	cin>>gene2;
	int misMatchPenalty = 3;
	int gapPenalty = 2;
	getMinimumPenalty(gene1, gene2,
		misMatchPenalty, gapPenalty);
	return 0;
}
```

6 Why we use Dynamic Programming

Dynamic programming is an algorithmic technique used commonly in sequence analysis. Dynamic programming is used when recursion could be used but would be inefficient because it would repeatedly solve the same subproblems. For example, consider the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, ... The first and second Fibonacci numbers are

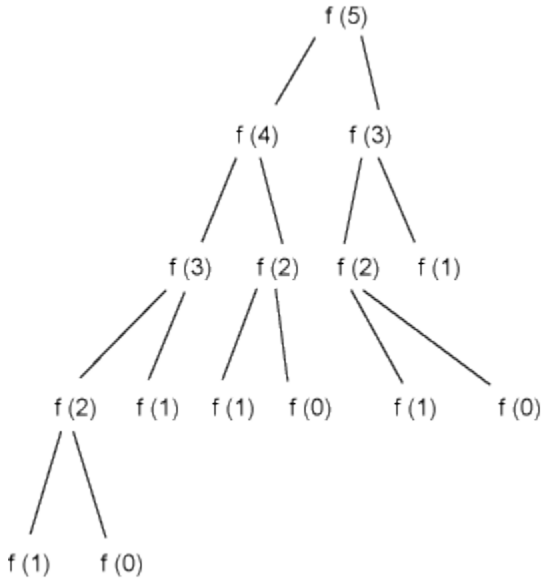
defined to be 0 and 1, respectively. The *n*th Fibonacci number is defined to be the sum of the two preceding Fibonacci numbers. So, you can calculate the *n*th Fibonacci number with the recursive function in Listing 1:

Listing 1. Recursive function for calculating *n*th Fibonacci number

```
      public int fibonacci1(int n) {
if (n == 0) {
return 0;
}
            else if (n == 1) {
return 1;
}
            else {
return fibonacci1(n - 1) + fibonacci1(n - 2);
}
      }
            }
```

But Listing 1's code is inefficient because it solves some of the same recursive subproblems repeatedly. For example, consider the computation of fibonacci1(5), represented in Figure 1:

Figure 1. Recursive computation of Fibonacci numbers



In Figure 1 you can see, for example, that fibonacci1(2) is computed three times. It would be much more efficient to build the Fibonacci numbers from the bottom up, as shown in Listing 2, rather than from the top down:

Listing 2. Building Fibonacci numbers from the bottom up

```
      public int fibonacci2(int n) {
int[] table = new int[n + 1];
for (int i = 0; i < table.length; i++) {
```

```
if (i == 0) {
table[i] = 0;
}
                    else if (i == 1) {
table[i] = 1;
}
                    else {
table[i] = table[i - 2] + table[i - 1];
}
}

return table[n];
      }
```

Listing 2 stores the intermediate results in a table so that you can reuse them, rather than throwing them away and computing them multiple times. It's true that storing the table is memory-inefficient because you use only two entries of the table at a time, but ignore that fact for now. I'm doing it this way to motivate your use of similar tables (although they will be two-dimensional) in this article's more complicated later examples. The point is that Listing 2's implementation is much more time-efficient than Listing 1's. Listing 2's implementation runs in *O(n)* time. I won't prove this, but the running time of Listing 1's naive, recursive implementation is exponential in *n*.

This is exactly how dynamic programming works. You take a problem that could be solved recursively from the top down and solve it iteratively from the bottom up instead. You store your intermediate results in a table for later use; otherwise, you would end up computing them repeatedly — an inefficient algorithm. But dynamic programming is usually applied to optimization problems like the rest of this article's examples, rather than to problems like the Fibonacci problem. The next example is a string algorithm, like those commonly used in computational biology.

We already know that; *Global sequence alignment* tries to find the best alignment between an entire sequence *S1* and another entire sequence *S2*. Consider these two DNA sequences:

- *S1* = GCCCTAGCG
- *S2* = GCGCAATG

If you award matches one point, penalize spaces by two points, and penalize mismatches by one point, the following is an optimal global alignment:

- *S1'* = GCCCTAGCG
- *S2'* = GCGC-AATG

A dash (-) denotes a space. There are five matches, one space in *S2'* (or, conversely, one insertion in *S1'*), and three mismatches. This yields a score of (5 * 1) + (1 * -2) + (3 * -1) = 0, which is the best you can do.

With *local sequence alignment*, you're not constrained to aligning the whole of both sequences; you can just use parts of each to obtain a maximum score. Using the same sequences *S1* and *S2* and the same scoring scheme, you obtain the following optimal local alignment *S1''* and *S2''*:

- *S1* = GCCCTAGCG
- *S1''* = GCG
- *S2''* = GCG
- *S2* = GCGCAATG

This local alignment doesn't happen to have any mismatches or spaces, although, in general, local alignments can have them. This local alignment has a score of (3 * 1) + (0 * -2) + (0 * -1) = 3. (The score of the best local alignment is greater than or equal to the score of the best global alignment, because a global alignment *is* a local alignment.)

The technique of dynamic programming can be applied to produce global alignments via the Needleman-Wunsch algorithm, and local alignments via the Smith-Waterman algorithm. In typical usage, protein alignments use a substitution matrix to assign scores to amino-acid matches or mismatches, and a gap penalty for matching an amino acid in one sequence to a gap in the other. DNA and RNA alignments may use a scoring matrix, but in practice often simply assign a positive match score, a negative mismatch score, and a negative gap penalty. (In standard dynamic programming, the score of each amino acid position is independent of the identity of its neighbors, and therefore base stacking effects are not taken into account. However, it is possible to account for such effects by modifying the algorithm.) A common extension to standard linear gap costs, is the usage of two different gap penalties for opening a gap and for extending a gap. Typically the former is much larger than the latter, e.g. -10 for gap open and -2 for gap extension. Thus, the number of gaps in an alignment is usually reduced and residues and gaps are kept together, which typically makes more biological sense. The Gotoh algorithm implements affine gap costs by using three matrices.

Dynamic programming can be useful in aligning nucleotide to protein sequences, a task complicated by the need to take into account frameshift mutations (usually insertions or deletions). The framesearch method produces a series of global or local pairwise alignments between a query nucleotide sequence and a search set of protein sequences, or vice versa. Its ability to evaluate frameshifts offset by an arbitrary number of nucleotides makes the method useful for sequences containing large numbers of indels, which can be very difficult to align with more efficient heuristic methods. In practice, the method requires large amounts of computing power or a system whose architecture is specialized for dynamic programming. The BLAST and EMBOSS suites provide basic tools for creating translated alignments (though some of these approaches take advantage of side-effects of sequence searching capabilities of the tools). More general methods are available from both commercial sources, such as *FrameSearch*, distributed as part of the Accelrys GCG package, and Open Source software such as Genewise.

The dynamic programming method is guaranteed to find an optimal alignment given a particular scoring function; however, identifying a good scoring function is often an empirical rather than a theoretical matter. Although dynamic programming is extensible to more than two sequences, it is prohibitively slow for large numbers of sequences or extremely long sequences.

7 Dynamic Programming Efficiency

Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. This idea is very insightful for solving bioinformatics problems. Aligning distantly related protein sequences is a long-standing problem in bioinformatics and a key for successful protein structure prediction. A fast and valid algorithm can benefit the whole process of biology research.

If we compare with the Time Complexity of some other Algorithms like Warshall, Greedy and Dijkstra, we can differentiate the efficiency of Dynamic Programming.

The time complexity of these Algorithms are given below:

Warshall:

FLOYD-WARSHALL(W)

1. n = W.rows
2. $D^{(0)} = W$
3. for k = 1 to n
4. Let $D^{(k)} = (d_{ij}^{(k)})$ be a new n x n matrix
5. for i = 1 to n
6. for j = 1 to n
7. $d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
8. return $D^{(n)}$

Complexity: $O(n^3)$

20

Greedy:

Greedy Strategy :

Analysis Summary of Algorithms:

Problem	Algorithm	Time Complexity	Space Complexity
Knapsack	Knapsack	O(nlogn)	O(n)
Job sequencing with deadlines	Sequenc1	$O(n^2)$	O(n)
	Sequenc2 or fast	**O(nlogn)**	O(n)
Optimal Merge Pattern	Huffman's	$O(n^2)$ using array O(nlonn) using Heap	Rec:O(logn) Iter: O(1)
Minimum Spanning Tree	Prim's	$O(n^2)$	O(n)
	Kruskal's	O(|E|log|E|)	O(n)
Shortest Path	Dijkstra's	$O(n^2)$	O(n)

Dijkstra:

Complexity Analysis

Minimum edge weight data structure	Time complexity (total)
adjacency matrix, searching	**O(V*V)**
binary heap and adjacency list	**O((V + E) log(V)) = O(E log(V))**
Fibonacci heap and adjacency list	**O(E + V log(V))**

Now compare with the Complexity of Dynamic Programming Algorithm:

Dynamic Programming:
Analysis Summary of Algorithms:

Problem	Algorithm	Time Complexity	Space Complexity
Binomial Coefficients	Pascal's Triangle	O(nk)	O(k)
Making (Amount = N, Deno = n)	Makechange	O(nN)	O(nN)
0-1-knapsack	0-1-knapsack	$O(2^n)$ - Recursion O(nm) - Table	O(n) - Recursion O(nm) - Table
Traveling Salesperson	tsp-dp	$O(n^2 2^n)$ or O(g(n)n!)	$O(n2^n)$
Optimal Binary Search Tree	OBST	$O(n^3)$ or $O(n^2)$	$O(n^2)$
Multistage Graph	Fgraph & Bgraph	**θ(|V| + |E|)**	**θ(n+k)**

8 Conclusion

Here, compare with some algorithms we can easily say that, the complexity of Dynamic Programming is lesser than other algorithms. So, it's efficient than others. This is why we are using this method to solve our problem.

This article has looked at three examples of problems that can be solved using dynamic programming. They all share these characteristics:

- The solution to each of them could be expressed as a recurrence relation.
- The naive implementation of this recurrence relation as a recursive method would have led to an inefficient solution involving multiple computations of subproblems.
- An optimal solution to the problem could be constructed from optimal solutions to subproblems of the original problem.

Dynamic programming is also used in matrix-chain multiplication, assembly-line scheduling, and computer chess programs. It's often needed to solve tough problems in programming contests. If we want to explore more than we can consult the book *Introduction to Algorithms* (see Related topics) for more details on when dynamic programming is applicable and how the correctness of dynamic programming algorithms is usually proved.

Dynamic programming is maybe the most important use of computer science in biology, but certainly not the only one. Bioinformatics and computational biology are interdisciplinary fields that are quickly becoming disciplines in themselves with academic programs dedicated to them. Many molecular biologists now know a little programming, and there's much interesting and important work to be done by programmers who can learn a little biology. If you want to learn more, see Related topics for pointers to potentially useful material.

9 Acknowledgments

Thanks to Md. Shamsujjoha Sir (Senior Lecturer, Department of Computer Science and Engineering, East West University, Dhaka, Bangladesh) for helping us to understand and complete this project and make us interested in computational biology and dynamic programming.

References

1. *Introduction to Algorithms* (2nd ed.) (Thomas H. Cormen et al., MIT Press, 2001): Explore dynamic programming and all other sorts of algorithms in depth. It's probably the best single-volume book on algorithms.
2. *Bioinformatics: Sequence and Genome Analysis* (2nd ed.) (David W. Mount, Cold Spring Harbor Laboratory Press, 2004): The best book the author has found so far for programmers wanting to learn biology. Each chapter has separate introductions for programmers and biologists.
3. *Bioinformatics Computing* (Bryan Bergeron, Prentice Hall, 2003): A guide to bioinformatics for biology students.
4. *Sequence alignment*, *Dynamic programming* and Bioinformatics from Wikipedia.
5. *Web services for bioinformatics* series (Mine Altunay, Daniel Colonnese, and Chetna Warade, developerWorks, May-June 2004): This three-part article series describes a framework for deploying bioinformatics applications as high-throughput Web services on the NC BioGrid.
6. BLAST: The BLAST project site.
7. A survey of sequence alignment algorithms for next-generation sequencing by Heng Li and Nils Homer. [https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2943993/]