Ruth Tilahun

CMSC 337

Lab 3

02/11/2024

Lab 3: Dynamic Programming

Introduction

One feature of spell checkers is their ability to suggest alternatives when an error is detected. In this Lab we will delve into one methodology that is used to determine these alternatives which is dynamic programming. This lab serves as a way to gain an understanding of dynamic programming. Dynamic programming is a programming technique where a problem is broken down into subproblems. The main goal of a dynamic programming solution is to store and reuse previously seen values to increase the efficiency of an algorithm. In other words, the idea is to store the results of subproblems, so that we do not have or recompute them when needed later.. According to Thomas Cormen, we can use dynamic programming when: (a) we are trying to find an optimal solution to a problem, (b) we can break an instance of the problem into instances of one or more subproblems, (c) we use solutions to the subproblem(s) to solve the original problem, and if we use a solution to a subproblem within an optimal solution to the original problem, then the subproblem solution we use must be optimal for the subproblem. (Cormen, 2013). In this lab, we will use a technique in dynamic programming known as tabulation where we store the results of the subproblems in a table and use these results to solve larger problems until we solve the entire problem.

Algorithm: String Transformation

We will use a string transformation algorithm Io suggest an alternative to a misspelled word. But first let's see how we can transform one string to another. Using a set of transformation operations namely *insert*, *delete*, *replace*, and *copy*, and the *cost* of each operation we can find a lowest-*cost* way to transform one string to another. Each operation comes with a *cost* which is a constant and depends on the type of operation. Using this algorithm, we can find a sequence of operations with a minimum total *cost*.

`Procedure COMPUTE-TRANSFORM-TABLES (X, Y, cC, cR, cD, cI)`

`Inputs:`

- `X and Y: two strings of length m and n, respectively.`
- `cC , cR, cD, cI : the` *`costs`* `of the` *`copy, replace, delete,`* `and` *`insert`* `operations, respectively.`

```
Output: Arrays cost[0..m, 0..n] and op[0..m, 0..n]. The value in cost[i, j]
is the minimum cost of transforming the prefix Xi into the prefix Yj , so
that cost[m ,n] is the minimum cost of transforming X into Y. The operation
in op[i, j] is the last operation performed when transforming Xi into Yj .

1. Let cost[0..m, 0..n] and op[0..m, 0..n] be new arrays.

2. Set cost[0, 0] to 0.

3. For i = 1 to m:

    A. Set cost[i, 0] to i * cD, and set op[i, 0] to delete xi.

4. For j = 1 to n:

    A. Set cost[0, j] to j * cI , and set op[0, j] to insert yj

5. For i = 1 to m:

    A. For j = 1 to n:

(Determine which of copy and replace applies, and set cost[i, j] and op[i, j]
according to which of the three applicable operations minimizes cost[i, j].)

      i. Set cost[i, j] and op[i, j] as follows:

            a. If xi and yj are the same, then set cost[i, j] to cost[i – 1,
            j – 1] + cC and set op[i, j] to copy xi .

            b. Otherwise (xi and yj differ), set costŒi; j to cost[i, j] to
            cost[i – 1, j – 1] + cR and set op[i, j] to replace xi by yj .

      ii. If cost[i - 1, j] + cD < cost[i, j]; j , then set cost[i, j] to
      cost[i - 1, j] + cD and set op[i, j] to delete xi.

      iii. If cost[i, j - 1] + cI < cost[i, j], then set cost[i, j] to
      cost[i, j - 1] + cI and set op[i, j] to insert ji.

6. Return the arrays cost and op.
```

In this algorithm, we fill a two 2-dimensional arrays or tables of size m by n row by row, where m is the length of string X and n is the length of string Y. The *cost* table with *cost*[i, j] will hold the minimum *cost* to transform Xi to Yj. For example, if X = abebe and Y = berbere, and we have *cost*[4,3] = 2, the minimum total *cost* of transforming prefix X4(abeb) to prefix Y3(ber) is 2. The op table with *op*[4, 3] holds the last operation done to achieve this transformation. This *cost* is dependent on previous values. Specifically we will look at the *cost*s of cells directly above(*cost*[i - 1, j]), to the left(*cost*[i, j – 1]) and above and to the left(*cost*[i -1, j – 1]). If our last operation is *delete*, meaning we *delete*d the ith character of X to transform Xi to Yj, then we must have transformed Xi-1 to Yj, thus *cost*[i, j] = *cost*[i – 1, j] + cD. Similarly, if the last operation was a *insert*, *cost*[i, j] = *cost*[i , j - 1] + cI. *Replace* and *copy* on Xi and Yj require that we use the both Xi-1 and Yj-1 because the conversion involves both i and j. Thus ,*cost*[i, j] = *cost*[i - 1, j - 1] + cR for *replace* and *cost*[i, j] = *cost*[i - 1 , j - 1] + cC for *copy*. To get a *cost* with the lowest value, first we consider the type of operation needed. If Xi is different from Yj, then the *replace* operation is needed. And if Xi is the same as Yj , the *copy* operation is needed. If *copy* is operation needed, we will take the minimum value of *cost*[i ,j] that can be obtained from the *copy*, *insert* or *delete* operations. Likewise, if the *replace* operation is needed we will choose the operation that minimizes *cost*[i, j] from one of the *replace*, *insert* or *delete* operations. To obtain these values that we are going to compare, we will use the *cost*s of transformations that we have previously stored – which is the application of dynamic programming in this algorithm.

This approach of programming gives an optimal solution to the overall problem by using the optimal solutions of the subproblems. To transform m characters into n, we know we have transformed m – 1 characters first with the lowest-*cost* solution. Similarly, to transform m – 1 characters we have transformed m – 2 characters first with the lowest-*cost* solution and so on. All cells are filled by considering the lowest *cost* operation, therefore the final *cost* will be the optimal solution.

This algorithm fills the tables in constant time. Because each of the tables contains (m + 1) * (n + 1) entries, COMPUTE-TRANSFORM-TABLES runs in $\Theta(m*n)$ time (Cormen, 2013).

Appendix:

```
import java.util.*;
import java.io.BufferedReader;
import java.io.FileReader;
public class costs {
    private static Integer[][] cost;
    private static String[][] op;
    public static void computeTransformTables(String X, String Y, int cC, int cR,
int cD, int cI ) {

        int m = X.length();
        int n = Y.length();

        cost = new Integer[m + 1][n + 1];
        op = new String[m + 1][n + 1];

        // Step 2
        cost[0][0] = 0;
        op[0][0] = "";

        // Step 3
        for (int i = 1; i <= m; i++) {
            cost[i][0] = i * cD;
            op[i][0] = "d"+ X.charAt(i - 1);
        }

        // Step 4
        for (int j = 1; j <= n; j++) {
            cost[0][j] = j * cI;
            op[0][j] = "i" + Y.charAt(j - 1);
        }

        // Step 5
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                // Determine which of copy and replace applies
                if (X.charAt(i - 1) == Y.charAt(j - 1)) {
                    cost[i][j] = cost[i - 1][j - 1] + cC;
                    op[i][j] = "c"+ X.charAt(i - 1);
                } else {
                    cR = replacementCost(X.charAt(i - 1), Y.charAt(j - 1));
                    cost[i][j] = cost[i - 1][j - 1] + cR;
                    op[i][j] = "r"+ X.charAt(i - 1) + "->" + Y.charAt(j - 1);
                }
```

```java
            // Check for delete and insert operations
            if (cost[i - 1][j] + cD < cost[i][j]) {
                cost[i][j] = cost[i - 1][j] + cD;
                op[i][j] = "d" + X.charAt(i - 1);
            }

            if (cost[i][j - 1] + cI < cost[i][j]) {
                cost[i][j] = cost[i][j - 1] + cI;
                op[i][j] = "i" + Y.charAt(j - 1);
            }
        }
    }

    // Step 6
    // Return the arrays cost and op
}

public static String assembleTransformation(String[][] op, int i, int j) {
    if (i == 0 && j == 0) {
        // Base case: both i and j are 0
        return "";
    } else {
        // At least one of i and j is positive
        if (op[i][j].charAt(0) == ('c') || op[i][j].charAt(0) == ('r')) {
            // If op[i][j] is a copy or replace operation
            return assembleTransformation(op, i - 1, j - 1) + op[i][j] + " ";
        } else if (op[i][j].charAt(0) == ('d')) {
            // If op[i][j] is a delete operation
            return assembleTransformation(op, i - 1, j) + op[i][j] + " ";
        } else {
            // If op[i][j] is an insert operation
            return assembleTransformation(op, i, j - 1) + op[i][j] + " ";
        }
    }
}
public static <E> void print(E[][] arr, int row, int col) {
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            System.out.print(arr[i][j] + "\t");
        }
        System.out.println();
    }
    System.out.println("\n\n");
```

```
    }

    public static int replacementCost(char chA, char chB) {
        int iA = (int) chA;
        int iB = (int) chB;
        return (int) (Math.sqrt(Math.abs(iA - iB)) + 1.0);
    }
     public static ArrayList<String> readFile(String fileName) {
            ArrayList<String> words = new ArrayList<String>();
            try (BufferedReader br = new BufferedReader(new
FileReader(fileName))) {
                String line;
                while ((line = br.readLine()) != null) {
                    words.add(line.trim());
                }
                return words;
            } catch (Exception ee) {
                System.err.println(ee);
                return words;
            }
        }

    public static void main(String args[]) {

         int cC = 0;
        int cR = 0;
        int cD = 3;
        int cI = 3;
        ArrayList<String> words = readFile("words");

        String X = "";
        String Y = "";

        int minCost = 0;
        for (int i = 0; i < args.length; i++) {
            ArrayList<WordCost> costs = new ArrayList<WordCost>();
            for (int j = 0; j < words.size(); j++){
                X = args[i];
                Y = words.get(j);
                computeTransformTables(X, Y, cC, cR, cD, cI);
                costs.add(new WordCost(Y, cost[X.length()][Y.length()]));
            }
            Collections.sort(costs);
            System.out.println(args[i] + " -> " + costs.get(0));
        }
```

```
    }
}
class WordCost implements Comparable<WordCost>{
    private int cost;
    private String word;

    public WordCost(String w, int c) {
        this.word = w;
        this.cost = c;
    }
    public String getWord() {
        return this.word;
    }
    public int getCost() {
        return this.cost;
    }
    public String toString() {
        return word + " with cost " + this.cost;
    }
    public int compareTo(WordCost data) {
         if (getCost() > data.getCost()) {
            return 1;
        } else if (getCost() == data.getCost()) {
            return 0;
        } else
            return -1;
    }
}
```

References

Cormen, Thomas H. *Algorithms Unlocked*. MIT Press, 2013.

OpenAI. (2024). *ChatGPT* (3.5) [Large language model]. https://chat.openai.com

Lec5.Pdf. https://people.seas.harvard.edu/~cs125/fall16/lec5.pdf. Accessed 7 Feb. 2024.