İskender Akyüz
150150150

# BLG336E – Homework III
## Report

- The program can be compiled by using this command line:
  $ g++ -std=c++11 main.cpp -o main
- In order to run the program, you should use this command line:
  $ ./main data.txt

## 1) Part 1

For the solution of part 1, I have used the KnapSack algorithm.

```
int KnapSackRec(int W, vector<TestSuite> testSuites, int i, int** dp) {
    if (i < 0)      // base condition
        return 0;
    if (dp[i][W] != -1)
        return dp[i][W];

    int weight = testSuites[i].getRunningTime();
    int value = testSuites[i].getBugsDetected();

    if (weight > W) {
        dp[i][W] = KnapSackRec(W, testSuites, i - 1, dp);
        return dp[i][W];
    }
    else {
        dp[i][W] = max(value + KnapSackRec(W - weight, testSuites, i - 1, dp), KnapSackRec(W, testSuites, i - 1, dp));
        return dp[i][W];
    }
}
```

The mathematical representation of the optimization function I have used:

$$OPT(w,i) = \begin{cases} 0 & if\ i = 0, \\ OPT(w, i-1) & if\ w_i > w, \\ \max\big(v_i + OPT(w - w_i, i-1), OPT(w, i-1)\big) & otherwise. \end{cases}$$

**Complexity Analysis:**

KnapSackRec function divides the problem into sub-problems to solve it efficiently. For each (w,i), (w, i-1) and (w-$w_i$, i-1) are also called. That means the weight can be a value between 1 and w and that results in n*w sub-problems.

$$O(n) = O(n*w)$$

- **Does your algorithm work if the running times of the test suites are given as real numbers instead of discrete values?**
  For the way I implemented the algorithm, it would not work because the columns in the 2D matrix represents the weight value. That's why the weight values, the running times in our case, should be discrete values.
  In order to solve this issue, we can convert the weight to integer values by multiplying them by multiplies of 10. Of course, this solution will increase the memory usage a lot. Another solution may be to round the weight values then apply the KnapSack algorithm. However, that will definitely decrease the precision of the solution.

**2) Part 2**

For the solution of part 2, I have used the Levenshtein Distance algorithm.

```
int LevensteinDistance(const vector<int> source, const vector<int> target, unsign
ed int insertCost, unsigned int deleteCost, unsigned int replaceCost) {
    if (source.size() > target.size())
        return LevensteinDistance(target, source, deleteCost, insertCost, replace
Cost);

    int min_size = source.size(), max_size = target.size();
    vector<int> levDist(min_size + 1);

    levDist[0] = 0;
    for (int i = 1; i <= min_size; ++i)
        levDist[i] = levDist[i - 1] + deleteCost;

    for (int j = 1; j <= max_size; ++j) {
        int prevDiagonal = levDist[0], prevDiagonalSave;
        levDist[0] += insertCost;

        for (int i = 1; i <= min_size; ++i) {
            prevDiagonalSave = levDist[i];
            if (source[i - 1] == target[j - 1])
                levDist[i] = prevDiagonal;
            else
                levDist[i] = min(min(levDist[i - 1] + deleteCost, levDist[i] + in
sertCost), prevDiagonal + replaceCost);
```

```
            prevDiagonal = prevDiagonalSave;
        }
    }

    return levDist[min_size];
}
```

The mathematical representation of the optimization function I have used:

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j) & if\ (\min(i,j) = 0, \\ min\begin{cases} lev_{a,b}(i-1,j) + deleteCost \\ lev_{a,b}(i,j-1) + insertCost \\ lev_{a,b}(i-1,j-1) + replaceCost \end{cases} & otherwise. \end{cases}$$

The default values for deleteCost, insertCost, and replaceCost in my algorithm are all 1's.

**Time Complexity:**

| | Statement | Steps | frequency | | Total steps | |
|---|---|---|---|---|---|---|
| | | | If-true | If-false | If-true | If-false |
| 1 | int LevensteinDistance(const vector<int> source, const vector<int> target, unsigned int insertCost, unsigned int deleteCost, unsigned int replaceCost) { int i, int** dp) { | | | | | |
| 2 | if (source.size() > target.size()) | 1 | 1 | 1 | 1 | 1 |
| 3 | return LevensteinDistance(target, source, deleteCost, insertCost, replaceCost); | x | 1 | 0 | x | 0 |
| 4 | int min_size = source.size(), max_size = target.size(); vector<int> levDist(min_size + 1); levDist[0] = 0; | 4 | 1 | 1 | 4 | 4 |
| 5 | for (int i = 1; i <= min_size; ++i) | 1 | m+1 | m+1 | m+1 | m+1 |
| 6 | levDist[i] = levDist[i - 1] + deleteCost; | 1 | m | m | m | m |
| 7 | for (int j = 1; j <= max_size; ++j) { | 1 | n+1 | n+1 | n+1 | n+1 |
| 8 | int prevDiagonal = levDist[0], prevDiagonalSave; levDist[0] += insertCost; | 2 | n | n | 2n | 2n |
| 9 | for (int i = 1; i <= min_size; ++i) { | 1 | n(m+1) | n(m+1) | n(m+1) | n(m+1) |
| 10 | prevDiagonalSave = levDist[i]; | 1 | n*m | n*m | n*m | n*m |
| 10 | if (source[i - 1] == target[j - 1]) | 1 | n*m | n*m | n*m | n*m |
| 10 | levDist[i] = prevDiagonal; | 1 | n*m | 0 | n*m | 0 |
| 10 | else | | | | | |
| 10 | levDist[i] = min(min(levDist[i - 1] + deleteCost, levDist[i] + insertCost), prevDiagonal + replaceCost); | 1 | 0 | n*m | 0 | n*m |
| 11 | prevDiagonal = prevDiagonalSave; | 1 | n*m | n*m | n*m | n*m |
| 12 | } | | | | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| 13 |     } | | | | | |
| 14 |     return levDist[min_size]; | 1 | 1 | 1 | 1 | 1 |
| 15 | } | | | | | |
| Total: | | | | | 5nm+2m+3n+8 | |

n: The length of the shortest sequence, m: The length of the longest sequence

$$O(n,m) = O(n*m)$$

The algorithm’s running time depends on the length of the sequences. In our case, the sequences are always equal to each other since we are comparing the test cases from the same test suite. Thus, the complexity becomes $O(n) = O(n^2)$ for our case.