Instituto Politécnico Nacional

Escuela Superior de Cómputo

Evolutionary Computing

Session 2
"Dynamic Programming"

Student: Naranjo Ferrara Guillermo

Teacher: Rosas Trigueros Jorge Luis

Lab session date: 11 February 2020

Delivery date: 19 February 2020

1 Theoretical Framework

1.1 Dynamic Programming

Dynamic Programming is mainly an optimization method.The idea is to simply store the results of sub-problems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.[1]

This method takes advantage from two characteristics of the problem to solve:

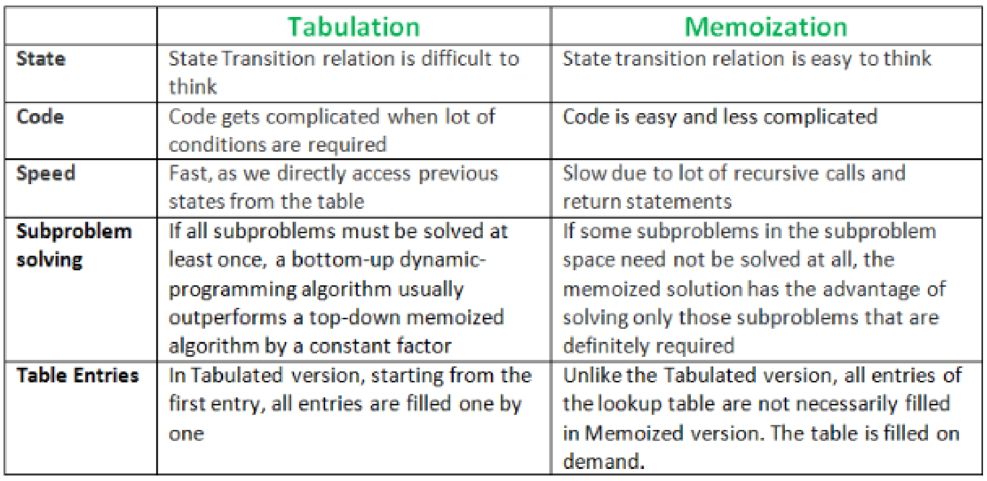
- **Optimal structure**. Optimal solutions to sub-problems can be used to construct the solution to the whole problem.[2]
- **Overlapped sub-problems**. A problem has overlapping sub-problems if a particular sub-problem can be used in the solving of more than one upper level problems. So that, recalculation at common sub-problems can be avoided by storing and reusing the calculated solutions.[2]

1.1.1 Tabulation and Memoization

There are two different ways to store the values so that the values of a sub-problem can be reused:

- **Tabulation** Follows a Bottom-Up approach, in which, as the name says, we start from the base case or state of the problem in order to reach our desired destination state.
- **Memoization** Follows a Top-Down approach, in which we start from our desired state and compute it's answer taking in count the values of states that can reach the destination state, till we reach the bottom most base state.

Here I show a table from [3] showing the differences between the two approaches for storing results in a DP solution



	Tabulation	Memoization
State	State Transition relation is difficult to think	State transition relation is easy to think
Code	Code gets complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous states from the table	Slow due to lot of recursive calls and return statements
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required
Table Entries	In Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. The table is filled on demand.

Table 1: Comparison between tabulation and memoization approaches for storing solutions of sub-problems of a DP problem.

1.1.2 Popular problems solved whit DP

Let's check three popular problems that can be solved with DP, the knapsack problem, the CMP problem and the Levenshtein Distance.

Knapsack problem

Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.[4]

There are some variants of this problem like the 0-1 where we choose to put or not only one of each item into the knapsack. Other variant is where we can choose more than once the same item. Another one is the fractional knapsack.

Minimum number of coins (CMP)

Given a number of denominations for coins we need to find the minimum number of coins required to give the change specified. We consider that the amount of coins we have is infinite.

As the knapsack problem, this is a typical problem for DP solution where we can see the two characteristics required for a problem to be solved with this method[5]

- **Optimal substructure**. To count the total number of solutions, we can divide all set solutions into two sets: 1)The ones that do not contain the n^{th} coin. 2) The ones that contain at least one valued coin.
- **Overlapping sub-problems**. To see this let's show an example. Note that in the next three the function $C(1,3)$ it's called 2 times.

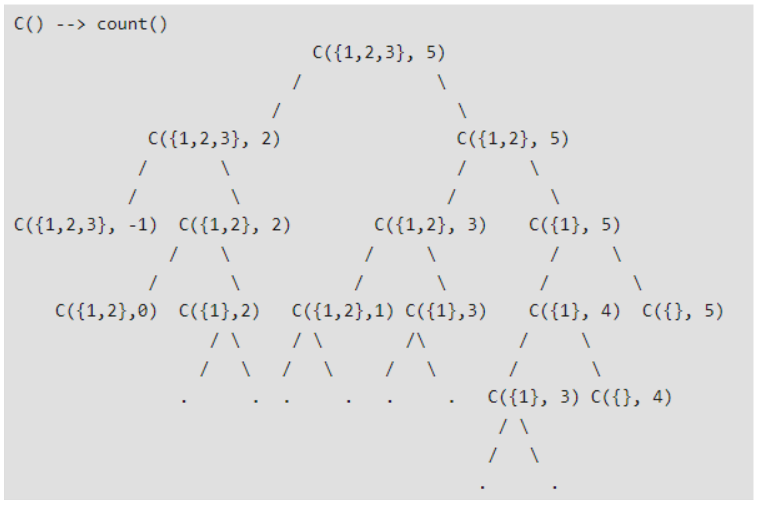


Figure 1: Recursion three of a CMP.[5]

Levenshtein Distance This string metric measures the difference between two sequences, i.e. the minimum number of single-character edits (insertions, deletions or substitutions) required to change one sequence into the other.[2] The smaller the Levenshtein distance, the more similar the strings are.

2 Material and Equipment

In order to this practice we need:

- A working computer.
- Any version (preferentially the last) of python installed in the computer.

3 Development

3.1 Dynamic Programming Algorithms

3.1.1 0/1 Knapsack problem

Write an algorithm for the 0/1 Knapsack problem. Prove it works correctly with the next parameters:
$C = 11$ $W = \{3, 4, 5, 9, 4\}$ $V = \{3, 4, 4, 10, 4\}$

In contrast whit the implementation of the solution for this problem developed the last session, because this time we're using DP and not Greedy, for this variant of the problem we guarantee to obtain the optimal solution.

Because I programmed this solution before the session, I used the same method to obtain the parameters of the problem as in the Greedy solution, and because I used it as a basis, I also transferred the same mistake of using two separate lists for values and weights, instead of using an only list of pair values.

Proceeding with the DP algorithm the first thing to do is to create the table, after that I used a for loop to resolve the trivial case in which we have no items to add. Then I used two nested for loops to fill the table depending on the case aroused:

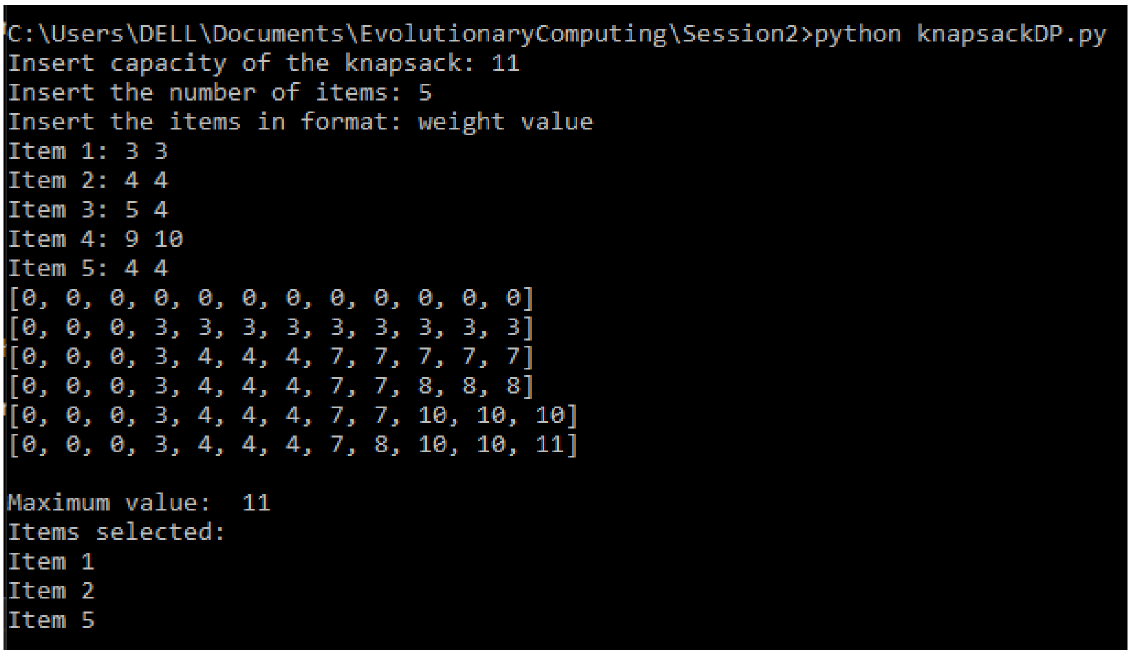
- If the weight of the item is bigger than the corresponding capacity, the value of the cell above the currently evaluating is transferred to it.
- In the contrary case, we take the max value between the value in the cell above and the value in the cell in which this object hasn't been packed yet plus the value of the new item packed to it.

Now the only thing left is to obtain the items selected to maximize the value, for this I use a while-loop.

The first thing I thought of was decreasing the value of the indexes one by one depending on the case, but for the conditions used I realized there were adversary cases and It would be awful continuing through that path. So I decided to use an approach more likely the one we used in class.

For this I used a variable that stores the maximum value to that point (initializing with the response to the problem) and when it comes to 0, breaks the loop. Now, in every iteration I compare this value with the value in the cell (starting from the last one). If the value differs I go one row up, If it doesn't I compare the final value to the value in the left cell. If it differs I decrease the value of the variable with the weight of the object and add this item to the list of selected items, whenever this last condition is true or false I go one column left.

Finally I order the array of selected items and show the maximum value of the knapsack to the user and the items selected for this.



```
C:\Users\DELL\Documents\EvolutionaryComputing\Session2>python knapsackDP.py
Insert capacity of the knapsack: 11
Insert the number of items: 5
Insert the items in format: weight value
Item 1: 3 3
Item 2: 4 4
Item 3: 5 4
Item 4: 9 10
Item 5: 4 4
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 3, 3, 3, 3, 3, 3, 3, 3, 3]
[0, 0, 0, 3, 4, 4, 4, 7, 7, 7, 7, 7]
[0, 0, 0, 3, 4, 4, 4, 7, 7, 8, 8, 8]
[0, 0, 0, 3, 4, 4, 4, 7, 7, 10, 10, 10]
[0, 0, 0, 3, 4, 4, 4, 7, 8, 10, 10, 11]

Maximum value:  11
Items selected:
Item 1
Item 2
Item 5
```

Figure 2: Working program for the knapsack problem.

3.1.2 Minimum number of coins (CMP) problem

Write an algorithm for the CMP problem. Prove it works correctly with the next parameters:
$T = 9$ $D = \{5, 2, 1\}$

As in the knapsack problem, I used the program of the last session as a basis, so I get the parameters in the same way.

For the DP solution, again, the first thing to do was to create the table and fill it's values for the trivial case (all the first column) with a for-loop.

To fill all the other cells I used two nested for-loops. Inside them there was a condition to be evaluated:

- If the value of the denomination was greater to the column in which I'm positioned, the cell evaluating takes the value of the left cell.

- In other case it's chosen the minimum value between the left cell and the cell that is in the row above and in the column resulting of subtracting the value of the actual denomination evaluating to the actual column and adding one to the value in the cell.

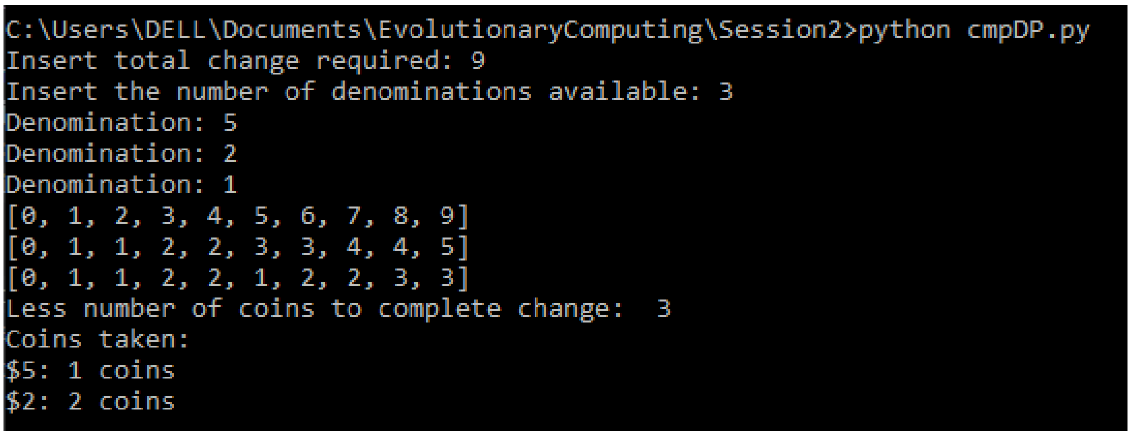
Now, calculate the number of coins taken of each denomination, was a little more complicated that knowing what items were selected in the knapsack problem. The advantage in this case is that I programmed the correct solution to this at first try.

First I think of using a list, but taking in consideration that I can take more than 1 coin of each denomination, instead I used a dictionary in which the keys were the denominations.

I used a while-loop that breaks until the number of coins (which initially takes the value of the response) is 0. Inside the loop I use a condition to know if a coin of certain value was taken.

If the value in the current cell (starting from the final position in the table) is equals to the value in the cell above, I move one row above. Otherwise, if the denomination hasn't been included into the dictionary, I added it with a value of 1, if it already is into the dictionary I update the item value by adding 1 to the current value.

Finally I show to the user the least number of coins needed to give the change, as well as the number of coins of each denomination required.



```
C:\Users\DELL\Documents\EvolutionaryComputing\Session2>python cmpDP.py
Insert total change required: 9
Insert the number of denominations available: 3
Denomination: 5
Denomination: 2
Denomination: 1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 1, 2, 2, 3, 3, 4, 4, 5]
[0, 1, 1, 2, 2, 1, 2, 2, 3, 3]
Less number of coins to complete change:  3
Coins taken:
$5: 1 coins
$2: 2 coins
```

Figure 3: Working program for the CMP.

3.1.3 Levenshtein Distance

Write an algorithm for the Levenshtein Distance problem. Prove it works correctly with the next examples:

- $str1 =$Nature $str2 =$Capture
- $str1 =$computing $str2 =$school

First I create the table with dimensions $(len(str1) + 1)x(len(str2) + 1)$, because of the consideration of the empty string. After that, with two independent for-loops I fill the trivial cases for the first column and row in the table.

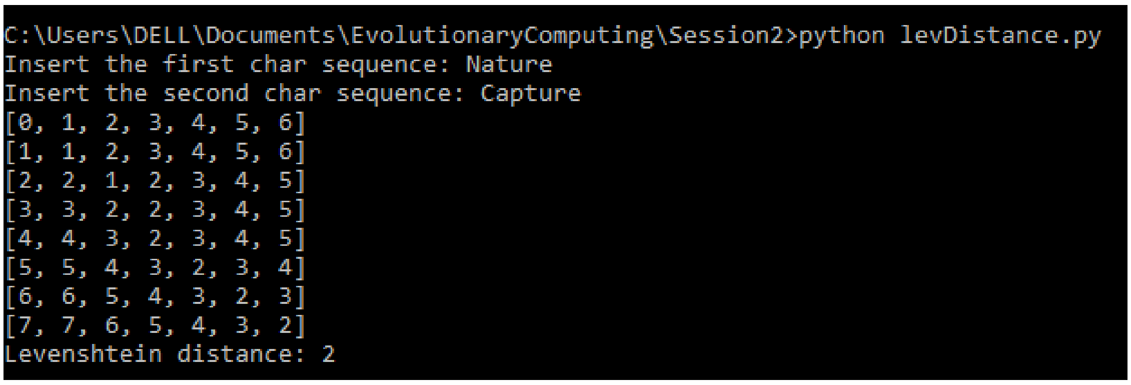
Then with two nested for-loops I fill the rest of the cells in the table.

A variable that indicates if the letters in the indexes of the strings currently evaluating are the same is always initialized to 1 and changed to 0 if it's not true; then I calculate the minimum value between the next three:

- The value of the cell above plus 1.
- The value of the cell at the left plus 1.
- The value of the cell above at the left plus the value of the flag always initialized in 1.

Then, if the minimum value is equals to 0, the cell will take the maximum value between the index that is being evaluated in the string 1, and the one being evaluated in the string 2. If the minimum value differs from 0 the cell value is the value of the minimum prior calculated.

Finally is shown to the user the Levenshtein Distance.



```
C:\Users\DELL\Documents\EvolutionaryComputing\Session2>python levDistance.py
Insert the first char sequence: Nature
Insert the second char sequence: Capture
[0, 1, 2, 3, 4, 5, 6]
[1, 1, 2, 3, 4, 5, 6]
[2, 2, 1, 2, 3, 4, 5]
[3, 3, 2, 2, 3, 4, 5]
[4, 4, 3, 2, 3, 4, 5]
[5, 5, 4, 3, 2, 3, 4]
[6, 6, 5, 4, 3, 2, 3]
[7, 7, 6, 5, 4, 3, 2]
Levenshtein distance: 2
```

Figure 4: Working program for the Levenshtein Distance problem.

4 Conclusion

This practice taught me the value of analyzing a problem and it's structure in order to apply some method to create a more efficient algorithm. In this case we use DP solutions to reuse the sub-problems solutions calculated to solve others.

For my experience working with this method, for me is easier to see the DP solution with the technique of memoization, using recursion, so I think this practice because of the use of tabulation suited me fine in order to improve my abilities and capacity to see DP solutions without using recursion.

5 Bibliography

[1]GeeksforGeeks. Dynamic Programming. [Online]. Available: https://www.geeksforgeeks.org/dynamic-programming/concepts

[2]"Evolutionary Computing", class notes, Academy of Computer Science, ESCOM-IPN, Primavera 2020.

[3]K. Nitish. Tabulation vs Memoization. [Online]. Available: https://www.geeksforgeeks.org/tabulation-vs-memoization/

[4]GeeksforGeeks. Fractional knapsack problem. [Online]. Available: https://www.geeksforgeeks.org/fractional-knapsack-problem/

[5]GeeksforGeeks. Coin Change — DP-7. [Online]. Available: https://www.geeksforgeeks.org/coin-change-dp-7/