Instituto Politécnico Nacional

Escuela Superior de Cómputo

Evolutionary Computing

Laboratory session #02: Introduction to Dynamic Programming

Student: Vargas Romero Erick Efrain

Professor: Rosas Trigueros Jorge Luis

Practice completion date: October 27, 2021

Report delivery date: September 24, 2021

1 Theoretical framework

From my point of view a phrase that describes in a simple way how dynamic programming works is:

Who does not know its history is condemned to repeat it

- Napoleón Bonaparte

1.1 Dynamic Programming

Dynamic programming is an optimization approach that transforms a complex problem into a sequence of simpler problems; its essential characteristic is the multistage nature of the optimization procedure. More so than the optimization techniques described previously, dynamic programming provides a general framework for analyzing many problem types. Within this framework a variety of optimization techniques can be employed to solve particular aspects of a more general formulation. Usually creativity is required to solve particular aspects of a more general formulation. Usually creativity is required before we can recognize that a particular problem can becast effectively as a dynamic program; and often subtle insights are necessary to restructure the formulation so that it can be solved effectively. [1]

1.1.1 Formalizing the dynamic programming approach

Dynamic programming problems have three important characteristics:

- **Stages:** The essential feature of the dynamic programming approach is the structuring of optimization problems into multiple *stages*, which are solved sequentially one stage at a time. Although each one-stage problem is solved as an ordinary optimization problem, its solution helps to define the characteristics of the next one-stage problem in sequence. The stages represent different time periods in the problems planning horizon, although sometimes there are problems that do not have implications but more difficult to recognize as a dynamic programming problem [2].

- **States:** The specification of the states of the system is perhaps the most critical design parameter of the dynamic-programming model. There are no set rules for doing this. In fact, for the most part, this is an art often requiring creativity and subtle insight about the problem being studied [2]. The essential properties that should motivate the selection states are:
  - States should convey enough information to make future decisions without regard to how process reached the current state.
  - The number of state variables should be small, since computational effor associated with the dynamic programming approach is prohibitely expensive when there are more than two, or possible three, state variables involved in the model formulation.
- **Recursive optimization:** Finally the general characteristic of the dynamic programming approach is the development of a recursive optimization procedure, which builds a solution overall *n-stage* problem by first solving a one stage problem and sequentially including one stage at a time and solving one stage problems until the overall optimum has been found. This procedure can be based on a backward induction process where the first stage to be analyzed is the final stage of the problem and problems are solved moving back one stage at a time until all stages are included [2].

  Dynamic programming algorithms are best developed in two distict stages [2]:

  1. Formulate the problem recusively: Write down a recursive formula or algorithm for the whole problem in terms of the answers to smaller subproblems. This is the hardest part. A complete recursive formulation has two parts:
     (a) **Specification:** Describe the problem that you want to solve recursively, in coherent and precise English-not how to solve that problem, but what problem are you trying to solve. Whithout this specification, it is impossible, even in principle, to determine wheter your solution is correct.
     (b) **Solution:** Give a clear recursive formula or algorithm fot the whole problem in terms of the answers to smaller instances of exactly the same problem
  2. Build solutions to your recurrence from the botton up: Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution, by considering intermediate subproblems in the correct order. This stage can be broken down into several smaller, relatively mechanical steps:
     - Identify subproblems.
     - Choose a memoization data structure.
     - Identify dependencies.
     - Find a good evaluation order.
     - Analyze space and running time.
     - Write down the algorithm.

For the present document, we will see three different cases when we can use dynamic programming to solve certain problems.

1.1.2 Knapsack problem

Suppose we are planning a hiking trip; and we are, therefore, interested in filling a knapsack with items that are considered necessary for the trip. There are n different item types that are deemed desirable; these could include bottle of water, apple, orange, sadwich, and sofort. Each item has a couple of attributes, namely a weight (or volume) and a value that quantifies the level of importance associated with each unit of that type item. Since the knapsack has a limited weight (or volume) capacity, the problem of interest is to figure out how to load the knapsack with a combination of units of the specified types of items that yields the greatest total value [3]
Formally, suppose we are given the following parameters:

- w_k = the weight of each $k-th$ item, for $k = 1,2,3,...,n$
- r_k = the value associated with each $k-th$ item, for $k = 1,2,3,...,n$
- c = the weight capacity of the knapsack

Then, our problem can be formulated as:

- Maximize
$$\sum_{k=1}^{n} r_k x_k \tag{1.1}$$
- Subject to:
$$\sum_{k=1}^{n} w_k x_w \leq c \tag{1.2}$$

Where $x_1, x_2, ..., x_n$ are non negative integer-valued decision variables, defined by x_k = the number of $k-th$ item that are loaded into the knapsack.

1.1.3 Change making problem

The change making problem is the problem of representing a given value with the fewest coins possible from a given set of coin denominations. Unboundedly many coins of each denomination are available. Formally, given a finite system $c_1 < c_2 < ... < c_m = n$ of positive integers (the coins) and a positive integer x, we wish to determine non negative integer coefficients $x_i, 1 \leq i \leq m$, so as to minimize [4].

$$\sum_{i=1}^{m} x_i \tag{1.3}$$

Subject to:

$$x = \sum_{i=1}^{m} x_i c_i \tag{1.4}$$

The sequence of coefficients $x_1, x_2, ..., x_m$ is called a representation of x. The quantity (1.3) that we wish to minimize is called the size of the representation is optimal if it is of minimum size. If $x_i > 0$, then we say that the coin c_i is used in the presentation.

1.1.4 Levanshtein distance (Edit distance problem)

The Levanshtein distance or edit distance problem between two *strings* is the number of insertions, deletions and substitutions needed to transform one string into other. This distance is of fundamental importance in several fields such as computational biology and text processing/searching and consequently, problems involving edit distance were studied extensively.
The basic problem is to compute the edit distance between two strings of length n over some alphabet [5]. Formally, we can represent this problem as following:

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & if \quad min(i,j)=0, \\ min \begin{cases} lev_{a,b}(i-1,j)+1 & if \quad a=b \\ lev_{a,b}(i,j-1)+1 & if \quad a=b \\ lev_{a,b}(i-1,j-1)+1 & otherwise \end{cases} & \end{cases} \tag{1.5}$$

2 Material and equipment

The necessary material for this practice is:

- A computer with the latest *Python* stable version installed
- A text editor

Or is possible to use the google site `https://colab.research.google.com/` that allows us to use a virtual machine with an *Ubuntu* operative system with *Python* installed.

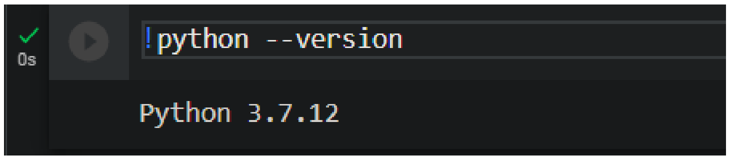
3 Practice development

3.1 Dynamic programming algorithms

To develop this practice we used the Google platform called *Colab* as this platform uses a virtual machine with linux (specifically Ubuntu) we can install some packets and of course we can verify if we have already *Python* installed. To check it we must use the command:

```
python --version
```

If we run this command in our linux terminal using *Colab* we can see the next as the result:



```
!python --version

Python 3.7.12
```

Figure 3.1: Verifying python version

Instructions: Write a Dynamic Programming algorithm in Python for the Knapsack problem, Change making problem and an extra one different of the mentioned before.

3.2 The knsapsack problem

To solve this problem, first of all we must create a table with the maxmum value that can be attained with weight less than or equal to w using items up to i (first items).

To get the number of items we can use the lenght of one of the arrays or the array that contains the items weight or items value.

Then, we create a matrix filled with zero values, this matrix will help us to store what has been the best solution with the current element (memoization).
Finally, we define the recursive solution as follows:

- $m[0,w] = 0$
- $m[i,w] = m[i-1,w]$ if $w_i > w$
- $m[i,w] = max(m[i-1,w], m[i-1,w-w_i]+v_i)$ if $w_i \leq w$

Where:

- m = the matrix that will store the best solution at (i,w) step.
- w = represents the current weight that we are trying to solve the problem.
- i = is the $i-th$ item (remember we have weight and values for each item)
- w_i = means the weight of the $i-th$ item
- v_i = the value of the $i-th$ item

In order to make easier to understand the code we have changed the name of the elements of the solution by:

- $m = DP$
- $w = currWeight$
- $i = currElement$
- $w_i = weight[i]$
- $v_i = values[i]$

```python
# THE KNAPSACK PROBLEM
import numpy as np

#INPUT VALUES
values = (3, 4, 5, 6)
weight = (2, 3, 4, 5)
maxWeight = 5

numberOfElements = len(values)

```

```python
#CALCULATING VALUES USING DYNAMIC PROGRAMMING

#We're using a matrix rows = currElement and col = currWeight
DP = np.zeros((numberOfElements + 1, maxWeight + 1))

#In cpp for(int currElement = 1; currElement < numberOfElements + 1; currElement++)
for currElement in range(1, numberOfElements + 1):
  #In cpp for(int currWeight = 1; currWeight < maxWeight + 1; currWeight++)
  for currWeight in range(1, maxWeight + 1):
    #Current element weight is greater than current weight (impossible to add)
    if weight[currElement - 1] > currWeight:
      DP[currElement][currWeight] = DP[currElement - 1][currWeight]
    else:
      #Transition list: as a summary the decisions that we can take to choose what's
     better at this point
      notTakeCurrentElement = DP[currElement - 1][currWeight]
      takeCurrentElement = DP[currElement - 1][currWeight - weight[currElement - 1]]
     + values[currElement - 1]
      DP[currElement][currWeight] = max(notTakeCurrentElement, takeCurrentElement)

print(f'Table result:\n{DP}')

#DP RECONSTRUCTION we can get the elements of the best solution
elementsUsed = []
itemValues = []
itemWeights = []


while currElement >= 0 and currWeight >= 0:
  #We move from bottom to up until the current element is different to the previous
    element
  while currElement > 0 and DP[currElement][currWeight] == DP[currElement - 1][
    currWeight]:
    currElement -= 1
  if currElement > 0:
    #If the current element is different to the previous one (with the same weight)
    we find an element that is part of the answer
    elementsUsed.append(currElement)
    itemValues.append(values[currElement - 1])
    itemWeights.append(weight[currElement - 1])
  currWeight -= (weight[currElement - 1])
  currElement -= 1

elementsUsed.reverse()
itemValues.reverse()
itemWeights.reverse()

print(f'\nThe best solution with a maximum weight of: {maxWeight} is: {DP[
    numberOfElements][maxWeight]} \nthe elements used were: {elementsUsed} (in that
    specific order) \nit means the items with values: {itemValues} and weights {
    itemWeights} respectively')
```

3.3 Making coin change problem

Like the *Knapsack problem*, we will create a table and then we will try to get the best solution. As we do not know if the coins are in order as the first step we must sort the coins array. Then, we create a data structure that allow us to store the subproblem solutions, for this problem we can use a matrix again, remember that this matrix must be initialized with zero values. This matrix have d rows, where d is the number of coins that we have and n columns where n is the change.

Something else that is different in this problem if we make a comparison with the *Knapsack problem* is that we want to maximize the solution instead of minimize.

Using the previous information we can get the solution for this problem:

- $m[1,n] = n$; $m[d,0] = 0$
- $m[1,n] = m[i-1,n]$ if $d_i > n$
- $m[i,n] = min(m[i-1,n], m[i,n-d_i]+1)$

Where:

- m = is the matrix that will store the best solution at $(i,n) step$
- n = is the change value
- i = is the $i-th$ coin
- d = is the denomination coins array
- d_i = is the $i-th$ coin value

In order to make easier to understand the solution we changed the name of the elements of the solution by:

- $m = DP$
- $n = value$ (change)
- $i = currentCoin$
- $d = coins$
- $d_i = coins[currentCoin]$

```python
#DYNAMIC PROGRAMMING

import numpy as np

coins = [1, 2, 5]
value = 11

nCoins = len(coins)
```

```

#currCoins = our coins and currValues = values (it starts from zero)
DP = np.zeros((nCoins, value + 1))

#for(currValue = 1; currValue < value + 1; currValue++)
for currValue in range(1, value + 1):
  DP[0][currValue] = currValue

for currCoin in range(1, nCoins):
  for currValue in range(1, value + 1):
    #Current coin value is impossible to use it because it's value is greater than
    the current change
    if coins[currCoin] > currValue:
      #We store the best solution at this point ignoring the current coin but using
    the previous one
      DP[currCoin][currValue] = DP[currCoin - 1][currValue]
    else:
      #We can use the current coin but also we can check if with the previous coin
    but with the same change we have a better solution
      #Transition list:
      notTakeCurrentCoin = DP[currCoin - 1][currValue]
      takeCurrentCoin = DP[currCoin][currValue - coins[currCoin]] + 1
      DP[currCoin][currValue] = min(notTakeCurrentCoin, takeCurrentCoin)

print(f'Table result: \n{DP}')

coinsUsed = []
coinsValues = []

while currCoin >= 0:
  while currValue >= 0:
    if DP[currCoin][currValue] != DP[currCoin - 1][currValue] and coins[currCoin] <=
     value:
      currValue -= coins[currCoin]
      coinsUsed.append(currCoin)
      coinsValues.append(coins[currCoin])
    elif currCoin == 0 and coins[currCoin] <= currValue:
      currValue -= coins[currCoin]
      coinsUsed.append(currCoin)
      coinsValues.append(coins[currCoin])
    else:
      currCoin -= 1
      break

print(f'With the change: {value} we need to use: {DP[nCoins - 1][value]} to minimize
     the number of coins, \nand the coins to use are: {coinsUsed} with the values {
    coinsValues}')
```

3.4 Levenshtein distance

The Levenshtein distance problem is very similar to other problems, we will create a table and then calculate the solution, the table has the dimensions mxn where m is the length of the first string and n is the lenght of the second string. Like in the other problems we must get the recurrence first and we got it previouslly with the eq. (1.5)

In code we get something like this:

```python
#DYNAMIC PROGRAMMING

import numpy as np

def areEqualCharacters(a, b):
  return 0 if a == b else 1

firstWord = "hello"
secondWord = "hallo"

maxFirstWord = len(firstWord)
maxSecondWord = len(secondWord)

DP = np.zeros((maxFirstWord + 1, maxSecondWord + 1))

#It is equals to for(int i = 0; i < firstWord + 1; i++)
for i in range(maxFirstWord + 1):
  #This case means the secondWord is empty so we do not need to change anything
  DP[i][0] = i

#It is equals to for(int j = 0; j < secondWord + 1; j++)
for j in range(maxSecondWord + 1):
  #Very similar to the previous case it means firstWord is empty we do not need to
    change anything
  DP[0][j] = j

for i in range(1, maxFirstWord + 1):
  for j in range(1, maxSecondWord + 1):
    #Trivial solution, one of the strings is empty in this subproblem
    if min(i, j) == 0:
      DP[i][j] = max(i, j)
    #Transition list
    #Fist transition: we move the first pointer
    moveFirstPointer = DP[i - 1][j] + 1
    #Second transition: we move the second pointer
    moveSecondPointer = DP[i][j - 1] + 1
    #third transition: we move both pointers and we check if both characters are
    equal or not (if they are equal we add 1 to the solution)
    moveBothPointers = DP[i - 1][j - 1] + areEqualCharacters(firstWord[i - 1],
    secondWord[j - 1])
    #We choose the best solution
    DP[i][j] = min(moveFirstPointer, moveSecondPointer, moveBothPointers)

print(f'Table result: \n{DP}')
print(f'\nMaximum moves needed: {DP[maxFirstWord][maxSecondWord]}')
```

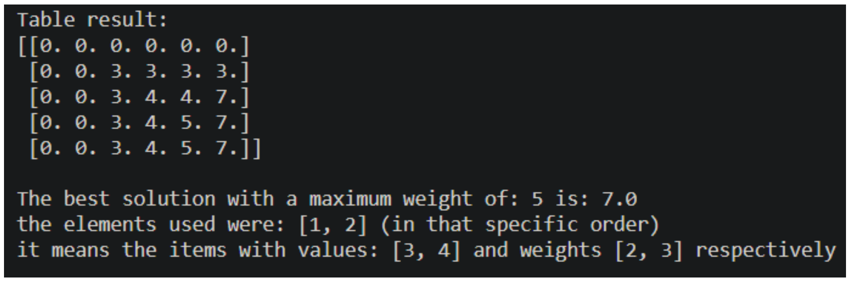
4 Screens, graphs and diagrams

Taking about the three problems explained here (knapsack, making coin change, Levenshtein distance) here we have some screen shots about some experiments.

4.1 Knapsack problem

For this experiment we use the items with the values (3,4,5,6) and weights (2,3,4,5) and our knapsack has a maximum weight of 5.

As we can see in the image 4.1 to solve this problem the best solution is get a value of 7 and we use two elements the first one and the second one, in other words we use items with values 3 and 4 and weights 2 and 3 respectively



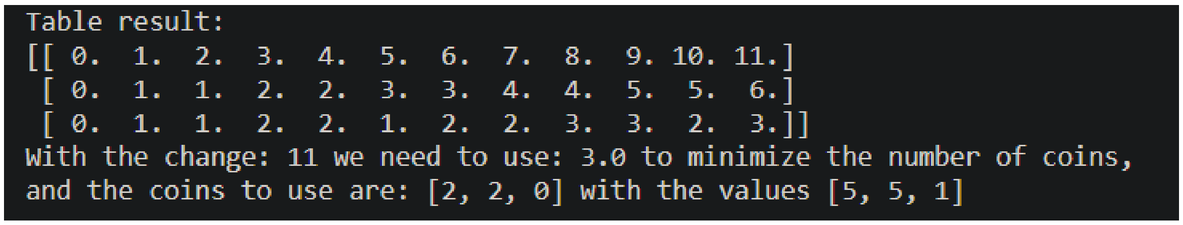
```
Table result:
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 3. 3. 3. 3.]
 [0. 0. 3. 4. 4. 7.]
 [0. 0. 3. 4. 5. 7.]
 [0. 0. 3. 4. 5. 7.]]

The best solution with a maximum weight of: 5 is: 7.0
the elements used were: [1, 2] (in that specific order)
it means the items with values: [3, 4] and weights [2, 3] respectively
```

Figure 4.1: Knapsack problem experiment

4.2 Making coin change problem

For this experiment we use 3 coins with values (1,2,5) and a change of 11. As we can see in the image 4.2 the best solution or the minimum number of coins needed to give 11 of change is 3 and we use the coins (2,2,0) (zero indexed), and the coins have the values (5,5,1) respectively



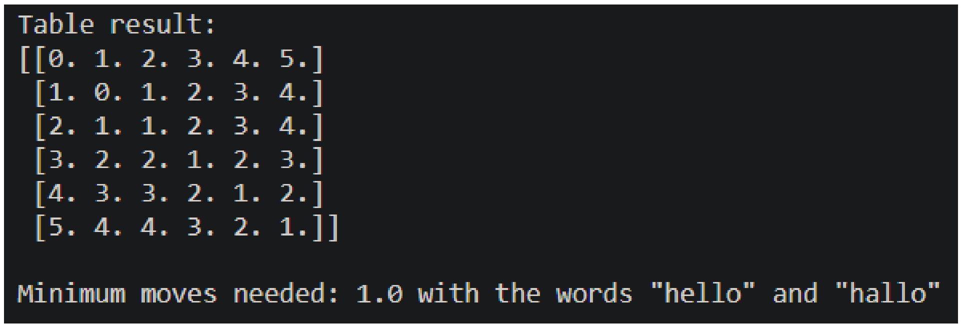
```
Table result:
[[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11.]
 [ 0.  1.  1.  2.  2.  3.  3.  4.  4.  5.  5.  6.]
 [ 0.  1.  1.  2.  2.  1.  2.  2.  3.  3.  2.  3.]]
With the change: 11 we need to use: 3.0 to minimize the number of coins,
and the coins to use are: [2, 2, 0] with the values [5, 5, 1]
```

Figure 4.2: Making coin change problem experiment

4.3 Levenshtein distance

Finally for this experiment we used two words the first one was "hello" and the second one was "hallo" as we can see we need to change the second character to make it equal, it is the best solution.

Again as we can see in the image 4.3 we can confirm that we need 1 operation to convert from "hello" to "hallo" using those words.



```
Table result:
[[0. 1. 2. 3. 4. 5.]
 [1. 0. 1. 2. 3. 4.]
 [2. 1. 1. 2. 3. 4.]
 [3. 2. 2. 1. 2. 3.]
 [4. 3. 3. 2. 1. 2.]
 [5. 4. 4. 3. 2. 1.]]

Minimum moves needed: 1.0 with the words "hello" and "hallo"
```

Figure 4.3: Edit distance problem experiment

5 Conclusions

Dynamic programming is a very interesting technique to solve problems where we can avoid recalculate subproblems, it helps to reduce a lot the time complexity but probably we increase the space complexity of our algorithm. Additionally we can solve dynamic programming problems using two ways bottom-up or top-down in other words we can try to solve this type of problems using recursion from the entire problem and split it in subproblems or try to solve our problem from the small case to the greatest subproblem.

Bibliography

[1] *Dynamic Programming*. MIT. Accessed on September 18th, 2021. Available online: `http://web.mit.edu/15.053/www/AMP-Chapter-11.pdf`

[2] *Dynamic Programming*. December 29th, 2018, 0th Edition (pre-publication draft). Accessed on September 18th, 2021. Available online:`http://jeffe.cs.illinois.edu/teaching/algorithms/book/03-dynprog.pdf`

[3] F.S. Hillier and G. J. Lieberman. *Introduction to operations Research*. 6th edition. McGraw-Hill, 1995. Accessed on September 19th, 2021. Available online: `https://personal.utdallas.edu/~scniu/OPRE-6201/documents/DP3-Knapsack.pdf`

[4] D. Kozen and S. Zaks. *Optimal Bounds for the Change-Making Problem*. Computer Science Department, Cornell University Ithaca, New York, USA. Accessed on September 19th, 2021. Available online `https://www.cs.cornell.edu/~kozen/Papers/change.pdf`

[5] A. Andoni and K. Ona. *Approaching edit distance in near-linear time*. MIT. Accessed on September 20th, 2021. Available online: `https://www.mit.edu/~andoni/papers/compEdit.pdf`