Instituto Politécnico Nacional
Escuela Superior de Cómputo
Evolutionary Computing
1 Introduction to Dynamic Programming
Martínez Coronel Brayan Yosafat
Rosas Trigueros Jorge Luis
17/09/2021
23/09/2021

Theorical framework

As an introduction, we want to solve a few problems with dynamic programming, that includes, and let us define every problem:

- Knapsack problem
- Change making problem
- Longest Common Subsequence

According to Wikipedia, Knapsack problem is defined as: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible [1]. Whereas Change making problem is the problem of representing a given amount of money with the fewest number of coins possible from a given set of coin denominations [2].

Finally, Longest Common Subsequence tell us that given two sequences, find the length of longest subsequence present in both. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, **“abc”, “abg”, “bdf”, “aeg”,” acefg”,** etc., are subsequences of **“abcdefg”** [3].

Before we dive deeper, we shall mention what does Dynamic programming mean, according to the site *InterviewBit* Dynamic Programming (commonly referred to as

DP) is an algorithmic technique for solving a problem by recursively breaking it down into simpler subproblems and using the fact that the optimal solution to the overall problem depends upon the optimal solution to its individual subproblems, developed by Richard Bellman in the fifties [4].

Material and equipment

We will use Google Colaboratory as well as Python (Numpy and Pandas), but we shall mention we copy and paste the code in PyCharm because it really puts a nice format. A personal computer with a Core i5 7th generation and 12 GB of RAM.

Practice development

Knapsack problem

As part of the practice, we wrote a script in a Jupiter Notebook, first we are going to talk about the Knapsack problem. Let v and w be vectors (lists in Python), they contain the values of the objects and its weights, respectively. Set W as the capacity of the knapsack and N as the number of objects. Then, in a matrix of zeros of N+1 x W+1, we iterate over the combinations as follows in section 2, this includes an example:

```
# Knapsack problem
# Section 1: Data definition
import numpy as np

v = (2, 3, 4, 5, 6)    # Values
w = (2, 4, 5, 5, 7)    # Weights
W = 9                  # Total weight of the Knapsack
N = len(w)             # Number of objects

m = np.zeros((N + 1, W + 1))

# Section 2: Dynamic programming
for row in range(1, N + 1):
    for col in range(1, W + 1):
        if w[row - 1] > col:
            m[row][col] = m[row - 1, col]
        else:
            m[row][col] = max(m[row - 1][col], m[row - 1][col - w[row -
1]] + v[row - 1])
```

```
# Section 3: Answer searching
indices = []
current = W

for row in range(N, 1, -1):
    for col in range(current, 1, -1):
        if m[row][col] != m[row - 1][col]:
            indices.append(row)
            current = col - w[row-1]
            break

# Section 4: Report
elements = [v[index-1] for index in indices]
weights = [w[index-1] for index in indices]
print(m)

print("\nAnswer report:")
print("Elements: ", elements)
print("Weights: ", weights)
```

Notice how the matrix is filled with initial values, we set the first row and the first column as 0. Of course, that is just half of the problem, with the matrix (Fig. 1) we need to find the actual answer, and we make as the section 3 of the code above. Lastly, but not least, we print the results. That part was kind of hard since I have never done something like this before. I mean, I did it in just another subject, but I already forget about it.

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 2. 2. 2. 2. 2. 2. 2. 2.]
 [0. 0. 2. 2. 3. 3. 5. 5. 5. 5.]
 [0. 0. 2. 2. 3. 4. 5. 6. 6. 7.]
 [0. 0. 2. 2. 3. 5. 5. 7. 7. 8.]
 [0. 0. 2. 2. 3. 5. 5. 7. 7. 8.]]
```

Fig. 1 Matrix from Knapsack problem

```
Answer report:
Elements:  [5, 3]
Weights:  [5, 4]
```

Fig. 2 Results from Knapsack problem example

So, to answer the problem, we start at position 6, 10 (the last one) and iterate over the matrix in reverse. If the number above our position is different, thus we go left the value of the current row. Else, we just go up until there is a different value above our position. If our current position has a value of 0, then we finished.

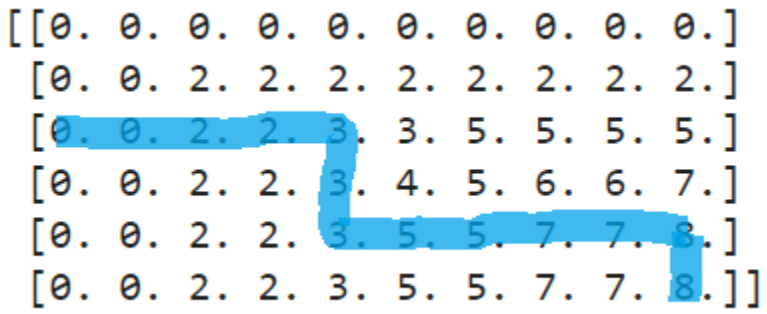

Fig. 3 Solution of the example

Change making problem

Now, let us talk about the change making problem, it is like the previous problem, in fact, we use a similar arrange. A matrix of zeros with a shape of N_den x N, where the first means Number of denominations and the second means Change.

```
# Change making problem
import numpy as np
import pandas as pd

# Section 1: Data definition
d = [1, 2, 5]    # Denominations
N = 9            # Change
N_den = len(d)
m = np.zeros((N_den, N + 1))

for col in range(1, N + 1):
    m[0][col] = col

# Section 2: Dynamic Programming
for row in range(1, N_den):
    for col in range(1, N + 1):
        if d[row] > col:
            m[row][col] = m[row - 1][col]
        else:
            m[row][col] = min(m[row - 1][col], m[row][col - d[row]] + 1)

# Section 3: Answer searching
current = N
```

```
solution = [0] * N_den

for row in range(N_den-1, -1, -1):
    if row == 0:
        solution[row] = m[row][current]
        break

    while m[row][current] != m[row-1][current]:
        solution[row] += 1
        current = current - d[row]

# Section 4: Answer report
print(m)

dictionary = {"Den": d, "Quantity": solution}
dt = pd.DataFrame.from_dict(dictionary)
dt = dt.set_index(["Den"])
dt = dt.convert_dtypes(convert_integer=["Quantity"])
print("\nChange required:", N)
print(dt)
```

So, as we can observe, the matrix is filled with zeros in its first column and filled with its index column in its first row (Fig. 4). Then we iterate over the combinations asking if we can use the current coin row, so, if the denomination is greater than the actual N, we just use the upper row, but, if they are equal then we put a one, and if they are lesser than the current N, we check if it does better job than the upper row.

```
[[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.]]
```

Fig. 4 Change making matrix

```
Change required: 9
     Quantity
Den
1           0
2           2
5           1
```

Fig. 5 Change making solution

Finding the solution is like the previous problem, we start at the end, if the cell above has the same value as our current position, then we move towards zero. If they are not the same, that means we should move to the left denomination times, where denomination is given by d[row]. If our current position has a value of zero, we shall end, or if we are at row zero (the first one), then we use the value of that cell as the number of coins of denomination one.

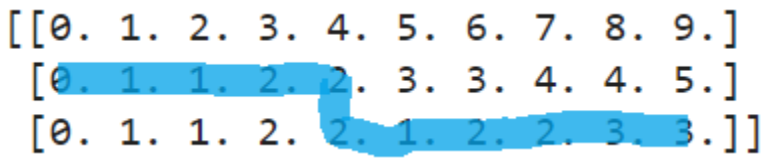

Fig. 6 Solution path of the example

Longest Common Subsequence

The first though could be making all combinations of subsequences and compare them, which is a terrible idea. Rather we use dynamic programming, starting at the end. Let us define a function L, which returns the length of the longest subsequence, letting A and B be a pair of strings with a length of n and m respectively, then L(A[0, ..., n-1], B[0, ..., m-1]) would return an integer greater or equal than zero. So, making it recursively, we find that:

If m is 0 or n is 0 then return 0

If A[m-1] is equal to B[n-1] then return 1 + L(A[0, ..., n-2], B[0, ..., m-2])

Else return max (L(A[0, ..., n-2], B[0, ..., m-1]), L(A[0, ..., n-1], B[0, ..., m-2]))

So, we could see this as a tree. So, this will be a little bit different about how we solve it with the matrix (it is just I could not see a way for using the same way as before), so I made a recursive function as follows in the code:

```
import numpy as np
```

```
# Section 1: Data definition
X = 'ABACDEH'
Y = 'BDFTE'
m = len(X)
n = len(Y)
matrix = np.zeros((m, n))


# Section 2: Dynamic programming
def lcs(string_a, string_b, l_a, l_b) -> int:
    answer: int

    if l_a == 0 or l_b == 0:
        return 0
    elif string_a[l_a - 1] == string_b[l_b - 1]:
        answer = 1 + lcs(string_a, string_b, l_a - 1, l_b - 1)
    else:
        answer = max(lcs(string_a, string_b, l_a, l_b - 1), lcs(string_a,
string_b, l_a - 1, l_b))

    matrix[l_a-1][l_b-1] = answer
    return answer


length = lcs(X, Y, m, n)

# Section 3: Answer search
print(matrix)

row = m-1
col = n-1
subsequence = []
while not (row == 0 or col == 0):
    if matrix[row-1][col] == matrix[row][col]:
        row -= 1
    elif matrix[row][col-1] == matrix[row][col]:
        col -= 1
    else:
        row -= 1
        col -= 1
        subsequence.append(X[row+1])

if col != 0 and row == 0:
    for letter in Y[0: col]:
        if X[row] == letter:
            subsequence.append(X[row])
            break
elif col == 0 and row != 0:
    for letter in Y[0: row]:
        if letter == Y[col]:
            subsequence.append(Y[col])
            break
else:
    if X[row] == Y[col]:
        subsequence.append(Y[col])

print('\n' + X)
```

```
print(Y)
print('Longest subsequence: ')
print(''.join(reversed(subsequence)))
```

One of the important things is that there is a function and uses the matrix to write the score of the longest subsequence so far.

```
[[0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 0.]
 [1. 1. 1. 1. 0.]
 [1. 1. 1. 1. 0.]
 [1. 2. 2. 2. 0.]
 [1. 2. 2. 2. 3.]
 [1. 2. 2. 2. 3.]]
```

Fig. 7 Matrix of the LCS problem

```
ABACDEH
BDFTE
Longest subsequence:
BDE
```

Fig 8. Result of the problem

This problem in particular was pretty difficult for me, as I said I do not really know how to use dynamic programming, so I read how does it work from Geeks for Geeks, and, using a similar way as they I change it to print a matrix, then, I look and saw that if there is a value (up or left) of the same integer as our current position, then we should move to that cell. If there is not an equal value, it means that character is part of the subsequence, and we move one column to the left and one row to top at the same time. We repeat until one or both iterators are 0.

But there is a particular case, the first character of the position could be in the subsequence, so we check it with the last if. And, of course, we printed it.

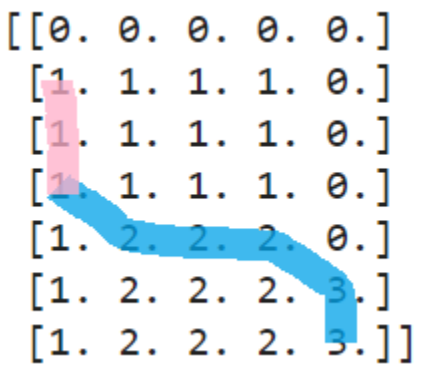

Fig. 9 Path of the solution, pink represents the if, blue the while

Conclusions and recommendations

Maybe one the hardest things I have ever done with these kinds of problems. But do not get me wrong, I have always wanted to get into algorithms, I coursed Algorithm' s analysis just when the pandemic came. It is still fun, I just need time, I really like how you introduce this topic, and I looked at my work and feel comfortable with it. I know it could be better, but been my first time, it is good, actually.

References

[1] Wikipedia. *Knapsack problem*. Aug. 26, 2021. Wikipedia. Accessed on: Sep. 22, 2021. [Online] Available: https://en.wikipedia.org/wiki/Knapsack_problem

[2] S. Goebbels *et. al. Change-making problems revisited: a parameterized point of view,* May. 30, 2017. Springer. Accessed on: Sep. 22, 2021. [Online] Available: https://doi.org/10.1007/s10878-017-0143-z

[3] Geeks for Geeks. *Longest Common Subsequence | DP-4.* Aug. 26, 2021. Geeks for Geeks. Accessed on: Sep. 22, 2021. [Online] Available: https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/

[4] InterviewBit. *Dynamic Programming.* InterviewBit. Accessed on: Sep. 22, 2021. [Online] Available: interviewbit.com/courses/programming/topics/dynamic-programming/