Project Development phase

Utilization Of Algorithms, Dynamic Programming, Optimal Memory Utilization

DATE	06 May 2023
Team ID	NM2023TMID18418
Project Name	CancerVision: Advanced Breast Cancer Prediction with Deep Learning

The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. ("Programming" in this context refers to a tabular method, not to writing computer code.) As we saw in Chapter 1, divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered.

Dynamic programming is typically applied to ***optimization problems***. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or

maximum) value. We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.

The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution in a bottom-up fashion.

4. Construct an optimal solution from computed information.

Steps 1-3 form the basis of a dynamic-programming solution to a problem. Step 4 can be omitted if only the value of an optimal solution is required. When we do perform step 4, we sometimes maintain additional information during the computation in step 3 to ease the construction of an optimal solution.

The sections that follow use the dynamic-programming method to solve some optimization problems. asks how we can multiply a chain of matrices so that the fewest total scalar multiplications are performed. Given this example of dynamic programming, discusses two key characteristics that a problem must have for dynamic programming to be a viable solution technique. then shows how to find the longest common subsequence of two sequences. Finally, uses dynamic programming to find an optimal triangulation of a convex polygon, a problem that is surprisingly similar to matrix-chain multiplication.

Algorithm

```
def FindMax(W, n, values, weights):
    MaxVals = [[0 for x in range(W + 1)] for x in range(n + 1)]
```

```
for i in range(n + 1):
        for w in range(W + 1):
                if i == 0 or w == 0:
                        MaxVals[i][w] = 0
                elif weights[i-1] <= w:
                        MaxVals[i][w] = max(values[i-1]
                                        + MaxVals[i-1][w-weights[i-1]],
                                                MaxVals[i-1][w])
                else:
                        MaxVals[i][w] = MaxVals[i-1][w]
```

Algorithm

- Initialize an array of size **n+1**, where n is the amount. Initialize the value of every index **i** in the array to be equal to the amount. This denotes the maximum number of coins (using coins of denomination 1) required to make up that amount.
- Since there is no denomination for 0, initialise the base case where **array[0] = 0**.
- For every other index **i**, we compare the value in it (which is initially set to **n+1**) with the value **array[i-k] +1**, where **k** is less than **i**. This essentially checks the entire array up till i-1 to find the minimum possible number of coins we can use.
- If the value at any **array[i-k] + 1** is lesser than the existing value at **array[i]**, replace the value at **array[i]** with the one at **array[i-k] +1**.

Code

```
def coin_change(d, amount, k):
    numbers = [0]*(amount+1)
```

```
    for j in range(1, amount+1):
        minimum = amount
        for i in range(1, k+1):
            if(j >= d[i]):
                minimum = min(minimum, 1 + numbers[j-
d[i]])
            numbers[j] = minimum

    return numbers[amount]
```