Dept. of Electronics and Electrical Communication Engineering

Indian Institute of Technology Kharagpur

ALGORITHMS
(EC31205)

CODING TASK-3

AUTHOR: JAYA KISHNANI
ROLL NUMBER: 20EC30020
DATE: 28/10/2022
THEME: DYNAMIC PROGRAMMING

Problem 1: Fractional Knapsack problem

Naïve (recursive) approach

In this we assumed base case when both weight and value are zero to be returned zero.
Then there were three possible cases:

1) When the weight of the given fractional nth item is greater than the maximum knapsack capacity, then we do not include it in our optimal solution.
2) Else when the weight of the given nth item is greater than the maximum knapsack capacity then we either take the fractional weight and value or exclude it from optimal solution.
3) Else we have three options either to choose the weight fully or fractionally or exclude it.

The code snippet is as follows

```
# A naive recursive implementation of Fractional Knapsack Problem

# Returns the maximum value that can be put in a knapsack of
# capacity W
def knapSack(W, wt, val, n, f):

  # Base Case
  if n == 0 or W == 0 :
    return 0

  # If weight of the fraction of nth item is more than Knapsack of capacity
  # W, then this item cannot be included in the optimal solution
  if (wt[n-1]*f > W):
    return knapSack(W, wt, val, n-1, f)

  # If the weight of nth item is more than Knapsack capacity
  # then either we return fractional nth item or we don't include it
  # whichever is maximum
  elif (wt[n-1] > W):
    return max(val[n-1]*f + knapSack(W-(wt[n-1]*f), wt, val, n-1, f), knapSack(W, wt, val, n-1, f))

  # return the maximum of three cases:
  # (1) nth item included fully
  # (2) nth item fraction included
  # (3) not included
  else:
    return max(val[n-1] + knapSack(W-wt[n-1], wt, val, n-1, f), val[n-1]*f + knapSack(W-(wt[n-1]*f), wt, val, n-1, f),
        knapSack(W, wt, val, n-1, f))

# end of function knapSack
```

The result we obtained was

```
val = np.array([60, 100, 120])
wt = np.array([10, 20, 30])
W = 50
n = len(val)
f = 0.3
print(knapSack(W, wt, val, n, f))
```

```
220
```

So here we obtained value as 220 by taking fractional factor as 0.3.
It's time complexity is exponential i.e. **O(2^n)**.

Dynamic programming approach

1) It involved building of a 2-D array of size (W+1, n+1) (where W is knapsack capacity and n is the length of value array) in bottom up approach.
2) In this we normalized the knapsack capacity and weight array as per the fractional factor. For eg: if fraction factor is 0.3 normalizing factor taken is 10, if 0.39 then 100. It is taken because while taking fractional weights we can't have array indices as fraction or float value, they are integral.
3) We used memoization in which we first stored the cases as defined in above approach instead of calling the function again.
4) This boils down the time complexity as **O(N*logN)**

The code snippet is as follows:

```
46] # Dynamic Programming approach
    # Program for Fractional Knapsack problem
    # Returns the maximum value that can
    # be put in a knapsack of capacity W
    def knapSack(W, wt, val, n, f):
      normalize = 10

      #Normalizing weights
      W = W*normalize
      wt = wt*normalize
      #print(wt)
      K = [[0 for x in range (W + 1)] for x in range(n + 1)]

      # Build table K[][] in bottom up manner
      for i in range(n + 1):
        for w in range(W + 1):
          if i == 0 or w == 0:
            K[i][w] = 0
          elif f*wt[i-1] <= w:
            if wt[i-1]>w:
              K[i][w] = max(f*val[i-1] + K[i-1][w-int(f*wt[i-1])], K[i-1][w])
            else:
              K[i][w] = max(f*val[i-1] + K[i-1][w-int(f*wt[i-1])], val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])
          else:
            K[i][w] = K[i-1][w]
      #print(K)
      return K[n][W]
```

The result of DP approach:

```
val = np.array([60, 100, 120])
wt = np.array([10, 20, 30])
W = 50
n = len(val)
f = 0.3
print(knapSack(W, wt, val, n, f))
```

```
220
```

We observed that both methods gave same results but DP approach is more optimal.

Problem 2: Travelling Salesman problem

We defined the graph as follows:

```
nodes = ["A", "B", "C", "D", "E", "F", "G

init_graph = {}
for node in nodes:
    init_graph[node] = {}

init_graph["A"]["B"] = 1
init_graph["B"]["E"] = 6
init_graph["B"]["D"] = 3
init_graph["C"]["D"] = 2
init_graph["D"]["E"] = 2
init_graph["D"]["F"] = 4
init_graph["C"]["F"] = 5
init_graph["E"]["F"] = 2
init_graph["E"]["G"] = 7
init_graph["F"]["G"] = 6
```

```
graph = Graph(nodes, init_graph)
```

We defined the TSP function implementing travelling salesperson algorithm and used itertools library for generating the subsets.

The code snippet for TSP algorithm implemented:

```
def tsp(graph,start_node):

  list_nodes = list(graph.get_nodes())
  n = len(list_nodes)

  subset_list = [[start_node]]

  C = [[sys.maxsize for x in range(n)] for x in range(2**(n - 1))]

  C[0][0] = 0

  for s in range(1,n):

    list_nodes.remove(start_node)
    subset_list_s = subsets(list_nodes,s)
    list_nodes.insert(0,start_node)
    for S in subset_list_s:

      S = [list(item) for item in S]
      S.insert(0,start_node)

    subset_list.extend(subset_list_s)
```

```
    #print(subset_list)
    for S in subset_list_s:

      S = [list(item) for item in S]
      S.insert(0,start_node)

      for j in range(n):
        if j == 0 or S.count(list_nodes[j]) == 0:
          continue
        neighbours = graph.get_outgoing_edges(list_nodes[j])
        min_dist = sys.maxsize
        temp = S

        temp.remove(list_nodes[j])

        for i in neighbours:

          i_ind = list_nodes.index(i)
          if min_dist > C[subset_list.index(temp)][i_ind] + graph.value(i,
list_nodes[j]):
            min_dist = C[subset_list.index(temp)][i_ind] + graph.value(i,l
ist_nodes[j])
        C[subset_list.index(S)][j] = min_dist

  return min(C[subset_list.index(list_nodes)])
```

It is an NP hard problem with time complexity O(n^2*(2^n)).