Question 1

Given the assignment problem

The cost matrix C represents the assignment costs C[i, j], where persons 1, 2, and 3 are denoted as persons a, b, and c, respectively.

1. Solving the assignment problem by the greedy approaches we get:

Row-by-row:

Row 1 : Person a Job 3

Row 2 : Person b Job 1

Row 3 : Person c Job 2

The optimal solution is {a 3, b 1, c 2} with the total cost = which is the minimal cost.

Column-by-Column:

Column 1

Column 2

Column 3

The optimal solution is {a 2, b 1, c 3} with the total cost = which is the minimal cost.

Entire matrix:

Choice 1

Choice 2

Choice 3

The optimal solution is {a 2, b, c 1} with the total cost = which is the minimal cost.

1. Given the assignment problem

The cost matrix C represents the assignment costs C[i, j], where persons 1, 2, 3, and 4 are denoted as persons a, b, c, and d, respectively.

Solving the assignment problem by a branch-and-bound technique we get:

The optimal solution would have a total cost at least this is represented at Node 0.

Level 1 consists of lower bounds for each value in row 1:

Node

Node 2 is the most promising node since it has the lowest lower bound value.

Expanding from Node 2 we get:

Node

Node 2 is the most promising node since it has the lowest lower bound value. Node 8

Node 9

The optimal solution found using promising nodes is {a 2, b, c 1, d 4} with a total cost of 64. Node 1 is expanded since it has a lower bound value lower than the solution found:

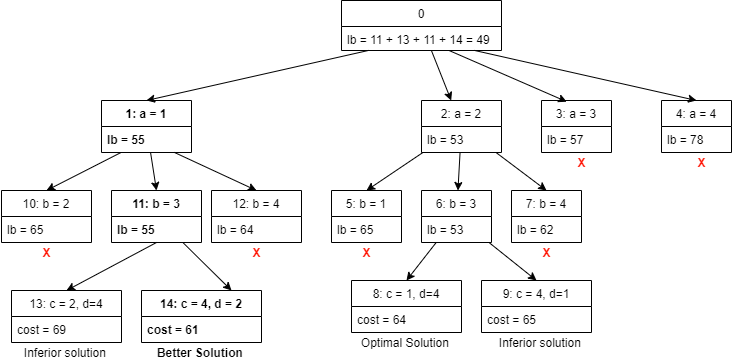
Expanding from Node 1 we get:

Node

Node 2 is expanded because it is the most promising with a lower bound smaller than the previously found solution:

Node 13

Node 14



The btter optimal solution is {a 1, b, c 4, d 2} with the total cost of 61.

Question 2

1. Solve the following instance of the 0/1 knapsack problem by a greedy technique.

|  |  |  |
| --- | --- | --- |
| Item | Weight | Value |
| 1 | 1 | $1 |
| 2 | 2 | $6 |
| 3 | 5 | $18 |
| 4 | 6 | $22 |
| 5 | 7 | $28 |

W=11

The value to weight ratios are computed where

|  |  |  |  |
| --- | --- | --- | --- |
| Item | Weight | Value |  |
| 1 | 1 | $1 | 1 |
| 2 | 2 | $6 | 3 |
| 3 | 5 | $18 | 3.6 |
| 4 | 6 | $22 | 3.66 |
| 5 | 7 | $28 | 4 |

The items are sorted in descending order according to the ratio:

|  |  |  |  |
| --- | --- | --- | --- |
| Item | Weight | Value |  |
| 5 | 7 | $28 | 4 |
| 4 | 6 | $22 | 3.66 |
| 3 | 5 | $18 | 3.6 |
| 2 | 2 | $6 | 3 |
| 1 | 1 | $1 | 1 |

Items are selected in descending order:

Select item 5, total weight = 7 < 11 Item selected. Knapsack contents: {5}

Add item 4, total weight = 13 > 11 Item not selected. Knapsack contents: {5}

Add item 3, total weight = 12 > 11 Item not selected. Knapsack contents: {5}

Add item 2, total weight = 9 < 11 Item selected. Knapsack contents: {5, 2}

Add item 1, total weight = 10 < 11 Item selected. Knapsack contents: {5, 2, 1}

The items selected are {5, 2, 1} with a total weight of 10 and value of 28 + 6 + 1 = $35.

1. Solve the following instance of the 0/1 knapsack problem by a dynamic programming technique:

|  |  |  |
| --- | --- | --- |
| Item | Weight | Value |
| 1 | 1 | $1 |
| 2 | 2 | $6 |
| 3 | 5 | $18 |
| 4 | 6 | $22 |
| 5 | 7 | $28 |

W=11

The following recurrence relation and the weights and values given above are used to calculate the values for the DP table below:

The recurrence relation above was used to create the following dynamic programming table:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 6 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| 3 | 0 | 1 | 6 | 7 | 7 | 18 | 19 | 24 | 25 | 25 | 25 | 25 |
| 4 | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 24 | 28 | 29 | 29 | 40 |
| 5 | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 28 | 29 | 34 | 35 | 40 |

The maximal value is F (5,11) = $40.

Using the backtracking algorithm, and the Dynamic programming table above (F):

move to F(4,11)move to F(3, 5) , move to F(2,0) move to F(1,0)Algorithm terminated.

The value of the optimal solution is $40, selected items are {4,3}, the total weight is 11.

Question 3

Question 4

Question 5

1. Solve the following instance of the 0/1 knapsack problem by a branch and bound technique.

|  |  |  |
| --- | --- | --- |
| Item | Weight | Value |
| 1 | 10 | $100 |
| 2 | 7 | $63 |
| 3 | 8 | $56 |
| 4 | 4 | $12 |

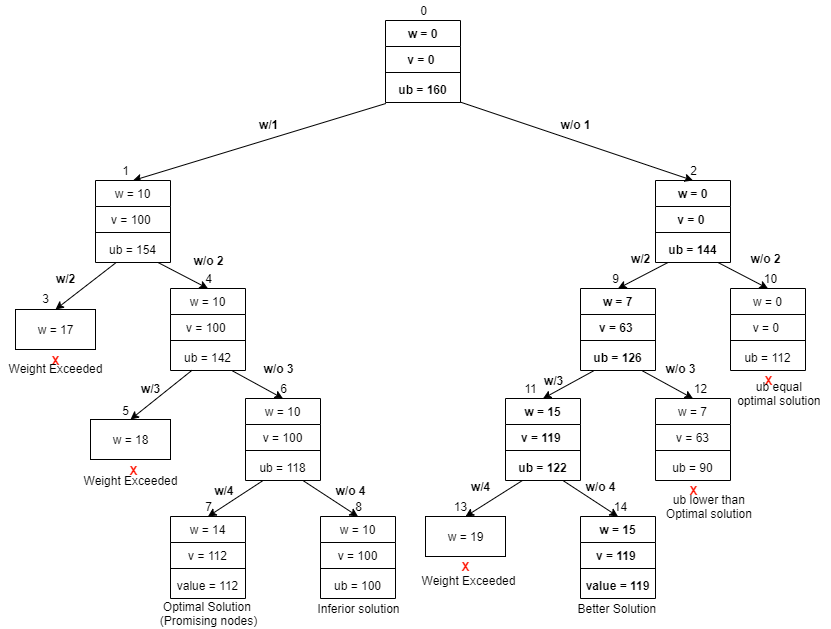
W=16

The value to weight ratios are computed where

|  |  |  |  |
| --- | --- | --- | --- |
| Item | Weight | Value |  |
| 1 | 10 | $100 | 10 |
| 2 | 7 | $63 | 9 |
| 3 | 8 | $56 | 7 |
| 4 | 4 | $12 | 3 |

The items are already sorted in descending order according to the ratio.

The upper bound



From node 0:

Node 0 Node 1 Node 2

Expanding from Node 1:

Node 3 Weight exceeded.  
Node 4

From Node 4:

Node 5 Weight exceeded.  
Node 6

From Node 6:

Node 7 (optimal solution)   
Node 8

The value of the optimal solution using only promising nodes is $112, selected items are {1,4}, the total weight is 14.

Expanding from Node 2:

Node 9 Node 10

From Node 9: (Since the ub is higher than the optimal solution)

Node 11 Node 12 (lower than optimal solution)

From Node 11: (Since the ub is higher than the optimal solution)

Node 13 Weight exceeded.  
Node 8 (better solution)

The value of the better optimal solution is $119, selected items are {2,3}, the total weight is 15.

1. def changeMaking(D, n):  
    # changeMaking algorithm adapted from the Dynamic Programming lecture  
    F = [0 for i in range(n+1)] # Creates the array to establish it's length  
    m = len(D)  
    for i in range(1, n+1):  
    temp = float('inf')  
    j = 1  
    while j <= m-1 and D[j] <= i:  
    temp = min(F[i - D[j]], temp)  
    j += 1  
    F[i] = temp +1  
    print("The minimum number of coins to make change for n = ", n, " is ", F[n])  
    return F[n], F # returns F instead of using a global variable  
     
     
   def Backtrack(D, n, minimum, F):  
    s = [] # stores the minimum coin set as it is generated  
    j = len(D)-1  
    while j >= 1 and n > 0:  
    # checking the computations for their component denominations  
    if F[n-D[j]] + 1 == minimum and D[j] <= n:  
    s.append(D[j])  
    n = n - D[j]   
    if n > 0:  
    j = len(D) - 1 # reset the pointer to the denominations array   
    minimum = F[n] # move to the next minimum produced  
    else:  
    j = 0  
    else:  
    j -= 1  
    return s  
     
     
   D = [0, 1, 5, 10]  
   n = 8  
   minm, F = changeMaking(D, n)  
   min\_coin\_set = Backtrack(D, n, minm, F)  
   print("The minimum coin set for n = ", n, "is", min\_coin\_set)