**Exercise - 10**

**Branch and Bound Technique**

**Aim:** To write the Python code to solve Knapsack and Travel Salesman Problem using Branch and Bound Technique.

**Algorithm:**

**Knapsack Problem:**

1. Initialize the current best solution as an empty set with a maximum value of 0.
2. Create a priority queue (also known as a heap) to store partial solutions (nodes) based on their potential maximum value.
3. Insert the root node into the priority queue, which represents the decision of including or excluding the first item.
4. Repeat the following steps until the priority queue becomes empty:

* Extract the node with the highest potential maximum value from the priority queue.
* If the node represents a complete solution (all items considered), update the current best solution if its value is higher.
* If the node represents an incomplete solution (not all items considered):
* Compute the potential maximum value for the node by considering two possibilities: - Including the next item and updating the current weight and value accordingly. - Excluding the next item and keeping the current weight and value.
* If the potential maximum value is higher than the current best solution value: - Create two child nodes: one representing the inclusion of the next item and the other representing the exclusion. - Insert the child nodes into the priority queue.

1. Return the current best solution as the optimal solution.

**Travel Salesman Problem:**

1. Initialize variables:

* Set the initial best distance to infinity.
* Create an empty list to store the best tour.
* Create an empty stack to store the subproblems.

1. Create the initial subproblem:

* Start with a partial tour containing only the starting node.
* Calculate the lower bound for the partial tour.

1. Push the initial subproblem onto the stack.
2. Repeat until the stack is empty:

* Pop the subproblem from the top of the stack.
* If the lower bound of the subproblem is higher than the current best distance, prune the subproblem and continue to the next iteration.
* If the subproblem represents a complete tour:
* Update the best distance and best tour if the current tour has a lower distance.
* Continue to the next iteration.
* If the subproblem is not a complete tour:
* Generate child subproblems by extending the current tour with each unvisited node.
* Calculate the lower bound for each child subproblem.
* Push the child subproblems onto the stack.

1. Return the best distance and best tour.

**Source Code:**

**Knapsack Problem:**

class Item:

    def \_\_init\_\_(self, weight, value):

        self.weight = weight

        self.value = value

        self.ratio = value / weight

def knapsack\_branch\_and\_bound(items, capacity):

    items.sort(key=lambda x: x.ratio, reverse=True)  # Sort items by value-to-weight ratio in descending order

    n = len(items)

    # Helper function to calculate the upper bound of a node

    def upper\_bound(level, weight, value):

        if weight >= capacity:

            return 0

        upper\_bound\_value = value

        j = level + 1

        total\_weight = weight

        while j < n and total\_weight + items[j].weight <= capacity:

            total\_weight += items[j].weight

            upper\_bound\_value += items[j].value

            j += 1

        if j < n:

            remaining\_capacity = capacity - total\_weight

            upper\_bound\_value += items[j].ratio \* remaining\_capacity  # Corrected line

        return upper\_bound\_value

    max\_value = 0

    best\_items = []

    stack = [(0, 0, 0, [])]  # (level, weight, value, chosen\_items)

    while stack:

        level, weight, value, chosen\_items = stack.pop()

        if level >= n:

            if value > max\_value:

                max\_value = value

                best\_items = chosen\_items

        else:

            if weight + items[level].weight <= capacity:

                new\_weight = weight + items[level].weight

                new\_value = value + items[level].value

                new\_chosen\_items = chosen\_items + [items[level]]

                stack.append((level + 1, new\_weight, new\_value, new\_chosen\_items))

            if upper\_bound(level, weight, value) > max\_value:

                stack.append((level + 1, weight, value, chosen\_items))

    return max\_value, best\_items

# User input

n = int(input("Enter the number of items: "))

items = []

for i in range(n):

    value = int(input(f"Enter the value of item {i+1}: "))

    weight = int(input(f"Enter the weight of item {i+1}: "))

    items.append(Item(weight, value))

capacity = int(input("Enter the capacity of the knapsack: "))

# Solve the knapsack problem using Branch and Bound

max\_value, best\_items = knapsack\_branch\_and\_bound(items, capacity)

# Print the result

print("Optimal Solution:")

print("Maximum Value:", max\_value)

print("Selected Items:")

for item in best\_items:

    print("Weight:", item.weight, "Value:", item.value)

**Travel Salesman Problem:**

import math

maxsize = float('inf')

def copyToFinal(curr\_path):

    final\_path[:N + 1] = curr\_path[:]

    final\_path[N] = curr\_path[0]

def firstMin(adj, i):

    min = maxsize

    for k in range(N):

        if adj[i][k] < min and i != k:

            min = adj[i][k]

    return min

def secondMin(adj, i):

    first, second = maxsize, maxsize

    for j in range(N):

        if i == j:

            continue

        if adj[i][j] <= first:

            second = first

            first = adj[i][j]

        elif(adj[i][j] <= second and

            adj[i][j] != first):

            second = adj[i][j]

    return second

def TSPRec(adj, curr\_bound, curr\_weight,

            level, curr\_path, visited):

    global final\_res

    if level == N:

        if adj[curr\_path[level - 1]][curr\_path[0]] != 0:

            curr\_res = curr\_weight + adj[curr\_path[level - 1]]\

                                        [curr\_path[0]]

            if curr\_res < final\_res:

                copyToFinal(curr\_path)

                final\_res = curr\_res

        return

    for i in range(N):

        if (adj[curr\_path[level-1]][i] != 0 and

                            visited[i] == False):

            temp = curr\_bound

            curr\_weight += adj[curr\_path[level - 1]][i]

            if level == 1:

                curr\_bound -= ((firstMin(adj, curr\_path[level - 1]) +

                                firstMin(adj, i)) / 2)

            else:

                curr\_bound -= ((secondMin(adj, curr\_path[level - 1]) +

                                firstMin(adj, i)) / 2)

            if curr\_bound + curr\_weight < final\_res:

                curr\_path[level] = i

                visited[i] = True

                TSPRec(adj, curr\_bound, curr\_weight,

                    level + 1, curr\_path, visited)

            curr\_weight -= adj[curr\_path[level - 1]][i]

            curr\_bound = temp

            visited = [False] \* len(visited)

            for j in range(level):

                if curr\_path[j] != -1:

                    visited[curr\_path[j]] = True

def TSP(adj):

    curr\_bound = 0

    curr\_path = [-1] \* (N + 1)

    visited = [False] \* N

    for i in range(N):

        curr\_bound += (firstMin(adj, i) +

                    secondMin(adj, i))

    curr\_bound = math.ceil(curr\_bound / 2)

    visited[0] = True

    curr\_path[0] = 0

    TSPRec(adj, curr\_bound, 0, 1, curr\_path, visited)

adj = [[0, 10, 15, 20],

       [10, 0, 35, 25],

       [15, 35, 0, 30],

       [20, 25, 30, 0]]

N = 4

final\_path = [None] \* (N + 1)

visited = [False] \* N

final\_res = maxsize

TSP(adj)

print("Minimum cost :", final\_res)

print("Path Taken : ", end = ' ')

for i in range(N + 1):

    print(final\_path[i], end = ' ')

**Sample input and output:**

**Knapsack Problem:**

A screenshot of a computer

Description automatically generated with medium confidence

**Travel Salesman Problem:**

Input:

A picture containing font, line, text, screenshot

Description automatically generated

Output:

A picture containing text, font, screenshot, line

Description automatically generated

**Result:**

Thus, an Approximation algorithm in Python to solve Knapsack Problem and Travel Salesman Problem has been successfully implemented and the output is verified.