**Practical No : 9**

**Name :** Samrudhi Pawar

**Roll No :** 44

**Title :**Develop a program to implement 0/1 Knapsack problem using branch and bound.

**Program :**

// C++ program to solve knapsack problem using

// branch and bound

#include <bits/stdc++.h>

using namespace std;

// Structure for Item which store weight and corresponding

// value of Item

struct Item

{

float weight;

int value;

};

// Node structure to store information of decision

// tree

struct Node

{

// level --> Level of node in decision tree (or index

// in arr[]

// profit --> Profit of nodes on path from root to this

// node (including this node)

// bound ---> Upper bound of maximum profit in subtree

// of this node/

int level, profit, bound;

float weight;

};

// Comparison function to sort Item according to

// val/weight ratio

bool cmp(Item a, Item b)

{

double r1 = (double)a.value / a.weight;

double r2 = (double)b.value / b.weight;

return r1 > r2;

}

// Returns bound of profit in subtree rooted with u.

// This function mainly uses Greedy solution to find

// an upper bound on maximum profit.

int bound(Node u, int n, int W, Item arr[])

{

// if weight overcomes the knapsack capacity, return

// 0 as expected bound

if (u.weight >= W)

return 0;

// initialize bound on profit by current profit

int profit\_bound = u.profit;

// start including items from index 1 more to current

// item index

int j = u.level + 1;

int totweight = u.weight;

// checking index condition and knapsack capacity

// condition

while ((j < n) && (totweight + arr[j].weight <= W))

{

totweight += arr[j].weight;

profit\_bound += arr[j].value;

j++;

}

// If k is not n, include last item partially for

// upper bound on profit

if (j < n)

profit\_bound += (W - totweight) \* arr[j].value /

arr[j].weight;

return profit\_bound;

}

// Returns maximum profit we can get with capacity W

int knapsack(int W, Item arr[], int n)

{

// sorting Item on basis of value per unit

// weight.

sort(arr, arr + n, cmp);

// make a queue for traversing the node

queue<Node> Q;

Node u, v;

// dummy node at starting

u.level = -1;

u.profit = u.weight = 0;

Q.push(u);

// One by one extract an item from decision tree

// compute profit of all children of extracted item

// and keep saving maxProfit

int maxProfit = 0;

while (!Q.empty())

{

// Dequeue a node

u = Q.front();

Q.pop();

// If it is starting node, assign level 0

if (u.level == -1)

v.level = 0;

// If there is nothing on next level

if (u.level == n - 1)

continue;

// Else if not last node, then increment level,

// and compute profit of children nodes.

v.level = u.level + 1;

// Taking current level's item add current

// level's weight and value to node u's

// weight and value

v.weight = u.weight + arr[v.level].weight;

v.profit = u.profit + arr[v.level].value;

// If cumulated weight is less than W and

// profit is greater than previous profit,

// update maxprofit

if (v.weight <= W && v.profit > maxProfit)

maxProfit = v.profit;

// Get the upper bound on profit to decide

// whether to add v to Q or not.

v.bound = bound(v, n, W, arr);

// If bound value is greater than profit,

// then only push into queue for further

// consideration

if (v.bound > maxProfit)

Q.push(v);

// Do the same thing, but Without taking

// the item in knapsack

v.weight = u.weight;

v.profit = u.profit;

v.bound = bound(v, n, W, arr);

if (v.bound > maxProfit)

Q.push(v);

}

return maxProfit;

}

// driver program to test above function

int main()

{

int W = 10; // Weight of knapsack

Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100}, {5, 95}, {3, 30}};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Maximum possible profit = "

<< knapsack(W, arr, n);

return 0;

}

/\* Output:

Maximum possible profit = 235

\*/