**Experiment No . 05: Branch & Bound Algorithm**

**DATE :- 08-05-2023**

**Aim:-** Write C program to implement the following using Branch and Bound Algorithm.

1. 0/1 Knapsack

**Theory**

* The Branch and Bound algorithm are used to solve optimization problems, particularly combinatorial optimization problems.
* It involves dividing the problem into smaller subproblems and constructing a search tree.
* The algorithm starts with an initial solution and initializes upper and lower bounds.
* The search process begins by exploring the search tree using a depth-first search (DFS) or breadth-first search (BFS) strategy.
* At each node of the search tree, the algorithm evaluates the bounds to determine if further exploration is warranted.
* If the lower bound of a node is higher than the current best solution, the subtree rooted at that node is pruned.
* If the upper bound of a node is lower than the current best solution, the algorithm backtracks to the parent node.
* At each node, a decision is made to include or exclude a particular element or constraint from the solution.
* The current solution and the bounds are updated accordingly.
* The algorithm continues exploring the search tree until all nodes have been visited or until the best solution is found.
* The final solution obtained is the optimal solution to the optimization problem.

1. **0/1 Knapsack**

**DATE :- 08-05-2023**

**Problem Statement**

Solve the 0/1 knapsack problem for the knapsack instance n = 4 and m = 35

(w1 . . . . w4) = (14, 17, 15, 13) (p1 . . . . . p4) = (80, 82, 45, 42)

**Algorithm**

Algorithm **UBound(cp,cw,k,m)**

//w[i] and p[i] are respectively the weight and profit of the ith object .

{

B:= cp ;

C:= cw;

For i:= k+1 to n do

{

If(c+w[i] m) then

{

c:=c + w[i] ;

b:=b-p[i];

}

}

return b;

}

Algorithm **bnb (level ,w,p)**

//bestp->best profit so far

{

If(w m )then

{

bestp := p;

copy elements from currentsol to bestsol

}

If((level =n ) or (ub (level , w ,p ) bestp ))

{

return;

}

Set currentsol[level] =1;

Bnb(level +1 , w+weight[level],p+profit[level])

Set currentsol[level] = 0;

Bnb(level +1,w,p)

}

**Time and Space Complexity**

Time Complexity = **O(2n + n2)**

Space complexity = **O(n)**

Where , n is the number of Elements .

**Program:**

#include <stdio.h>

#define N 100

struct ITEM

{

    int w, p;

};

int m, n, currentsol[N], bestsol[N], cw, cp, bestp;

struct ITEM items[N];

void fillz()

{

    for (int i = 0; i < n; i++)

        currentsol[i] = bestsol[i] = 0;

}

int ub(int level, int w, int p)

{

    int r = m - w; // remaining knapsack cap

    int upperbound = p;

    for (int i = level; i < n; i++)

    {

        if (items[i].w <= r)

        {

            r -= items[i].w;

            upperbound += items[i].p;

        }

        else

        {

            upperbound += (r / items[i].w) \* items[i].p; // fractional weight

            break;

        }

    }

    return upperbound;

}

void \_sort()

{

    for (int i = 0; i < n; i++)

    {

        for (int j = 0; j < n - i - 1; j++)

        {

            if ((items[j].p / items[j].w) < (items[j + 1].p / items[j + 1].w))

            {

                int tempp = items[j].p, tempw = items[j].w;

                items[j].p = items[j + 1].p, items[j].w = items[j + 1].w;

                items[j + 1].p = tempp, items[j + 1].w = tempw;

            }

        }

    }

}

void bnb(int level, int w, int p)

{

    if (w <= m) // current weight is less than the knapsack cap

    {

        if (p > bestp)

        {

            bestp = p;

            for (int i = 0; i < n; i++)

                bestsol[i] = currentsol[i];

        }

    }

    if (level == n || ub(level, w, p) <= bestp) // if all the items are exhausted or the upper bound of the subtree is less than the current best p

        return;

    currentsol[level] = 1;

    bnb(level + 1, w + items[level].w, p + items[level].p); // including the item at the current level

    currentsol[level] = 0;

    bnb(level + 1, w, p); // exclusing the item at the current level

}

int main()

{

    printf("Enter knapsack capacity: ");

    scanf("%d", &m);

    printf("Enter the number of items: ");

    scanf("%d", &n);

    printf("Enter the profit and weight for each item\n");

    for (int i = 0; i < n; i++) {

        scanf("%d", &items[i].p);

        scanf("%d", &items[i].w);

    }

    cw = cp = bestp = 0;

    fillz();

    \_sort();

    bnb(0, cw, cp);

    printf("ouput in the form of p/w decendig order\n");

    printf("Knapsack Solution\nMax Profit = %d\nItems Included: ", bestp);

    for (int i = 0; i < n; i++)

        printf(" %d", bestsol[i]);

    printf("\n");

    return 0;

}

**Output:**

C:\P Jeevesh Naidu\college\second year\lV sem\madf codes>cd "c:\P Jeevesh Naidu\college\second year\lV sem\madf codes\" && gcc tempCodeRunnerFile.c -o tempCodeRunnerFile && "c:\P Jeevesh Naidu\college\second year\lV sem\madf codes\"tempCodeRunnerFile

Enter knapsack capacity: 12

Enter the number of items: 4

Enter the profit and weight for each item

30 5

28 7

20 4

24 2

ouput in the form of p/w decendig order

Knapsack Solution

Max Profit = 74

Items Included: 1 1 1 0

**Conclusion:**

Branch & Bound algorithm was studied . The program for 0/1 Knapsack was studied and implemented successfully.