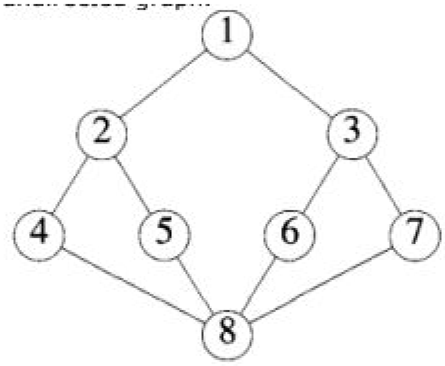
LAB 7 : BRANCH AND BOUND

Date : 18.4.18

Submitted By : Abhineet Singh

1. ***Write a program to implement a Breadth first search (BFS) Traversal Algorithm for an undirected graph.***




```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

void BFS(vector< vector<int> > A, int start, int N, int *levels) {
	int flag = 0, k = 0;
	queue <int> Q;
	vector<bool> visited(N, false);
	visited[start] = true;
	Q.push(start);
	while (!Q.empty()) {
	int z = Q.front();
	cout << z << " ";
	Q.pop();
	vector<int> :: iterator i;
	if (z == start) {
	levels[z] = 0;
	}
	for ( i = A[z].begin(); i != A[z].end(); ++i) {
	if (visited[*i] == false) {
```

```
            visited[*i] = true;
            Q.push(*i);
            levels[*i] = levels[z] + 1;
        }
        }
        }
}

        int main() {
        int N, M;
        cout<<"enter the number of nodes"<<endl;
        cin >> N;
        cout<<"enter the number of edges"<<endl;
        cin >> M;
        cout<<"enter the nodes which have an edge between them"<<endl;
        vector< vector<int> > A(10);
        int u, v, m;
        for (int i = 0; i < M; ++i) {
        cin >> u >> v;
        m = max(u, v);
        A[u].push_back(v);
        A[v].push_back(u);
        }
        int levels[m + 1];
        for (int i = 1; i <= m; ++i) {
        levels[i] = 0;
        }
        int start;
        cout << "Enter start vertex: ";
        cin >> start;
        cout<<"the bfs of the graph is"<<endl;
        BFS(A, start, N, levels);
        cout << endl;
        cout<<"the corresponding levels are"<<endl;
        for (int i = 1; i <= m; ++i) {
        cout << levels[i] << " ";
        }
        return 0;
}
```

```
iiitd009@iiitd009-HP-406-G1-MT: ~/Desktop
iiitd009@iiitd009-HP-406-G1-MT:~/Desktop$ g++ bfs.cpp
iiitd009@iiitd009-HP-406-G1-MT:~/Desktop$ ./a.out
enter the number of nodes
8
enter the number of edges
10
enter the nodes which have an edge between them
1 2
1 3
2 4
2 5
3 6
3 7
4 8
5 8
6 8
7 8
Enter start vertex: 1
the bfs of the graph is
1 2 3 4 5 6 7 8
the corresponding levels are
0 1 1 2 2 2 2 3 iiitd009@iiitd009-HP-406-G1-MT:~/Desktop$
```

2. ***Write a program to Implement a 0/1 knapsack problem using Branch and Bound Algorithm for the knapsack instance n=4, (p1 p2 p3 p4) =(10, 10, 12, 18), (w1 w2 w3 w4)=(2 4 5 9), and m=15.***

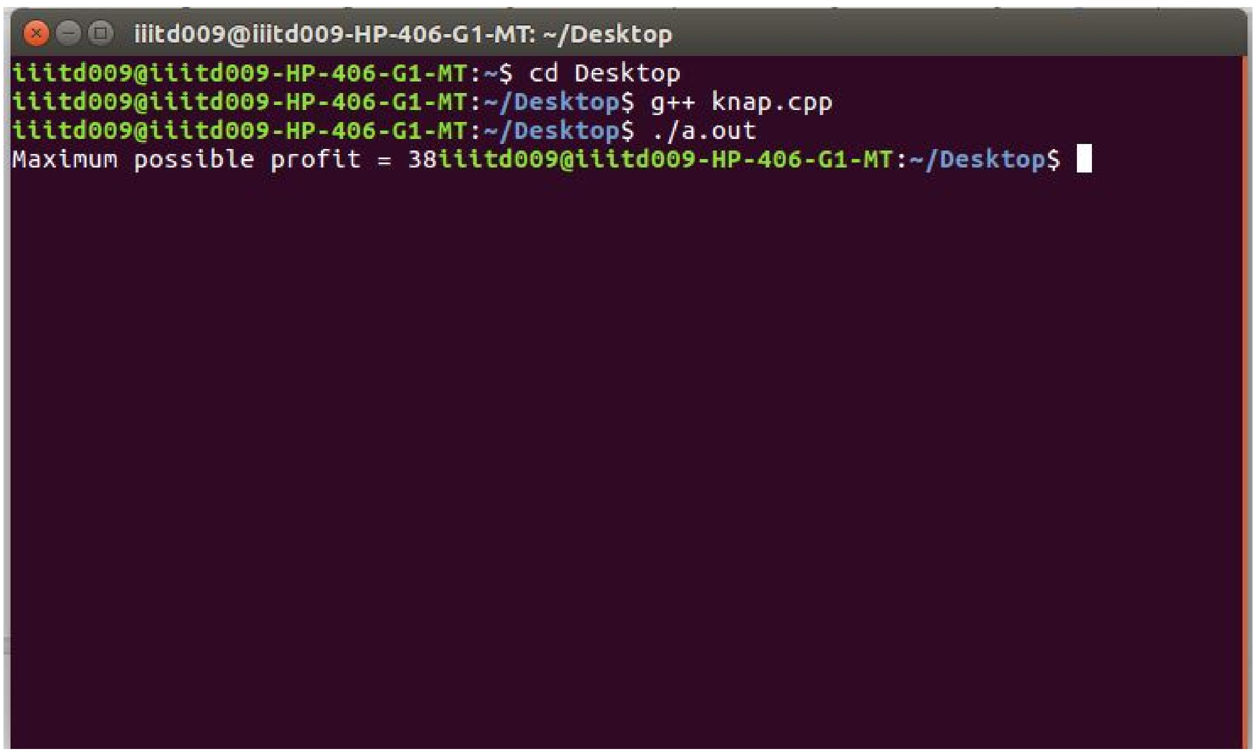
```
// C++ program to solve knapsack problem using
// branch and bound
#include <bits/stdc++.h>
using namespace std;

// Stucture 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 = 15;   // Weight of knapsack
    Item arr[] = {{2, 10}, {4, 10}, {5, 12},
                  {9,18}};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Maximum possible profit = "
         << knapsack(W, arr, n);
```

```
    return 0;
}
```



iiitd009@iiitd009-HP-406-G1-MT: ~/Desktop
iiitd009@iiitd009-HP-406-G1-MT:~$ cd Desktop
iiitd009@iiitd009-HP-406-G1-MT:~/Desktop$ g++ knap.cpp
iiitd009@iiitd009-HP-406-G1-MT:~/Desktop$ ./a.out
Maximum possible profit = 38iiitd009@iiitd009-HP-406-G1-MT:~/Desktop$