Name : Sonal Dilip Gholap

UID : 2021600020

Branch : CSE-AIML

Batch : B

Experiment : 8

Aim : Experiment on Branch and Bound Algorithm.

- **Objectives :**

1. To study branch and bound algorithms like 0/1 knapsack problem and travelling salesman problem.
2. To implement 0/1 knapsack problem and travelling salesman problem using branch and bound.
3. To derive and analyze the time complexity of 0/1 knapsack problem and travelling salesman problem.

- **Theory :**

1. Branch and bound is a well-known algorithmic technique used for solving optimization problems.
2. It works by dividing the problem into smaller subproblems, solving each subproblem independently, and then combining their solutions to obtain the solution to the original problem.
3. The "branch" part of the algorithm refers to the process of dividing the problem into smaller subproblems.
4. This is typically done by generating a set of candidate solutions, and then selecting one of them to further explore. The selected solution is modified or "branched" to create new subproblems, which are then explored in a similar way.
5. The "bound" part of the algorithm refers to the process of evaluating the subproblems to determine whether they are worth exploring further.
6. This involves calculating an upper bound on the potential value of the subproblem's solution, as well as a lower bound on its actual value.
7. If the upper bound is less than the current best solution, the subproblem can be discarded without further exploration. Similarly, if the lower bound is greater than the current best solution, the subproblem's solution can be immediately accepted.

8. Overall, branch and bound is a powerful optimization technique that can be used to solve a wide range of problems.
9. Its key strengths are its ability to handle large and complex problems, and its ability to quickly prune away unproductive search paths to focus on the most promising solutions.

- **0/1 knapsack problem**

Pseudocode for 0/1 knapsack problem

bound(u, n, W, arr):

```
if u.weight >= W:
    return 0

profit_bound = u.profit
j = u.level + 1
total_weight = u.weight

while j < n and total_weight + arr[j].weight <= W:
    total_weight += arr[j].weight
    profit_bound += arr[j].value
    j += 1

if j < n:
    profit_bound += (W - total_weight) * (arr[j].value / arr[j].weight)

return profit_bound
```

knapsack(W, arr, n):

```
    sort arr in non-increasing order by value/weight ratio
    queue Q
    node u
    u.level = -1
```

```
u.profit = u.weight = 0;
max_profit = 0
Q.push(u)

while Q is not empty do
        u = Q.front()
        Q.pop()

        if u is a leaf node then
                continue

        node v
        v.level = u.level + 1
        v.weight = u.weight + arr[v.level].weight
        v.profit = u.profit + arr[v.level].value

        if v.weight <= W and v.profit > max_profit then
                max_profit = v.profit

        bound_value = bound(v, n, W, arr)
        if bound_value > max_profit then
                Q.push(v)

        v.weight = u.weight
        v.profit = u.profit

return max_profit
```

Derivation of Time Complexity

The worst case time complexity of 0/1 Knapsack problem using branch and bound is $O(2^n)$, where n is the number of items. The worst-case scenario occurs when all possible nodes are explored. The number of nodes in this case is 2^n, since there are 2 choices for each item: either include it or exclude it from the knapsack. The best-case time complexity O(n) as only one path through the tree will have to be traversed in the best case.

Solution :

PAGE No.
DATE / /

Branch and Bound

I) 0/1 Knapsack Problem

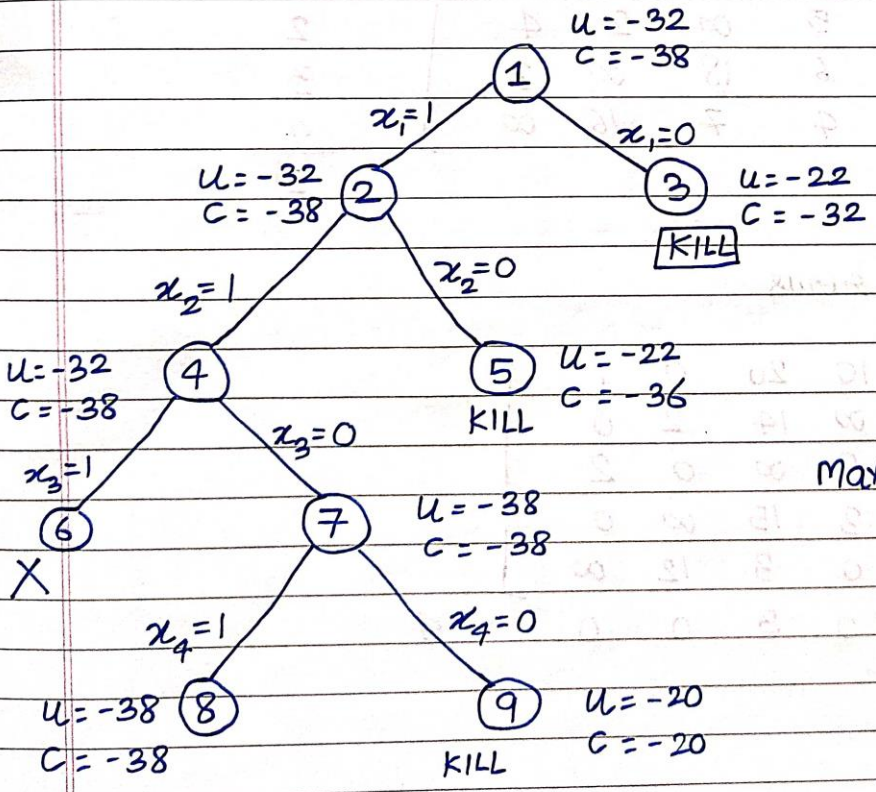
	1	2	3	4
Profit	10	10	12	18
Weight	2	4	6	9

$M = 15$

$n = 4$

Upper Bound = $\sum_{i=1}^{n} p_i x_i \leq M$

Cost = $\sum_{i=1}^{n} p_i x_i$ (with fraction)



Upper Bound
~~∞~~ ~~-32~~
-38

x {1, 1, 0, 1}

Max profit = 10 + 10 + 18
= 38

Code :

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

using namespace std;

struct Item {
    int weight;
    int value;
};

struct Node {
    int level;
    int profit;
    int weight;
};

bool cmp(Item a, Item b) {
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

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

    // If the node's weight exceeds the maximum weight, then return 0 as it is not feasible
    if (u.weight >= W) {
        return 0;
    }
    int profit_bound = u.profit;   // Calculate the profit so far

    // Start from the next level and keep adding items to the knapsack until it reaches the
maximum weight
    int j = u.level + 1;
    int total_weight = u.weight;
    while ((j < n) && (total_weight + arr[j].weight <= W)) {
        total_weight += arr[j].weight;
        profit_bound += arr[j].value;
        j++;
    }

    // If there are still items remaining, add a fractional amount of the next item to fill
the knapsack
    if (j < n) {
        profit_bound += (W - total_weight) * ((double)arr[j].value / arr[j].weight);
    }
    return profit_bound;
}
```

```
int knapsack(int W, Item arr[], int n) {
    sort(arr, arr + n, cmp);
    queue<Node> Q;
    Node u, v;
    u.level = -1;
    u.profit = u.weight = 0;
    int max_profit = 0;
    Q.push(u);
    while (!Q.empty()) {  // Traverse the search tree in BFS order
        u = Q.front();
        Q.pop();
        // If the node represents a leaf node or its upper bound is less than the current
maximum profit, skip it
        if (u.level == n-1) {
            continue;
        }
        v.level = u.level + 1;   // Create the child node where the next item is added to the
knapsack
        v.weight = u.weight + arr[v.level].weight;
        v.profit = u.profit + arr[v.level].value;

        // If the child node is feasible and its profit is greater than the current maximum
profit, update the maximum profit
        if (v.weight <= W && v.profit > max_profit) {
            max_profit = v.profit;
        }

        // Calculate the upper bound of profit for the child node and if it is greater than
the current maximum profit, add it to the queue
        int bound_value = bound(v, n, W, arr);
        if (bound_value > max_profit) {
            Q.push(v);
        }
        v.weight = u.weight;
        v.profit = u.profit;
        bound_value = bound(v, n, W, arr);
        if (bound_value > max_profit) {
            Q.push(v);
        }
    }
    return max_profit;
}

int main() {
    int n, W;
    cout << "Enter the number of items: ";
    cin >> n;
    Item arr[n];
    cout << "Enter the weight and value of each item:\n";
```

```
    for (int i = 0; i < n; i++) {
        cout << "Item " << i+1 << ": ";
        cin >> arr[i].weight >> arr[i].value;
    }
    cout << "Enter the maximum weight of the knapsack: ";
    cin >> W;
    cout << "Maximum value we can obtain : " << knapsack(W, arr, n) << endl;
    return 0;
}
```

Output :

```
PS C:\C++ Learning Course>  & 'c:\Users\Sonal Dilip Gholap\.vscode\extensions\ms-vscode.cpptoo
ls-1.15.2-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In
-o0zvs2gs.bh5' '--stdout=Microsoft-MIEngine-Out-oupraeiz.z2b' '--stderr=Microsoft-MIEngine-Err
or-gnew0vjm.b5v' '--pid=Microsoft-MIEngine-Pid-am2akm1b.nat' '--dbgExe=C:\Program Files (x86)\
mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\gdb.exe' '--interpreter=mi'
Enter the number of items: 4
Enter the weight and value of each item:
Item 1: 2 10
Item 2: 4 10
Item 3: 6 12
Item 4: 9 18
Enter the maximum weight of the knapsack: 15
Maximum value we can obtain : 38
PS C:\C++ Learning Course>
```

- **Travelling salesman problem**

Pseudocode for travelling salesman problem

tsp_bb(u, level, cost, path, visited):

```
if cost >= minCost:
    return

if level == N:
    cost += dist[u][0]
```

```
        if cost < minCost:
            minCost = cost
            minPath = path
        return

    for v = 1 to N - 1:
        if visited[v] == false:
            visited[v] = true
            path.push_back(v)
            tsp_bb(v, level + 1, cost + dist[u][v], path, visited)
            visited[v] = false
            path.pop_back()
```

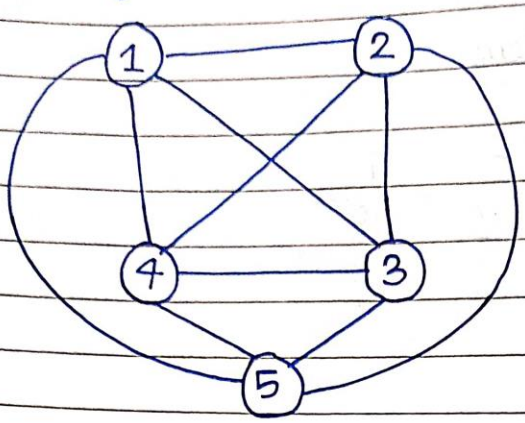
Derivation of Time Complexity

The time complexity of the Travelling Salesman Problem (TSP) using branch and bound algorithm is generally given by O(n^2 * 2^n). In the worst case, we need to generate all possible permutations of the n nodes, and each permutation requires O(n) time to calculate the cost of the path. Thus, the overall time complexity is O(n * n!) or O(n^2 * 2^n).

Solution :

DATE / /

II) Travelling Salesman Problem



Adjacency matrix :

	1	2	3	4	5	min
1	∞	20	30	10	11	10
2	15	∞	16	4	2	2
3	3	5	∞	2	4	2
4	19	6	18	∞	3	3
5	16	4	7	16	∞	4
						21

Reducing rows

	∞	10	20	0	1
	13	∞	14	2	0
	1	3	∞	0	2
	16	3	15	∞	0
	12	0	3	12	∞
min	1	0	3	0	0

= 4

DATE / /

Reducing columns →

∴ Reduced matrix →

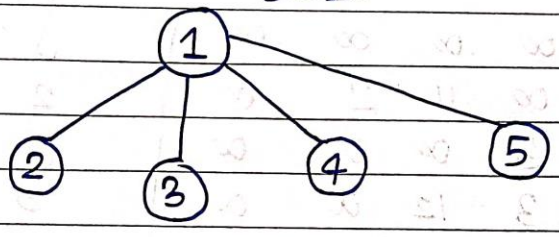
∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

Reduced cost = 21 + 4

r = 25

C = 25

Upper bound = ∞



For 1 → 2

∞	∞	∞	∞	∞
∞	∞	11	2	0
0	∞	∞	0	2
15	∞	12	∞	0
11	∞	0	12	∞

$C = C(1,2) + r + \hat{r}$

$= 10 + 25 + 0$

$= 35$

For 1 → 3

∞	∞	∞	∞	∞
12	∞	∞	2	0
∞	3	∞	0	2
15	3	∞	∞	0
11	0	∞	12	∞
min 11	0	0	0	0 = 11

Reduced →

∞	∞	∞	∞	∞
1	∞	∞	2	0
∞	3	∞	0	2
4	3	∞	∞	0
0	0	∞	12	∞

$C = C(1,3) + r + \hat{r} = 17 + 25 + 11 = 53$

For 1,4

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

$C = c(1,4) + r + \hat{r}$

$= 0 + 25 + 0$

$= 25$

For 1,5

					min
∞	∞	∞	∞	∞	
12	∞	11	2	∞	0
0	3	∞	0	∞	2
15	3	12	∞	∞	0
∞	0	0	12	∞	3
					0
					5

↓ Reducing rows

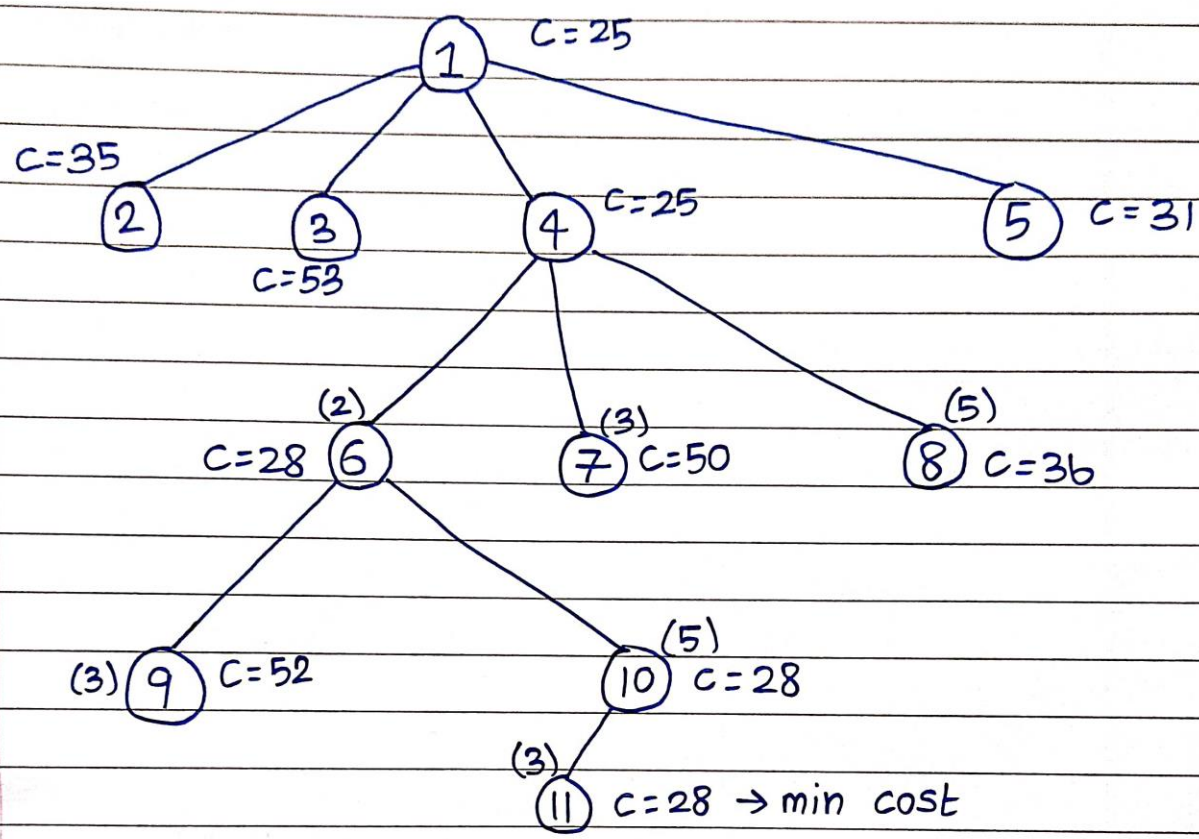
∞	∞	∞	∞	∞
10	∞	9	0	∞
0	3	∞	0	∞
12	0	9	∞	∞
∞	0	0	12	∞

$C = c(1,5) + r + \hat{r}$

$= 1 + 25 + 5$

$= 31$

State space tree :



Update upper bound to 28 & kill all nodes with cost > 28

Minimum cost = 28

Path of the tour = 1 → 4 → 2 → 5 → 3

Code :

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

```cpp
using namespace std;

const int MAXN = 15;
const int INF = 1e9;

int N;
int dist[MAXN][MAXN];
int minCost = INF;
vector<int> minPath;

void tsp_bb(int u, int level, int cost, vector<int> &path, vector<bool> &visited) {
    // If the current cost is already greater than or equal to the minimum cost found so far, stop searching further
    if (cost >= minCost) {
        return;
    }

    // If we have visited all the nodes, update the minimum cost and path if the current cost is less
    if (level == N) {
        cost += dist[u][0];
        if (cost < minCost) {
            minCost = cost;
            minPath = path;
        }
        return;
    }

    for (int v = 1; v < N; v++) {
        if (!visited[v]) {
            visited[v] = true;
            path.push_back(v);
            tsp_bb(v, level + 1, cost + dist[u][v], path, visited);
            visited[v] = false;
            path.pop_back();
        }
    }
}

int main() {

    cout << "Enter number of cities: ";
    cin >> N;

    cout << "Enter adjacency matrix: \n";
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            cin >> dist[i][j];
        }
    }
```

```
    vector<int> path = {0};
    vector<bool> visited(N, false);
    visited[0] = true;
    tsp_bb(0, 1, 0, path, visited);

    cout << "Minimum Cost: " << minCost << endl;
    cout << "Path: ";
    for (int i = 0; i < minPath.size(); i++) {
        cout << minPath[i] << "->";
    }
    cout << "0" << endl;

    return 0;
}
```

Output :

```
PS C:\C++ Learning Course>  & 'c:\Users\Sonal Dilip Gholap\.vscode\extensions\ms-vscode.cpptoo
ls-1.15.2-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In
-aahkqbrv.lmf' '--stdout=Microsoft-MIEngine-Out-zlwxdnxr.v3f' '--stderr=Microsoft-MIEngine-Err
or-bzpfycm5.wd1' '--pid=Microsoft-MIEngine-Pid-bd1b1zee.yw0' '--dbgExe=C:\Program Files (x86)\
mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\gdb.exe' '--interpreter=mi'
Enter number of cities: 5
Enter adjacency matrix:
0 20 30 10 11
15 0 16 4 2
3 5 0 2 4
19 6 18 0 3
16 4 7 16 0
Minimum Cost: 28
Path: 0->3->1->4->2->0
PS C:\C++ Learning Course>
```

Result :

1. Time complexity of 0/1 knapsack problem – **O(2^n)**
2. Time complexity of travelling salesman problem – **O(n!)**

Conclusion :

1. Branch and bound can be a highly efficient algorithm for solving large-scale optimization problems like 0/1 Knapsack and TSP.
2. Branch and bound reduces the search space by systematically eliminating large parts of the search space that do not contain optimal solutions.
3. Branch and bound uses a divide-and-conquer strategy to solve complex problems by breaking them down into smaller subproblems.