**Task 13: Branch and Bound – Traveling Salesman Problem**

**Aim:**

Create to C program to implement Traveling Salesman Problem using Branch and Bound algorithm.

**Algorithm**:

**Step1.** Input the first node as root. Identify the root, if there is still a node to expand then do the

branching if there is no more node (empty) then the search is done.

**Step2.** From the branch, choose the node (*i*) with the minimum cost. If exist several nodes with the same cost, then choose arbitrarily.

**Step3.** Identify the node (*i*),

**Step 3.1** If the (*i*) is the solution –no more queue node– then stop the search,

**Step 3.2** If the (*i*) is not the solution, then do the branching and choose the node (*i*) with the minimum cost. If exist several nodes with the same cost, then choose arbitrarily.

**Step 4.** Back to step 3.

**Program:**

#include<stdio.h>

#include<conio.h>

int a[10][10], visited[10], n, cost = 0;

void get()

{

int i, j;

printf("Enter No. of Cities: ");

scanf("%d", &n);

printf("\nEnter Cost Matrix: \n");

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

{

printf("\n Enter Elements of Row# : %d\n", i + 1);

for (j = 0; j < n; j++)

scanf("%d", &a[i][j]);

visited[i] = 0;

}

printf("\n\nThe cost list is:\n\n");

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

{

printf("\n\n");

for (j = 0; j < n; j++)

printf("\t % d", a[i][j]);

}

}

void mincost(int city)

{

int i, ncity;

visited[city] = 1;

printf("%d –>", city + 1);

ncity = least(city);

if (ncity == 999)

{

ncity = 0;

printf("%d", ncity + 1);

cost += a[city][ncity];

return;

}

mincost(ncity);

}

int least(int c)

{

int i, nc = 999;

int min = 999, kmin;

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

{

if ((a[c][i] != 0) && (visited[i] == 0))

if (a[c][i] < min)

{

min = a[i][0] + a[c][i];

kmin = a[c][i];

nc = i;

}

}

if (min != 999)

cost += kmin;

return nc;

}

void put()

{

printf("\n\nMinimum cost:");

printf("%d", cost);

}

void main()

{

get();

printf("\n\nThe Path is:\n\n");

mincost(0);

put();

}

**Output:**

Enter No. of Cities: 6

Enter Cost Matrix:

99 10 15 20 99 8

5 99 9 10 8 99

6 13 99 12 99 5

8 8 9 99 6 99

99 10 99 6 99 99

10 99 5 99 99 99

Enter Elements of Row# : 1

Enter Elements of Row# : 2

Enter Elements of Row# : 3

Enter Elements of Row# : 4

Enter Elements of Row# : 5

Enter Elements of Row# : 6

The Path is:

1 –>6 –>3 –>4 –>5 –>2 –>1

Minimum cost: 46

**Task 13: Branch and Bound – Traveling Salesman Problem**

**Test Case 1:** Implement Job Assignment Problem by applying the concept of branch and bound.

**Problem Description:**

Let there be N workers and N jobs in ABC Pvt. Ltd. Here, any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.

**Procedure:**

There are two approaches to calculate the cost function:

1. For each worker, we choose job with minimum cost from list of unassigned jobs (take minimum entry from each row).
2. For each job, we choose a worker with lowest cost for that job from list of unassigned workers (take minimum entry from each column).

**Algorithm/ Pseudo code:**

/\* findMinCost uses Least() and Add() to maintain the

list of live nodes

Least() finds a live node with least cost, deletes

it from the list and returns it

Add(x) calculates cost of x and adds it to the list

of live nodes

Implements list of live nodes as a min heap \*/

// Search Space Tree Node

node

{

int job\_number;

int worker\_number;

node parent;

int cost;

}

// Input: Cost Matrix of Job Assignment problem

// Output: Optimal cost and Assignment of Jobs

algorithm findMinCost (costMatrix mat[][])

{

// Initialize list of live nodes(min-Heap)

// with root of search tree i.e. a Dummy node

while (true)

{

// Find a live node with least estimated cost

E = Least();

// The found node is deleted from the list

// of live nodes

if (E is a leaf node)

{

printSolution();

return;

}

for each child x of E

{

Add(x); // Add x to list of live nodes;

x->parent = E; // Pointer for path to root

}

}

}

**Result:**

Thus the Traveling Salesman Problem using Branch and Bound algorithm was executed successfully.

**Task 14: Branch and Bound – Knapsack Algorithm**

**Aim:**

Create to C program to implement Algorithm using Branch and Bound algorithm.

**Algorithm**:

**Step1:** Sort all items in decreasing order of ratio of value per unit weight so that an upper bound

can be computed using Greedy Approach.

**Step2:** Initialize maximum profit, maxProfit = 0

**Step3:** Create an empty queue, Q.

**Step4:** Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy

node are 0.

**Step5:** Do following while Q is not empty.

**5.1:** Extract an item from Q. Let the extracted item be u.

**5.2:** Compute profit of next level node. If the profit is more than maxProfit, then update

maxProfit.

**5.3:** Compute bound of next level node. If bound is more than maxProfit, then add next

level node to Q.

**5.4:** Consider the case when next level node is not considered as part of solution and add

a node to queue with level as next, but weight and profit without considering next

level nodes.

**Program:**

#inclue <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef enum { NO, YES } BOOL;

int N;

int vals[100];

int wts[100];

int cap = 0;

int mval = 0;

void getWeightAndValue (BOOL incl[N], int \*weight, int \*value)

{

int i, w = 0, v = 0;

for (i = 0; i < N; ++i)

{

if (incl[i])

{

w += wts[i];

v += vals[i];

}

}

\*weight = w;

\*value = v;

}

void printSubset (BOOL incl[N])

{

int i;

int val = 0;

printf("Included = { ");

for (i = 0; i < N; ++i)

{

if (incl[i])

{

printf("%d ", wts[i]);

val += vals[i];

}

}

printf("}; Total value = %d\n", val);

}

void findKnapsack (BOOL incl[N], int i)

{

int cwt, cval;

getWeightAndValue(incl, &cwt, &cval);

if (cwt <= cap)

{

if (cval > mval)

{

printSubset(incl);

mval = cval;

}

}

if (i == N || cwt >= cap)

{

return;

}

int x = wts[i];

BOOL use[N], nouse[N];

memcpy(use, incl, sizeof(use));

memcpy(nouse, incl, sizeof(nouse));

use[i] = YES;

nouse[i] = NO;

findKnapsack(use, i+1);

findKnapsack(nouse, i+1);

}

int main(int argc, char const \* argv[])

{

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

scanf(" %d", &N);

BOOL incl[N];

int i;

for (i = 0; i < N; ++i)

{

printf("Enter weight and value for element %d: ", i+1);

scanf(" %d %d", &wts[i], &vals[i]);

incl[i] = NO;

}

printf("Enter knapsack capacity: ");

scanf(" %d", &cap);

findKnapsack(incl, 0);

return 0;

}

**Sample input: 4**

1 15

5 10

3 9

4 5

**Knapsack Capacity:**

8

**Sample output:**

Included = { 1 }; Total value = 15

Included = { 1 5 }; Total value = 25

Included = { 1 3 4 }; Total value = 29