1. Graph coloring

#include<bits/stdc++.h>

usingnamespace std;

#defineV4

voidprintSolution(intcolor[]);

boolisSafe(boolgraph[V][V],intcolor[])

{

    for(inti=0;i<V;i++)

        for(intj=i+1;j<V;j++)

            if(graph[i][j]&&color[j]==color[i])

                returnfalse;

    returntrue;

}

boolgraphColoring(boolgraph[V][V],intm,inti,

                intcolor[V])

{

    if(i==V){

        if(isSafe(graph,color)){

            printSolution(color);

            returntrue;

        }

        returnfalse;

    }

    for(intj=1;j<=m;j++){

        color[i]=j;

        if(graphColoring(graph,m,i+1,color))

            returntrue;

        color[i]=0;

    }

    returnfalse;

}

voidprintSolution(intcolor[])

{

    cout<<"Solution Exists:"" Following are the assigned colors \n";

    for(inti=0;i<V;i++)

        cout<<"  "<<color[i];

    cout<<"\n";

}

intmain()

{

    boolgraph[V][V]={

        {0,1,1,1},

        {1,0,1,0},

        {1,1,0,1},

        {1,0,1,0},

    };

    intm=3;

    intcolor[V];

    for(inti=0;i<V;i++)

        color[i]=0;

    if(!graphColoring(graph,m,0,color))

        cout<<"Solution does not exist";

    return0;

}

1. Hamiltonian cycle in graph

#include<bits/stdc++.h>

usingnamespace std;

#defineV5

voidprintSolution(intpath[]);

boolisSafe(intv,boolgraph[V][V],

            intpath[],intpos)

{

    if(graph[path[pos-1]][v]==0)

        returnfalse;

    for(inti=0;i<pos;i++)

        if(path[i]==v)

            returnfalse;

    returntrue;

}

boolhamCycleUtil(boolgraph[V][V],

                  intpath[],intpos)

{

    if(pos==V)

    {

        if(graph[path[pos-1]][path[0]]==1)

            returntrue;

        else

            returnfalse;

    }

    for(intv=1;v<V;v++)

    {

        if(isSafe(v,graph,path,pos))

        {

            path[pos]=v;

            if(hamCycleUtil(graph,path,pos+1)==true)

                returntrue;

            path[pos]=-1;

        }

    }

    returnfalse;

}

boolhamCycle(boolgraph[V][V])

{

    int\*path=newint[V];

    for(inti=0;i<V;i++)

        path[i]=-1;

    path[0]=0;

    if(hamCycleUtil(graph,path,1)==false)

    {

        cout<<"\nSolution does not exist";

        returnfalse;

    }

    printSolution(path);

    returntrue;

}

voidprintSolution(intpath[])

{

    cout<<"Solution Exists:"

            " Following is one Hamiltonian Cycle \n";

    for(inti=0;i<V;i++)

        cout<<path[i]<<"";

    cout<<path[0]<<"";

    cout<<endl;

}

intmain()

{

    boolgraph1[V][V]={{0,1,0,1,0},

                        {1,0,1,1,1},

                        {0,1,0,0,1},

                        {1,1,0,0,1},

                        {0,1,1,1,0}};

    hamCycle(graph1);

    boolgraph2[V][V]={{0,1,0,1,0},

                         {1,0,1,1,1},

                         {0,1,0,0,1},

                         {1,1,0,0,0},

                         {0,1,1,0,0}};

    hamCycle(graph2);

    return0;

}

**Branch-and-Bound**

1. 15-puzzle problem

#include<iostream>

#defineN4

usingnamespace std;

intgetInvCount(intarr[])

{

    intinv\_count=0;

    for(inti=0;i<N\*N-1;i++)

    {

        for(intj=i+1;j<N\*N;j++)

        {

            if(arr[j]&&arr[i]&&arr[i]>arr[j])

                inv\_count++;

        }

    }

    returninv\_count;

}

intfindXPosition(intpuzzle[N][N])

{

    for(inti=N-1;i>=0;i--)

        for(intj=N-1;j>=0;j--)

            if(puzzle[i][j]==0)

                returnN-i;

}

boolisSolvable(intpuzzle[N][N])

{

    intinvCount=getInvCount((int\*)puzzle);

    if(N&1)

        return!(invCount&1);

    else

    {

        intpos=findXPosition(puzzle);

        if(pos&1)

            return!(invCount&1);

        else

            returninvCount&1;

    }

}

intmain()

{

    intpuzzle[N][N]=

    {

        {12,1,10,2},

        {7,11,4,14},

        {5,0,9,15},

        {8,13,6,3},

    };

    isSolvable(puzzle)?cout<<"Solvable":

                        cout<<"Not Solvable";

    return0;

}

1. LC Branch-and-bound job sequencing problem

// Program to solve Job Assignment problem

// using Branch and Bound

#include <bits/stdc++.h>

**usingnamespace**std;

#define N 4

// state space tree node

**struct**Node

{

    // stores parent node of current node

    // helps in tracing path when answer is found

    Node\* parent;

    // contains cost for ancestors nodes

    // including current node

**int**pathCost;

    // contains least promising cost

**int**cost;

    // contain worker number

**int**workerID;

    // contains Job ID

**int**jobID;

    // Boolean array assigned will contains

    // info about available jobs

**bool**assigned[N];

};

// Function to allocate a new search tree node

// Here Person x is assigned to job y

Node\* newNode(**int**x, **int**y, **bool**assigned[],

              Node\* parent)

{

    Node\* node = **new**Node;

**for**(**int**j = 0; j < N; j++)

        node->assigned[j] = assigned[j];

    node->assigned[y] = **true**;

    node->parent = parent;

    node->workerID = x;

    node->jobID = y;

**return**node;

}

// Function to calculate the least promising cost

// of node after worker x is assigned to job y.

**int**calculateCost(**int**costMatrix[N][N], **int**x,

**int**y, **bool**assigned[])

{

**int**cost = 0;

    // to store unavailable jobs

**bool**available[N] = {**true**};

    // start from next worker

**for**(**int**i = x + 1; i < N; i++)

    {

**int**min = INT\_MAX, minIndex = -1;

        // do for each job

**for**(**int**j = 0; j < N; j++)

        {

            // if job is unassigned

**if**(!assigned[j] && available[j] &&

                costMatrix[i][j] < min)

            {

                // store job number

                minIndex = j;

                // store cost

                min = costMatrix[i][j];

            }

        }

        // add cost of next worker

        cost += min;

        // job becomes unavailable

        available[minIndex] = **false**;

    }

**return**cost;

}

// Comparison object to be used to order the heap

**struct**comp

{

**bool**operator()(**const**Node\* lhs,

**const**Node\* rhs) **const**

    {

**return**lhs->cost > rhs->cost;

    }

};

// print Assignments

**void**printAssignments(Node \*min)

{

**if**(min->parent==NULL)

**return**;

    printAssignments(min->parent);

    cout <<"Assign Worker "<<**char**(min->workerID + 'A')

         <<" to Job "<< min->jobID << endl;

}

// Finds minimum cost using Branch and Bound.

**int**findMinCost(**int**costMatrix[N][N])

{

    // Create a priority queue to store live nodes of

    // search tree;

    priority\_queue<Node\*, std::vector<Node\*>, comp> pq;

    // initialize heap to dummy node with cost 0

**bool**assigned[N] = {**false**};

    Node\* root = newNode(-1, -1, assigned, NULL);

    root->pathCost = root->cost = 0;

    root->workerID = -1;

    // Add dummy node to list of live nodes;

    pq.push(root);

    // Finds a live node with least cost,

    // add its childrens to list of live nodes and

    // finally deletes it from the list.

**while**(!pq.empty())

    {

      // Find a live node with least estimated cost

      Node\* min = pq.top();

      // The found node is deleted from the list of

      // live nodes

      pq.pop();

      // i stores next worker

**int**i = min->workerID + 1;

      // if all workers are assigned a job

**if**(i == N)

      {

          printAssignments(min);

**return**min->cost;

      }

      // do for each job

**for**(**int**j = 0; j < N; j++)

      {

        // If unassigned

**if**(!min->assigned[j])

        {

          // create a new tree node

          Node\* child = newNode(i, j, min->assigned, min);

          // cost for ancestors nodes including current node

          child->pathCost = min->pathCost + costMatrix[i][j];

          // calculate its lower bound

          child->cost = child->pathCost +

            calculateCost(costMatrix, i, j, child->assigned);

          // Add child to list of live nodes;

          pq.push(child);

        }

      }

    }

}

// Driver code

**int**main()

{

    // x-coordinate represents a Worker

    // y-coordinate represents a Job

**int**costMatrix[N][N] =

    {

        {9, 2, 7, 8},

        {6, 4, 3, 7},

        {5, 8, 1, 8},

        {7, 6, 9, 4}

    };

    /\* int costMatrix[N][N] =

    {

        {82, 83, 69, 92},

        {77, 37, 49, 92},

        {11, 69,  5, 86},

        { 8,  9, 98, 23}

    };

    \*/

    /\* int costMatrix[N][N] =

    {

        {2500, 4000, 3500},

        {4000, 6000, 3500},

        {2000, 4000, 2500}

    };\*/

    /\*int costMatrix[N][N] =

    {

        {90, 75, 75, 80},

        {30, 85, 55, 65},

        {125, 95, 90, 105},

        {45, 110, 95, 115}

    };\*/

    cout <<"\nOptimal Cost is "

        << findMinCost(costMatrix);

**return**0;

}

1. LC branch and bound algorithm for 0/1-Knapsack problem

*/ C++ Program to implement 0/1*

*// knapsack using LC Branch and Bound*

#include<bits/stdc++.h>

usingnamespace std;

*// Stores the number of items*

intsize;

*// Stores the knapsack capacity*

floatcapacity;

typedefstructItem{

*// Stores the weight of items*

    floatweight;

*// Stores the value of items*

    intvalue;

*// Stores the index of items*

    intidx;

}Item;

typedefstructNode{

*// Upper Bound: Best case*

*// (Fractional Knapsack)*

    floatub;

*// Lower Bound: Worst case (0/1)*

    floatlb;

*// Level of the node*

*// in the decision tree*

    intlevel;

*// Stores if the current item is*

*// selected or not*

    boolflag;

*// Total Value: Stores the sum of the*

*// values of the items included*

    floattv;

*// Total Weight: Stores the sum of the*

*// weights of the items included*

    floattw;

}Node;

*// Function to calculate upper bound*

*// (includes fractional part of the items)*

floatupper\_bound(floattv,floattw,

                  intidx,vector<Item>&arr)

{

    floatvalue=tv;

    floatweight=tw;

    for(inti=idx;i<size;i++){

        if(weight+arr[i].weight

            <=capacity){

            weight+=arr[i].weight;

            value-=arr[i].value;

        }

        else{

            value-=(float)(capacity

                             -weight)

                     /arr[i].weight

                     \*arr[i].value;

            break;

        }

    }

    returnvalue;

}

*// Function to calculate lower bound (doesn't*

*// include fractional part of the items)*

floatlower\_bound(floattv,floattw,

                  intidx,vector<Item>&arr)

{

    floatvalue=tv;

    floatweight=tw;

    for(inti=idx;i<size;i++){

        if(weight+arr[i].weight

            <=capacity){

            weight+=arr[i].weight;

            value-=arr[i].value;

        }

        else{

            break;

        }

    }

    returnvalue;

}

classcomp{

public:

    booloperator()(Nodea,Nodeb)

    {

        returna.lb>b.lb;

    }

};

voidassign(Node&a,floatub,floatlb,

            intlevel,boolflag,

            floattv,floattw)

{

    a.ub=ub;

    a.lb=lb;

    a.level=level;

    a.flag=flag;

    a.tv=tv;

    a.tw=tw;

}

voidknapsack(vector<Item>&arr)

{

*// Sort the items based on the*

*// profit/weight ratio*

    sort(arr.begin(),arr.end(),

         [&](Item&a,Item&b){

             returna.value/a.weight

                    >b.value/b.weight;

         });

*// min\_lb -> Minimum lower bound*

*// of all the nodes explored*

*// final\_lb -> Minimum lower bound*

*// of all the paths that reached*

*// the final level*

    floatmin\_lb=0,final\_lb=INT\_MAX;

*// curr\_path -> Boolean array to store*

*// at every index if the element is*

*// included or not*

*// final\_path -> Boolean array to store*

*// the result of selection array when*

*// it reached the last level*

    boolcurr\_path[size],final\_path[size];

*// Priority queue to store the nodes*

*// based on lower bounds*

    priority\_queue<Node,vector<Node>,

                   comp>

        pq;

    Nodecurrent,left,right;

    current.lb=current.ub=current.tw

        =current.tv=current.level

        =current.flag=0;

*// Insert a dummy node*

    pq.push(current);

    for(inti=0;i<size;i++)

        curr\_path[i]=final\_path[i]

            =false;

    while(!pq.empty()){

        current=pq.top();

        pq.pop();

        if(current.ub>min\_lb

            ||current.ub>=final\_lb){

*// If the current node's best case*

*// value is not optimal than min\_lb,*

*// then there is no reason to explore*

*// that path including final\_lb*

*// eliminates all those paths whose*

*// best values is equal to final\_lb*

            continue;

        }

*// update the path*

        if(current.level!=0)

            curr\_path[current.level-1]

                =current.flag;

        if(current.level==size){

*// Reached last level*

            if(current.lb<final\_lb)

                for(inti=0;i<size;i++)

                    final\_path[arr[i].idx]

                        =curr\_path[i];

            final\_lb=min(current.lb,final\_lb);

            continue;

        }

        intlevel=current.level;

*// right node -> Excludes current item*

*// Hence, cp, cw will obtain the value*

*// of that of parent*

        assign(right,

               upper\_bound(current.tv,

                           current.tw,level+1,

                           arr),

               lower\_bound(current.tv,current.tw,

                           level+1,arr),

               level+1,false,

               current.tv,current.tw);

*// Check whether adding the current*

*// item will not exceed the knapsack weight*

        if(current.tw+arr[current.level].weight

            <=capacity){

*// left node -> includes current item*

*// c and lb should be calculated*

*// including the current item.*

            left.ub

                =upper\_bound(

                    current.tv

                        -arr[level].value,

                    current.tw

                        +arr[level].weight,

                    level+1,arr);

            left.lb

                =lower\_bound(

                    current.tv

                        -arr[level].value,

                    current.tw

                        +arr[level].weight,

                    level+1,arr);

            assign(left,left.ub,left.lb,

                   level+1,true,

                   current.tv-arr[level].value,

                   current.tw

                       +arr[level].weight);

        }

*// If Left node cannot be inserted*

        else{

*// Stop the left node from*

*// getting added to the*

*// priority queue*

            left.ub=left.lb=1;

        }

*// Update the lower bound*

        min\_lb=min(min\_lb,left.lb);

        min\_lb=min(min\_lb,right.lb);

*// Exploring nodes whose*

*// upper bound is greater than*

*// min\_lb will never give*

*// the optimal result*

        if(min\_lb>=left.ub)

            pq.push(left);

        if(min\_lb>=right.ub)

            pq.push(right);

    }

    cout<<"Items taken into the"

         <<" knapsack are : \n";

    if(final\_lb==INT\_MAX)

        final\_lb=0;

    for(inti=0;i<size;i++)

        cout<<final\_path[i]<<"";

    cout<<"\n";

    cout<<"Maximum profit is : "

         <<(-final\_lb)<<"\n";

}

*// Driver Code*

intmain()

{

    size=4;

    capacity=15;

    vector<Item>arr;

    arr.push\_back({ 2,10,0 });

    arr.push\_back({ 4,10,1 });

    arr.push\_back({ 6,12,2 });

    arr.push\_back({ 9,18,3 });

    knapsack(arr);

    return0;

}