```
//  File:  constraints.h
//
//Massachusetts Institute of Technology
//16.412J/6.834J Cognitive Robotics
//
//Russian Doll Search
//
//Problem Set #2
//Due: in class Wed, 3/9/05
//
//Lawrence Bush, Brian Bairstow
//{ bushl2, bairstow }@mit.edu
// __________________________________________
//
//

#ifndef _constraints_h_
#define _constraints_h_
#include <string>
#include <vector>
#include <algorithm>
#include "variable.h"
#include "variables.h"
#include "tuples.h"
using namespace std;

// constraints class
//
//
class constraints {


public:
    //constraints() {}  // default constructor // not needed or wanted
    // Constructor - assigns all variable attributes
    constraints(int number_of_variables_in, int maximum_domain_size_in, int             ↙
    number_of_constraint_groups_in, int  global_upper_bound_in, variables  X_in)
        : number_of_variables(number_of_variables_in), maximum_domain_size              ↙
    (maximum_domain_size_in),number_of_constraint_groups(number_of_constraint_groups_in), ↙
    global_upper_bound(global_upper_bound_in), X(X_in)
    {
        global_lower_bound = 0;
    }

    void insert_tuples(tuples ts_in){ // insert a variable object into the container
        tuples_vector.push_back(ts_in);
    }


    int get_number_of_constraint_groups()   const {return number_of_constraint_groups;}
    int get_number_of_variables()   const {return number_of_variables;}

    // assessor operator, returns player k
    tuples operator[](int k)  const
    {
        return tuples_vector[k];
    }

    int get_number_of_nodes() {
        vector<variable> vl = X.get_variable_list();

        int number_of_nodes=1;
        for(vector<variable>::iterator i = vl.begin(); i != vl.end(); i++)
        {
            number_of_nodes *= i->get_domain_size();// metric counter
        }
        return number_of_nodes;
```

```
}

int size() { return tuples_vector.size(); } // returns the number of tuple in the ↙
container

double initialize_upper_bound(variables & sa, variables ca, vector<unsigned long long ↙
int> & operations) {

    upper_bound = 999999999;
    double temp;
    variables tempvar;
    int i;

    for(i = 0; i < sa.size(); i++)
    {
        ca.insert(sa[i]);
    }

    for(i = 0; i < ca[0].get_domain_size(); i++)
    {
        temp = evaluate(ca,operations);
        if(temp < upper_bound)
        {
            upper_bound = temp;
            tempvar = ca;
        }
        if(i < ca[0].get_domain_size()-1)
            ca = increment_first_value(ca);
        operations[2]+=1;
    }

    sa = tempvar;
    //if(upper_bound==sa_eval) {upper_bound -= 1;   cout<<"ub == sa_eval"<<endl;} // ↙
obsolete not
    return upper_bound;
}

double get_upper_bound() {
    return upper_bound;
}
double get_global_upper_bound() {
    return global_upper_bound;
}



// print tuple container
void print(std::ostream & out) {

    out << "-----------------------------------------------------"<<endl;
    out << "Tuple Sets Statistics: \n";
    out << "-----------------------------------------------------"<<endl;
    out << "Number of Variables:  "<< number_of_variables << endl;
    out << "Domain Sizes:         "<< maximum_domain_size << endl;
    out << "Number of Tuple Sets: "<< number_of_constraint_groups << endl;
    out << "Global Upper Bound:   "<< global_upper_bound << endl;
    out << "-----------------------------------------------------"<<endl;
    out << "Print all Tuple Sets: \n";
    out << "-----------------------------------------------------"<<endl;

    for( int i = 0 ; i < size() ; i++ ) {
        tuples_vector[i].print(out);
    }
}

bool is_last_value(variables & ca_in)
{
    if(ca_in.back().get_domain_value()==(ca_in.back().get_domain_size()-1)){
        return true;
    } else {
        return false;
    }
}


bool is_last_variable(variables & ca_in)
{
    if(ca_in.back().get_var_index()==X.back().get_var_index()){
        return true;
    } else {
        return false;
    }
}

variables initialize_assignment(int initial ){
    current_assignment = variables();
    next_variable = initial;
    next_value = 0;
    current_assignment.insert(variable(next_variable,X[next_variable].get_domain_size ↙
(),next_value ));
    return current_assignment;
}

variables increment_first_value(variables ca_in)
{
    int n = ca_in.size();
    vector<variable> temp(n);

    for(int i = n - 1; i > 0; i--)
    {
        temp[i] = ca_in[i];
        ca_in.remove();
    }
    ca_in = get_next_value(ca_in);

    for(int count_n = 1; count_n < n; count_n ++)
    {
        ca_in.insert(temp[count_n]);
    }
    return ca_in;
}

variables get_next_value(variables & ca_in){

    variable temp = ca_in.back();
    ca_in.remove();
    if(!temp.increment_domain_value()) {
        cout<<"Error, domain exceeded!\n";
    }
    ca_in.insert(temp);

    return ca_in;

    return ca_in;
}


variables get_next_variable(variables ca_in){

    // insert the next variable if I can
    ca_in.insert(variable(X[ca_in.back().get_var_index()+1].get_var_index(),X[ca_in. ↙
back().get_var_index()+1].get_domain_size(),0));
    return ca_in;
}
variables back_up(variables & ca_in){

    if(ca_in.size()==0){return ca_in;}
    ca_in.remove();
    if(ca_in.size()==0){return ca_in;}

    if(ca_in.back().increment_domain_value()) {
        ca_in = get_next_value(ca_in);
    } else {
        ca_in = back_up(ca_in);
    }
    return ca_in;
}


variables get_next_assignment(){

    // insert the next variable if I can
    if(next_variable+1 != number_of_variables){
        next_variable++;
        current_assignment.insert(variable(next_variable,X[next_variable].         ↙
get_domain_size(),0));
        return current_assignment;
    }
    // else increment the value
    while( 1 ){

        variable temp = current_assignment.back();
        current_assignment.remove();
        next_variable--;

        if(!temp.next_domain_value()){
            current_assignment.insert(temp);

            next_variable++;
            return current_assignment;
        }
    }
    return current_assignment;
}


double evaluate(variables & ca_in,vector<unsigned long long int> & operations, double ↙
& additional_cost, double & upper_bound){

    //operations += ca_in.size();// obsolete counter

    double return_value = 0;
    for( int i = 0 ; i < size() ; i++ ) {//size is the number of tuple sets

        return_value += tuples_vector[i].evaluate(ca_in, operations);
        if( (return_value + additional_cost) > upper_bound) { return return_value; }
    }
    return return_value;
}


double evaluate(variables ca_in,vector<unsigned long long int> & operations){

    //operations += ca_in.size();// obsolete counter

    double return_value = 0;
    for( int i = 0 ; i < size() ; i++ ) {
        return_value += tuples_vector[i].evaluate(ca_in, operations);
    }
    return return_value;
}

variables bind_next_assignment(){
    // this function goes to the next value, rather than going deeper
    // increment the value
    while( 1 ){

        variable temp = current_assignment.back();
        current_assignment.remove();
        next_variable--;

        if(!temp.next_domain_value()){
            current_assignment.insert(temp);

            next_variable++;
            return current_assignment;
        }
    }
    return current_assignment;
}
// constraint sorts
void sort_tuples_by_num_non_default_cost(){
    sort(tuples_vector.begin(),tuples_vector.end(),less_size());
}
class less_size {
public:
    bool operator()(tuples x, tuples y) const { return (x.get_number_of_tuples() < y. ↙
get_number_of_tuples()); }
};


private:

int number_of_variables;
int maximum_domain_size;
int number_of_constraint_groups;
double global_upper_bound;
double upper_bound;
double global_lower_bound;
variables X;
vector<tuples> tuples_vector;


variables current_assignment;
int next_variable;
int next_value;


};

#endif
```