Algoritmos Genéticos

AlgoritmosGeneticos.cpp

```cpp
#include < bits / stdc++.h >
const int INF = 1 << 30;
using namespace std;

// Usamos este tipo de dato para evitar negativos
typedef unsigned long int ulong;
typedef  function < bool(pair < int, int > , pair < int, int > ) > Comparator;

int nVariables, nRestricciones, nOpcion, nPoblacion, nIteraciones, nBits, nTotalBits = 0,
 nSurvivors = 0;
vector < double > aFormulaObjetivo, aFormulaEstatica, aValuesZ, aAcumulateZ;
vector < pair < double, double > > aLimites;
vector < pair < int, int > > aCountedValues;
vector < vector < double > > aRestricciones, aValuesPoblation;
vector < int > aBitsVariables, aMaxVariables;
vector < vector < ulong > > aPobladores;

// Map usado para contar el número de apariciones y encontrar los vectores más dominantes
unordered_map < int, int > mapValues;
// Map usado para evitar que se repitan los sujetos a evaluar.
unordered_map < double, int > mapRepetitions;
// En este elemento guardaremos el elemento final junto a su clave
pair < int, double > finalValueOfZ;
// El comparador nos ordenará el map con base en su valor ( # de apariciones ) en lugar de hacerlo por śu llave
Comparator compFunctor = [](pair < int, int > elem1, pair < int, int > elem2) {
    return elem1.second > elem2.second;
};

// Los datos se piden en una función secundaria para facilitar el trabajo
#include "PideDatos.h"

// Función que ordenará el map depositando el resultado en un set
void sortMap() {
        set < pair < int, int > , Comparator > setOfValues(mapValues.begin(), mapValues.end(), compFunctor);
        aCountedValues.clear();
        aCountedValues.resize(setOfValues.size());
        copy(setOfValues.begin(), setOfValues.end(), aCountedValues.begin());
    }

// Función que generará elementos al azar entre 0 y 1, comparándolos contra la acumulada de Z
void countRandomValues() {
        mapValues.clear();
        for (int i = 0; i < nPoblacion; ++i) {
            double dRandom = ((double) rand() / (RAND_MAX));
            for (int j = 0; j < nPoblacion; ++j) {
                if (dRandom < aAcumulateZ[j]) {
                    mapValues[j]++;
                    break;
                }
```

```
            }
        }
        sortMap();
    }

// Función que nos indicará si el poblador cumple con las restricciones, al fallar en alguna, la función devolverá un false
bool checkRestrictions(vector < double > aTest) {
        for (int i = 0; i < nRestricciones; ++i) {
            double dAux = 0;
            int j = 0;
            for (j = 0; j < nVariables; ++j) {
                dAux += aRestricciones[i][j] * aTest[j];
            }
            if (aRestricciones[i][j] == 1) {
                if (dAux > aRestricciones[i][++j]) {
                    return false;
                }
            } else {
                if (dAux < aRestricciones[i][++j]) {
                    return false;
                }
            }
        }
        return true;
    }

// Función que mutará un poblador con base en su clave específica
void mutatePoblator(int nIndex) {
        vector < ulong > aFuturePoblator;
        vector < double > aValueFuturePoblator(nVariables);
        do {
            int nRandomVariable = rand() % nVariables;
            int nRandomBit = rand() % aBitsVariables[nRandomVariable];
            aFuturePoblator = aPobladores[nIndex];
            bitset < sizeof(ulong) * 8 > bFuturePoblator(aFuturePoblator[nRandomVariable]);
            bFuturePoblator.flip(nRandomBit);
            aFuturePoblator[nRandomVariable] = bFuturePoblator.to_ulong();
            aValueFuturePoblator = aValuesPoblation[nIndex];
            double dAux = (double)(bFuturePoblator.to_ulong()) * aFormulaEstatica[nRandomVariable];
            aValueFuturePoblator[nRandomVariable] = aLimites[nRandomVariable].first + dAux;
        } while (!checkRestrictions(aValueFuturePoblator));
        aPobladores[nIndex] = aFuturePoblator;
        aValuesPoblation[nIndex] = aValueFuturePoblator;
    }

// Función que mutará un poblador con base al vector más dominante y lo añadirá a los demás pobladores.
void mutatePoblator() {
        vector < ulong > aFuturePoblator;
        vector < double > aValueFuturePoblator(nVariables);
        do {
            int nRandomVariable = rand() % nVariables;
                int nRandomBit = rand() % aBitsVariables[nRandomVariable];
                aFuturePoblator = aPobladores[0];
                bitset < sizeof(ulong) * 8 > bFuturePoblator(aFuturePoblator[nRandomVariable]);
                bFuturePoblator.flip(nRandomBit);
```

```
104.                    aFuturePoblator[nRandomVariable] = bFuturePoblator.to_ulong();
105.                    aValueFuturePoblator = aValuesPoblation[0];
106.                    double dAux = (double)(bFuturePoblator.to_ulong()) * aFormulaEstatica[
    nRandomVariable];
107.                    aValueFuturePoblator[nRandomVariable] = aLimites[nRandomVariable].firs
    t + dAux;
108.                } while (!checkRestrictions(aValueFuturePoblator));
109.                aPobladores.push_back(aFuturePoblator);
110.                aValuesPoblation.push_back(aValueFuturePoblator);
111.            }
112.
113.        // Función que generará un poblador al azar añadiéndolo en un espacio específico,
    se usa al iniciar el programa
114.        void generatePoblator(int nIndex) {
115.                do {
116.                    for (int j = 0; j < nVariables; ++j) {
117.                        aPobladores[nIndex][j] = rand() % aMaxVariables[j];
118.                        double dAux = (double) aPobladores[nIndex][j] * aFormulaEstatica[j
    ];
119.                        aValuesPoblation[nIndex][j] = aLimites[j].first + dAux;
120.                    }
121.                } while (!checkRestrictions(aValuesPoblation[nIndex]));
122.            }
123.
124.        // Función que calcula el valor de z y lo añade a su respectiva posición
125.        void calculateZ(int nIndex) {
126.                double dAux = 0;
127.                cout << "Value of z" << nIndex << " for";
128.                for (int i = 0; i < nVariables; ++i) {
129.                    cout << " x" << i << " = " << aValuesPoblation[nIndex][i] << " * " <<
    aFormulaObjetivo[i];
130.                    dAux += aFormulaObjetivo[i] * aValuesPoblation[nIndex][i];
131.                }
132.                cout << " is = " << dAux << endl;
133.                aValuesZ[nIndex] = dAux;
134.            } // Función que calculará los valores de Z, así como su acumulada
135.        void calculateValues() {
136.                mapRepetitions.clear();
137.                aValuesZ.clear();
138.                aValuesZ.resize(nPoblacion);
139.                double dAux = 0, minorValue = 0, maxValue = 0;
140.                int i = 0;
141.                while (i < nPoblacion) {
142.                    calculateZ(i);
143.                    if (aValuesZ[i] > maxValue) {
144.                        maxValue = aValuesZ[i];
145.                    }
146.                    if (aValuesZ[i] < 0) {
147.                        if (aValuesZ[i] < minorValue) {
148.                            dAux += (minorValue - aValuesZ[i]) * i;
149.                            minorValue = aValuesZ[i];
150.                        } else {
151.                            dAux -= (minorValue - aValuesZ[i]);
152.                        }
153.                    } else {
154.                        dAux += (aValuesZ[i] - minorValue);
155.                    }
156.                    unordered_map < double, int > ::const_iterator got = mapRepetitions.fi
    nd(aValuesZ[i]);
157.                    if (got == mapRepetitions.end()) {
158.                        mapRepetitions[aValuesZ[i]]++;
```

```
159.                     i++;
160.                 } else mutatePoblator(i);
161.             }
162.             cout << "Minor value is " << minorValue << endl;
163.             maxValue -= minorValue;
164.             cout << "Max value is " << maxValue << endl;
165.             aAcumulateZ.clear();
166.             aAcumulateZ.resize(nPoblacion);
167.             i = 0;
168.             if (nOpcion == 1) {
169.                 aAcumulateZ[i] = aValuesZ[i] / dAux;
170.             } else {
171.                 aAcumulateZ[i] = (maxValue - (aValuesZ[i] - minorValue)) / dAux;
172.             }
173.             for (i = 1; i < nPoblacion - 1; ++i) {
174.                 if (nOpcion == 1) {
175.                     aAcumulateZ[i] = (aValuesZ[i] / dAux) + aAcumulateZ[i - 1];
176.                 } else {
177.                     aAcumulateZ[i] = ((maxValue - (aValuesZ[i] - minorValue)) / dAux)
    + aAcumulateZ[i - 1];
178.                 }
179.             }
180.             aAcumulateZ[i] = 1;
181.         }
182.
183.     // En la última iteración únicamente se sacan los valores de Z y se guarda el más
    alto o el más bajo según se haya solicitado
184.     void calculateValuesFinal() {
185.             aValuesZ.clear();
186.             aValuesZ.resize(nPoblacion);
187.             cout << "\n===== Iteración Final =====\n" << endl;
188.             if (nOpcion == 1) {
189.                 finalValueOfZ = make_pair(-1, 0.0);
190.                 for (int i = 0; i < nPoblacion; ++i) {
191.                     calculateZ(i);
192.                     if (aValuesZ[i] > finalValueOfZ.second) {
193.                         finalValueOfZ = make_pair(i, aValuesZ[i]);
194.                     }
195.                 }
196.             } else {
197.                 finalValueOfZ = make_pair(-1, INF);
198.                 for (int i = 0; i < nPoblacion; ++i) {
199.                     calculateZ(i);
200.                     if (aValuesZ[i] < finalValueOfZ.second) {
201.                         finalValueOfZ = make_pair(i, aValuesZ[i]);
202.                     }
203.                 }
204.             }
205.         }
206.
207.     // Función que inicializará la población y calculará la fórmula necesaria para obt
    ener el valor de la variable
208.     void startPoblation() {
209.             for (int i = 0; i < nVariables; ++i) {
210.                 double dRange = aLimites[i].second - aLimites[i].first;
211.                 double dExponent = pow(2, aBitsVariables[i]) - 1;
212.                 aFormulaEstatica[i] = (dRange / dExponent);
213.             }
214.             for (int i = 0; i < nPoblacion; ++i) {
215.                 generatePoblator(i);
216.             }
```

```
217.           }
218.
219.       // Función que calculará el número de bits necesarios para cada variable, así como
    su máximo valor posible
220.       void calculateBits() {
221.               for (int i = 0; i < nVariables; ++i) {
222.                   double dRange = aLimites[i].second - aLimites[i].first;
223.                   double dExponent = pow(10, nBits);
224.                   double dLog = log2(dRange * dExponent);
225.                   double dAux = ceil(dLog);
226.                   aBitsVariables[i] = dAux;
227.                   aMaxVariables[i] = pow(2, aBitsVariables[i]);
228.                   nTotalBits += dAux;
229.               }
230.           }
231.
232.       // Inicializamos por primera vez los vectores a utilizar
233.       void initializeVectors() {
234.           aBitsVariables.resize(nVariables);
235.           aMaxVariables.resize(nVariables);
236.           aFormulaEstatica.resize(nVariables);
237.           aPobladores.resize(nPoblacion, vector < ulong > (nVariables));
238.           aValuesPoblation.resize(nPoblacion, vector < double > (nVariables));
239.           aValuesZ.resize(nPoblacion);
240.           aAcumulateZ.resize(nPoblacion, 0);
241.       }
242.       int main(int argc, char
243.           const * argv[]) {
244.       // Función que pide los datos desde terminal
245.           askData();
246.       // Función que genera los valores random desde el reloj en lugar del algoritmo
247.           srand(time(0));
248.       // Iniciamos vectores, calculamos bits y generamos la primer población necesaria
249.           initializeVectors();
250.           calculateBits();
251.           startPoblation();
252.           cout << endl;
253.       // Iniciamos las iteraciones
254.           for (int i = 0; i < nIteraciones; ++i) {
255.               cout << "\n===== Iteración " << i << " =====\n" << endl;
256.       // En caso de que falten pobladores, se generan a partir del más dominante
257.               while (aPobladores.size() < nPoblacion) {
258.                   mutatePoblator();
259.               }
260.       // Calculamos los valores necesarios, tanto Z como la acumulada de Z
261.               calculateValues();
262.       // Generamos los valores random a calcular en la acumulada
263.               countRandomValues();
264.       // Generamos vectores auxiliares donde copiaremos los individuos más dominantes
265.               vector < vector < ulong > > aPobladoresAux(nPoblacion, vector < ulong > (n
    Variables));
266.               vector < vector < double > > aValuesPoblationAux(nPoblacion, vector < doub
    le > (nVariables));
267.               for (int j = 0; j < aCountedValues.size(); ++j) {
268.                   aPobladoresAux[j] = aPobladores[aCountedValues[j].first];
269.                   aValuesPoblationAux[j] = aValuesPoblation[aCountedValues[j].first];
270.               }
271.               aPobladores.clear();
272.               aValuesPoblation.clear();
273.               for (int j = 0; j < aCountedValues.size(); ++j) {
274.                   aPobladores.push_back(aPobladoresAux[j]);
```

```
                    aValuesPoblation.push_back(aValuesPoblationAux[j]);
                }
            }
        // En la última iteración, rellenamos los pobladores faltantes y calculamos sus va
    lores
            while (aPobladores.size() < nPoblacion) {
                mutatePoblator();
            }
            calculateValuesFinal();
            cout << "\nSolucion optima para Z es: " << finalValueOfZ.second << " \nCon:\n"
    ;
            for (int i = 0; i < nVariables; ++i) {
                cout << "\tx" << i << " = " << aValuesPoblation[finalValueOfZ.first][i] <<
     endl;
            }
            return 0;
        }
```

PideDatos.h

```
void askData() {
    cout << "Bienvenido al sistema de solucion de Problemas de Programacion Lineal" << en
    dl;
    cout << "Introduca el numero de variables (Maximo 4): ";
    cin >> nVariables;
    while (nVariables > 4) {
        cout << "Numero de variables excedido, intente nuevamente: ";
        cin >> nVariables;
    }
    aFormulaObjetivo.resize(nVariables, 0);
    cout << "Introduzca el valor de la variables en la funcion Z:\n";
    for (int i = 0; i < nVariables; ++i) {
        cout << "\tx" << i << " = ";
        cin >> aFormulaObjetivo[i]; //cout<<endl;
    }
    aLimites.resize(nVariables);
    for (int i = 0; i < nVariables; ++i) {
        cout << "Introduzca el limite izquierdo de la variable x" << i << ":\n";
        cin >> aLimites[i].first;
        cout << "";
        cout << "Introduzca el limite derecho de la variable x" << i << ":\n";
        cin >> aLimites[i].second;
        cout << "";
        cout << endl;
    }
    cout << "\nQue desea hacer?\n\t1) Maximizar\n\t2) Minimizar" << endl;
    cin >> nOpcion;
    while (nOpcion != 1 && nOpcion != 2) {
        cout << "Opcion no valida, intente nuevamente: ";
        cin >> nOpcion;
    }
    cout << "\nIntroduzca el numero de restricciones (Maximo 5): ";
    cin >> nRestricciones;
    while (nRestricciones > 5) {
        cout << "Numero de restricciones excedido, intente nuevamente: ";
        cin >> nRestricciones;
```

```
    }
    aRestricciones.resize(nRestricciones, vector < double > (nVariables + 2));
    for (int i = 0; i < nRestricciones; ++i) {
        cout << "\tIntroduzca el valor de la variables para la R" << i << ":\n";
        int j;
        for (j = 0; j < nVariables; ++j) {
            cout << "\t\tx" << j << " = ";
            cin >> aRestricciones[i][j];
        }
        cout << "\n\tQue desea hacer?\n\t\t1) <=\n\t\t2) >=\n\t";
        cin >> aRestricciones[i][j];
        while (aRestricciones[i][j] != 1 && aRestricciones[i][j] != 2) {
            cout << "Opcion no valida, intente nuevamente: ";
            cin >> aRestricciones[i][j];
        }
        cout << "\tIntroduzca el valor de la variable a igualar para la R" << i << ": ";

        cin >> aRestricciones[i][++j];
        cout << "";
        cout << endl;
    }
    cout << "\nIntroduzca el numero de poblacion: ";
    cin >> nPoblacion;
    cout << "\nIntroduzca el numero de iteraciones (maximo 100): ";
    cin >> nIteraciones;
    while (nIteraciones > 100) {
        cout << "Numero de iteraciones excedido, intente nuevamente: ";
        cin >> nIteraciones;
    }
    cout << "\nIntroduzca el numero de bits de precision: ";
    cin >> nBits;
}
```