# Algoritmos Genéticos

## AlgoritmosGeneticos.cpp

1. #include < bits / stdc++.h >
2. const int INF = 1 << 30;
3. using namespace std;
4. // Usamos este tipo de dato para evitar negativos
5. typedef unsigned long int ulong;
6. typedef  function < bool(pair < int, int > , pair < int, int > ) > Comparator;
7. int nVariables, nRestricciones, nOpcion, nPoblacion, nIteraciones, nBits, nTotalBits = 0, nSurvivors = 0;
8. vector < double > aFormulaObjetivo, aFormulaEstatica, aValuesZ, aAcumulateZ;
9. vector < pair < double, double > > aLimites;
10. vector < pair < int, int > > aCountedValues;
11. vector < vector < double > > aRestricciones, aValuesPoblation;
12. vector < int > aBitsVariables, aMaxVariables;
13. vector < vector < ulong > > aPobladores;
14. // Map usado para contar el número de apariciones y encontrar los vectores más dominantes
15. unordered\_map < int, int > mapValues;
16. // Map usado para evitar que se repitan los sujetos a evaluar.
17. unordered\_map < double, int > mapRepetitions;
18. // En este elemento guardaremos el elemento final junto a su clave
19. pair < int, double > finalValueOfZ;
20. // El comparador nos ordenará el map con base en su valor ( # de apariciones ) en lugar de hacerlo por śu llave
21. Comparator compFunctor = [](pair < int, int > elem1, pair < int, int > elem2) {
22. return elem1.second > elem2.second;
23. };
24. // Los datos se piden en una función secundaria para facilitar el trabajo
25. #include "PideDatos.h"
26. // Función que ordenará el map depositando el resultado en un set
27. void sortMap() {
28. set < pair < int, int > , Comparator > setOfValues(mapValues.begin(), mapValues.end(), compFunctor);
29. aCountedValues.clear();
30. aCountedValues.resize(setOfValues.size());
31. copy(setOfValues.begin(), setOfValues.end(), aCountedValues.begin());
32. }
33. // Función que generará elementos al azar entre 0 y 1, comparándolos contra la acumulada de Z
34. void countRandomValues() {
35. mapValues.clear();
36. for (int i = 0; i < nPoblacion; ++i) {
37. double dRandom = ((double) rand() / (RAND\_MAX));
38. for (int j = 0; j < nPoblacion; ++j) {
39. if (dRandom < aAcumulateZ[j]) {
40. mapValues[j]++;
41. break;
42. }
43. }
44. }
45. sortMap();
46. }
47. // Función que nos indicará si el poblador cumple con las restricciones, al fallar en alguna, la función devolverá un false
48. bool checkRestrictions(vector < double > aTest) {
49. for (int i = 0; i < nRestricciones; ++i) {
50. double dAux = 0;
51. int j = 0;
52. for (j = 0; j < nVariables; ++j) {
53. dAux += aRestricciones[i][j] \* aTest[j];
54. }
55. if (aRestricciones[i][j] == 1) {
56. if (dAux > aRestricciones[i][++j]) {
57. return false;
58. }
59. } else {
60. if (dAux < aRestricciones[i][++j]) {
61. return false;
62. }
63. }
64. }
65. return true;
66. }
67. // Función que mutará un poblador con base en su clave específica
68. void mutatePoblator(int nIndex) {
69. vector < ulong > aFuturePoblator;
70. vector < double > aValueFuturePoblator(nVariables);
71. do {
72. int nRandomVariable = rand() % nVariables;
73. int nRandomBit = rand() % aBitsVariables[nRandomVariable];
74. aFuturePoblator = aPobladores[nIndex];
75. bitset < sizeof(ulong) \* 8 > bFuturePoblator(aFuturePoblator[nRandomVariable]);
76. bFuturePoblator.flip(nRandomBit);
77. aFuturePoblator[nRandomVariable] = bFuturePoblator.to\_ulong();
78. aValueFuturePoblator = aValuesPoblation[nIndex];
79. double dAux = (double)(bFuturePoblator.to\_ulong()) \* aFormulaEstatica[nRandomVariable];
80. aValueFuturePoblator[nRandomVariable] = aLimites[nRandomVariable].first + dAux;
81. } while (!checkRestrictions(aValueFuturePoblator));
82. aPobladores[nIndex] = aFuturePoblator;
83. aValuesPoblation[nIndex] = aValueFuturePoblator;
84. }
85. // Función que mutará un poblador con base al vector más dominante y lo añadirá a los demás pobladores.
86. void mutatePoblator() {
87. vector < ulong > aFuturePoblator;
88. vector < double > aValueFuturePoblator(nVariables);
89. do {
90. int nRandomVariable = rand() % nVariables;
91. int nRandomBit = rand() % aBitsVariables[nRandomVariable];
92. aFuturePoblator = aPobladores[0];
93. bitset < sizeof(ulong) \* 8 > bFuturePoblator(aFuturePoblator[nRandomVariable]);
94. bFuturePoblator.flip(nRandomBit);
95. aFuturePoblator[nRandomVariable] = bFuturePoblator.to\_ulong();
96. aValueFuturePoblator = aValuesPoblation[0];
97. double dAux = (double)(bFuturePoblator.to\_ulong()) \* aFormulaEstatica[nRandomVariable];
98. aValueFuturePoblator[nRandomVariable] = aLimites[nRandomVariable].first + dAux;
99. } while (!checkRestrictions(aValueFuturePoblator));
100. aPobladores.push\_back(aFuturePoblator);
101. aValuesPoblation.push\_back(aValueFuturePoblator);
102. }
103. // Función que generará un poblador al azar añadiéndolo en un espacio específico, se usa al iniciar el programa
104. void generatePoblator(int nIndex) {
105. do {
106. for (int j = 0; j < nVariables; ++j) {
107. aPobladores[nIndex][j] = rand() % aMaxVariables[j];
108. double dAux = (double) aPobladores[nIndex][j] \* aFormulaEstatica[j];
109. aValuesPoblation[nIndex][j] = aLimites[j].first + dAux;
110. }
111. } while (!checkRestrictions(aValuesPoblation[nIndex]));
112. }
113. // Función que calcula el valor de z y lo añade a su respectiva posición
114. void calculateZ(int nIndex) {
115. double dAux = 0;
116. cout << "Value of z" << nIndex << " for";
117. for (int i = 0; i < nVariables; ++i) {
118. cout << " x" << i << " = " << aValuesPoblation[nIndex][i] << " \* " << aFormulaObjetivo[i];
119. dAux += aFormulaObjetivo[i] \* aValuesPoblation[nIndex][i];
120. }
121. cout << " is = " << dAux << endl;
122. aValuesZ[nIndex] = dAux;
123. } // Función que calculará los valores de Z, así como su acumulada
124. void calculateValues() {
125. mapRepetitions.clear();
126. aValuesZ.clear();
127. aValuesZ.resize(nPoblacion);
128. double dAux = 0, minorValue = 0, maxValue = 0;
129. int i = 0;
130. while (i < nPoblacion) {
131. calculateZ(i);
132. if (aValuesZ[i] > maxValue) {
133. maxValue = aValuesZ[i];
134. }
135. if (aValuesZ[i] < 0) {
136. if (aValuesZ[i] < minorValue) {
137. dAux += (minorValue - aValuesZ[i]) \* i;
138. minorValue = aValuesZ[i];
139. } else {
140. dAux -= (minorValue - aValuesZ[i]);
141. }
142. } else {
143. dAux += (aValuesZ[i] - minorValue);
144. }
145. unordered\_map < double, int > ::const\_iterator got = mapRepetitions.find(aValuesZ[i]);
146. if (got == mapRepetitions.end()) {
147. mapRepetitions[aValuesZ[i]]++;
148. i++;
149. } else mutatePoblator(i);
150. }
151. cout << "Minor value is " << minorValue << endl;
152. maxValue -= minorValue;
153. cout << "Max value is " << maxValue << endl;
154. aAcumulateZ.clear();
155. aAcumulateZ.resize(nPoblacion);
156. i = 0;
157. if (nOpcion == 1) {
158. aAcumulateZ[i] = aValuesZ[i] / dAux;
159. } else {
160. aAcumulateZ[i] = (maxValue - (aValuesZ[i] - minorValue)) / dAux;
161. }
162. for (i = 1; i < nPoblacion - 1; ++i) {
163. if (nOpcion == 1) {
164. aAcumulateZ[i] = (aValuesZ[i] / dAux) + aAcumulateZ[i - 1];
165. } else {
166. aAcumulateZ[i] = ((maxValue - (aValuesZ[i] - minorValue)) / dAux) + aAcumulateZ[i - 1];
167. }
168. }
169. aAcumulateZ[i] = 1;
170. }
171. // 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
172. void calculateValuesFinal() {
173. aValuesZ.clear();
174. aValuesZ.resize(nPoblacion);
175. cout << "\n===== Iteración Final =====\n" << endl;
176. if (nOpcion == 1) {
177. finalValueOfZ = make\_pair(-1, 0.0);
178. for (int i = 0; i < nPoblacion; ++i) {
179. calculateZ(i);
180. if (aValuesZ[i] > finalValueOfZ.second) {
181. finalValueOfZ = make\_pair(i, aValuesZ[i]);
182. }
183. }
184. } else {
185. finalValueOfZ = make\_pair(-1, INF);
186. for (int i = 0; i < nPoblacion; ++i) {
187. calculateZ(i);
188. if (aValuesZ[i] < finalValueOfZ.second) {
189. finalValueOfZ = make\_pair(i, aValuesZ[i]);
190. }
191. }
192. }
193. }
194. // Función que inicializará la población y calculará la fórmula necesaria para obtener el valor de la variable
195. void startPoblation() {
196. for (int i = 0; i < nVariables; ++i) {
197. double dRange = aLimites[i].second - aLimites[i].first;
198. double dExponent = pow(2, aBitsVariables[i]) - 1;
199. aFormulaEstatica[i] = (dRange / dExponent);
200. }
201. for (int i = 0; i < nPoblacion; ++i) {
202. generatePoblator(i);
203. }
204. }
205. // Función que calculará el número de bits necesarios para cada variable, así como su máximo valor posible
206. void calculateBits() {
207. for (int i = 0; i < nVariables; ++i) {
208. double dRange = aLimites[i].second - aLimites[i].first;
209. double dExponent = pow(10, nBits);
210. double dLog = log2(dRange \* dExponent);
211. double dAux = ceil(dLog);
212. aBitsVariables[i] = dAux;
213. aMaxVariables[i] = pow(2, aBitsVariables[i]);
214. nTotalBits += dAux;
215. }
216. }
217. // Inicializamos por primera vez los vectores a utilizar
218. void initializeVectors() {
219. aBitsVariables.resize(nVariables);
220. aMaxVariables.resize(nVariables);
221. aFormulaEstatica.resize(nVariables);
222. aPobladores.resize(nPoblacion, vector < ulong > (nVariables));
223. aValuesPoblation.resize(nPoblacion, vector < double > (nVariables));
224. aValuesZ.resize(nPoblacion);
225. aAcumulateZ.resize(nPoblacion, 0);
226. }
227. int main(int argc, char
228. const \* argv[]) {
229. // Función que pide los datos desde terminal
230. askData();
231. // Función que genera los valores random desde el reloj en lugar del algoritmo
232. srand(time(0));
233. // Iniciamos vectores, calculamos bits y generamos la primer población necesaria
234. initializeVectors();
235. calculateBits();
236. startPoblation();
237. cout << endl;
238. // Iniciamos las iteraciones
239. for (int i = 0; i < nIteraciones; ++i) {
240. cout << "\n===== Iteración " << i << " =====\n" << endl;
241. // En caso de que falten pobladores, se generan a partir del más dominante
242. while (aPobladores.size() < nPoblacion) {
243. mutatePoblator();
244. }
245. // Calculamos los valores necesarios, tanto Z como la acumulada de Z
246. calculateValues();
247. // Generamos los valores random a calcular en la acumulada
248. countRandomValues();
249. // Generamos vectores auxiliares donde copiaremos los individuos más dominantes
250. vector < vector < ulong > > aPobladoresAux(nPoblacion, vector < ulong > (nVariables));
251. vector < vector < double > > aValuesPoblationAux(nPoblacion, vector < double > (nVariables));
252. for (int j = 0; j < aCountedValues.size(); ++j) {
253. aPobladoresAux[j] = aPobladores[aCountedValues[j].first];
254. aValuesPoblationAux[j] = aValuesPoblation[aCountedValues[j].first];
255. }
256. aPobladores.clear();
257. aValuesPoblation.clear();
258. for (int j = 0; j < aCountedValues.size(); ++j) {
259. aPobladores.push\_back(aPobladoresAux[j]);
260. aValuesPoblation.push\_back(aValuesPoblationAux[j]);
261. }
262. }
263. // En la última iteración, rellenamos los pobladores faltantes y calculamos sus valores
264. while (aPobladores.size() < nPoblacion) {
265. mutatePoblator();
266. }
267. calculateValuesFinal();
268. cout << "\nSolucion optima para Z es: " << finalValueOfZ.second << " \nCon:\n";
269. for (int i = 0; i < nVariables; ++i) {
270. cout << "\tx" << i << " = " << aValuesPoblation[finalValueOfZ.first][i] << endl;
271. }
272. return 0;
273. }

## PideDatos.h

1. void askData() {
2. cout << "Bienvenido al sistema de solucion de Problemas de Programacion Lineal" << endl;
3. cout << "Introduca el numero de variables (Maximo 4): ";
4. cin >> nVariables;
5. while (nVariables > 4) {
6. cout << "Numero de variables excedido, intente nuevamente: ";
7. cin >> nVariables;
8. }
9. aFormulaObjetivo.resize(nVariables, 0);
10. cout << "Introduzca el valor de la variables en la funcion Z:\n";
11. for (int i = 0; i < nVariables; ++i) {
12. cout << "\tx" << i << " = ";
13. cin >> aFormulaObjetivo[i]; //cout<<endl;
14. }
15. aLimites.resize(nVariables);
16. for (int i = 0; i < nVariables; ++i) {
17. cout << "Introduzca el limite izquierdo de la variable x" << i << ":\n";
18. cin >> aLimites[i].first;
19. cout << "";
20. cout << "Introduzca el limite derecho de la variable x" << i << ":\n";
21. cin >> aLimites[i].second;
22. cout << "";
23. cout << endl;
24. }
25. cout << "\nQue desea hacer?\n\t1) Maximizar\n\t2) Minimizar" << endl;
26. cin >> nOpcion;
27. while (nOpcion != 1 && nOpcion != 2) {
28. cout << "Opcion no valida, intente nuevamente: ";
29. cin >> nOpcion;
30. }
31. cout << "\nIntroduzca el numero de restricciones (Maximo 5): ";
32. cin >> nRestricciones;
33. while (nRestricciones > 5) {
34. cout << "Numero de restricciones excedido, intente nuevamente: ";
35. cin >> nRestricciones;
36. }
37. aRestricciones.resize(nRestricciones, vector < double > (nVariables + 2));
38. for (int i = 0; i < nRestricciones; ++i) {
39. cout << "\tIntroduzca el valor de la variables para la R" << i << ":\n";
40. int j;
41. for (j = 0; j < nVariables; ++j) {
42. cout << "\t\tx" << j << " = ";
43. cin >> aRestricciones[i][j];
44. }
45. cout << "\n\tQue desea hacer?\n\t\t1) <=\n\t\t2) >=\n\t";
46. cin >> aRestricciones[i][j];
47. while (aRestricciones[i][j] != 1 && aRestricciones[i][j] != 2) {
48. cout << "Opcion no valida, intente nuevamente: ";
49. cin >> aRestricciones[i][j];
50. }
51. cout << "\tIntroduzca el valor de la variable a igualar para la R" << i << ": ";
52. cin >> aRestricciones[i][++j];
53. cout << "";
54. cout << endl;
55. }
56. cout << "\nIntroduzca el numero de poblacion: ";
57. cin >> nPoblacion;
58. cout << "\nIntroduzca el numero de iteraciones (maximo 100): ";
59. cin >> nIteraciones;
60. while (nIteraciones > 100) {
61. cout << "Numero de iteraciones excedido, intente nuevamente: ";
62. cin >> nIteraciones;
63. }
64. cout << "\nIntroduzca el numero de bits de precision: ";
65. cin >> nBits;
66. }