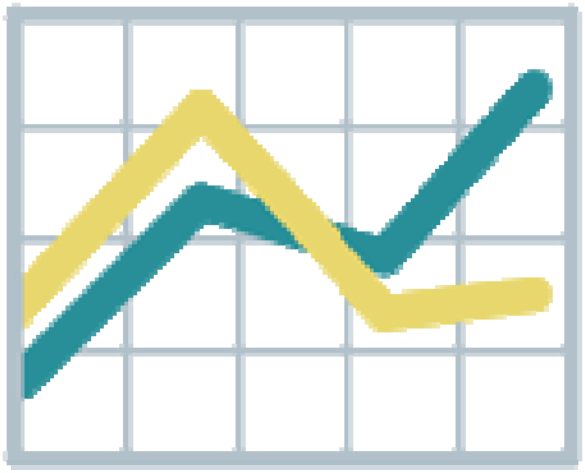
Proyecto Final



Manual de Usuario

David Jiménez Xolalpa A00994415

Alejandro Méndez González A01210989

Introducción:

Para resolver el problema decidimos utilizar el algoritmo de Recocido Simulado. Es un método de optimización inspirado en el proceso de templado de metales usado desde alrededor de los 5000 años antes de Cristo. El proceso de templado de metales consiste de tres fases: una fase de calentamiento a una temperatura determinada; en la segunda fase se sostiene la temperatura alta lo cual permite a las moléculas acomodarse en estados de mínima energía; y se sigue de una fase de enfriamiento controlado para aumentar el tamaño de sus cristales y reducir sus defectos.

1. Por qué se seleccionó dicho algoritmo.

Consideramos también el algoritmo de búsqueda tabú y de colonia de hormigas. Al final nos decidimos por el recocido simulado porque además de ser relativamente fácil de implementar, estamos más familiarizados con la teoría detrás de él ya que lo estudiamos en cursos anteriores. Por lo anterior creímos que nos sería más fácil adaptarlo para solucionar el problema que se nos propuso.

2. El diseño y selección de todos los parámetros que se requieren.

Para indicar el problema de optimización a solucionar se especifican el número de variables y de restricciones, junto con los valores de estas últimas. Para el recocido simulado, se especifica la temperatura inicial, la tasa de disminución de ésta y el número de iteraciones máximo. Estos últimos son los valores a modificar para encontrar una respuesta que se acerque más al óptimo.

3. Cómo se incorporaron las restricciones.

Las representamos por medio de un arreglo bidimensional de tamaño [número de restricciones][número de variables +2] donde las casillas extras de la segunda dimensión representan el tipo de desigualdad (>=,<=,=) y el “lado derecho” de la misma, respectivamente. Se toman en cuenta para la generación de la solución inicial para que ésta quede dentro de un rango razonable, además de influir en el cálculo del nivel de “bondad” de las soluciones posibles generadas.

4. Cómo genera los puntos vecinos a los que se puede mover.

Decidimos restringir los números a dos decimales. De esta manera tendríamos un número finito de posibilidades. Para generar los vecinos se modifica la solución actual en un paso hacia arriba y hacia abajo.

5. Cómo selecciona el punto vecino al que se debe mover.

La selección del punto vecino se realiza al azar. Se juntó con el paso de la generación de los vecinos. En cada variable se tiene la posibilidad de aumentar o disminuir su valor en un paso. De esta manera se genera un vecino al azar de los posibles diferentes vecinos. Éste es el vecino seleccionado.

Algoritmo en Pseudocódigo:

El siguiente código representa una versión general del algoritmo de recocido simulado:

1. s := GeneraUnaSolucionInicial();
2. T := T0;
4. **mientras** (CondicionesParoNoActivar(g, T)) **hacer**
5. s′:= TomaUnVecinoAleatorioDe(s);
6. **si** (f(s′) < f(s))
7. s := s′;
8. **o bien**
9. **si** (Random(0, 1.0) < exp((f(s) − f(s′))/T))
10. s := s′;
11. **fin si**;
12. **fin si**;
13. g := g + 1; T := Actualiza(g, T);
15. **fin mientras**

Código Fuente:

```
package com.solver;

import java.text.DecimalFormat;
import java.text.NumberFormat;

public class Solver {

	public static NumberFormat df = new DecimalFormat("#0.000");

	int noRestricciones = 3;
	int noVariables = 2;
	boolean max = true;
	final double LT = 1;
	final double EQ = 2;
	final double GT = 3;

	public double temperatura;
	public double temperaturaPaso = 0.99995;

	public double[] objetivo = new double[noVariables];
	public double[][] restricciones = new double[noRestricciones][noVariables+2];
	public double[][] rangos = new double[noVariables][2];

	public double[] solucionActual;

	int midValue = 2000;

	/*****************************************David*************************
*****************/

	public final double precisionFinal = 0.01;
```

```
	/*****************************************Alejandro**********************
*****************/


	public Solver() {
		// TODO: Inicializar temperatura
		temperatura = 90000000;
		for (int i = 0; i < rangos.length; i++) {
			//double[] inicial = { Double.NaN, Double.NaN };
			double[] inicial = { 0, 0 };
			rangos[i] = inicial;
		}
	}

	public void entrada() {
		double[] objetivo = {10, 15};
		this.objetivo = objetivo;
		double[] r1 = {5, 10, LT, 100};
		double[] r2 = {2.5, 2.5, LT, 2250};
		double[] r3 = {2, 1, LT, 1200};
		// No negatividad
		double[] r4 = {1, 0, GT, 0};
		double[] r5 = {0, 1, GT, 0};
		this.restricciones[0] = r1;
		this.restricciones[1] = r2;
		this.restricciones[2] = r3;
	}

	/** solo funciona con dos variables por el momento y para LT*/
	public void getRangos() {
		for (int i = 0; i < restricciones.length; i ++) {
			for (int j = 0; j < restricciones.length; j++) {
				if (j == i) continue; // skip one self

				// Construir matriz para resolver
				double[][] matriz = construirMatriz(restricciones[i],
restricciones[j]);
				double[] resultadosMatriz = Gauss_Jordan.resolver(matriz);
				System.out.println(Solver.printArray(resultadosMatriz));
				for (int k = 0; k < noVariables; k++) {
					// Checar el piso
					if (Double.isNaN(rangos[k][0]) &&
resultadosMatriz[k] >= 0) rangos[k][0] = resultadosMatriz[k];
					else if (rangos[k][0] > resultadosMatriz[k] &&
resultadosMatriz[k] >= 0) rangos[k][0] = resultadosMatriz[k];
					// Checar el techo
					if (Double.isNaN(rangos[k][1])) rangos[k][1] =
resultadosMatriz[k];
					else if (rangos[k][1] < resultadosMatriz[k])
rangos[k][1] = resultadosMatriz[k];
				}
			}
		}
```

```
	}

	public double[][] construirMatriz(double[] ... args) {
		double[][] matriz = new double[args.length][args[0].length-1]; // Remove
sign

		for (int i = 0; i < args.length; i++) {
			double[] fila = new double[args[i].length - 1];
			System.arraycopy(args[i], 0, fila, 0, args[i].length-2);
			fila[fila.length-1] = args[i][args[i].length-1];
			matriz[i] = fila;
		}
		return matriz;
	}

	/** Genera una solucion aleatoria dentro del rango de las restricciones
(factible)*/
	public double[] generaUnaSolucionInicial() {
		double[] solucion = new double[noVariables];
		for(int i = 0; i < solucion.length; i++) {
			double diferencia = rangos[i][1] - rangos[i][0];
			solucion[i] = Math.random() * diferencia + rangos[i][0];
//			solucion[i] = 400;
		}
		solucionActual = solucion;
		return solucion;
	}

	/** revisa que los valores de la solucion "posible" cumplan con la restriccion
"restriccion" */
	public double pruebaRestriccion(double [] posible, double [] restriccion) {
		boolean res = false;
		double temp = 0.0;
		for(int i = 0; i < posible.length; i++)
			temp += restriccion[i] * posible[i];
		double diferencia = Math.abs(restriccion[restriccion.length-1] - temp);
		if(restriccion[restriccion.length-2] == this.LT){
			if(temp < restriccion[restriccion.length-1])
				return diferencia;
			else
				return -diferencia;
		} else if(restriccion[restriccion.length-2] == this.EQ){
			if(temp == restriccion[restriccion.length-1])
				return diferencia;
			else
				return -diferencia;
		} else if(restriccion[restriccion.length-2] == this.GT){
			if(temp > restriccion[restriccion.length-1])
				return diferencia;
			else
				return -diferencia;
		}
		return -diferencia;
	}
```

```
	/** disminuye siempre la temperatura en X cantidad */
	public void actualizaTemperatura() {
		temperatura *= temperaturaPaso;
	}

	/** Funcion de evaluacion: Separar en casos si no se cumplen las restricciones se le da un valor negativo e ignorar Z
	 *  Si cumple con las restricciones, solo tomar en cuenta el valor de Z */
	public double queTanBuenoEs(double[] posible) {
		boolean cumple = true;
		for(int i = 0; i < restricciones.length && cumple; i++){
			cumple &= pruebaRestriccion(posible, restricciones[i]) >= 0;
		}

		if(cumple)
			return midValue - calculaZ(posible); // Al maximizar, debe ser la menor z posible
		else{
			double diferencia = 0.0;
			for(int i = 0; i < restricciones.length; i++){
				if(pruebaRestriccion(posible, restricciones[i]) < 0)
					diferencia += pruebaRestriccion(posible, restricciones[i]);
			}
			//Diferencia ya es negativo
			return midValue - diferencia;
		}

	}

	/** Calcula Z */
	public double calculaZ(double[] posible){
		double res = 0.0;
		for(int i = 0; i < posible.length; i++)
			res += this.objetivo[i] * posible[i];
		return res;
	}


	/** Probabilidad de aceptar */
	public boolean boltzmann(double actual, double posible) {
		double prob;

		if (max && posible < actual) return true;
		if (!max && posible > actual) return true;

		prob = Math.exp(Math.abs(posible-actual)/this.temperatura);

		return Math.random() < prob;
	}

	/** los vecinos a una distancia definida por <b>precision</b> */
	public double[] vecinos() {

		double precision = getPrecision();
```

```
			double[] vecinoElegido = new double[noVariables];
			for (int i = 0; i < noVariables; i++) {
				double rand = Math.random();
				if (rand >= 0.5) vecinoElegido[i] = solucionActual[i] +
precision;
				else vecinoElegido[i] = solucionActual[i] - precision;
				// Mantenemos la soluciÃ³n en nuestro universo finito
				if (vecinoElegido[i] > rangos[i][1]) vecinoElegido[i] =
rangos[i][1];
				if (vecinoElegido[i] < rangos[i][0]) vecinoElegido[i] =
rangos[i][0];
			}
			return vecinoElegido;
		}

		/** Se obtiene una precisiÃ³n dependiente de la temperatura */
		public double getPrecision() {
			double precision = temperatura * 0.1;
//			return precision>precisionFinal?precision : precisionFinal;
			return 1;
		}

		public static String printArray(double[][] a) {
			StringBuilder b = new StringBuilder();
			b.append("{");
			for (double[] d : a) b.append(printArray(d));
			b.append("}");
			return b.toString();
		}

		public static String printArray(double[] a) {
			StringBuilder b = new StringBuilder();
			b.append("[");
			for (double d : a) {
				b.append(df.format(d));
				b.append(" | ");
			}
			b.append("]");
			return b.toString();
		}

		public static void main(String[] args) {
			Solver s = new Solver();
			s.entrada();
			s.getRangos();
			System.out.println("_____RANGOS_____________");
			System.out.println(printArray(s.rangos));
			System.out.println("_____INICIAL_____");
			System.out.println(printArray(s.generaUnaSolucionInicial()));
			System.out.println("_____RECOCIDO_____");

			int i = 0;
			while (s.temperatura > 0 && i < 1000000) {
				double[] posible = s.vecinos();
```

```
                    boolean acepto = s.boltzmann(s.queTanBuenoEs(s.solucionActual),
s.queTanBuenoEs(posible));
                    if (acepto) s.solucionActual = posible;
                    s.actualizaTemperatura();
                    if (i % 10000 == 0) System.out.println(i + " - " +
printArray(s.solucionActual) + "=>" + s.calculaZ(s.solucionActual)  + ", " +
s.temperatura + " -> " + s.queTanBuenoEs(s.solucionActual));
//                  System.out.println(i + " - " + printArray(s.solucionActual) +
"=>" + s.calculaZ(s.solucionActual) + ", " + s.temperatura + " -> " +
s.queTanBuenoEs(s.solucionActual));
                    i++;
//                  if (i % 10000 == 0) break;

            }
            System.out.println(printArray(s.solucionActual));
            double[] prueba = {400, 400};
            System.out.println(s.queTanBuenoEs(prueba));
      }

}
```

```
package com.solver;

//  Tomado de http://www.lawebdelprogramador.com/foros/Algoritmia/710641-
Algoritmo_de_gauss_jordan_en_java.html

import java.util.*;
public class Gauss_Jordan
{
      static void muestramatriz(double matriz[][], int var)
      {
//            for(int x=0;x<var;x++)
//            {
//                  for(int y=0;y<(var+1);y++)
//                        //System.out.print(" "+matriz[x][y]+" |");
//                  //System.out.println("");
//            }

      }
      static void pivote(double matriz[][],int piv,int var)
      {
            double temp=0;
            temp=matriz[piv][piv];
            for(int y=0;y<(var+1);y++)
            {

                  matriz[piv][y]=matriz[piv][y]/temp;
            }
      }
      static void hacerceros(double matriz[][],int piv,int var)
      {
            for(int x=0;x<var;x++)
            {
                  if(x!=piv)
                  {
```

```
                        double c=matriz[x][piv];
                        for(int z=0;z<(var+1);z++)
                              matriz[x][z]=((-1*c)*matriz[piv][z])+matriz[x][z];
                  }
            }
      }

      /** Debe incluir el resultado */
      public static double[] resolver(double[][] matriz)
      {
            int var = matriz.length;
            int piv = 0;
            for(int a=0;a<matriz.length;a++)
            {
                  pivote(matriz,piv,var);

                  //System.out.println("\tRenglon "+(a+1)+" entre el pivote");
                  muestramatriz(matriz,var);

                  //System.out.println("");

                  //System.out.println("\tHaciendo ceros");
                  hacerceros(matriz,piv,var);

                  muestramatriz(matriz,var);
                  //System.out.println("");
                  piv++;
            }

            double[] respuesta = new double[var];

            for(int x=0;x<var;x++)
            {
                  //System.out.println("La variable X"+(x+1)+" es:
"+matriz[x][var]);
                  respuesta[x] = matriz[x][var];
            }

            return respuesta;
      }

}
```

Manual de Usuario:

El programa se realizó sin ninguna interfaz gráfica para introducir la información. Se realiza desde el código, y al compilarse da el resultado.

Instrucciones de uso:

Para introducir el modelo es necesario sustituir los valores del arreglo *objetivo*, y del arreglo bidimensional *restricciones.* Para lo anterior se agregó el método entrada( ) en el cual se encuentra un ejemplo de cómo introducir un problema al programa. Las constantes LT, EQ y GT representan un menor que, igual, y mayor que respectivamente. Finalmente se tiene que cambiar manualmente el valor de la variables **noRestricciones, noVariables** y **max**, que indican respectivamente el número de restricciones, número de variables y si se trata de un problema de maximización o de minimización.

Ejemplo:

Max Z = 10x1 + 15x2
s.a. 5 x1 + 10 x2 <= 6000
2.5 x1 + 2.5 x2 <= 2250
2 x1 + 1 x2 <= 1200

```
int noRestricciones = 3;
int noVariables = 2;
boolean max = true;

.
.
.
.

public void entrada() {
        double[] objetivo = {10, 15};
        this.objetivo = objetivo;
        double[] r1 = {5, 10, LT, 100};
        double[] r2 = {2.5, 2.5, LT, 2250};
        double[] r3 = {2, 1, LT, 1200};
        // No negatividad
        double[] r4 = {1, 0, GT, 0};
        double[] r5 = {0, 1, GT, 0};
        this.restricciones[0] = r1;
        this.restricciones[1] = r2;
        this.restricciones[2] = r3;
}
```

Comentarios y Conclusiones:

Uno de los primeros retos a los que nos enfrentamos fue poder generar una solución inicial que fuera factible. Nuestro primer acercamiento fue generar una solución no factible, y que el recocido también llevara a una solución factible para después buscar la z deseada. En este acercamiento tuvimos el problema de poder dar un valor de energía adecuado.
Para poder mejorar la solución inicial, pensamos en obtener los rangos mínimos y máximos de las variables usando Gauss - Jordan. De esta manera la primera respuesta generada ya se encuentra dentro o muy cerca del universo posible de soluciones.
El otro problema al que nos enfrentamos fue probar los parámetros de temperatura y la tasa de enfriamiento para que el problema se acercara más a solución correcta.