Algorítmica II. Algoritmos Metaheurísticos

Joaqín Roiz Pagador[1]

Universidad Pablo de Olavide, Carretera Utrera s/n, Sevilla, España,
quiniroiz@gmail.com

Abstract. El siguiente documento pertenece a una serie de documentos que pretende servir a modo de resumen para el temario de Algorítmica II para el Grado en Ingeniería Informática en Sistemas de Información.

Keywords: algoritmos, metaheurística, enfriamiento, simulado, termodinámica

Enfriamiento Simulado.

El enfriamiento simulado(SA) aplicado a los problemas de optimización emerge del trabajo de S. Kirkpatrick y V.Cerny. En la época de 1980, SA ha tenido un mayor impacto que las búsquedas heurísticas debido a su simpleza y eficiencia solventando problemas de optimización combinacional. Ha sido extendido por su continua optimización de problemas.

SA se basa en el principio de mecánica estadística donde se simula el proceso de enfriamiento lento que se aplica para obtener una estructura resistente de cristal.

Podemos basar el enfriamiento simulado como una simulación de critalización que utiliza los fundamentos termodinámicos. Frente a los algoritmos clásicos presenta la mejora de que, a grandioso espacio a recorrer, no podremos asegurar un óptimo global en un tiempo razonable.

Por tanto, utilizaremos tres estrategias posibles:

- Aceptar peores soluciones (Ref. Hill Climbing).
- Modificar estructuras de entornos (de forma similar al caballo de ajedrez modificando sus posibles movimientos).
- Comenzar búsquedas desde otras soluciones iniciales.

Nosotros utilizaremos la primera de ellas.

Sabremos el estado de la búsqueda hacia la solución mediante un parámetro de control, que modela la función de probabilidad. Disminuye así la probabilidad de estos movimientos hacia soluciones peores.

La probabilidad de aceptación disminuye conforme se va adelantando camino.

Referenciando a David D. de Vega: "De forma similar a una búsqueda de montículos en un campo de fútbol, nuestro algoritmo realizará un barrido de sus vecinos en el punto que esté (mirará a su alrededor). Llegado a un punto de probabilidad de aceptación reducido, se le enviará a otra parte del campo para buscar en otro punto diferente, de forma que podamos dar con un óptimo local mejor y, tal vez, con el óptimo global".

Esta probabilidad se parametrizará en temperatura (probabilidad de aceptación para la solución 'mala') y enfriamiento. Buscaremos por entoros por criterio de aceptación, así pues.

Características

- Método de búsqueda por entornos caracterizado por un criterio de aceptación de soluciones vecinas que varía en tiempo de ejecución.
- Variables de entrada:
  - Temperatura → En qué medida se aceptarán las soluciones vecinas 'peores'.
    - T_0 → Valor alto, temperatura(T) inicial.
    - T_f → Condición de parada.
  - T se va reduciendo en cada iteración (enfriamiento).
- En cada iteración se genera un número de vecinos L(T)
  - Pueden ser fijos siempre.
  - Depende de cada iteración.
- Cada vez que se genera un vecino, se aplica la probabilidad de aceptación para ver si sustituye a la actual.
  - Si la solución vecina es mejor que la actual, se acepta automáticamente, como una búsqueda local clásica.
  - Si es peor, existe aun la probabilidad de que el vecino sustituya la solución actual (Salir de óptimos locales).

- Consideraciones de T:
  - A mayor T, mayor probabilidad(P_a) de soluciones peores (cuantas más iteraciones demos, menos soluciones peores aceptaremos).
  - Aceptar soluciones mucho peores al principio, pero no al final.
- Consideraciones sobre δ:
  - A menor δ, mayor P_a de soluciones peores.
  - Tras cada iteración se enfría la temperatura y se pasa a la siguiente iteración.

A continuación se listan las antologías al sistema de enfriamiento:

Termodinámica	SA
Estado del Sistema	Soluciones factibles
Energía	Coste
Cambios de estado	Solución del entorno
Temperatura	**Parámetro de control**
Estado congelado	Solución heurística

Resumiendo:

En cada iteración, un vecino aleatorio es generado. Se comprobará que el coste de la función es siempre aceptado (mejor). En caso contrario, será seleccionado según la probabilidad de aceptación que depende de la temperatura y el total de su degradación δE de la función objetivo. Este valor representa la diferencia del valor objetivo. En cada movimiento la probabilidad bajará, por lo general representada de la siguiente forma:

$$P_a(\delta, T) = \epsilon^{-\frac{\delta}{T}}$$

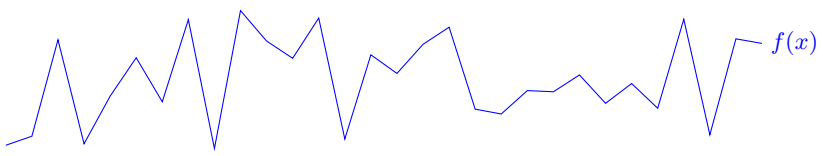


Fig. 1. Función donde podemos observar diferentes óptimos locales y uno global escondido entre ellos.

input : Cooling schedule.
$s \leftarrow s_0$
$T \leftarrow T_{max}$
do
  do
    Generate a random neighbor s'
    $\delta E = f(s') - f(s)$
    if $\delta E \leq 0$ **then**
      $s = s'$
    end
    Accept s' with a probability $\epsilon^{-\frac{\delta E}{T}}$
  while *Equilibrium condition*;
  $T = g(T)$
while *Stopping criteria Satisfied*;
output: Best Solution found.

Algorithm 1: Algoritmo de Enfriamiento Simulado

– La función de aceptación de probabilidad: Es el elemento principal del enfriamiento simulado que permite utilizar movimientos de vecinos poco aceptables.

– El encargado de enfriamiento: Enfriará la temperatura en cada paso del algoritmo. Ésta es la potencia de efectividad del algoritmo.

Ejemplo, ejercicios de prácticas.

Ejercicio 1 Queremos hallar el máximo de $f(x) = x^3 + 900x + 100$ entre $x = 0$ y $x = 31$. Para resolver el problema usando SA, se propone seguir la siguiente estrategia. En primer lugar, discretizamos el rango de valores de x con vectores binarios de 5 componentes entre 00000 y 11111. Estos 32 vectores constituyen S las soluciones factibles del problema.

Le damos un valor inicial T intuitivamente, por ejemplo, $T_0 = 100$ o 500 y en cada iteración del algoritmo lo reduciremos un 10%, es decir, utilizando la estrategia de descenso geométrico: $T_k = 0.9 \cdot T_{k-1}$.

Cada iteración consiste en lo siguiente:

1. El número de vecinos queda fijado a 5, siendo éstos variaciones de un bit de la solución. Por ejemplo, si partimos de 00011, los 5 posibles vecinos resultantes serían: 10011, 01011, 00111, 00001, 00010.
2. Para aplicar el criterio de aceptación, escogeremos un vecino, buscamos su coste asociado en la tabla auxiliar que se proporciona y calculamos la diferencia con la solución actual. Si está má cerca del óptimo se acepta, sino, se aplica $P_a(\delta, T) = \exp^{-\frac{\delta}{T}}$. Siendo T la temperatura actual y δ la diferencia de costes entre la solución candidata y la actual.
3. Concluya la búsqueda cuando el proceso se enfríe o cuando no se acepte ninguna solución de su vecindad.

Solución propuesta en Clase de Prácticas:

```
private static int[] enfriamientoSimulado() {
        int[] bestSolution = {0, 0, 0, 0, 0};
        int[] solutions = {0, 0, 0, 0, 0};
        double mejorValor = Integer.MIN_VALUE;
        double temp = 100;
        int nuevo;
        double valorFuncion;
        while (temp > 1) {
            nuevo = (int) (Math.random() * solutions.length);
            Util.permuta(solutions, nuevo);
            for (int i = 0; i < solutions.length; i++) {
                Util.permuta(solutions, i);
                valorFuncion = funcionCalc(solutions);
                if (aceptarProbabilidad(valorFuncion, mejorValor, temp) > Math.random()) {
                    mejorValor = valorFuncion;
                } else {
                    Util.permuta(solutions, i);
                }
                if (mejorValor > funcionCalc(bestSolution)) {
                    bestSolution = Arrays.copyOf(solutions, solutions.length);
                }
            }
            temp -= 0.1 * temp;
        }
        return bestSolution;
    }

    private static void permuta(int[] sVecino, int n) {
        if (sVecino[n] == 1) {
            sVecino[n] = 0;
        } else {
            sVecino[n] = 1;
        }
    }
```

```
private static double funcionCalc(int[] array) {
    int n = binaryToDecimal(array);
    return Math.pow(n, 3) - 60 * Math.pow(n, 2) + 900 * n + 100;
}

private static double aceptarProbabilidad(double valorFuncion, double mejorValor,
     double temperatura) {
    if (valorFuncion > mejorValor) {
        return 1.0;
    }
    return Math.exp((mejorValor - valorFuncion) / temperatura);
}

private static int binaryToDecimal(int[] array) {
    int n = 0;
    for (int i = 0; i < array.length; i++) {
        if (array[i] != 0) {
            n += Math.pow(2, i);
        }
    }
    return n;
}
```

Ejercicio 2 Queremos resolver el problema de la mochila utilizando SA. Para ello, suponga que la capacidad de la mochila es q = 180 y que puede insertar hasta 100 elementos, con pesos aleatorios comprendidos entre [1,100], esto es, considere el siguiente vector P = $\{p_1, p_2, \ldots, p_{100}\}$, con $p_i = [1, 100]$. Se permite que dos elementos pesen lo mismo.

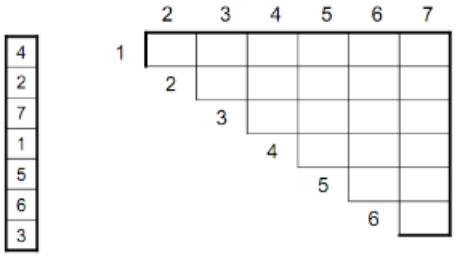
1. **¿Qué codificación escogería para modelar el problema?** Para comenzar, un vector con cada uno de los elementos almacenados, éstos ordenados de menor a mayor. A su vez, otro vector con las posiciones de estos elementos y el valor 0, en caso de no incluírlo o 1 en caso de hacerlo, será nuestro vector de soluciones.
2. **¿Qué temperatura inicial pondría?** Para comenzar, elegiremos una temperatura bastante elevada,10000, dejando así un lento enfriamiento y mayor probabilidad de óptimo global.
3. **¿Cuántas combinaciones reales existen?** Suponiendo la representación del espacio de soluciones escogidos en el primer apartado, la cantidad de combinaciones posibles es de 2^n.

Implementación propuesta:

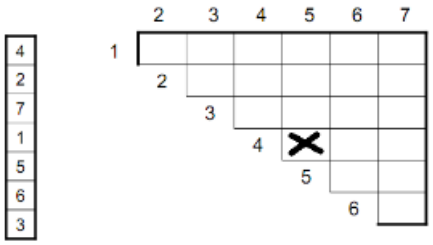
```
public static int[] enfriamientoSimulado(int[] elementos, int n, int capacidad) {
        int[] bestSolution = new int[n];
        int[] solutions = new int[n];
        int pesoGlobal = 0;
        double temp = 10000;
        int peso;
        while (temp > 1) {
            Util.permuta(solutions, (int) (Math.random() * solutions.length));
            int[] vecino = Arrays.copyOf(solutions, solutions.length);
            Util.permuta(vecino, (int) (Math.random() * vecino.length));
            peso = calcularPeso(vecino, elementos);
            if (peso < capacidad && aceptarProbabilidad(peso, pesoGlobal, capacidad, temp)
                > Math.random()) {
                pesoGlobal = peso;
            }
            int mejorPeso = calcularPeso(bestSolution, elementos);
            if (pesoGlobal <= capacidad && pesoGlobal > mejorPeso) {
                bestSolution = vecino;
            }

            temp -= 0.1 * temp;
        }

        return bestSolution;
    }

    private static double aceptarProbabilidad(int peso, int capacidad, int pesoGlobal,
        double temperatura) {
        if (peso > pesoGlobal && peso < capacidad) {
            return 1.0;
        }
        return Math.exp((pesoGlobal - peso) / temperatura);
    }

    public static int calcularPeso(int[] solution, int[] elementos) {
        int res = 0;
        for (int i = 0; i < elementos.length; i++) {
            if (solution[i] != 0) {
                res += elementos[i];
            }
        }
        return res;
}
```

Problema 1 Se desea fabricar un material aislante compuesto por siete materiales distintos. Se desea encontrar cuál es el orden en que deben mezclarse dichos materiales para asegurar que sean lo más aislante posible.

Suponga la siguiente situación:



Se puede apreciar un vector en el que aparece el orden en el que se combinan los materiales y, además, una matriz triangular para identificar posibles permutaciones de orden, es decir, posibles soluciones a las que ir.



Así, la posición (4,5) estaría haciendo referencia a que se intercambie el orden de los materiales 4 por el 5.

Para evaluar la cantidad aislante de la solución, suponga que ´sta se mide por la suma de los tres primeros componentes, esto es, sobre la figura de arriba sería $4 + 2 + 7 = 13$. Si realizamos la permutación 4 5, entonces, la nueva solución candidata sería $[5, 2, 7, 1, 4, 6, 3]$, siendo su coste $5 + 2 + 7 = 14 (> 13)$ y por tanto se aceptaría. De esta condición se deduce que el máximo se alcanzará cuando tengamos en las tres posiciones superiores $\{5, 6, 7\}$, en cualquier orden posible o, en otras palabras, que el máximo global sería $5 + 6 + 7 = 18$.

Solución Propuesta:

```
public class Material {

    private List<Integer> materiales;

    public Material() {
        Set<Integer> aux = new HashSet<>();
        do {
            aux.add((int) (Math.random() * 7) + 1);
        } while (aux.size() < 7);
        materiales = new ArrayList<>(aux);
    }
```

```
    public void generarVecino() {
        generarVecino(materiales);
    }

    public void generarVecino(List<Integer> materiales) {
        int n1 = (int) (Math.random() * materiales.size());
        int n2 = (int) (Math.random() * materiales.size());
        int aux = materiales.get(n1);
        materiales.set(n1, materiales.get(n2));
        materiales.set(n2, aux);
    }

    public int cantidadAislante() {
        return cantidadAislante(materiales);
    }

    public int cantidadAislante(List<Integer> materiales) {
        int res = 0;
        for (int i = 0; i < 3; i++) {
            res += materiales.get(i);
        }
        return res;
    }

    public List<Integer> listarMateriales() {
        return materiales;
    }
}



public class EnfriamientoMateriales {

    public static Material encontrarMejorMaterial() {
        Material vecino = new Material();
        Material mejor = vecino;
        int cantidadMejor = 0;
        double tmp = 10000;
        while (tmp >= 10) {
            vecino.generarVecino();
            if (aceptarProbabilidad(vecino, mejor, tmp) > Math.random()) {
                cantidadMejor = vecino.cantidadAislante();
            }
            if (cantidadMejor > mejor.cantidadAislante()) {
                mejor = vecino;
            }
            tmp -= tmp * 0.1;
        }
        return mejor;
    }

    private static double aceptarProbabilidad(Material vecino, Material mejor, double
         temp) {
        if (vecino.cantidadAislante() > mejor.cantidadAislante()) {
            return 1.0;
        }
        return Math.exp((mejor.cantidadAislante() - vecino.cantidadAislante()) / temp);
    }
}
```

Problema 2 Un modelo de coche se configura a partir de n componentes distintos. Cada uno de esos componentes puede tomar $m_i (i = 1, \ldots, m)$ posibles valores (v_{ij}). La afinidad de los consumidores para cada posible valor v_{ij} es a_{ij} y el coste c_{ij}. Se conoce también la importancia, w_i, que los consumidores atribuyen a cada componente. Se desea encontrar una combinación de componentes que alcance la máxima afinidad global con los gustos de los consumidores y cuyo coste no supere un umbral M.

Para poder comprobar que la metaheurística diseñada funciona, suponga que $n = m = 4$. Los valores a_{ij} , v_{ij} y w_i estarán entre $[0, 1]$. Por último, el coste estará entre $[1, 100]$.

Solución Propuesta:

```
public class Componente {

    private double afinidad;
    private double valor;
    private double importancia;
    private double coste;

    public Componente() {
        afinidad = Math.random();
        valor = Math.random();
        importancia = Math.random();
        coste = Math.random() * 100;
    }

    public double afinidadGlobal() {
        return afinidad + valor + importancia;
    }

    public double getCoste() {
        return coste;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Componente) {
            Componente o = (Componente) obj;
            return o.getCoste() == getCoste() && o.afinidadGlobal() == afinidadGlobal();
        }
        return false;
    }

    @Override
    public String toString() {
        return "\nAfinidad: " + afinidadGlobal()
                    + ", Coste: " + getCoste();
    }

}


public class ModeloCoche implements Cloneable {

    private List<Componente> componentes;
    private int[] coche;

    public ModeloCoche(int n) {
        componentes = new ArrayList<>();
        coche = new int[n];
```

```
        for (int i = 0; i < n; i++) {
            componentes.add(new Componente());
        }
    }

    public void generarVecino() {
        int pos = (int) (Math.random() * coche.length);
        permuta(coche, pos);
    }

    private static void permuta(int[] sVecino, int n) {
        if (sVecino[n] == 1) {
            sVecino[n] = 0;
        } else {
            sVecino[n] = 1;
        }
    }

    public double afinidadGlobalCoche() {
        double af = 0;
        for (int i = 0; i < coche.length; i++) {
            if (coche[i] != 0) {
                af += componentes.get(i).afinidadGlobal();
            }
        }
        return af;
    }

    public double costeCoche() {
        double coste = 0;
        for (int i = 0; i < coche.length; i++) {
            if (coche[i] != 0) {
                coste += componentes.get(i).getCoste();
            }
        }
        return coste;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof ModeloCoche) {
            ModeloCoche o = (ModeloCoche) obj;
            return coche.equals(o.coche);
        }
        return false;
    }

    @Override
    public String toString() {
        return "Componentes aceptados: " + Arrays.toString(coche)
                + "\nDatos Componentes: " + componentes
                + "\nPrecio: " + costeCoche()
                + "\nAfinidad: " + afinidadGlobalCoche();
    }

}
```

```java
public class EnfriamientoComponentes {

    public static ModeloCoche encontrarMejorMaterial(int n) throws
         CloneNotSupportedException {
        // nuestro tope de precio lo fijaremos en 80
        int maxPrecio = 80;
        ModeloCoche vecino = new ModeloCoche(n);
        ModeloCoche mejor = new ModeloCoche(n);
        double afinidad = 0, afinidadGlobal = 0;
        double tmp = 10000;
        while (tmp >= 10) {
            boolean encontrado = false;
            int i = 0;
            // buscaremos el primer vecino vlido
            // en caso de no existir limitamos a 100 intentos
            while (!encontrado && i < 100) {
                // la generacin de vecino la hace la propia clase vecino
                vecino.generarVecino();
                // si el vecino no cumple esta condicin no es un vecino vlido a considerar
                if (vecino.costeCoche() < maxPrecio) {
                    encontrado = true;
                } else {
                    i++;
                }
            }
            // sometemos el vecino a la aceptacin de probabilidad
            if (aceptarProbabilidad(vecino, mejor, maxPrecio, tmp) > Math.random()) {
                // en caso de pasarla, actualizamos la nueva afinidad del aspirante
                afinidad = vecino.afinidadGlobalCoche();
            }
            // si la nueva afinidad mejora la afinidad global
            if (afinidad > afinidadGlobal) {
                // actualizamos la nueva mejor solucin encontrada
                afinidadGlobal = afinidad;
                mejor = vecino;
            }
            // actualizamos la temperatura
            tmp -= tmp * 0.1;
        }
        return mejor;
    }

    private static double aceptarProbabilidad(ModeloCoche vecino, ModeloCoche mejor, int
         maxPrecio, double temp) {
        if (maxPrecio > vecino.costeCoche() && vecino.afinidadGlobalCoche() >
             mejor.afinidadGlobalCoche()) {
            return 1.0;
        }
        return Math.exp((mejor.afinidadGlobalCoche() - vecino.afinidadGlobalCoche()) /
             temp);
    }
}
```

Problema 3 Se dispone de un conjunto de n procesos y un ordenador con m procesadores (de características no necesariamente iguales). Se conoce el tiempo que requiere el procesador j-ésimo para realizar el proceso i-ésimo, t_{ij}. Se desea encontrar un reparto de procesos entre los m procesadores tal que el tiempo de finalización sea lo más corto posible. Tome tantas decisiones como estime convenientemente e intente comparar distintas soluciones con distintas configuraciones iniciales.

```
public static int[] encontrarReparto(int n, double[][] t) {
        int[] reparto = new int[n];
        for (int i = 0; i < n; i++) {
            reparto[i] = i;
        }
        int[] vecino = Arrays.copyOf(reparto, n);
        double mejorTiempo = Integer.MAX_VALUE;
        double candidato = Integer.MAX_VALUE;

        double tmp = 10000;
        // mientras no se enfrie
        while (tmp >= 10) {

            // buscaremos el primer vecino vlido
            int n1 = (int) (Math.random() * n);
            int n2 = (int) (Math.random() * n);
            int aux = vecino[n1];
            vecino[n1] = vecino[n2];
            vecino[n2] = aux;

            double tiempoVecino = Util.tiempo(vecino, t, n);
            // sometemos el vecino a la aceptacin de probabilidad
            if (aceptarProbabilidad(mejorTiempo, tiempoVecino, tmp) > Math.random()) {
                // en caso de pasarla, actualizamos la nueva afinidad del aspirante
                candidato = tiempoVecino;
            }
            // si la nueva afinidad mejora la afinidad global
            if (candidato < mejorTiempo) {
                // actualizamos la nueva mejor solucin encontrada
                mejorTiempo = candidato;
                reparto = Arrays.copyOf(vecino, n);

            }
            // actualizamos la temperatura
            tmp -= tmp * 0.1;

        }

        return reparto;
    }

    private static double aceptarProbabilidad(double mejorTiempo, double tiempoVecino,
         double tmp) {
        if (tiempoVecino < mejorTiempo) {
            return 1.0;
        }
        return Math.exp((tiempoVecino - mejorTiempo) / tmp);
    }

    public static double tiempo(int[] vecino, double[][] tiempos, int n) {
        double tiempo = 0;
        for (int i = 0; i < n; i++) {
            tiempo += tiempos[vecino[i]][i];
        }
```

```
        return tiempo;
    }
```

References

1. Material asignatura, *Metaheurísticas*, UGR. `http://sci2s.ugr.es/graduateCourses/Metaheuristicas`
2. E.-G. Talbi, Metaheuristics. From design to implementation, Wiley, 2009 Material.
3. C. Blum, A Roli, Metaheuristics in Combinatorial Optimization: overview and conceptual comparison. ACM Computing Surveys, 35(3), 2003, 268-308.