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:

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

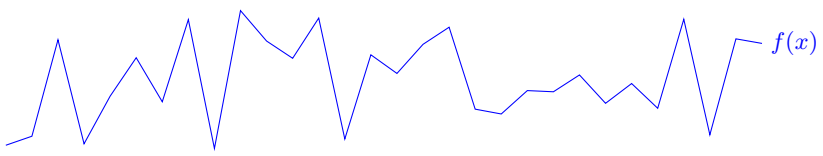

Fig. 1. Función donde podemos observar diferentes óptimos locales y uno global escondido entre ellos.

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:

```
private 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, nuevo;
        while (temp > 1) {
            nuevo = (int) (Math.random() * solutions.length);
            Util.permuta(solutions, nuevo);
            peso = elementos[nuevo] + pesoGlobal;
            for (int i = 0; i < n; i++) {
                Util.permuta(solutions, i);
                peso += elementos[i];
                if (aceptarProbabilidad(peso, pesoGlobal, capacidad, temp) >
                     Math.random()) {
                    pesoGlobal = peso;
                } else {
                    peso -= elementos[i];
                    Util.permuta(solutions, i);
                }
                if (pesoGlobal > calcularPeso(bestSolution, elementos) && pesoGlobal <
                     capacidad) {
                    bestSolution = Arrays.copyOf(solutions, solutions.length);
                }
            }
            temp -= 0.1 * temp;
        }

        return bestSolution;
    }

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

    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);
    }

    private 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;
    }
```

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.