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 resúmen para el temario de Algorítmica II para el Grado en Ingeniería Informática en Sistemas de Información.

Keywords: algoritmos, metaheurística, búsqueda, tabú

Búsqueda Tabú.

El algoritmo de búsqueda tabú fue propuesto por Glover en 1986. En 1990, este algoritmo llegó a ser muy popular solventando problemas de optimización de forma aproximada. Hoy en día es uno de los algoritmos metaheurísticos más extendidos.

Comparando con el enfriamiento simulado, podemos observar que esta técnica es menos aleatoria, TS acepta peores movimientos y modifica entornos. También utiliza la memoria adaptativa y estrategias especiales de resolución de problemas.

Principales ventajas de TS:

1. Restringe el entorno de búsqueda.
   - Evitamos recorridos Cíclicos.
   - Generamos vecinos modificados.
2. Introducimos mecanismos de reinicialización.
   - Intensificamos en zonas exploradas.
   - Diversificamos en zonas poco visitadas.

Se utilizarán dos tipos de memoria:

1. Memorias a corto plazo o lista tabú ← Decisión de vecinos. Guarda información que permite guiar la búsqueda de forma inmediata, desde el comienzo del procedimiento (entornos restringidos/tabú)
2. Memorias a largo plazo ← Búsqueda de nuevo máximo. Guarda información que permite guiar la búsqueda a *posteriori*, después de una primera etapa en la que se han realizado una o varias ejecuciones del algoritmo aplicando la memoria a corto plazo.

La lista será dinámica, es decir, habrá cambios.

El principio de TS podría resumirse como: "Es mejor una mala decisión basada en información que una buena decisión al azar, ya que, en un sistema que emplea memoria, una mala elección basada en una estraategia proporcionará claves útiles para continuar la búsqueda. Una buena elección fruto del azar no proporcionará ninguna información para posteriores acciones." (F.Glover)

Memoria + Aprendizaje = Búsqueda inteligente

TS explícitamente usa el historial de búsqueda. Aplica una búsqueda local y usa una memoria a corto plazo para escapar del mínimo local y evitar ciclos. La memoria a corto plazo implementa una lista tabú que almacenará un listado de vecinos visitados con soluciones y prohíbe utilizarlas

nuevamente.

En los siguientes puntos describiremos las memorias a corto y a largo plazo, pero a groso modo podemos decir que, por cada iteración en la lista tabú, a modo de cola de prioridades, un elemento se insertará, y el más antiguo se eliminará, similar a un sistema FIFO.

El algoritmo parará cuando una condición de parada es encontrada, por ejemplo, cuando el número de soluciones prohibidas en la lista tabú es completo.

Una lista tabú de tamaño pequeño hará que se concentre la búsqueda en pequeñas áreas del espacio de búsqueda. Por el contrario, una lista tabú de mayor tama˜no fuerza el proceso de búsquedas en regiones más extensas, debido a que ésto prohíbe revisitar un alto número de soluciones.

Memoria a Corto Plazo

- Modificación de las estructuras de entorno.
  - La TS restringe la búsqueda local sustituyendo E(S_{act}) por otro entorno E*(S_{act}). En cada iteración, se acepta siempre el mejor vecino de dicho entorno, tanto si es peor como si es mejor que S_{act}.
  - La memoria de corto placo (**Lista tabú**) permite a la TS determinar E*(S_{act}) y así organizar la manera en la que se explora el espacio.
- Las soluciones admitidas en E*(S_{act}) dependen d ela estructura de la lista tabú:
  - **Lista de soluciones tabú**. Soluciones ya visitadas que se marcan como tabú para no volver a ellas, eliminándolas del vecindario de S_{act}.
  - **Lista de movimientos tabú**. Se eliminan del entorno todos los vecinos resultantes de aplicar sobre S_{act} un movimiento realizado anteriormente.
  - **Lista de valores de atributos tabú**. Se eliminan del entorno todos aquellos vecinos con un par (atributo, valor) determinado que ya presentara alguna solución explorada anteriormente (NOTA: en TSP no nos vendría bien, mejor dependencias como día, clima, tráfico...).
- Tendencia tabú. Tiempo que una solución permanece en la lista tabú.
- Niveles de aspirración: Introduce flexibilidad.
  - Se genera una solución mejor que cualquiera anterior.
  - Se alcanza valor aceptable.
  - Se alcanza el mejor valor conocido.
- Estrategias para listas de candidatos.
  - Restringen el número de vecinos examinados en una iteración dada, para los casos en los que E*(S_{act}) es grande o la evaluación costosa.
  - Se busca el mejor movimiento disponible que pueda ser determinado con esfuerzo equilibrado.
  - Se genera una parte del espacio y se toma el mejor vecino.

Una selección adecuada puede reducir el Tiempo Computacional.

Memoria a Largo Plazo

En algunas aplicaciones, las componentes de la memoria TS de corto plazo son suficientes para producir soluciones de muy alta calidad. Pero TS se vuelve mucho más potente incluyendo memoria de largo plazo. La memoria de largo plazo se puede usar de dos modos:

- Intensificación y diversificación. Memoria de frecuencias y visitas a una solución.
- Reinicializar la búsqueda, mantener registro de mejores soluciones.. Se pueden usar para diversificar.
- Flowchart → Serie de iteraciones a corto plazo, donde extraemos la mejor de ellas.
- Nueva solución a partir de lo almacenado, evitaremos repetir.

Una estructura muy empleada es la memoria de frecuencias, que registra el número de veces que cada valor de un atributo ha pertenecido a soluciones visitadas en la búsqueda.

Estrategias de Intensificación

- Se basan en una reinicialización de la búsqueda que efectúa un regreso a regiones atractivas del espacio para buscar en ellas más extensamente.
- Se mantiene un registro de las mejores soluciones visitadas, insertando una nueva solución cada vez que se convierte en la mejor.
- Se puede introducir una medida de diversificación para asegurar que las soluciones registradas difieran una de otra en un grado deseado.

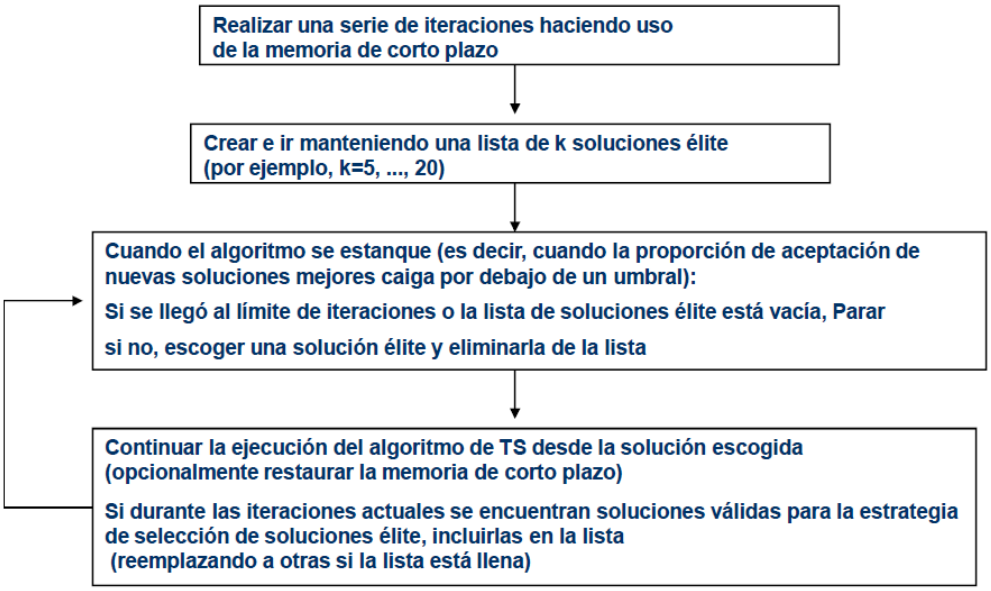


Fig. 1. Flowchart Intensificación

Estrategias de Diversificación

- Conducen la búsqueda hacia nuevas regiones del espacio de búsqueda no exploradas aún.
- La búsqueda se reinicializa cuando se estanca, partiendo de una solución no visitada.
- Esta solución se genera a partir de la memoria de frecuencias, dando mayor probabilidad de aparición a los valores menor habituales.

Estructuras de memoria largo plazo

- Si las variables son binarias, se puede usar un vector de dimensión n, para almacenar el número de veces que cada variable tomó el valor 0 (ó 1).
- Si son enteras, se utiliza una matriz bidimensional, como contador de las veces que la variable i toma el valor k: m[i,k].
- Si son permutaciones de orden, se puede utilizar una matriz bidimensional, como contador de las veces que el valor i ha ido seguido de j: M[i,j]

Uso de las memorias de frecuencias

1. Generar directamente la nueva solución inicial a partir de la información almacenada en la memoria de frecuencias M, dando mayor probabilidad de aparición a los valores menos habituales.

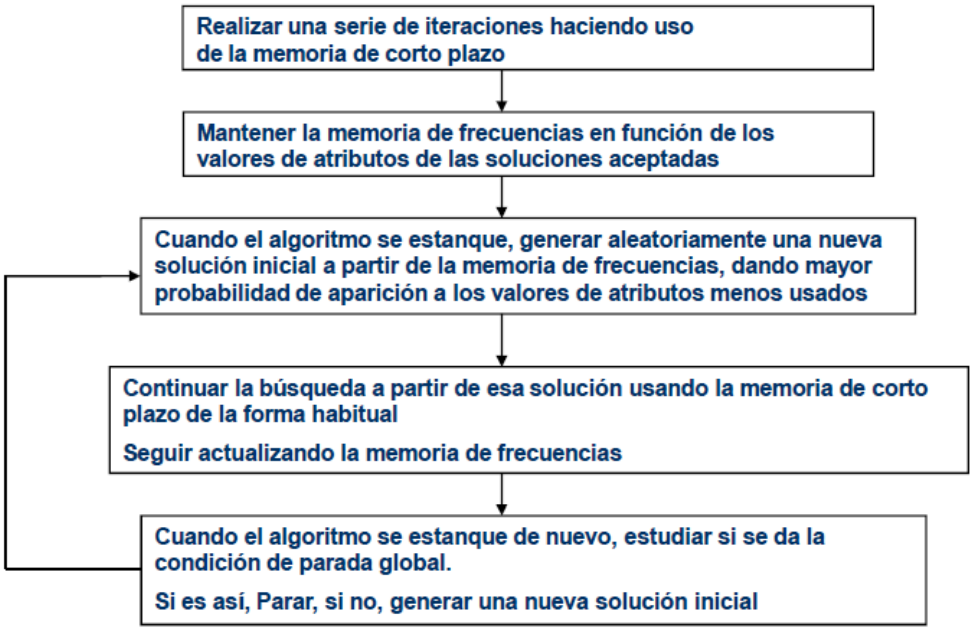


Fig. 2. Flowchart Diversificación

2. Usar la información almacenada en M para modificar temporalmente el caso del problema, potenciando los valores heurísticos de los atributos menos usados en la búsqueda. Aplicar un algoritmo greedy sobre ese caso modificado para generar la solución inicial. Restaurar el caso original del problema antes de continuar con la búsqueda.

$sBest \leftarrow s_0$
$tabuList \leftarrow []$
while $notstoppingCondition()$ **do**
  $canidateList \leftarrow []$
  $bestCandidate \leftarrow null$
  for $sCandidateinsNeightborhood$ **do**
    if $(nottabuList.contains(sCandidate))and(fitness(sCandidate) > fitness(bestCandidate))$ **then**
      $bestCandidate \leftarrow sCandidate$
    end
  end
  if $fitness(bestCandidate) > fitness(sBest)$ **then**
    $sBest \leftarrow bestCandidate$
  end
  $tabuList.push(bestCandidate)$
  if $tabuList.size > maxTabuSize$ **then**
    $tabuList.removeFist()$
  end
end
output: Best Solution found.

Algorithm 1: Algoritmo de Búsqueda Tabú con memoria a corto plazo.

Si quisiéramos añadir una condición de aspiración, en la condición perteneciente a la iteración de los vecinos deberíamos añadir "or sCandidate.isAspiration()".

Ampliación de temario: `http://leeds-faculty.colorado.edu/glover/fred%20pubs/329%20-%20Introduccion%20a%20la%20Busqueda%20Tabu%20TS_Spanish%20w%20Belen(11-9-06).pdf`

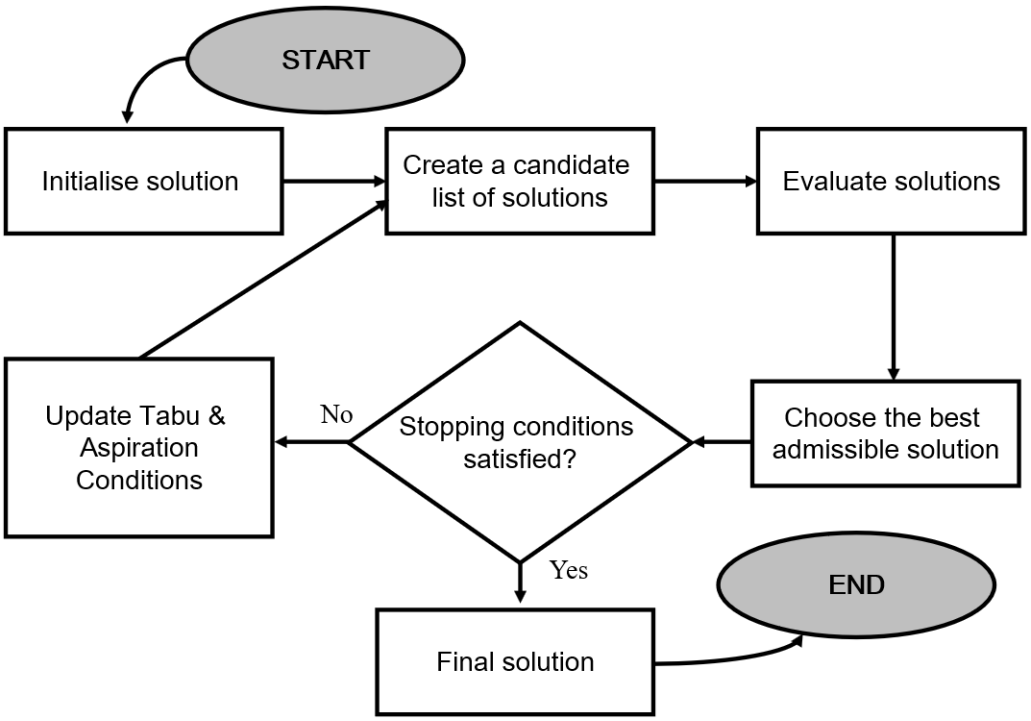


Fig. 3. Diagrama de flujo Búsqueda Tabú

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:

```
public class Funcion {

    public static int[] busquedaTabu() {
        Queue<int[]> tabu = new LinkedList<>();
        int[] bestSolution = {0, 0, 0, 0, 0};
```

```
        int[] actualSolution = {0, 0, 0, 0, 0};

        int maxTabu = 3;
        double maxTotal = 0;
        double maxVecinos = 0;

        // condicin de parada
        for (int it = 0; it < 100; it++) {
            // salto del mnimo local
            int posNueva = (int) (Math.random() * actualSolution.length);
            Util.permuta(actualSolution, posNueva);

            for (int i = 0; i < bestSolution.length; i++) {

                // genero vecino
                int[] vecino = Arrays.copyOf(actualSolution, actualSolution.length);
                Util.permuta(vecino, i);

                // si en lista tab ni la probamos y vamos al siguiente vecino
                if (!tabu.contains(vecino)) {
                    double maxActual = funcionCalc(vecino);
                    // fitness
                    if (maxActual > maxVecinos) {
                        maxVecinos = maxActual;
                        actualSolution = Arrays.copyOf(vecino, vecino.length);
                    }
                }
            }
            if (maxVecinos > maxTotal) {
                maxTotal = maxVecinos;
                bestSolution = Arrays.copyOf(actualSolution, actualSolution.length);
            }
            tabu.add(actualSolution);
            if (tabu.size() > maxTabu) {
                tabu.remove();
            }
        }

        return bestSolution;
    }

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

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.

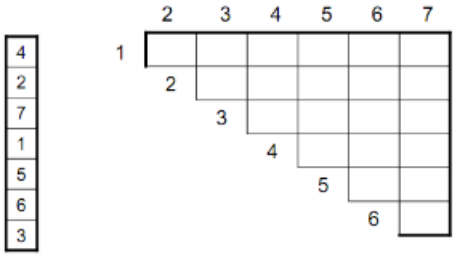
Implementación propuesta:

```
public class Mochila {

    public static int[] busquedaTabu(int[] elementos, int n, int capacidad) {
        Queue<int[]> tabu = new LinkedList<>();
        for (int i = 0; i < n; i++) {
            int r;
            do {
                r = (int) (Math.random() * 100);
            } while (r < 1);
            elementos[i] = r;
        }
        int[] bestSolution = new int[n];
        int[] solutions = new int[n];
        int maxTabu = 3;
        int pesoGlobal = 0;
        int peso, nuevo, pesoCandidato;
        // condicin de parada
        for (int it = 0; it < 100; it++) {
            // salto del mnimo local
            nuevo = (int) (Math.random() * solutions.length);
            Util.permuta(solutions, nuevo);
            pesoCandidato = 0;
            peso = elementos[nuevo] + pesoGlobal;
            for (int i = 0; i < n && peso < capacidad; i++) {
                // genero vecino
                int[] vecino = Arrays.copyOf(solutions, solutions.length);
                Util.permuta(vecino, i);

                // si en lista tab ni la probamos y vamos al siguiente vecino
                if (!tabu.contains(vecino)) {
                    peso = calcularPeso(vecino, elementos);
                    tabu.add(vecino);
                    // fitness
                    if (peso > pesoCandidato && peso < capacidad) {
                        pesoCandidato = peso;
                        solutions = Arrays.copyOf(vecino, vecino.length);
                    }
                }
                if (tabu.size() > maxTabu) {
                    tabu.remove();
                }
            }
            if (pesoCandidato > pesoGlobal && pesoCandidato <= capacidad) {
```

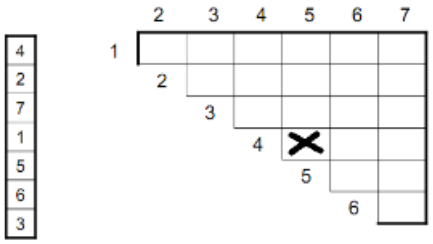
```
                pesoGlobal = pesoCandidato;
                bestSolution = Arrays.copyOf(solutions, solutions.length);
            }
        }

        return bestSolution;
    }

    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 posicióm (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:

```java
public class Material implements Cloneable {

    private final 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(int n1, int n2) {
        int aux = materiales.get(n1);
        materiales.set(n1, materiales.get(n2));
        materiales.set(n2, aux);
    }

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

    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;
    }

    @Override
    public boolean equals(Object obj) {
        Material aux = (Material) obj;
        return materiales.equals(aux.getMateriales());
    }

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

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

}
```

```
public class TabuMateriales {

    public static Material encontrarMejorMaterial() throws CloneNotSupportedException {
        Material vecino = new Material();
        Material mejor = new Material();
        Material aspirante;
        Queue<Material> listaTabu = new LinkedList<>();
        int maxIt = 1000, maxTabu = 3;

        for (int i = 0; i < maxIt; i++) {
            vecino.generarVecino();
            aspirante = (Material) vecino.clone();
            vecino.generarVecino(j, j + 1);
            if (!listaTabu.contains(vecino)) {
                if (vecino.cantidadAislante() > aspirante.cantidadAislante()) {
                    aspirante = vecino;
                }
                Material aux = (Material) vecino.clone();
                listaTabu.add(aux);
            }
            if (listaTabu.size() >= maxTabu) {
                listaTabu.remove();
           }
            if (aspirante.cantidadAislante() > mejor.cantidadAislante()) {
                mejor = aspirante;
            }
        }
        return mejor;

    }
}
```

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;
    }
}

public class ModeloCoche {

    private List<Componente> componentes;

    public ModeloCoche(int n) {
        componentes = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            componentes.add(new Componente());
        }
    }

    public void generarVecino(int i, int[] coche) {
        permuta(coche, i);
    }

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

public static int[] encontrarMejorMaterial(int n, ModeloCoche modelo) throws
    CloneNotSupportedException {
        // nuestro tope de precio lo fijaremos en 80
        int maxPrecio = 80;
        // la lista tab  guardar  hasta 3 elementos
        int maxTabu = 3;
        int nuevo;
        Queue<int[]> listaTabu = new LinkedList<>();
        int[] vecino = new int[n];
        int[] candidato, original;
        int[] mejor = new int[n];
        for (int j = 0; j < 10000; j++) {
            // buscaremos el mejor vecino
            candidato = Arrays.copyOf(vecino, n);
            original = Arrays.copyOf(candidato, n);
            nuevo = (int) (Math.random() * n);
            modelo.generarVecino(nuevo, vecino);
            // la generacion de vecino la hace la propia clase ModeloCoche
            vecino = Arrays.copyOf(original, n);
            modelo.generarVecino(i, vecino);
            // si el vecino no est en la lista no es un vecino valido a considerar
            if (modelo.costeCoche(vecino) < maxPrecio && !listaTabu.contains(vecino)) {
                // si el coste est dentro del mximo y la afinidad supera el resto
                if (modelo.afinidadGlobalCoche(vecino) >
                    modelo.afinidadGlobalCoche(candidato)) {
                    candidato = Arrays.copyOf(vecino, n);
                }
                // el vecino se adhiere a la lista tabu
                listaTabu.add(vecino);
                // si hemos metido mas de la cuenta eliminamos el ultimo
                if (listaTabu.size() >= maxTabu) {
                  listaTabu.remove();
                }
            }
        // si el candidato es el mejor, actualizamos
        if (modelo.costeCoche(candidato) < maxPrecio
                    && modelo.afinidadGlobalCoche(candidato) >
                        modelo.afinidadGlobalCoche(mejor)) {
```

```
            mejor = Arrays.copyOf(candidato, n);
        }
       }
       return mejor;
   }
```

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.
4. B. Melián, F. Glover. Introducción a la búqueda tabú, 2006. `http://leeds-faculty.colorado.edu/glover/fred%20pubs/329%20-%20Introduccion%20a%20la%20Busqueda%20Tabu%20TS_Spanish%20w%20Belen(11-9-06).pdf`