Universidad Politécnica
de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

Máster Universitario en Ingeniería Informática

Trabajo Fin de Máster

Explicación en Bases de Conocimiento de Sistemas de Ayuda a la Decisión

Autor: Rafael Timermans Díez

Tutor: Juan Antonio Fernández del Pozo de Salamanca

Madrid, junio de 2021

Este Trabajo Fin de Máster se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Máster

Máster Universitario en Ingeniería Informática

Título: Explicación en Bases de Conocimiento de Sistemas de Ayuda a la Decisión

Junio de 2021

Autor(a): Rafael Timermans Díez

Tutor(a):

Juan Antonio Fernández del Pozo de Salamanca

Departamento de Inteligencia Artificial

ETSI Informáticos

Universidad Politécnica de Madrid

Resumen

La toma de decisiones es un elemento común en todos los ámbitos de la vida, las cuales cobran mucha importancia en el ámbito de las organizaciones y de las empresas. Por ello, surgen los sistemas de apoyo a la toma de decisiones como apoyo a los profesionales para considerar todas las posibilidades y tomar buenas decisiones de manera informada.

Aunque estas herramientas son muy útiles y obtienen muy buenos resultados, en problemas donde existen una gran cantidad de variables que influyen en la decisión, surgen algunos problemas como su almacenamiento en memoria o la interpretación y extracción de conocimiento del resultado por parte del usuario.

En el caso concreto de las tablas de decisión, surge una propuesta para solucionar estas limitaciones, y son las llamadas KBM2L. Estas KBM2L una representación de las tablas de decisión en la cual se comprime toda la información en una lista, mediante la agrupación de los casos en base a la decisión de dichas entradas. Para ello, será necesaria una reorganización en la tabla mediante una serie de permutaciones en las filas y las columnas, de tal forma que conservemos toda la información.

Debido a esto, el objetivo de este trabajo consiste en la implementación de metaheurísticas de optimización combinatoria para poder realizar estas permutaciones, y desarrollar un software capaz de optimizar la creación de las KBM2L. Así mismo, se realizarán una serie de pruebas donde medirá el rendimiento y viabilidad de este software.

Abstract

Decision-making is a key factor throughout life, which is very important in the field of organizations and companies. For this reason, decision support systems have emerged as a support tool for professionals to consider all the possibilities and make good, informed decisions.

Although these tools are very useful and obtain very good results, some problems arise in cases where there are a lot of variables that influence these decisions, like computer resources to use that huge information, or that it is very difficult for the user to interpret and to obtain knowledge of that problem.

In the specific case of decision tables, in order to solve these limitations, the so called KBM2L are created. A KBM2L is a list that contains all the information of the decision table in a reduced way, by grouping all the cases based on the decision of those entries. To do that, we need to reorder the table through a series of permutation in the rows and columns, in such way that we still retain all the information.

Hence, the aim of the project is to implement metaheuristics of combinatorial optimization to be able to do said reorganization, and to develop a software capable of optimizing the creation of KBM2Ls. Likewise, we will perform some test to measure the performance of this software.

Tabla de contenidos

Tabla de ilustraciones

Índice de tablas

1 Introducción

Durante los últimos años, ha habido un auge en los sistemas de ayuda a la toma de decisiones. Estos sistemas analizan problemas del mundo real en los que influyen las decisiones humanas además de factores externos los cuales pueden producir distintas situaciones, y al solucionarlos mediante distintos métodos, crean un plan de acción a seguir para obtener los mejores resultados.

Normalmente estos sistemas no tratan de sustituir a las personas que toman estas decisiones, si no que como su nombre indica, son sistemas de apoyo para que los expertos del área del problema puedan tomar mejores decisiones, viendo las salidas de estos sistemas.

Una vez se ha resuelto el problema, estos sistemas suelen utilizar algunas herramientas visuales [1] para poder mostrar esta solución a los usuarios, de tal forma que puedan interpretarlos, ya que el objetivo de estos sistemas no es indicar únicamente el mejor plan de acción, sino mostrar los distintos caminos posibles con las consecuencias asociadas. Estas consecuencias suelen representarse mediante una escala numérica (utilidad), que muestra el valor ganado final, con lo que se puede establecer una preferencia en las decisiones.

De esta forma, estos sistemas no solo buscan hallar el mejor camino a seguir, sino que el usuario pueda obtener conocimiento, al tener los resultados previstos de cada acción, pudiendo obtener patrones de comportamiento.

Algunas de las herramientas de la representación del problema y para su evaluación optima son los árboles de decisiones, las tablas de decisiones y los diagramas de influencia [2]. Estas herramientas son muy útiles, pero cuando el problema se vuelve demasiado complejo con gran cantidad de variables a analizar, la representación de las soluciones crece de forma exponencial.

En concreto, las tablas de decisiones tienen que representar todas las posibles combinaciones de las variables independientes, dado que a cada combinación se le asociará una decisión distinta. En problemas del mundo real, es común tener al menos 20 variables, lo cual puede producir en la tabla millones de filas. Esto supone varios problemas:

- La tabla ocupa una gran cantidad de memoria en el ordenador. Esto produce que incluso simples consultas a estas tablas sean costosas, ralentizando los procesos.
- Es imposible que la utilice un ser humano debido a su enorme tamaño. Con tablas tan grandes, la única alternativa real es realizar una consulta para encontrar la alternativa de la que se desea saber la solución correcta.
- No se aprecia la estructura del contenido de la tabla.
- No es posible razonar sobre las decisiones. Al dar una decisión por cada combinación, estas no se encuentran agrupadas, mientras que en

muchos casos podrían obtenerse patrones de decisión, ya que en cada decisión influirán más unos atributos que otros. Por tanto, no es posible obtener conocimiento de estas tablas como en otros casos.

Problemas parecidos surgen con el resto de las herramientas. Para solucionar estos problemas surgen distintas alternativas que tratan el tema desde distintos enfoques, siendo uno de ellos las Knowledge Based Matrix To List [3] (KBM2L), las cuales son extensiones de las tablas de decisiones.

La idea de las KBM2L es conservar toda la información de la tabla original sin que sea necesario almacenar todos los registros, reduciendo así el tamaño. Para ello, la propuesta que surge es agrupar las entradas de la tabla por la decisión tomada, y guardar solo el último elemento de cada bloque, siendo el representante de su clase, y sabiendo que todas las entradas anteriores a este representante tienen la misma decisión que este.

Para reordenar la tabla para hacer posible esta agrupación, se propone alterar el orden de los atributos (columnas de la tabla), obligando a esta a reordenarse para mantener un sentido ascendente respecto a los valores en cada atributo de cada fila. Que se mantenga este orden es importante, porque si no, con solo el representante de la clase no podríamos saber cuáles son las entradas que van antes que él.

Con esta lista reducida no solo estamos haciendo que se ocupe menos memoria, sino que al estar agrupada por decisiones permite al usuario extraer conocimiento de forma mucho más rápida y eficiente, pudiendo identificar los patrones de las decisiones comunes.

El objetivo es desarrollar e implementar un algoritmo que sea capaz de ordenar estas KBM2L para dar lugar a la combinación optima, que es aquella que tiene el menor número de bloques. Esto no supone una tarea sencilla, sino que se convierte en un problema de optimización combinatoria, ya que cuantos más atributos haya en la tabla más combinaciones habrá, llegando a más de billones de combinaciones.

Debido a esto, habrá que utilizar metaheurísticas de optimización combinatoria y aplicarlas a este problema en concreto. Estas metaheurísticas son algoritmos que se utilizan en problemas con una gran cantidad de posibilidades (como este caso), y donde no es tan importante hallar la mejor solución, sino soluciones aproximadas de un valor razonable que se puedan encontrar en un límite de tiempo acotado, ya que el explorar todas las soluciones es totalmente inviable.

De esta forma, el objetivo último de este trabajo consiste en implementar algoritmos de metaheurísticas de optimización combinatoria y aplicarlos al problema de las KBM2L, para obtener tablas de decisiones de un tamaño reducido y agrupadas por decisiones, lo cual permitiría al usuario poder interpretar y obtener conocimiento de la tabla y de su estructura de una forma sencilla.

2 Trabajos previos

Debido a la naturaleza del trabajo, será necesario estudiar tres ámbitos distintos para la comprensión del mismo, y son: las KBM2L, ya que es la base fundamental del proyecto; el funcionamiento de las herramientas de apoyo para los sistemas para la ayuda a la toma de decisiones, ya que las KBM2L se basan en estas herramientas, en concreto las tablas de decisiones; y las metaheurísticas de optimización combinatoria para poder explorar todo el espacio de búsqueda.

Como la creación y optimización de las listas se basa en los sistemas de decisiones y en los métodos de optimización combinatoria, explicaremos las listas de conocimiento al final de este capítulo.

2.1 Sistemas de ayuda a las decisiones

Un sistema de ayuda a las decisiones [2], en inglés *decisión support making (DSS)*, es un sistema que se utiliza para tomar una decisión en un problema, el cual no suele dar una respuesta de la decisión que se debe tomar, si no que proporciona una ayuda a un experto encargado de tomar la decisión.

Este apoyo se suele proporcionar mediante el cálculo de un valor (o utilidad) a cada una de las posibles decisiones, siendo la decisión de mayor utilidad la mejor según el sistema.

Aunque teóricamente un sistema de ayuda a las decisiones es cualquier herramienta que ayude a tomar la mejor decisión en un problema, debido a las características de estos se suele utilizar este método solo para aplicaciones software informáticas, ya que estas son capaces de analizar los datos de forma eficiente.

Debido a que la definición es muy general, a lo largo de la historia han surgido múltiples definiciones distintas de lo que se considera un DSS, y aunque no hay ninguna mundialmente aceptada, sí que es de consenso general que es “algo” que ayuda a tomar decisiones.

Como podemos suponer, esta definición tan general comprende un área enorme de herramientas y aplicaciones. En esta memoria no nos vamos a centrar tanto en estos sistemas, si no que vamos a explicar algunas de las herramientas utilizadas por estos sistemas de apoyo, que se utilizan para la representación del conocimiento proporcionada por estos sistemas.

2.1.1 Árboles de decisiones

Los árboles de decisiones son una herramienta para representar un modelo de decisiones con forma de árbol. Como el resto de DSS, esta herramienta ayuda a tomar una decisión en base a los parámetros, por lo que, a diferencia de los árboles de clasificación, no hacen una clasificación del problema, sino que ofrecen información sobre las distintas decisiones.

Esta herramienta utiliza una forma de árbol, usada en más problemas, que consiste en un grafo unidireccional acíclico. En estos grafos hay dos elementos distintos:

- Aristas: Las aristas son los caminos para ir de un nodo a otro. Aunque en un grafo en general puede haber tantas aristas como se quiera, en un árbol solo puede haber un camino directo de un vértice a otro. De cada nodo, menos del nodo final, surgen varias aristas, y cada ramificación indica una posible acción o resultado.
- Nodos: Los nodos representan los individuos de un problema, representan las clases. Así, estos pueden ser ciudades, decisiones, objetos, etc. En el caso de los árboles de decisiones, estos vértices (o nodos) son las decisiones o bifurcaciones que pueden darse en un problema. Hay tres tipos distintos de nodos:
    - De decisión. Indica una decisión que tiene que tomar el usuario, de la cual saldrán distintas ramificaciones para cada uno de los resultados dependiendo de la acción realizada.
    - De probabilidad. Parecidos a los anteriores, estos nodos se ramifican en varios resultados, pero estos no dependen de las acciones que se tomen, sino de un acontecimiento incierto al que hay asociado una probabilidad para cada una de las posibles resoluciones.
    - Final. Nodos donde se indica la utilidad de la decisión final.

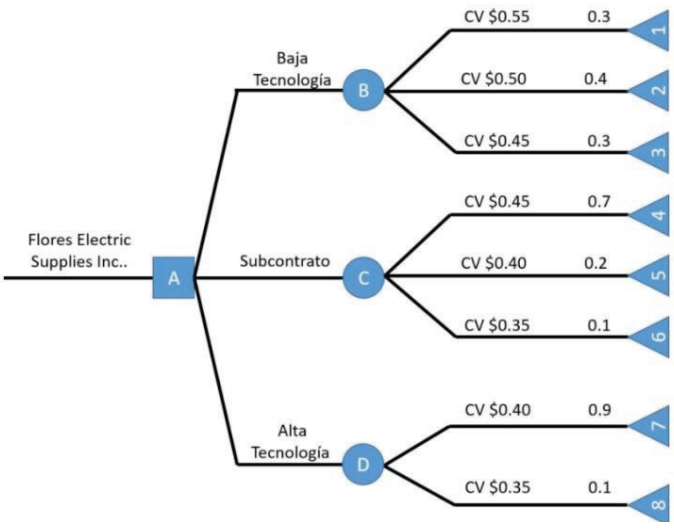

Ilustración 1 Ejemplo árbol de decisiones

En el ejemplo podemos ver todos los elementos. En este caso, se trata de una proveedora de electricidad que está pensando en montar una fábrica, para lo cual su única decisión (representada en el nodo A), es elegir el tipo de tecnología que piensan usar en la planta (nodos B, C y D): baja tecnología, alta tecnología, o subcontratar la generación de la energía.

Según la decisión que tomen, para generar cada caballo de vapor (CV, siendo esta una medida de energía alternativa a los watios), tendrá un coste distinto. Sin embargo, este coste no depende solo de la decisión de la empresa, sino que vemos que los nodos se ramifican.

De esta forma, si hemos elegido utilizar baja tecnología (nodo B), hay una probabilidad del 30% de que cada CV cueste 0.55$, 40% de que cueste 0.50$ y otro 30% de que cueste 0.45$, siendo estos los nodos finales.

2.1.2 Tablas de decisiones

Son una representación visual de la solución de un algoritmo de toma de decisiones, en la cual se indica la mejor acción posible ante unas determinadas condiciones. En estas tablas, se puede expresar el mismo conocimiento que en los árboles de decisiones, es solo otra representación visual.

Podemos ver un ejemplo de tabla de decisiones en la siguiente imagen [4]:

Play golf dataset

Independent variables				Dep. var
OUTLOOK	TEMPERATURE	HUMIDITY	WINDY	PLAY
sunny	85	85	FALSE	Don't Play
sunny	80	90	TRUE	Don't Play
overcast	83	78	FALSE	Play
rain	70	96	FALSE	Play
rain	68	80	FALSE	Play
rain	65	70	TRUE	Don't Play
overcast	64	65	TRUE	Play
sunny	72	95	FALSE	Don't Play
sunny	69	70	FALSE	Play
rain	75	80	FALSE	Play
sunny	75	70	TRUE	Play
overcast	72	90	TRUE	Play
overcast	81	75	FALSE	Play
rain	71	80	TRUE	Don't Play

Ilustración 2 Tabla de decisiones

En esta taba, se intenta responder a la pregunta de si jugar o no al golf dadas ciertas condiciones meteorológicas. Así, cada fila en una tabla será una entrada distinta, y cada columna serán las variables o atributos de cada entrada. En este caso las variables son el pronóstico (soleado, lluvioso, etc), la temperatura,

la humedad y si va a hacer viento o no. También vemos que la respuesta a estas variables puede ser de cualquier tipo: puede ser binaria, categórica, continua, discreta…

Por último, la variable dependiente es la respuesta al problema para dicha entrada, es decir, dadas las condiciones representadas en esa fila, cuál es la mejor opción, la respuesta con más utilidad. En este caso se intenta responder a si jugar al golf, por lo que la respuesta es binaria, pero de igual forma que con el resto de las variables, el dominio de la respuesta podría ser mayor. Además, aunque en este caso no es así, también podría haber varias variables dependientes distintas. Así, podría haber otra columna de respuesta que indicase el tipo de palos que se van a llevar al partido de golf, ya que estos podrían depender también de la meteorología.

Estas tablas son sencillas de interpretar ya que permiten elegir la mejor respuesta dadas las condiciones actuales.

El problema de estas tablas es que cuando el problema es grande, estas crecen en tamaño de forma exponencial, ya que tienen que contener una fila por cada combinación de atributos, por lo que, si hay gran cantidad de atributos o estos tienen un dominio de respuesta elevado, el número de combinaciones se vuelve enorme.

En problemas del mundo real, estas tablas acostumbran a tener más de 20 atributos, lo que conlleva tablas de millones de filas. Esto supone distintos problemas:

- Interpretación de la tabla: Es imposible para una persona sacar conclusiones o explicaciones de los resultados de una tabla tan grande, por lo que la única utilidad de la tabla se convierte en dar la mejor solución dado un caso, pero esta no puede ser explicada ni interpretada.
- Tamaño en memoria: Estas tablas de millones de filas ocupan su espacio en la memoria del ordenador, y no es despreciable. Adicionalmente a esto, como la tabla es muy grande, cualquier operación sobre ella tendrá un coste en el tiempo elevado.

2.1.3 Diagramas de influencia

Los diagramas de influencia [5] forma gráfica de modelar un sistema, el cual lo presentaremos en modo de diagrama. Esta técnica nos permite analizar un sistema de causa-efecto, lo que nos facilita la identificación de la estructura del sistema, los diferentes elementos que influyen y las diferentes relaciones entre estos. A diferencia de los árboles de decisiones, estos se enfocan en los objetivos, las entradas, y las decisiones fundamentales.

Para ello suelen utilizarse herramientas software [6] que permitan incorporar la incertidumbre al proceso, de modo que se quede en segundo plano permitiéndonos diseñar un modelo de decisiones compleja.

De forma similar a los árboles de decisión, este modelo se compone de nodos y relaciones que los conectan. Los nodos pueden ser de tres formas, de decisión, de probabilidad y de azar (como en los árboles). Los nodos de valor representan el nodo final que personifican el objetivo a maximizar simbolizado normalmente mediante un rombo.

Las relaciones pueden ser de dos tipos también, o arcos condicionales o informativos. Los condicionales son aquellos que están dirigidos hacia nodos de azar, y representan dependencia probabilística si van a un nodo de azar, y dependencia funcional si va a un nodo de valor. Los arcos informativos son los que llegan los nodos de decisión, y a diferencia de los arcos anteriores, estos sí que implican cronología y causalidad.

Podemos ver un ejemplo en la siguiente imagen:

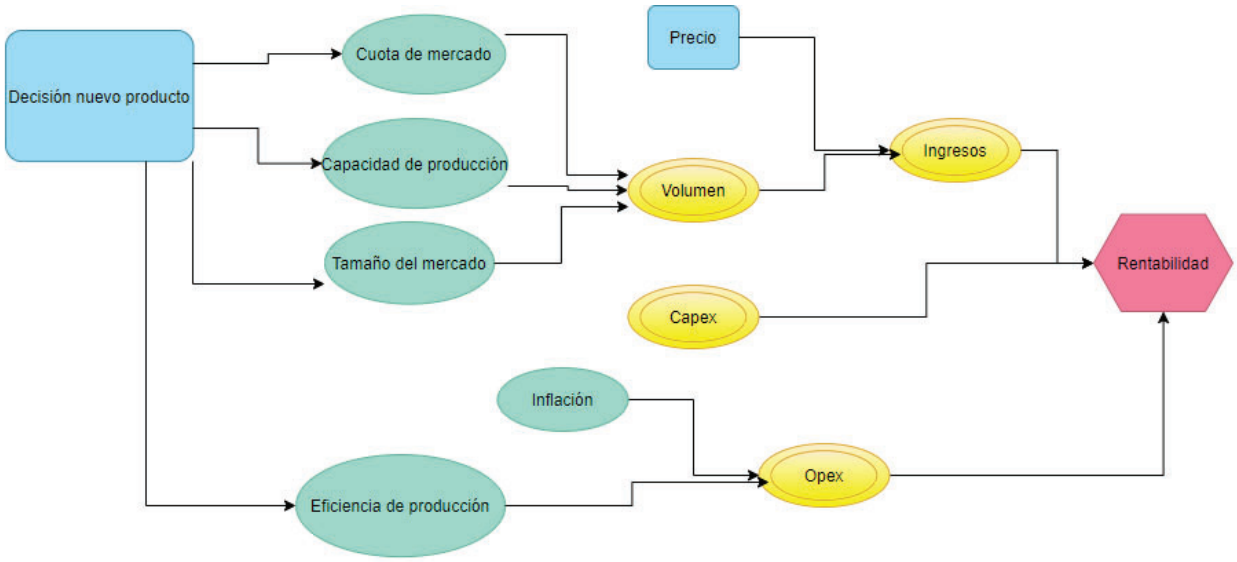

Ilustración 3 Diagrama de influencia

Vemos que hay dos nodos de decisión, que son la decisión de qué producto nuevo sacar al mercado, y el precio de venta. El nodo final es la rentabilidad que sacaremos del producto. También se aprecia los distintos colores que se usan. De esta forma los nodos amarillos representan a los efectos y causas que surgen de los nodos anteriores del sistema. Así, el nodo ingresos, será el resultado del precio que hayamos elegido y del volumen de ventas (el cual a su vez es la consecuencia de los nodos anteriores). También podemos apreciar nodos de incertidumbre, como es el caso de la inflación.

Ventajas:

- Permite incluir más factores que el árbol de decisión.
- La representación visual permite comprender el problema de forma más sencilla.

Desventajas

- Aunque permiten más variables que el árbol de decisiones, si hay demasiadas variables es recomendable utilizar otros modelos más complejos.
- Resulta complicado ver los posibles resultados asociados con una decisión o camino concreto debido a toda la incertidumbre.

2.2 Metaheurísticas de optimización combinatoria

La optimización combinatoria [7] es el nombre general para el proceso de buscar una solución dentro de una gran cantidad de posibles soluciones, aunque estas deben de ser finitas.

De esta forma, un problema de este tipo está definido por [8]:

- Un conjunto de variables: X = {x1,x2,x3…xn}
- El dominio de dichas variables: D1,D2,D3…Dn
- Restricciones de las variables
- Una función a optimizar f(x)

Una solución sería aquella que para todas las variables tenga un valor dentro del dominio de cada una además de que cumpla las restricciones posibles. Según esta definición existen una gran cantidad de soluciones posibles (todas las posibles combinaciones), y a este conjunto de soluciones las llamaremos el espacio de búsqueda. Sin embargo, para solucionar el problema de optimización combinatoria no basta con coger cualquiera, sino que es necesario encontrar aquella que optimice la función dada. En algunos casos esto significará maximizar dicha función y en otros minimizarlo.

Escrito de forma más técnica, solucionar el problema consiste en, dado el espacio de búsqueda S, encontrar una solución $s' \in S$, tal que $\forall s \in S$, $f(s') \leq f(s)$. De esta forma, s' es una solución global del problema.

Existen muchos ejemplos famosos de este tipo de problemas como el problema del cartero chino, el problema de la mochila, etc. En el caso del problema de la mochila, hay una serie de objetos y cada uno tiene asociado un valor y un peso. Así mismo, existe un peso límite que podemos meter en la mochila (la restricción), por lo que la suma del peso de los objetos que llevamos tiene que ser siempre menor que el máximo permitido. El problema consiste en seleccionar los objetos que nos aporten el mayor valor posible.

En este caso, el conjunto de variables son los objetos, y el dominio de cada una es solo 0 o 1: 0 en el caso de que no cojamos el objeto, y 1 en el caso contrario. La función será muy sencilla también: sea vi el valor que proporciona el objeto xi, $f(x)= \Sigma xn*vn$.

En el caso del problema del cartero chino, existe un cartero que tiene que pasar por todas las ciudades de una zona de la forma más rápida posible, por lo que el problema es elegir el orden correcto para visitarlas.

Aunque estos dos ejemplos puedan parecer sencillos, existe un gran abanico de posibilidades. Supongamos que el cartero tiene que recorrer 20 ciudades. Como se trata de ordenar estas ciudades (elegir el orden en el que las va a visitar), la cantidad total de combinaciones posibles es de 20!, que equivale a 2.4 cuatrillones de posibilidades, es decir $2,43x10^{18}$. Esto supone un espacio de búsqueda enorme, el cual es imposible recorrerlo entero para hallar la mejor solución.

Es en este punto donde surgen las metaheurísticas de optimización combinatoria. Estas heurísticas, son técnicas o algoritmos que proponen una forma de recorrer el espacio de búsqueda de forma parcial, intentando encontrar la mejor solución posible. Como no se pueden recorrer la totalidad de las soluciones en un espacio acotado de tiempo, las soluciones proporcionadas no tienen por qué ser la solución óptima, si no que se busca obtener soluciones aceptables aproximadas en un tiempo razonable.

Hay varias formas de clasificar este tipo de heurísticas, aunque la que usaremos en esta memoria serán las heurísticas de trayectoria frente a las de población. Así, en las heurísticas de trayectoria se recorre un conjunto de soluciones en busca de una mejor solución, la cual vamos sustituyendo, mientras que en las heurísticas de población, se parte de un conjunto inicial de soluciones el cual va evolucionando, como es el caso de los algoritmos genéticos.

2.2.1 Metaheurísticas de trayectoria

Estas metaheurísticas se llaman así porque realizan un proceso en el espacio de búsqueda mediante una trayectoria en la que se van obteniendo mejores soluciones. De esta forma, se empieza desde un estado inicial (una solución inicial), que es muy común que se inicialice de forma aleatoria, y se van añadiendo mejores soluciones con cada iteración, creándose así un camino de soluciones (trayectoria).

Una de las ventajas de este tipo de métodos es que es posible observar el funcionamiento del algoritmo, comprobando qué soluciones va añadiendo, y la forma en la que recorre el espacio de búsqueda acercándose a un resultado final.

Algunos ejemplos de estos algoritmos son la búsqueda básica local, la búsqueda tabú, la técnica de recocido simulado y la búsqueda de entorno variable.

Explicaremos el funcionamiento de algunos de estos en los apartados sucesivos.

2.2.1.1 Búsqueda de entorno variable

La "búsqueda de entorno variable" [9], en inglés *variable neigborhood search*, es una metaheurística para problemas de optimización combinatoria, propuesta por Mladenović & Hansen en el año 1997.

Estos problemas de optimización combinatoria son a los que nos hemos referido en apartados anteriores, por lo que son necesarias metaheurísticas que no exploren todo el espacio de soluciones, debido a la imposibilidad física de esto, sino que solo exploren una parte, llegando a soluciones aproximadas.

En el caso concreto de este algoritmo, se basa en lo que describe como vecindarios o *neigborhood*. Un vecindario $N(x)$ es aquel en el que todos los puntos del vecindario son modificaciones simples de la solución x siguiendo un criterio *N*. Un ejemplo sería, dado un vector que hay que ordenar, el vecindario podría ser todas las soluciones resultantes al cambiar de sitio dos de los elementos del vector *x*. Adicionalmente, al cambiar el vector *x* por otro distinto *x'* ese vecindario cambia, ya que las permutaciones de dos elementos de un vector no son las mismas que las de otro (aunque algunas puedan coincidir), siendo evidente que cada vector pertenece a muchos vecindarios a la vez.

Una vez definido este concepto clave de esta heurística podemos definir el algoritmo básico, cuya idea es sencilla.

Lo primero será decidir una serie de estructuras de vecindario. Estas estructuras variarán depende del problema, pero dado el ejemplo anterior del vector, una estructura sería intercambiar dos elementos de sitio, otra distinta sería intercambiar tres elementos entre sí, otra intercambiar 4, etc. Una vez tenemos una lista de estructuras de vecindario será necesario generar una solución inicial *s*. Como en muchas de estas heurísticas, esta puede obtenerse al azar o mediante un método previo rápido.

Después entraremos en un bucle donde haremos lo siguiente. A partir de esa solución inicial, buscaremos en la primera estructura de vecindario ($N_0(s)$) una solución mejor *s'*, es decir, que en el caso de optimizar la función $f(x)$, $f(s)<f(s')$. Este algoritmo tiene distintas variaciones de las cuales luego explicaremos alguna, pero una diversificación clásica es la forma de encontrar *s'*. Las dos formas habituales de obtenerla son o hacer una búsqueda local, donde encontramos la mejor solución del vecindario $N_i(s)$ (*Best Improve)*, o nos conformamos con la primera solución que encontremos cuya $f(s')>f(s)$ *(First Improve)*.

Una vez obtenida *s'*, si esta es peor que *s*, lo que haremos será repetir el proceso en la siguiente estructura de vecindario $N_{i+1}(s)$. la condición de parada del algoritmo será que no exista un siguiente vecindario, es decir un $N_{i+1}(s)$ en el que podamos buscar, y consideraremos la solución *s* como solución final.

Este cambio de vecindario se debe a que el mínimo global de la función de optimización que estamos buscando, será también mínimo local de todos los vecindarios a los que pertenezca, por lo que al buscar un mínimo de todos los vecindarios estamos buscando justo eso.

Si, por el contrario, *f(s')>f(s)*, empezaremos de nuevo otra iteración con la nueva solución como solución inicial y empezando desde el primer vecindario.

Como hemos visto, el punto de parada es haber recorrido todas las estructuras de vecindario de una misma solución sin haber encontrado una mejora, pero se pueden añadir otras condiciones de parada como puede ser tiempo de CPU, tiempo de CPU sin una nueva mejora, etc.

Hay distintas variaciones de este método, por lo que vamos a explorar algunas de las alternativas.

2.2.1.1.1 VNS Básico

Este es básicamente el que hemos explicado en el apartado anterior, ya que coge la base de la búsqueda de entorno variable y la aplica directamente sin demasiadas variaciones. Hay que tener en cuenta que todos estos métodos aceptan distintas variaciones pequeñas, como las explicadas anteriormente de *First Improve* y *Best Improve*.

Un ejemplo de pseudocódigo lo podemos ver en la siguiente imagen [7], el cual está simplificado y lo explica muy bien:

```
VNS ALGORITHM:
    Select a set of neighborhood structures N_k(s), with k =1... max;
        // 0. Initialization phase, e.g. |N_1(s)| < |N_2(s)| < ... < |N_max(s)|
    s = Generate-Initial-Solution();
    while termination conditions not met do
      k = 1;
      while k < max do          // Inner loop
        s' = Pick-At-Random (N_k(s));    // 1. Shaking phase
        ŝ = Local-Search (s');   // 2. Local search phase
        if (f(ŝ) < f(s)) then
          s = ŝ;          // 3. Move phase
          k = 1;
        else
          k = k + 1;
        endif
      Endwhile          // end inner loop
    Endwhile
```

Ilustración 4. Pseudocódigo de VNS básico

Como hemos explicado, inicializa unas estructuras de vecindario además de una solución inicial *s*, momento en el cual entra en el bucle, el cual repite hasta que no se alcancen las condiciones de parada. En este hace una búsqueda para encontrar una solución mejor, la cual si encuentra volverá a buscar otra mejor

en la primera estructura de vecindario de esa nueva solución, y en caso contrario avanzará a la estructura siguiente.

Una diferencia de este código respecto a la explicación anterior es la forma de buscar una solución mejor, y es que, aunque se hace una búsqueda local (se explora todo el vecindario), no se hace desde la solución inicial *s* si no desde una solución elegida aleatoriamente de $N_i(s)$, lo que se conoce como *shaking phase*.

2.2.1.1.2 Skewed VNS

Este caso mezcla el VNS básico con el *Simulated Annealing,* para tratar de escapar de mínimos locales [7].

Select a set of neighborhood structures $N_k(s)$, with $k = 1 \ldots max$;
    // 0. Initialization phase, e.g. $|N_1(s)| < |N_2(s)| < \ldots < |N_{max}(s)|$
s = Generate-Initial-Solution();
while termination conditions not met **do**
  $k = 1$;
  while $k < max$ **do** // Inner loop
    s' = Chose-Best-Of ($N_k(s)$);
    if ($f(s') < f(s)$) **then**
      $s = s'$; // Move phase
    else
      if rand() < $e^{-(f(s')-f(s))}$ **then**
        $s = s'$; // Flexible move phase: Accepting a worse s' as new solution with probability $e^{-(f(s')-f(s))}$
      else
        $k = k + 1$; // no improvements: local minimum reached; let's investigate in another neighbor
      endif
    endif
  Endwhile // end inner loop
Endwhile

Ilustración 5. VND

Como podemos ver, el cambio más destacable de este algoritmo base es que este acepta soluciones peores a la actual con cierta probabilidad (grande al principio y pequeña al final) de la misma forma que el *Simulated Annealing*.

Otra pequeña variación a destacar es que lo habitual en esta alternativa es que en la búsqueda de soluciones en el vecindario se explore por completo, eligiendo la mejor solución. Esto se debe parcialmente a que, debido a que es posible seleccionar soluciones peores, es mejor para la precisión del algoritmo que esta sea al menos la mejor del vecindario.

2.2.1.1.3 Otras variantes

Existen más variaciones del VNS, pero por destacar dos últimas vamos a explicar dos alternativas más complejas que las anteriores. Estas son la búsqueda descompuesta y el *Primal-Dual VNS*.

En la búsqueda descompuesta, la idea es crear sub-problemas mediante la fragmentación de los vecindarios, yendo de grupos más pequeños a más grandes. De esta forma, vamos obteniendo soluciones parciales, en las que por ejemplo solo podemos cambiar un número reducido de atributos, y vamos ampliando la solución ampliando el espacio de búsqueda hasta llegar a la solución local.

En la variante del *Primal-Dual*, consiste en obtener una solución dual en la que tengamos una solución primaria y otra secundaria la cual no es factible, y a partir de estas dos y mediante una heurística obtener unos límites inferiores y superiores a partir de los cuales obtener la solución óptima al problema original.

2.2.1.2 Algoritmo de recocido simulado

Otra de las metaheurísticas para los problemas de optimización combinatoria se trata del algoritmo de recocido simulado [10] o *Simulated Annealing*.

Este algoritmo es utilizado en casos donde el espacio de búsqueda es excesivamente grande; recordemos el caso del cartero chino, donde tiene que recorrer 20 ciudades en orden, por lo que el espacio de búsqueda es de 20!, que equivale a 2.4 cuatrillones de posibilidades. En este tipo de casos, no nos interesa tanto una solución exacta y óptima, sino una aproximada de resultados aceptables.

El nombre de este método proviene del proceso metalúrgico del recocido, donde se calienta y enfrían fragmentos de algún material (acero o cerámica) para incrementar el tamaño de los cristales a la vez que reducir su tamaño. El calor causa energía que hace que los átomos se activen, cambiando de posición más rápido, mientras que al enfriarse comienzan a dar saltos más pequeños hasta que recuperan una nueva posición.

En esto se basa el recocido simulado, el cual parte de una solución inicial, y va dando “saltos” aleatorios cambiando de solución, dando más al principio y menos al final, a semejanza con los átomos. De forma más específica, estos “saltos” que significan un cambio en la solución, se traduce en el algoritmo como que siempre se aceptará una solución mejor, y las soluciones peores será más probable que las aceptemos al principio y menos al final.

Para ello, este algoritmo define también el concepto de vecindario de forma similar a la búsqueda de entorno variable. Así, una solución solo puede cambiar a una de sus soluciones vecinas. Para ello necesitamos definir en cada problema el concepto del vecindario, por ejemplo, en el caso de ordenar un vector, podríamos utilizar el vecindario de cambiar dos elementos de sitio, o podríamos intercambiar más elementos. Hay que tener cuidado y no definir vecindarios

excesivamente grandes, si no que los saltos sean entre soluciones cercanas, ya que un gran salto aleatorio hacia una mala solución podría empeorar el resultado final. En el caso extremo, cojamos el caso del cartero chino, donde hay 20 ciudades. Si definimos el vecindario como intercambiar las 20 ciudades entre sí (el vecindario es la totalidad del espacio de búsqueda) estaríamos cogiendo soluciones al azar hasta que encontremos una aceptable. Teniendo en cuenta la enorme cantidad de posibilidades que hay, es improbable que acabemos con un mínimo global como solución en un tiempo acotado.

Al coger vecindarios más pequeños, conseguimos acercarnos a la solución poco a poco, y en caso de aceptar una solución peor, no nos hemos desviado tanto de donde estábamos.

Este punto nos lleva a preguntarnos cuál es el sentido de aceptar soluciones peores. Lo podemos ver de forma más clara en la siguiente imagen [11]

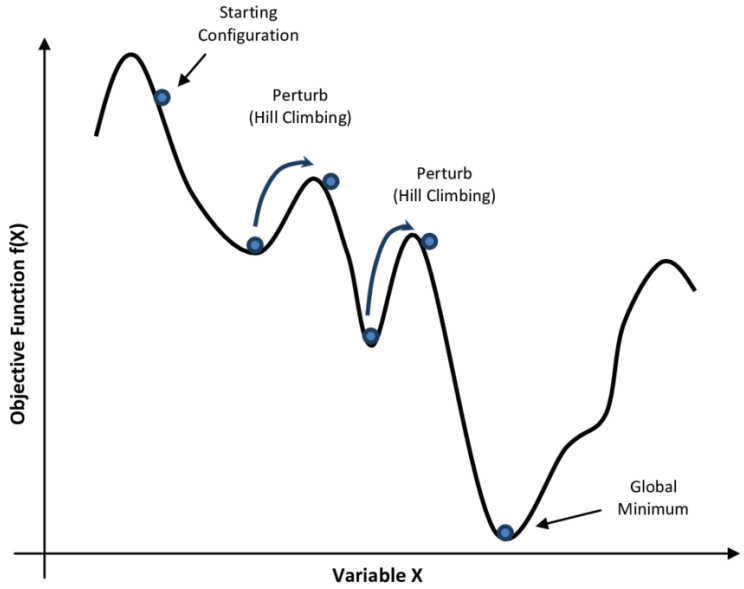

Ilustración 6. Función de soluciones de recocido simulado

Supongamos que la función a optimizar tiene la forma de la imagen, que queremos minimizar la función, y que el orden en el que elegimos las soluciones va de izquierda a derecha, es decir, el primer punto se ha seleccionado antes que el segundo, que ha sido seleccionado antes del tercero, etc). Del primer punto, llegamos al segundo mediante cierto número de cambios de solución, los cuales están permitidos porque vamos en sentido descendente, cogiendo mejores soluciones. Sin embargo, sería muy complicado llegar del punto dos a otro mejor (el cuarto), porque hay una colina, y recordemos que debido a los vecindarios no podemos seleccionar soluciones muy alejadas. Debido a esto, la única forma de mejorar sería aceptando el punto 3 como solución, a partir de la cual si podemos llegar a soluciones mejores.

De esta forma, aceptar soluciones peores es un método para escapar de mínimos locales, y buscar el mínimo global. Según pase el tiempo, aceptaremos menos soluciones peores, porque entendemos que el algoritmo se está acercando poco a poco al mínimo global, por lo que llega un punto que pierde sentido aceptar un estado peor.

Una vez explicado el algoritmo, vamos a ver un ejemplo de pseudocódigo que lo ilustre [7].

```
s = s0;  //Generate-Initial-Solution()
T = T0;
e = e0;  //e0 > emax
k = 0;
while k < kmax and e > emax do
  s' = neighbor(s);  //Pick-At-Random(N(s))
  if f(s') < f(s) then
    s = s';  //s' replaces s
  else
    if rand() < e^(-(f(s')-f(s))/T) then
      s = s';        //Accepting a worse s' as new solution with a
                     given probability
    endif
  endif
  Update(T);
  k = k + 1;
Endwhile  //termination conditions met
return s;
```

Ilustración 7 Pseudocódigo del recocido simulado

El algoritmo comienza seleccionando una solución inicial. Como ya hemos comentado en otras heurísticas, hay diversas alternativas para ello, ya que podemos seleccionar una solución inicial de forma totalmente aleatoria, o tener un algoritmo rápido que encuentre un buen punto desde el que partir, ya sea mediante otras heurísticas o por conocimiento adicional al problema específico.

Una vez seleccionada la solución inicial, es momento de establecer el parámetro clave de este algoritmo, la temperatura. Esta variable, de forma simular al método metalúrgico, determina la probabilidad de aceptar soluciones peores, por lo que, a mayor temperatura, mayor probabilidad habrá, sobre todo en las primeras iteraciones.

Una vez declaradas estas variables, ya podemos comenzar con el bucle para hallar el mínimo global, el cual terminaremos cuando se cumplan las condiciones de parada definidas, igual que en las otras heurísticas, donde podemos elegir número de iteraciones, tiempo de CPU, etc. Dentro de este bucle lo primero será elegir una nueva solución del vecindario de forma aleatoria, y

comprobar si esta es mejor que la solución anterior. Si lo es, la nueva solución reemplazará a la anterior. En caso de que sea peor, se pasará por una función que determinará si se descarta, o si se acepta.

Aquí hay varias opciones, pero explicaremos una de las más utilizadas. Se calcula la diferencia entre los resultados de la función de utilidad con cada una de las soluciones, este valor lo dividimos entre el valor de la temperatura, lo multiplicamos por menos uno, y elevamos el número *e* por el resultado de esta operación, la cual queda de la forma:

$$Probabilidad = e^{-\frac{f(s')-f(s)}{T}}$$

Ecuación 1 Probabilidad de aceptar soluciones peores

Por otra parte, generaremos un número aleatorio entre 0 y 1, y si número es menor que el resultado de la ecuación anterior, aceptaremos la nueva solución, aun siendo peor que la anterior, reemplazándola. De esta forma, hay mayor probabilidad de aceptar una solución peor cuanto mayor sea el resultado de la ecuación. Esto se consigue si la temperatura es muy elevada, o si la diferencia entre los resultados de la función de optimización no es muy grande, es decir, cuanto peor sea la nueva solución, más diferencia habrá, por lo que habrá menos probabilidades de aceptarla.

Antes de volver a ejecutar una nueva iteración del bucle, solo queda actualizar la temperatura, reduciéndola para que se vaya acercando al 0, momento en el cuál no se aceptarían soluciones peores. Para esto, las 2 opciones fundamentales son multiplicarla por un factor menor que 1, o restarle una constante, como la temperatura inicial entre el número de iteraciones.

Por norma general, en problemas donde el número de posibles soluciones es enorme, y donde no hace falta una gran exactitud sino una solución aproximada, este algoritmo es de gran utilidad y consigue resultados muy buenos.

2.2.2 Metaheurísticas basadas en poblaciones

A diferencia de las anteriores, en este tipo de técnicas no se va de una única mala solución a una mejor, si no que en cada iteración se trata con una lista de soluciones (población) más grande. Esto permite estudiar el espacio de búsqueda de una forma más amplia.

Algunas de las principales técnicas basadas en población son la computación evolutiva y las técnicas de la inteligencia de enjambre.

2.2.2.1 Computación Evolutiva

Este tipo de técnicas se basa en la teoría de la evolución de los seres vivos. En esta se defiende que los seres vivos de la actualidad descienden de otros organismos completamente distintos, los cuales, mediante combinaciones entre distintos organismos y mutaciones propias, crearon nuevos organismos, de los cuáles solo sobrevivieron los más fuertes (los mejores preparados para sobrevivir).

De esta forma, aunque hay distintos tipos de algoritmos de computación evolutiva [12] y cada uno se apoya más en unos principios que en otros, todos utilizan tres operaciones fundamentales basadas en la teoría de la evolución:

- Reproducción. En esta se combinan dos o más individuos (antepasados), creando nuevos individuos (hijos), los cuales tendrán características mezcladas de los antepasados.
- Mutación. Cada individuo puede mutar, cambiando por sí mismo algunos de sus atributos. Esta operación es más similar a buscar una nueva solución en los algoritmos de trayectoria, ya que el objetivo no es cambiar al individuo por completo, sino producir pequeñas modificaciones.
- Selección de los individuos "más fuertes". A los individuos se les evalúa mediante una función de desempeño (*fitness)*, la cual determina el valor de una solución para pertenecer a la población. De esta forma, de una iteración a otra eliminaremos a los individuos menos valiosos, aunque no de una forma estricta, sino que a mayor *fitness* presente un individuo, mayor probabilidad tendrá de pertenecer a la siguiente población.

Por tanto, el objetivo principal de estos métodos en cada iteración consiste en ampliar la población mediante las dos primeras operaciones, y descartar los individuos peores con la tercera operación, teniendo una población mejor en cada iteración, acercándonos así al mínimo global. Al igual que en el recocido simulado, hay probabilidad de quedarnos con soluciones peores de una población a otra, para evitar quedar atrapados en mínimos locales.

Cosas a tener en cuenta de estos métodos es que hacen falta determinar ciertos componentes para afrontar un problema:

- Una forma de crear una población inicial. Para asegurarnos alcanzar un buen resultado, es necesario coger una muestra diversificada que incluya gran cantidad de atributos para que estos puedan darse en sus sucesores.
- Una función de evaluación que valore cada individuo para asociarle su *fitness*. En este caso, esta función podría ser la misma función de optimización de las heurísticas de trayectoria.
- Definir las operaciones que se ejecutarán en la población. Algunos métodos les dan más importancia a algunos operadores que a otros. De igual forma, no se ejecutarán todas las operaciones en todos los individuos, si no que normalmente, los individuos con mayor *fitness* tendrán más probabilidad de reproducirse o de mutar, ya que son los que aportan mayor valor.

- Valores para los parámetros del algoritmo. Hay diversos parámetros que pueden modificarse y que influyen en el rendimiento del algoritmo. Algunos ejemplos son el tamaño de la población inicial, la probabilidad de aplicar las operaciones genéticas a cada individuo, probabilidad de aceptación de los individuos, etc.

Como podemos ver, estos métodos son más complejos de forma general en su implementación a un problema que los basados en trayectoria, y la buena definición de estos parámetros depende en gran medida de conocer en profundidad el problema al que nos enfrentamos. A pesar de esto, estos algoritmos suelen obtener resultados muy buenos y son tenidos en cuenta normalmente.

Las tres corrientes más aceptadas de la computación evolutiva son:

- Algoritmos genéticos.
- Estrategia evolutiva.
- Programación evolutiva.

Por norma general, los algoritmos genéticos se apoyan más en la reproducción de los individuos, mientras que las otras dos vertientes también hacen uso de las mutaciones.

2.2.2.2 Inteligencia de enjambre

La inteligencia de enjambre es una rama de la inteligencia artificial que se encarga de estudiar comportamientos colectivos, los cuales suelen ser descentralizados y no organizados. Por tanto, los problemas de esta rama suelen estar representados por un conjunto de individuos (el enjambre), los cuales actúan de forma independiente unos de otros siguiendo unas normas sencillas. Para actuar, estos individuos interaccionan y se basan tanto con el medio ambiente como con el resto del enjambre, por lo que se acaban generando unos patrones de comportamiento globales para toda la población.

Aunque no parece que este ámbito sea aplicable para el problema que nos ocupa, este proporciona una forma alternativa de alcanzar una solución, mediante el encargo al enjambre de encontrar soluciones óptimas, las cuales comprenden el espacio de búsqueda.

Hay muchos algoritmos distintos dentro de este ámbito, por lo que nos vamos a centrar en dos de los más famosos que se utilicen para problemas de optimización combinatoria, esos son el algoritmo de la colonia de hormigas y la optimización por enjambre de partículas.

2.2.2.2.1 Algoritmo de la colonia de hormigas

Como es evidente, este algoritmo [13] debe su nombre al comportamiento real de las hormigas en el momento de buscar comida. Cuando las hormigas encuentran comida, vuelven a su colonia dejando un rastro de feromonas para el resto de las hormigas. Depende de la calidad y de la cantidad de la comida encontrada, dejará un rastro más fuerte o débil. En el mundo real, este las hormigas no dejan el rastro necesariamente de una intensidad distinta, sino que, además, si la comida está muy lejos, este rastro desaparecerá antes debido a la tardanza en volver. Así, las hormigas son atraídas con cierta probabilidad, dependiendo de la fuerza de las feromonas, a los caminos establecidos, con lo que los mejores caminos poco a poco irán volviéndose más concurridos, hasta el momento en el que los caminos ineficientes pierdan totalmente el rastro de feromonas y todas las hormigas transiten el mejor camino.

El algoritmo funciona de forma parecida. En este caso, el espacio de búsqueda tiene forma de grafo, donde cada vértice es una solución, y las aristas el camino a recorrer. En cada iteración un número predeterminado de hormigas escogen un camino, al principio será aleatorio, pero una vez llegan a la solución y vuelven, cada hormiga dejará un rastro de feromonas. La cantidad de estas feromonas (numérica), dependerá del valor de la función objetivo de esa solución, dejando más feromonas cuanto mejor sea la solución, y cuantas más hormigas recorran ese camino, más feromonas habrá en total.

De esta forma, las hormigas empezarán a coger los caminos con mayor cantidad de feromonas mediante una función probabilística, siendo más probable escoger el camino con más feromonas, aunque sin cerrarse a escoger otros peores. A su vez, en cada iteración se reducirán las feromonas de cada camino, simulando a la evaporación de la vida real. Una forma típica de reducirlo es multiplicándolo por una constante de desaparición (feromonas multiplicado por un valor menor que uno), y reduciéndolo también en base a la distancia a esa solución.

Según pase el tiempo, las hormigas empezarán a converger en los caminos más cortos, desapareciendo todo el rastro de feromonas del resto de caminos, momento en el cuál podremos seleccionar la solución final.

2.2.2.2.2 Optimización por enjambre de partículas

La optimización por enjambre de partículas [14] es una metaheurística que, al igual que muchas otras metaheurísticas que estamos viendo, se basan en aspectos del mundo real.

El algoritmo se creó originalmente para el estudio de comportamientos sociales de ciertos animales, y cómo se comportaban al estar en grupo (como bandadas de pájaros), pero tras una simplificación al problema empezó a usarse para problemas de optimización combinatoria, con lo que era independiente del problema al que se aplicase, característica común de estos métodos.

El algoritmo parte de un conjunto de partículas que forman el enjambre. Estas partículas se mueven por el espacio de soluciones en base a dos criterios, la mejor posición (solución) que ha llegado a encontrar esa misma partícula, y la mejor posición encontrada por el enjambre. De esta forma, se aplicará una fórmula para ponderar esos dos criterios y guiar a cada partícula, por lo que, aunque todas se mueven de forma independiente, una de las componentes les hace moverse como un grupo.

Mediante la iteración de este paso, el enjambre se va moviendo (y es común que se divida en varios grupos), buscando mejores soluciones hasta que se cumplan las condiciones de parada, las cuales se definen de igual forma que en el resto de los métodos estudiados, ya la mejor solución encontrada por todo el enjambre será la que se devuelva como solución final.

En cada iteración, cada partícula calculará la distancia a la mejor solución del enjambre y a su mejor solución, y ponderándola, calculará una velocidad de movimiento en esa dirección. La partícula no se moverá hasta el punto calculado ponderado, sino que aquí interviene el ratio de aprendizaje, y es que proporción (de 0 hasta 1) de todo ese espacio se moverá. De esta forma, elegir los valores correctos para estos parámetros será de gran importancia para la buena ejecución del algoritmo.

Al igual que muchos otros, este algoritmo no garantiza alcanzar el mínimo global aportando solo una solución aproximada, por lo que no es recomendable si se busca una solución exacta.

Esta explicación del algoritmo es su versión básica, aunque de forma adicional hay distintas variaciones que pueden aplicarse para intentar encontrar soluciones de forma más eficiente y eficaz. Dos cambios comunes son combinar este método con otras heurísticas, como viene siendo común, y otro es añadir ciertos parámetros para evitar la convergencia inicial del enjambre.

Esto último se debe a que, si una partícula encuentra una solución muy buena desde el principio, el enjambre tenderá a converger en ese punto, pudiendo ser arrastrado hacia un mínimo local, ya que, además, si todos llegan a esa zona sin encontrar una solución mejor, esta heurística no tiene un mecanismo para escapar de él, ya que su mecanismo es que el enjambre explore gran parte del espacio de búsqueda de forma simultánea.

2.3 Knowledge Based Matrix To List

Como hemos visto en puntos anteriores, existen formas de representar el conocimiento asociado a problemas de decisión, pero cuando estos son excesivamente grandes, surgen grandes dificultades.

Aunque en todos los sistemas los problemas son parecidos, vamos a centrarnos en las tablas de decisiones. Cuando el problema es muy grande, estas contienen

millones de filas, lo cual trae los problemas de la memoria ocupada en el ordenador, y la imposibilidad de las personas humanas de interpretar esa tabla.

Para solventar esos problemas se proponen las *Knowledge Based Matrix To List* [3] (KBM2L). Basadas en las tablas de decisiones, las KBM2L buscan reducir esta matriz en gran medida, para posteriormente convertirla en una lista, ya que el formato que se propone es también más reducido que una matriz (tabla).

La idea fundamental que se propone para reducir la tabla es agrupar los casos por la decisión que se toma, y guardar la información de todo el bloque entero que comparta la misma decisión, sin tener que guardar cada una de las entradas que lo componen.

Por tanto, aunque existen otras aproximaciones a este tipo de problemas (como truncar o aproximar los árboles de toma de decisiones), una de las cosas que diferencia este enfoque es que se busca conservar la misma información de la que disponíamos. Por ello, las transformaciones que se harán se compondrán solo de permutaciones en las columnas y reordenación en las filas, pero sin eliminar información.

Al agrupar y reducir la tabla en base a la decisión tomada, estamos solventando los dos problemas fundamentales que teníamos:

- La lista ocupará mucho menos porque no estamos guardando todas las entradas, sino solo información de los bloques, los cuales nos permiten recuperar las filas que lo componen.
- La lista ahora sí que contiene conocimiento. Como estamos agrupando los casos por la decisión tomada, los usuarios pueden ver patrones, y los atributos que influyen en que se tomen decisiones distintas. Así, esto ya no es solo una tabla de decisiones, sino que es una lista de conocimiento.

Vamos a explorar cómo se construyen estas tablas, y las formas de optimizar la búsqueda en ellas.

2.3.1 Construcción

Como hemos visto, este tipo de tablas son una extensión de las tablas de decisiones, por lo que tienen muchos elementos en común, aunque estos necesiten una nueva representación. Por tal motivo, necesitaremos expresar estos elementos de una forma distinta.

Así, al conjunto de todos los atributos (columnas de la tabla) lo denominaremos esquema, y si ponemos estos atributos en un orden concreto, lo definiremos como una base. Si cambiamos de orden la base original de una tabla de decisiones, estaremos cambiando las columnas de sitio, pero seguiremos conservando la misma información, aunque su representación sea distinta.

El dominio de estos atributos suele ser numérico, y en el caso de que no lo sea (atributos categóricos) para su representación en estas tablas se convierte a una

representación numérica discreta en la que cada número identifica a una clase. Con esto surge el concepto de índice (o *index*), que es un vector que contiene (en el mismo orden que la base) los valores de para cada uno de los atributos del esquema para una entrada de la tabla en concreto.

De esta forma, dado un determinado orden en el dominio de cada atributo y dada una base específica, la tabla se puede ordenar yendo de la entrada con menor valor en estos atributos hasta la entrada con el máximo de todas.

Con esta definición, lo que tenemos es una matriz multidimensional (MM) en la que las filas representan a cada observación, y cada celda es el valor para un atributo para esa entrada.

La lista que queremos crear tiene que estar ordenada siempre en orden ascendente en función de los valores de los atributos. Por tanto, a este vector de coordenadas que es el índice, hay un vector del mismo tamaño asociado conocido como los pesos. Estos pesos son similares a cualquier otro sistema de representación numérica. Pongamos el ejemplo de la notación decimal, en la que, para obtener el número final, hay que multiplicar cada cifra por un factor, y luego sumar los resultados. En ese caso, a las unidades se las multiplica por uno, a las decenas por 10, a las centenas por 100, etc. En nuestro caso es lo mismo, y el vector de pesos es el vector que define por qué número hay que multiplicar cada valor de un vector de coordenadas (el índice) para obtener la posición que ocupa esta entrada en la tabla.

La diferencia es que nosotros no estamos en una escala decimal, sino que a cada elemento hay que multiplicarlo por la cantidad de combinaciones que se pueden crear con las coordenadas previas a ese atributo. Vamos a explicar cómo se hallaría este vector de pesos.

- El último atributo (el de menos peso), su peso asociado es 1 de igual forma que en la escala decimal, ya que no hay atributos previos.
- El penúltimo peso habrá que multiplicarlo por las combinaciones previas a este, es decir, el dominio del último atributo. Así, si el último atributo es binario, solo habría que multiplicarlo por dos.
- El antepenúltimo valor del vector de pesos habrá que multiplicarlo de igual forma por las posibles combinaciones anteriores, lo que supone la multiplicación del dominio del último atributo por el dominio del penúltimo atributo, o lo que es lo mismo, el valor del peso de atributo anterior por su dominio.
- Así sucesivamente hasta completar el vector.

Con este vector, y haciendo la suma de los elementos del resultado de hacer el producto vectorial del vector de pesos por las coordenadas del índice, podemos hallar la posición numérica de una entrada en la tabla. A esta posición le llamaremos el *offset*.

De forma inversa, y análogamente a al caso decimal, dividiendo un determinado offset por u elemento del vector de pesos hallaríamos el valor del índice de un atributo, por lo que también se puede hacer la transformación de offset a index.

De esta forma, cuando cambiamos de base (reordenamos los atributos) también se reordenan los índices y el vector de pesos, por lo que las entradas de la tabla también alteran su posición (offset), de tal forma que esta siempre esté ordenada de forma creciente. Vemos que al alterar el orden de los atributos no estamos cambiando nada referente a la información que aporta la tabla, ya que sigue conteniendo los mismos datos, aunque sea en distinto orden.

Para los siguientes pasos en la construcción de una KBM2L, vamos a poner un ejemplo real. Supongamos la siguiente tabla de decisión, donde A_i son los distintos atributos binarios y D_i las decisiones a tomar de cada entrada. En este caso solo tenemos un valor de decisión, pero recordemos que puede haber múltiples decisiones.

Offset	**A1**	**A2**	**A3**	**D1**
0	0	0	0	0
1	0	0	1	0
2	0	1	0	1
3	0	1	1	1
4	1	0	0	1
5	1	0	1	2
6	1	1	0	2
7	1	1	1	2

Tabla 1 Ejemplo 1 de KBM2L

Esta es una tabla de decisión cualquiera, en la que hay tres atributos y una decisión. Los dominios binarios para todos los atributos (0,1), y el dominio es (0,1 y 2) para la decisión. La columna de la izquierda representa el offset.

Como el nombre indica, lo que queremos crear es una lista, y una forma de crear una lista crear un elemento por cada entrada de la tabla, en el que conservemos toda la información. Dado lo que hemos explicado, nos podemos ahorrar toda la representación de los atributos, y ahorraremos mucho espacio si lo que almacenamos en vez es el offset (recordemos que cada offset representa a una única combinación de valores para cada atributo). Así, de cada elemento tendremos 2 atributos, el offset y la decisión.

En caso de haber más decisiones, estas también se pueden representar de la misma forma que el offset, mediante una conversión con los pesos en el vector que contiene a las decisiones. Así, la lista quedaría de la siguiente forma:

(0,0) , (1,0) , (2,1) , (3,1) , (4,1) , (5,2) , (6,2) , (7,2)

Lo que queremos hacer es agrupar estas por decisiones, y coger un representante de cada clase, para lo cual cogeremos el elemento con mayor offset de cada bloque de entradas seguidas con la misma decisión. A cada bloque le llamaremos *ítem*, y a este elemento mayor lo llamaremos supremo de ese ítem en específico. De la misma forma, a la entrada con menor offset de un ítem la llamaremos ínfimo.

De esta forma, los dos primeros elementos de esa lista (offset 0 y 1) compondrían un bloque, ya que comparten la misma decisión (0), los dos siguientes elementos comparten la decisión 1, y las últimas 3 entradas de la tabla tienen la decisión 2. Por tanto, nos hallaríamos ante una lista de 3 items, cada uno de los cuales lo representaremos con el supremo de ese ítem.

Una vez sintetizada, nos quedaría la siguiente lista:

<1,0 | <4,1 | <7,2

Para ello vamos a usar una notación común a lo largo del trabajo. Para ello estamos separando los ítems con "|", e indicamos que empieza la definición de un ítem con el símbolo "<". Lo que quiere decir esta lista, es que todos los elementos con offset igual o menor que 1, tienen la decisión 0, los que estén entre 4 y 1 (cuatro incluido), tendrán la decisión 1, y el resto (menores o iguales que 7, aunque mayores que 4, ya que tienen su propio ítem) tendrán la decisión 2. De esta forma, hemos pasado de una matriz de 32 elementos (8x4) a una lista de 8 tuplas, para luego conseguir una lista de 3 tuplas, en total 6 elementos. A esta última lista, será lo que llamaremos una KBM2L, y en esta en específico solo tenemos 3 ítems. Con otra base, la ordenación de la lista sería distinta por lo que para una misma tabla construiríamos un KBM2L con un número distinto de ítems, donde en el peor de los casos tendremos tantos ítems como entradas en la tabla, cuando cada entrada tenga una decisión distinta de la anterior.

Vemos de esta forma que hemos pasado de una matriz multidimensional a una lista, por lo que, haciendo referencia al significado de KBM2L (*knowledge based matrix to list*) hemos transformado una matriz basada en conocimiento (la solución de toma de decisiones de un problema) y la hemos transformado en una lista que mantiene el mismo conocimiento.

Otra forma de representar la lista es mediante el index en vez del offset. Esta lista ofrece la misma información, pero no indica la posición del supremo en la tabla, sino sus coordenadas del índice, y la lista final anterior quedaría de la siguiente forma:

<(0,0,1) ,0 | <(1,0,0) ,1 | <(1,1,1) ,2

Vamos a ampliar este ejemplo cambiando la base de la tabla anterior, para que se pueda apreciar el cambio en el número de ítems. Así, si antes la base era B_0=(A1,A2,A3), la nueva base será B_1=(A2,A3,A1), por lo que la tabla, al cambiar las columnas quedaría de la siguiente forma:

A2	**A3**	**A1**	**D1**
A2	**A3**	**A1**	**D1**
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	1
0	1	1	2
1	0	1	2
1	1	1	2

Tabla 2 Ejemplo 2 de KBM2L desordenada

De momento solo hemos cambiado las columnas de sitio, por lo que ahora faltaría reordenar las filas para que vayan en sentido ascendente según la nueva base y así poder calcular el nuevo offset (el cuál es distinto con cada base).

Offset	A2	A3	A1	D1
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	2
4	1	0	0	1
5	1	0	1	2
6	1	1	0	1
7	1	1	1	2

Tabla 3 Ejemplo 2 KBM2L ordenado

A primera vista ya podemos intuir que la lista va a ser más larga, debido a que las decisiones están menos agrupadas. Creando la KBM2L obtenemos lo siguiente:

<0,0 | <1,1 | <2,0 |<3,2 | <4,1 | <5,2 | <6,1 | <7,2

Aunque seguimos ahorrándonos memoria debido a que esta lista ocupa menos que la matriz original, nos hallamos en el peor de los casos: no hemos conseguido reducir la lista de entradas. Podemos ver que para representar la tabla de 8 entradas hemos necesitado 8 ítems, por lo que seguimos almacenando la información de todos los casos.

Es justo en este mismo caso donde podemos ver el objetivo final de este trabajo: crear un software capaz de obtener la mejor base posible.

El ejemplo 1 era un ejemplo irreal creado para la explicación de las KBM2L, donde hemos puesto todas las decisiones concentradas, pero en ejemplos del mundo real esto no suele suceder así, y es más normal encontrarnos con una tabla similar a la segunda o al menos un punto intermedio, donde no somos capaces de reducir tanto el número de casos simplemente pasándolo a lista. Por ello, es en estos casos donde necesitamos buscar entre todas las bases aquellas que nos permitan obtener el menor número de ítems posibles. Esta búsqueda la haremos mediante metaheurísticas de optimización combinatoria, los cuales se explicarán más adelante a lo largo de esta memoria.

Por otra parte, estas listas no siempre se construyen de una única tabla, sino que a veces el problema está dividido en tablas más pequeñas, las cuales contienen un sub-grupo distinto de entradas, que al unirlas todas crearían la tabla completa. Esto se debe a que como hemos dicho a menudo la matriz

original es demasiado grande para tratarlo de ninguna forma. En estos casos, mientras estamos construyendo la KBM2L se pueden quedar espacios en blanco de los que no tenemos datos (no tenemos la decisión que se toma, pero el índice sí, ya que al conocer los atributos y los dominios sabemos todas las posibilidades que existen). En este caso, estas entradas que todavía no tengamos la información de la decisión que se toma se le asignará a la decisión el valor "-1", el cuál es un valor arbitrario que estamos utilizando como notación para indicar al programa (y al usuario), que esa decisión se desconoce (*unknown*).

Este valor "-1" no solo se utiliza durante la construcción de la lista, sino que puede haber problemas para los que no tengamos la decisión óptima calculada para cada una de las entradas, por lo que a estas se le asignará también este valor. Esta decisión "-1" se trata como el resto de las decisiones, y la KBM2L las agrupará como si fuese otro número normal. En el caso extremo, si en la tabla anterior no dispusiésemos de ninguna de las decisiones, es decir, todas estuviesen rellenas con un "-1", la KBM2L sería de la siguiente forma:

<7,-1

Esta lista indica justo eso, que no tenemos la decisión para ninguno de los casos, por lo que, aunque solo tenga un ítem, esta lista no tiene ninguna utilidad.

2.3.2 Optimización de la búsqueda de bases

Una vez hemos comprendido el funcionamiento de las KBM2L, es el turno de explicar algunas ideas que nos permitirán optimizar los procesos de búsqueda para encontrar las mejores bases.

Debido a la complejidad del problema, el espacio de búsqueda es extremadamente grande, por lo que resulta imposible explorarlo en su totalidad en un tiempo acotado para encontrar la mejor solución. Esto se debe a que el problema crece de forma factorial, siendo las posibilidades de ordenar las bases d!, siendo d el número de atributos, convirtiéndose en un problema de optimización es espacio de permutaciones (n!). Si calculamos el número de posibilidades según el número de atributos (empezando por dos atributos), la lista de posibilidades en orden sería: 2, 6, 24, 120, 720, 5040, 40320, 362880… En el caso de 20 atributos, ya estaríamos hablando de cuatrillones de posibilidades, lo que crea la necesidad de encontrar heurísticas y métodos que nos ayuden en la búsqueda.

2.3.2.1 Explotación de los índices

Del valor de los índices podemos sacar un doble conocimiento: por una parte, podemos extraer conocimiento sobre el esquema de la toma de decisiones obteniendo una explicación, y por otro, pueden ayudar a obtener mejores bases.

En problemas donde la cantidad de atributos es especialmente grande, hay una probabilidad muy alta de empezar el proceso con una base bastante mala en lo referente al número de ítems que proporciona, así que una alternativa es hacer una pequeña transformación para empezar con una base mejor, y partir desde ahí.

Para esto hay varias alternativas y lo más eficiente sería que con el suficiente conocimiento del problema, un experto de ese ámbito podría decir cuáles son los atributos más influyentes y cuáles menos, con lo cual podría organizarse una base mejor. Como este trabajo trata todos los problemas de la misma forma sin tener en cuenta el tema al que representan, esa no es una opción, y lo que se propone es estudiar la parte fija de los atributos de los ítems.

La parte fija son aquellos atributos que tienen el mismo valor para todos los elementos de un mismo ítem, mientras que la parte variable son aquellos atributos que toman distintos valores. Esta parte fija, aunque no explica por completo por qué comparten política, si es una base orientativa de por qué puede ser. Por el contrario, la parte variable es aquella en el que los elementos de un mismo ítem toman distintos valores, por lo que esta parte no nos aporta a priori ninguna información. Debido a la ordenación de la tabla, para obtener la parte fija de un ítem no es necesario revisar todos los casos de ese bloque, si no que bastaría con aplicar la puerta lógica AND al supremo y al ínfimo de ese mismo ítem, y el resultado será esa parte fija.

Así pues, una estrategia para obtener una base mejor que la inicial es localizar la parte fija de los ítems que compartan una misma política, y llevar estos atributos a posiciones de más peso, para intentar agrupar estos ítems. Esta agrupación se debe a que si dos ítems distintos comparten una parte fija, es probable que estos ítems puedan ser agrupados, ya que comparten esa parte explicativa que hemos llamado parte fija, pero puede ser que la base no esté ordenada correctamente.

Llamaremos posición de más peso (o importancia) cuanto más a la izquierda del vector esté, de forma análoga al vector de pesos, mientras que los de bajo peso estarán a la derecha. Así, intercambiando de posición los atributos 2 y 3, estaríamos haciendo una pequeña transformación que nos permitiría empezar el problema desde una mejor base. Podemos verlo con un ejemplo práctico.

Supongamos el siguiente problema el cual tiene 5 atributos, todos binarios, la decisión es también binaria, siendo x e y las políticas, y la base inicial B_0=[0,1,2,3,4], y la KBM2L resultante es la siguiente.

< 3, x | < 7, y | < 9, x | < 31, y

Y la representación en formato índice:

< (0, 0, 0, 1, 1), x | < (0, 0, 1, 1, 1), y |, < (0, 1, 0, 0, 1), x | < (1, 1, 1, 1, 1), y

Podemos observar que hay cuatro ítems, dos con política *x* y dos con policía *y*, pero que están alternados, por lo que la base ideal (si es que existe), obtendría únicamente dos ítems. Vamos a coger la parte fija del primer y tercer ítem (política *x*).

- El primer ítem tiene como parte fija los atributos 0,1 y 2, tomando siempre el valor 0.
- El tercer ítem tiene como parte fija los 4 primeros atributos, tomando todos el valor 0, menos el segundo atributo que vale 1.

De aquí se pueden obtener dos conclusiones. La primera es que el último atributo nunca es parte fija, por lo que la información que aporta para esta decisión es nula. La segunda conclusión es que los dos ítems comparten como parte fija los atributos 0 y 2, por lo que hace pensar que estos atributos son los que explican que la política de todos estos casos sea la descrita. Así pues, vamos a cambiar la base para poner el atributo 2 en una posición de más importancia que el 1 (el 0 ya está en la posición más importante) con lo que la nueva base será B_1=[0,2,1,3,4]. Haciendo la transformación a la KBM2L nos queda la siguiente lista:

< 5, x | < 31, y

Con este sencillo cambio, hemos juntados los ítems de la misma política. Hemos obtenido un beneficio doble, ya que al juntar los ítems con la decisión *x*, se han fusionado también los ítems de otras políticas, ya que los otros bloques no están en medio de ellas ahora.

Con esta aproximación, reforzamos la idea de que el resultado que se obtiene de optimizar una KBM2L es doble:

- Por un lado, estamos convirtiendo las tablas en elementos que ocupan mucha menos memoria y son más fáciles de navegar y de tratar.
- Por otro, estamos extrayendo conocimiento de por qué las políticas son las que son, estamos obteniendo esos patrones de decisión y cuáles son los valores de los atributos de los que depende esa política. Así que estamos extrayendo el conocimiento final del esquema de la toma de decisiones.

2.3.2.2 Vecindarios

Derivado de la reflexión anterior, surge ahora el concepto de vecindarios. No todas las posiciones son iguales, y algunos atributos están mejor colocados en una posición dependiendo de su influencia e importancia. De forma adicional, puede haber atributos que, para una decisión dada, compartan información mutua [15] y se complementen. Este caso no ha sido estudiado, por lo que queda

abierto para trabajos posteriores, pero es posible que dados dos atributos de este tipo, tengan un peso similar, por lo que sabemos que irán contiguos en la base ofreciéndonos una facilidad extra para ordenarla.

De esta forma, al cambiar el orden de la base estamos alterando el orden de las entradas, moviendo pequeños bloques de filas de un sitio a otro. Sin embargo, aquí entran en escena varias consideraciones. La primera y más evidente es que cuantos más atributos cambiemos más grande será el cambio en la lista.

Debido a esto, una primera aproximación sería utilizar la distancia de Hamming [16] (H), la cual determina la diferencia que hay entre un vector y otro en base a los elementos cambiados. Esto significa que, si intercambiamos el orden de dos atributos de una base B_0, la nueva base B_1 estará a una distancia de Hamming 2, y si hubiésemos reordenado 4 elementos, la distancia H sería 4.

Sin embargo, este enfoque es incompleto. Como el peso de los atributos en la tabla de decisiones no es la misma, se puede intuir que no es lo mismo cambiar dos atributos de mucho peso que dos atributos de poco peso, si no que tendrá distinto efecto. Con esto, se pueden definir cinco tipos distintos de cambio de base:

1. Cambiar atributos de mucho peso. Esto tiene el efecto de cambiar grandes bloques de información, los cuales se cambian a cualquier otro lugar de la lista, son cambios globales. De esta forma, este tipo de cambios son útiles cuando tenemos una lista con pocos ítems y nos estamos acercando a la meta, ya que, si todavía hay muchos ítems, en cada bloque que estés cambiando va a haber todavía muchos ítems mezclados, por lo que el cambio de base no será tan eficaz.
2. Cambiar atributos de poco peso. Este es el cambio contrario al anterior, con lo que conseguimos cambiar pequeños bloques de información a posiciones cercanas. Es muy útil cuando la tabla todavía es muy dispersa, ya que nos permite buscar ítems más grandes de forma local y no global.
3. Cambiar atributos de peso medio. Es un pequeño solapamiento cuyo efecto es cambiar bloques de tamaño medio de forma más o menos global a lo largo de la lista.
4. Cambio de atributos mezclando atributos de mucho peso y de poco peso. Esto supone cambios impredecibles que transforman la lista por completo. Son cambios caóticos que suponen un buen comienzo para tratar el problema, ya que, en el punto inicial, es muy fácil obtener una base mejor de forma pseudo-aleatoria.

Podemos ver por tanto que empieza a surgir una idea de estrategia para optimizar la búsqueda, en la cual primero haremos cambios radicales que alteren la lista por completo hasta encontrar una base más apropiada, a la cual empezaremos a aplicarle el cambio 3, intentando hacer pequeños ítems en zonas reducidas, los cuales iremos haciendo más grandes con el cambio número

2, que al reordenarlos de forma global con el cambio 1 nos permitirá tener una KBM2L final y reducida.

Para seguir esta estrategia, todavía faltan por definir algunos conceptos, como:

- ¿Qué consideramos atributos de mucho peso y cuándo se convierten en atributos de menos peso?
- Todos los atributos tienen distinto peso, por lo que no es lo mismo cambiar dos atributos de mucho peso en particular que otros dos que también tengan mucho peso.

Para solucionar estos problemas, se propone una nueva medida de distancia, que combina la distancia de Hamming con estos conceptos, y la llamaremos distancia G. Esta G se calcula mediante la siguiente fórmula:

$$G(B, B') = r_{left} + \left(\sum_{i=0}^{a} r_i - H\left(B, \dot{B'}\right) - 1 \right)$$

Ecuación 2 Distancia G

Siendo:

- r_{left} : El peso del atributo (i) más importante de aquellos que se cambian de sitio . Esta se medirá en una escala numérica como el número de atributos a la izquierda de dicho atributo, por lo que el atributo de más peso de la base tendrá el valor 0.
- r_i : La distancia que se mueve un atributo *i*, medida como el número de atributos intermedios entre la posición final y la inicial. Vemos que al haber un sumatorio, estamos calculando la distancia que se mueve todos los atributos que cambian el orden. A modo de ejemplo, si un atributo *j* estaba en la posición 2 en la base B, y en la base B' se encuentra en la posición 5, r_j=2, ya que en medio se encuentran dos atributos (el 3 y el 4).
- H(B,B'): distancia de Hamming de las bases inicial y final.
- El -1 es un factor de corrección para que al hacer este cálculo obtengamos que la distancia G entre una misma base y ella misma sea 0, que es lo que tendría que pasar ya que es la misma base.

Al incluir la distancia de Hamming estamos aumentando de forma lineal la distancia G en base a esa distancia, por lo que está directamente influida. Además, estamos teniendo en cuenta dos cosas más: el peso del atributo más importante, y la distancia que recorre cada atributo. La segunda medida nos permite dar un valor numérico a la distancia que van a recorrer los bloques de forma local que se moverán en la lista, siendo más grande cuanto más se vayan a desplazar. Lo que nos indica la segunda medida es cuánto de global va a ser el cambio más grande, ya que este está determinado por los atributos de más

peso que se cambie, sin embargo, esta medida va al revés, es menor cuanto más peso sea el atributo.

De esto podemos deducir que cuanto más grande sea la distancia, más bloques se van a mover (y por tanto más pequeños), y además estos bloques se van a mover de forma local, y conforme se vaya reduciendo la distancia estos bloques serán más grandes y se moverán de forma global a lo largo de toda la lista. En el caso extremo, G(B,B)=0, habrá un solo bloque, que se cambiará de forma global consigo mismo, quedando la misma lista.

Para ver el resultado, vamos a tomar un ejemplo. Supongamos dos bases, B_0=[0,1,2,3,4] y B_1=[0,3,2,4,1]. De forma inmediata podemos ver que hemos alterado 3 atributos, por lo que $H(B_0,B_1)$ es 3, y que el atributo de mayor peso tienen un "peso" de 1, por lo que r_{left}=1. Ahora vamos a calcular la distancia viajada de cada atributo.

- $r_{3=}1$: hay un atributo entre la posición inicial y final
- $r_{1=}2$: hay dos atributo entre la posición inicial y final
- $r_{4=}0$: se ha movido a una posición contigua

Si quisiésemos calcular esta distancia del resto de atributos también nos daría 0, puesto que no se han movido. Así, ya podríamos calcular la distancia G, siendo:

$G(B_0,B_1)$= 1+ (1+2+0+3-1) = 4

Este es un valor relativo a la misma tabla, ya que cuanto mayor sea la tabla mayores valores se podrán alcanzar (se pueden cambiar más atributos y los de menor importancia alcanzarán valores de r más altos), por lo que este resultado no nos dice nada como tal, sino que habría que compararlos con otros cambios de base, para ver si una tiene mayor distancia G o menos, pero por sí mismo no indica prácticamente nada. Sin embargo, comparándolos con otros podemos establecer estos vecindarios que buscábamos, ya que podemos asumir que dos cambios de base que tengan el mismo valor G van a mover la misma cantidad de información a sitios parecidos.

Con este enfoque estamos complementando la estrategia previamente comentada, ya que en los comienzos del problema intentaremos hacer cambios de base con uno distancia G elevada (en relación a otros cambios posibles), mientras que según avancemos acercándonos a la solución iremos haciendo cambios de distancia más reducida.

Este concepto permite introducir los llamados vecindarios, y es que, dada una base inicial, se pueden agrupar sus permutaciones en grupos basados en la distancia G a esta base inicial. Así, depende de la acción que tengamos que hacer (acciones de cambio de base), podemos explorar las bases pertenecientes al vecindario adecuado.

Como podemos intuir, esto está fuertemente ligado al algoritmo de optimización explicado llamado búsqueda de entorno variable, ya que por fin tenemos una forma de definir estos vecindarios. De esta forma, podemos hacer una lista de vecindarios de más distancia G a menos y recorrerlos en ese orden.

2.3.2.3 Algoritmos genéticos

De igual forma que en otros problemas de optimización combinatoria [17], es posible aplicar los algoritmos genéticos a este caso concreto.

En el caso de las KBM2L, tenemos un espacio de búsqueda en el que cada solución está representada también por genes, los atributos de cada base. Así, el objetivo del algoritmo genético será ir combinando o mutando los atributos de aquellas bases que tengan mejores resultados, hasta que no se encuentre más mejora.

Así, el artículo describe una operación para hacer combinaciones entre dos ancestros para crear un hijo, donde para elegir qué atributos de cada uno escoger, se valora tanto el *fitness* del ancestro como la posición de importancia de cada atributo.

Sin embargo, utilizar algoritmos genéticos en las listas KBM2L tiene un problema fundamental, y es la dificultad de elegir una población inicial. Debido a que el espacio de soluciones es enorme y que buscamos un algoritmo genérico para tratar cualquier problema, obtener una población inicial representativa supone una tarea demasiado compleja, por lo que no se tratará más esta aproximación.

2.3.2.4 Cambio rápido de base

Uno de los problemas a los que nos vamos a enfrentar en el trabajo es cambiar de base la KBM2L, debido a que cada vez que hacemos un cambio, hay que reordenar todas las filas de la lista, y cuando el número de elementos es elevado, esto se vuelve una tarea larga.

Como durante el proceso de optimizar la lista vamos a necesitar una gran cantidad de cambios de base, surge la necesidad de elaborar un método más rápido de hacer el cambio. Así se desarrolla la llamada *fast copy* o copia rápida.

Cuando se cambian atributos de peso elevado y se mantienen los atributos de poco peso, recordemos que se mueven grandes bloques, aunque no todos por norma general. De esta forma, la idea es no crear la nueva lista desde 0, sino copiar la tabla anterior, y solo re-calcular aquellos bloques que se han movido, los cuales pueden habitualmente moverse como bloques. Veamos un ejemplo ilustrado.

[*ABCD*] Initial	Offset [*ABCD*]	Policy		[*BACD*] New	Offset [*ABCD*]	Policy
0000	0	d_0	*	0000	0	d_0
0001	1	d_0		0001	1	d_0
0010	2	d_0		0010	2	d_0
0011	3	d_0		0011	3	d_0
0100	4	d_0		0100	8	d_0
0101	5	d_1		0101	9	d_0
0110	6	d_1		0110	10	d_0
0111	7	d_2		0111	11	d_0
1000	8	d_0		1000	4	d_0
1001	9	d_0		1001	5	d_1
1010	10	d_0		1010	6	d_1
1011	11	d_0		1011	7	d_2
1100	12	d_2	*	1100	12	d_2
1101	13	d_2		1101	13	d_2
1110	14	d_2		1110	14	d_2
1111	15	d_2		1111	15	d_2

Ilustración 8 Fast copy

En la tabla anterior, se está cambiando la base [A,B,C,D] por [B,A,C,D]. Así, a la izquierda de la tabla podemos ver los índices iniciales, los offset iniciales, y la política de cada caso. En la zona de la derecha se observan los nuevos índices, los offset correspondientes a la lista anterior, y la política de cada caso.

Vemos que, al cambiar dos atributos de mucha importancia, solo estamos moviendo dos bloques entre sí, los offset del 4 al 7 y los del 8 al 11, mientras que se mantienen los otros dos (marcados con *)

Esto permite el desarrollo de un algoritmo que cambiase de base de forma muy rápida, que tendría que hacer lo siguiente:

- Clonar la lista en la nueva.
- Calcular los nuevos offset de los elementos.
- Mover los bloques necesarios.

De esta forma, aunque el nuevo programa tendría que seguir calculando los nuevos offset, no tendría que reordenar toda la tabla, sino solo una parte, la cual además estará constituida por bloques que pueden moverse a la vez, reduciendo en gran medida el tiempo que se tarda en la creación de una nueva lista ante el cambio de base.

2.3.2.5 Testeo de una nueva base

De forma similar al caso anterior, durante el proceso de búsqueda de una base óptima, es necesario comprobar si una nueva base es mejor que la anterior, para ver si la reemplazamos o no una cantidad ingente de veces. Recordemos que el espacio de búsqueda es inmenso, por lo que, aunque solo exploremos una diminuta porción, esto sigue siendo un gran número de cambios.

La forma más fácil es cambiar la lista con la nueva base, y comprobar los ítems de los que dispone, que al compararlos con la base anterior podemos ver si es mejor o no. En esto entra el problema descrito en el apartado anterior, y es que cada cambio de base y reordenación de la lista supone mucho trabajo, por lo que no es una medida eficiente para tablas de gran tamaño.

El artículo propone utilizar los supremos e ínfimos de los ítems, ya que estos determinan la longitud y los extremos de cada ítem. Así, se propone eliminar el resto de las entradas de la tabla y utilizar solo los elementos supremos, y ordenar estas tablas conforme a las nuevas bases.

Así, cuando un supremo tenga una política distinta al anterior, lo llamaremos *switches* en vez de ítems, ya que no pueden ser iguales al ser una tabla reducida.

De esta forma, los *switches* representan una cota mínima en los ítems que existirán, por lo que, aunque no sea la información completa, si puede servir como algo aproximado, por lo que se propone descartar aquellas bases con un número de *switches* mayor que la actual.

2.3.3 Infraestructura

Para manejar estas estructuras, existe desarrollado un código base que nos permite almacenar y manejar estas KBM2L, así como cambiar su base, reordenándose la lista de forma automática. El código existente es bastante extenso, por lo que aquí vamos a explicar únicamente el funcionamiento básico y necesario para el desarrollo de este trabajo.

Para ello, se definen las KBM2L como una clase S4 de R [18], en la cual se definen varios atributos, entre ellos:

- Lista de Respuestas: Lista que contienen las columnas de decisión. Como en un problema pueden tener que tomarse distintas decisiones independientes, cada una estará representada por un atributo de decisión distinto. Si hubiese dos decisiones que tomar (D1 y D2), esta lista sería (D1, D2).
- Cardinales de la lista de respuesta: Indica el número de alternativas de cada atributo de respuesta, es decir, su dominio. Así, en el caso de que haya dos decisiones binarias, en este valor se almacenará el siguiente vector: (2,2). Este indica que los dos atributos de decisión contienen dos valores.

- Dominio de la lista de respuestas: Las posibles alternativas que pueden tomar cada una de las decisiones.
- Lista de atributos: Contiene la lista de atributos, y se almacena de forma análoga que la lista de respuestas, mediante un vector que lista las etiquetas de los distintos atributos.
- Cardinales de la lista de atributos: De la misma forma que en el caso de las respuestas, indica el número de posibilidades puede tomar cada atributo como valor.
- Dominio de la lista de atributos: No indica el número de posibilidades que pueden tomar los atributos, sino que almacena una lista con los valores que se pueden tomar.
- Peso de atributos: Como se mencionó antes, para calcular el *offset* de un elemento (su posición en la lista) hace falta un vector que asigne un peso a cada atributo. Este es el elemento que lo contiene en la infraestructura.
- Base: La base en la que está la lista actualmente. Define el orden en el que están colocados los atributos de la lista de atributos.
- Offset: Es la KBM2L en sí, sin las columnas de decisiones. Esto es la lista, ya que contiene un vector con los offset de los supremos de cada ítem, por lo que es la lista ya comprimida y compacta.
- Index: Igual que el apartado anterior, pero contiene los índices en lugar de los offset, para poder representar la lista de otra forma.
- Respuesta: Las políticas de decisiones asociadas a cada entrada de la tabla. Con esta lista y con la de offset, tenemos la KBM2L a la que estamos acostumbrados.

Con todos estos elementos, ya tenemos todo lo necesario para representar las KBM2L. Además de esta infraestructura, hay creadas múltiples funciones auxiliares que nos ayudan a manejarlas:

- Creación de KBM2L: Recibe una tabla de decisiones (en forma de matriz), y la convierte en KBM2L. Esta viene habitualmente de un fichero con extensión *csv*, aunque también podemos utilizar un dataframe de R para su creación. Para crearlo, es necesario indicar el número de columnas de los atributos y el número de columnas de las decisiones, a partir de lo cual la función se encarga de obtener los dominios y de agrupar la lista en ítems en función de la base actual, es decir, el orden en el que vienen inicialmente los atributos. Durante este proceso, hace varias comprobaciones para detectar posibles errores en la tabla de decisiones, como entradas duplicadas, por ejemplo.
- Organización de la tabla: Cuando se detecta un cambio de base, la tabla se reorganiza de forma automática para mantener la integridad, agrupando las entradas en los nuevos ítems resultantes.

- Cambio de base: Originalmente la librería solo permite cambios a una base aleatoria. Esto no es muy útil, por lo que de las primeras tareas será crear un método que nos permita cambiar a una base de nuestra elección.

Aunque estas son todas las funciones de utilidad de la librería, también nos encontramos con muchos métodos de prueba, así como ejemplos, que nos permiten explorar el funcionamiento de estos métodos.

Sin embargo, también nos encontramos algunas carencias. La mayor de ellas y que se detectó al principio del proyecto es que no existe un método para hacer un cambio de base de un KBM hacia una nueva base definida. Sí que existe una función (*swap.base.kbm*) que realiza el cambio de base una KBM, pero recibe como argumentos dos objetos KBM, el primero el original, y el segundo es un objeto vacío, cuyos dominios de los atributos y respuestas son clonados, pero sin la necesidad de todas las entradas de la tabla. Es este segundo objeto el que contiene la base a la que se quiere hacer el cambio. Por tanto, la forma más fácil es conseguir esta lista vacía, el problema es que la infraestructura solo proporciona las herramientas para clonarla tal cual (sin el cambio de base), o con una base aleatoria, por lo que necesitaremos definir nuestro propio método.

Para crear esta función vamos a apoyarnos en otro de los métodos definidos por la infraestructura, denominado *empty.kbm*, cuya función es clonar una KBM2L pasada como argumento, pero sin la información de las entradas, con lo cual es una creación mucho más rápida.

El problema de esta función es que también clona la base, y esta no puede cambiarse solo en el atributo de la base, sino que hay que cambiarlo en el resto de los atributos, ya que hay varias propiedades de las KBM2L que dependen de la base, ya que la información se almacena ordenada en base a este orden de los atributos. Estas propiedades son algunas como el cardinal de los atributos, el dominio de los atributos, la propia base, y también la lista de los pesos, ya que como no todos los atributos tienen el mismo dominio, al modificar la base esta también se verá alterada.

Por tanto, la primera función que se ha desarrollado es la llamada *custom.base* (cuyo código se encuentra en el anexo "7.1 Código de custom.base"). Esta recibe dos parámetros como argumento: la KBM original, y un array con la nueva base. Esta función hace una llamada a *empty.kbm* con el primero de los argumentos, creando el objeto vacío, para después reordenar todas las propiedades necesarias para hacer el cambio de base de forma efectivo.

Con esto, ya tenemos todo lo necesario para hacer un cambio de base, por lo que, durante todo el proyecto, cada vez que se quiera hacer un cambio de base ya sea para testear la longitud de la nueva lista, o porque se ha encontrado una nueva solución, se seguirán estos dos pasos:

- Invocaremos a *custom.base*, con lo cual crearemos una KBM clonada de la actual, pero con las entradas vacías y con la nueva base.
- Llamaremos a la función *swap.base.*kbm, proporcionada por la infraestructura, a la que pasaremos como parámetro la lista actual y el nuevo objeto vacío con la base deseada.

Como vemos, es una infraestructura muy extensa que nos permitirá partir con una base sólida para la creación del algoritmo de optimización de las listas.

3 Desarrollo

Une vez comprendido el concepto de las KBM2L, así como heurísticas de optimización combinatoria, queda el grueso del trabajo, que es la implementación de un software el cuál aplique estas heurísticas a listas no optimizadas para encontrar las mejores bases que permitan reducir su tamaño.

Para ello, se van a elegir distintas metaheurísticas para probar su eficiencia y efectividad, y comprobar cuáles obtienen mejores resultados.

De todas las opciones disponibles se han elegido dos: la búsqueda de entorno variable y el algoritmo de recocido simulado. Las metaheurísticas de población (en concreto los algoritmos genéticos) han sido descartados porque, aunque son métodos prometedores, tienen el problema de tener que elegir una población inicial, que como hemos explicado no es fácil para lo que queremos obtener, ya que el objetivo es hacer un algoritmo genético, mientras que la población inicial habría que elegirla de una forma distinta dependiendo del problema representado. Además, este tipo de algoritmos requiere de un conocimiento extenso del problema que se quiere solucionar, no siendo este el caso.

De forma distinta sucede con el recocido simulado y la búsqueda de entorno variable. Estos dos métodos no requieren de un conocimiento previo tan elevado del problema ni la topología del mismo, y suelen tener buenos resultados aplicados a problemas con gran cantidad de posibles soluciones, por lo cual los convierte en idóneos para nuestro trabajo.

Así pues, el objetivo del trabajo será implementar y adaptar estas heurísticas para que puedan trabajar en nuestro problema con la infraestructura proporcionada.

Una vez desarrollados, se harán pequeñas variantes de cada algoritmo ya que estos tienen distintas formas de ejecutarse. Cuando tengamos todos los algoritmos y sus variantes implementados, se les ejecutarán pruebas de rendimiento para comprobar la viabilidad de estas, y comprobar cuáles son mejores en cada caso, ya que también es posible que unos funcionen mejor en determinadas circunstancias, por lo que será necesario un pequeño análisis de los resultados.

3.1 Búsqueda de entorno variable

La metaheurística de la búsqueda de entorno variable (o VNS) es una gran alternativa a la hora de recorrer espacios de búsqueda enormes como a los que nos enfrentamos. Además de eso, y como ya hemos visto en capítulos previos, los cambios de base de los KBM2L guardan mucha similitud con el concepto de los vecindarios de esta heurística, ya que la distancia entre la base original y la

que se está haciendo el cambio, define de alguna forma el tipo de modificación que va a sufrir la lista, siendo cada tipo más apropiado para distintos momentos.

Por tanto, los cambios de base que pueden realizarse desde una base inicial pueden ser agrupados según el tipo de alteración que van a provocar, es decir, se pueden agrupar mediante la distancia G. Así, estas agrupaciones guardan mucha similitud con los vecindarios de esta heurística, los cuales van recorriéndose en orden.

Es por estas razones por las que vamos a implementar esta heurística como primera solución, ya que parece la más apropiada para tratar nuestro problema, además de que las similitudes hacen que parezca factible adaptarlo. Por lo que vamos a coger el funcionamiento de la metaheurística general, y adaptarla para que se pueda aplicar en nuestro trabajo.

Recordemos brevemente los pasos del algoritmo:

- Se definen las estructuras de vecindario que fragmentan el espacio de búsqueda.
- Se recorren en orden, y cada vez que se encuentra una mejor solución, se vuelve al primer vecindario.
- Si recorremos todos los vecindarios sin conseguir una mejora, se termina el proceso.

Podemos observar que el punto clave de este método es el concepto de vecindarios. Así, para no explorar el inmenso espacio de búsqueda entero, esta heurística divide este espacio en vecindarios, los cuales son relativos a una solución en particular, ya que, si un vecindario para las bases se define como aquellas bases a distancia de Hamming 2 respecto a la actual, no serán los mismos vecindarios para todas las bases.

La idea de los vecindarios es que el mínimo global del problema será el mínimo global de todos los distintos vecindarios de todas las bases. Como recorrer todos los vecindarios de todas las bases es inviable, ya que es lo mismo que recorrer todo el espacio de búsqueda, cuando recorramos todos los vecindarios de una misma base sin encontrar ninguna mejora, supondremos que es mínimo global.

Esto no tiene por qué ser así, ya que falta mucho espacio de búsqueda por recorrer, pero recordemos que estos métodos buscan soluciones aproximadas en un tiempo finito.

Así pues, lo primero será definir los vecindarios. Como hemos visto en los trabajos previos, la distancia entre dos bases parece una medida fundamental, en concreto la distancia G. Esta distancia no solo nos da una medida que separa las dos bases, sino que indica cómo va a ser el cambio en la KBM2L al aplicar el cambio de base.

El problema con esta distancia es que el rango de distancias depende del número de atributos de la tabla de decisiones, por lo que no hay un valor acotado para todas las tablas en general. A modo de ejemplo, la máxima distancia G entre dos bases con 5 atributos rondará el valor 20, mientras que una base que tenga 20 atributos la máxima distancia será más de 100.

Debido a esto, hace falta definir un vecindario para que, independientemente del valor absoluto de G, separe correctamente distancias altas frente a distancias bajas para todas las listas. La idea que se propone es definir un número fijo de vecindarios, agrupando las distancias por cercanías. Si por ejemplo quisiésemos establecer solo 10 vecindarios, en una tabla cuya máxima distancia G entre dos bases fuese 10, cogeríamos el primer vecindario para todos los cambios de base a distancia 10, el siguiente para los cambios de base a distancia 9, luego 8, etc.

Como ya se ha explicado, los primeros vecindarios que se recorran deberán ser aquellos con una distancia G más elevada.

Si quisiésemos establecer estos mismos 10 vecindarios en una tabla cuya máxima G entre dos bases fuese 20, el primer vecindario contendría todos los cambios de base cuya distancia fuese 20 o 19, el siguiente sería 18 y 17, etc.

Otra forma de verlo es que estamos separando los vecindarios por porcentajes. Así, con 10 vecindarios, asignamos todos los cambios de base que supongan una distancia del 90% al 100% de la máxima G posible al primer vecindario, y el segundo tendría del 80% al 90%. Si en vez de 10 vecindarios quisiésemos solo 5, estaríamos organizándolos en bloques del 20%.

Así, cuantos más bloques tengamos, más pequeños serán estos y más rápido se recorrerán, con la diferencia de que habrá que recorrer más vecindarios más veces, ya que no solo hay más vecindarios que recorrer hasta el punto de parada, sino que, además, con vecindarios más grandes en cada iteración, aunque más larga, estamos recorriendo mucho más espacio de búsqueda.

Otro punto a tener en cuenta es que, con vecindarios pequeños, podemos ver mejor la trayectoria que se está tomando en la búsqueda de base, ya que hay más cambios de base, mientras que con vecindarios más grandes hay menos, ya que antes de realizar un cambio se explora un espacio mucho más grande.

Por tanto, se harán pruebas para encontrar los tamaños óptimos, aunque estos pueden variar en función del tamaño de la tabla, ya que es posible que tablas más grandes necesiten vecindarios más pequeños, ya que el tamaño de estos vecindarios escala en gran medida con el número de atributos, por lo que necesitamos coger cuadrantes de búsqueda de un tamaño manejable.

El otro problema de estos vecindarios es que, aunque estamos reduciendo bastante el espacio de búsqueda al organizarlo en estos bloques, estos bloques siguen siendo enormes y contienen muchos cambios de base que, aunque

tengan la misma distancia, pueden no representar los mismos cambios, ya que una G alta podría indicar un cambio de base en el que se intercambian pocos atributos de poco peso, o un cambio en el que se altera el orden de muchos atributos de mucho peso.

Por tanto, aunque esta distancia da una pequeña aproximación de lo que buscamos no es suficiente. La opción que se ha elegido finalmente es considerar solo aquellos cambios de base con distancia de Hamming 2, es decir, solo se pueden hacer cambios de base donde se intercambien dos elementos. Con esto obtenemos dos beneficios:

Solucionamos el problema recién comentado, y es que la distancia G no representa por completo el cambio que vamos a generar en la lista. Al solo cambiar dos atributos siempre, estos cambios sí que están reflejados en la distancia G, ya que al quitarnos de la fórmula la variable del número de atributos que estamos cambiando, solo estamos considerando el peso de los dos atributos a intercambiar, con lo que definimos el tipo de cambio de igual forma para distancias G parecidas.

Nos permite un mejor seguimiento de la progresión del algoritmo. De la misma forma que con un tamaño pequeño de vecindarios, el reducir la posibilidad de cambio de base a solo dos atributos, nos obliga a hacer muchos más cambios de base hasta encontrar aquella que sea óptima. Por otro lado, estos vecindarios se hacen mucho más pequeños (menos posibilidades) por lo que se recorren más rápido, además de que podemos seguir más fácil la trayectoria que toma la base, ya que sin limitar la distancia H, el cambio de base en los primeros pasos podría estar alterando la posición de todos los atributos, lo que haría que fuese muy difícil monitorizar el proceso.

Aunque falta por concretar el tamaño (y por tanto el número) de los vecindarios, ya tenemos definida la estructura de estos, por lo que ya podemos empezar a concretar el algoritmo. Este puede definirse en un pequeño número de fases:

1. *Recibe una KBM2L no optimizada y el número de vecindarios que quieren utilizarse*. Aunque hay un número de vecindarios definido por defecto, el usuario puede introducir los vecindarios que desee. Esto permite cierta flexibilidad para poder variar los resultados en función de si el KBM2L recibido es muy grande, en cuyo caso puede ser útil poder utilizar un número de vecindarios mayor, y viceversa.
2. *Explora el vecindario actual devolviendo la mejor base*. Recordemos que estos vecindarios se exploran en orden. En nuestro caso, se exploran primero los vecindarios con una distancia G más elevada, por lo que, si el algoritmo acaba de empezar a ejecutarse, se explorará aquél con los mayores G respecto a la base actual.
3. *Comprobamos si se ha producido mejora*. Se comprueba si con la nueva base, se consigue una lista mejor o peor, es decir, la nueva KBM2L con

la lista obtenida del paso anterior proporciona un número mayor o menor de ítems. Depende del resultado se procederá de una forma o de otra.

   a. Se ha producido una mejora: Si la nueva base es mejor que la anterior, se pasa a sustituirla y a calcular el KBM2L resultante. Con esta nueva lista volveremos al paso 2 con el primer vecindario de todos. Como hemos cambiado de lista completamente, no podemos seguir explorando los siguientes vecindarios, ya que hay más probabilidades de que podamos obtener mejores oportunidades con los cambios de base con G altos, ya que nunca se puede saber cuánto de lejos estamos de la solución. Por lo que cada vez que consigamos una mejora, volveremos al primer vecindario con la nueva lista.
   b. No se ha producido mejora. Que no encontremos una base mejor en un vecindario es buena noticia, significa que nuestro KBM2L es mínimo local del vecindario, con lo que es posible que estemos más cerca del mínimo global. Recordemos además que el mínimo global es a su vez mínimo local de todos los vecindarios, por lo que el objetivo es encontrar aquella base que no pueda mejorarse en ningún vecindario. Por tanto, repetiremos el paso dos con la lista y la base actual en el siguiente vecindario, es decir, el vecindario más cercano con una distancia G menor. Si no quedan más vecindarios por explorar, significa que hemos encontrado el mínimo aproximado de la función, por lo que es el punto de parada, y la solución actual se considera la solución al problema.

4. Adicionalmente, se podrá introducir un número máximo de iteraciones para evitar esperas no acotadas, aunque este se puede variar, ya que una tabla con más atributos necesitará realizar un mayor número de iteraciones. De esta forma, esto es solo un mecanismo de control, ya que además con este algoritmo no pueden producirse bucles infinitos, ya que solo nos quedamos con soluciones mejores, no equivalentes, por lo que lo que se espera es que el algoritmo se acabe con el punto de parada del paso tres en el que no se encuentran nuevas mejoras, y no con este.

Vemos que es una búsqueda de entorno variable muy clásica. A partir de este esqueleto vamos a elaborar unas pocas versiones distintas del algoritmo, ya que algunos de estos pasos permiten cierta flexibilidad, por lo que comprobaremos cuáles son aquellas variantes que nos aportan unos mejores resultados.

3.1.1 Búsqueda de entorno variable básico

Ya hemos definido los conceptos necesarios para la adaptación de este algoritmo a nuestro problema, por lo que ahora es el momento de implementarlo.

Como primera variante, utilizaremos una variante básica, la cual destaca por la exploración de los vecindarios en su totalidad. Como primer enfoque, todavía no está claro los resultados que pueden obtenerse ni el tiempo que puede durar resolver este tipo de problemas, así que esta primera aproximación va a hacer una búsqueda en profundidad.

Vamos a ver el código entero para poder explicarlo:

```
vns.rafa<-function(rafa.kbm,proporcionIni=0.8, resta=0.2){
  ## inicializamos la base y la solucion
  proporcion=proporcionIni
  base=rafa.kbm@base
  bestkbm<-rafa.kbm
  for (i in 1:200){

  #funcion que obtiene la mejor base posible del vecindario proporcionado
    bestBase=getBestSwap(bestkbm, proporcion, resta)

    ##si no mejora, reducimos vecindario
    if(all(bestBase==base)){
      proporcion=proporcion-resta
    }

    #si mejora, actualizamos la solucion y volvemos al primer vecindario
    else{
      base=bestBase
      apoyokbm<-custom.base(bestkbm, base)
      bestkbm<-swap.base.kbm(bestkbm,apoyokbm)

      proporcion=proporcionIni
    }

    # sin no quedan más vecindarios, finalizar
    if (proporcion<0){
      break
    }
  }
  return(bestkbm)
  }
```

Ilustración 9 Código del algoritmo de búsqueda de entorno variable

Podemos ver que es una función definida en R, llamada vns.rafa, y debe su nombre al autor y a la traducción de la heurística en inglés (*variable neighborhood search*). Vamos a analizar los pasos de los que se compone.

3.1.1.1.1 Argumentos y vecindarios

Esta función recibe tres argumentos:

- *Un objeto de tipo* KBM2L. Este es el único parámetro obligatorio, y representa la lista que quiere optimizarse. En el caso del código, este parámetro se llama rafa.kbm.
- *Dos parámetros para definir el número y tamaño de los vecindarios*. Estos dos parámetros, llamados *proporcionIni* y *resta* son opcionales, y aunque el usuario puede introducir los valores que considere, se ofrecen dos por defecto.

Para definir los vecindarios, hemos elegido el sistema de porcentajes explicado en el punto anterior. Así, el valor de *proporcionIni* representa una proporción o porcentaje, el cual determina el punto donde empieza el primer vecindario. Este punto se expresa como un porcentaje del valor de las distancias G posibles. Así, recibir un 0.8 (que en este caso es el valor por defecto), significa que el primer vecindario contiene a las bases cuya distancia G a la base actual sea mayor al 80% del máximo de las distancias G posibles desde la base actual con distancia de Hamming 2.

Por otra parte, la variable *resta* indica el tamaño de los vecindarios también en porcentaje. De esta forma, si los valores son 0.8 para *proporcionIni* y 0.2 para *resta*, el primer vecindario iría del 80% al 100%, el segundo del 60% al 80%, etc.

Podemos observar que con las dos variables estamos añadiendo información redundante, ya que únicamente con el valor de la primera variable, podríamos establecer el tamaño del vecindario como la resta entre 1 y ese valor. Sin embargo, al añadir la segunda variable, estamos permitiendo al usuario cierta flexibilidad, donde puede hacer que el primero y el último vecindario sean más pequeños que el resto, ya que, si el usuario o introduce un valor en *resta* tal que la suma de ese valor más la suma de *proporcionIni* es mayor que uno, estamos haciendo ese vecindario menor que el resto, ya que no hay bases con distancia G mayor al 100%, que es el máximo.

De igual forma, si mediante sucesivos decrementos de *resta* al valor de *proporcionIni* no podemos llegar al valor 0 de forma exacta, significará que el último vecindario será también más pequeño, ya que este vecindario tendrá parte del rango menor de 0, y no existen bases que cumplan esa condición.

Podemos ver el funcionamiento de los vecindarios mediante un ejemplo. Imaginemos una KBM2L de 5 atributos, con base inicial B_0=[0,1,2,3,4]. Como ya hemos dicho, los cambios solo pueden realizarse a bases cuya distancia de Hamming sea 2, es decir, solo podemos intercambiar dos atributos, por lo que vamos a calcular todas las posibilidades antes de separar estas bases en vecindarios.

Los cambios de base posibles desde B_0 son a las bases:

[1,0,2,3,4], [2,1,0,3,4], [3,1,2,0,4], [4,1,2,3,0], [0,2,1,3,4], [0,3,2,1,4], [0,4,2,3,1,], [0,1,3,2,4], [0,1,4,3,2], [0,1,2,4,3]

Vemos que en este caso hay 10 cambios posibles, no son muchos por lo que se podrían explorar en su totalidad sin necesidad de separarlos en vecindarios, sin embargo, la cantidad de posibilidades crece de forma factorial, por lo que cuando aumentemos el número de atributos la separación en vecindarios será importante. Así, con 10 atributos tendremos 45 combinaciones, con 20 tendremos 190. De forma adicional, hay que tener en cuenta que además este proceso se repite muchas veces, por lo que cualquier pequeña optimización ahorrará mucho tiempo de cálculo.

Ahora que tenemos las combinaciones posibles, vamos a calcular las distancias a cada una de ellas desde la base inicial. Para nombrarlas, lo haremos en el orden en el que las hemos descrito en la lista, por lo que B_1 será [1,0,2,3,4], B_2 será [2,1,0,3,4], etc.

Base	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10
Distancia G desde B_0	1	2	6	7	2	7	6	3	5	4

Tabla 4 Distancias G a una base B0

Podemos ver que para el caso seleccionado la máxima distancia es 7. Por otra parte, la mínima distancia siempre será 1 para el caso en el que se intercambien los dos primeros atributos, o 0 si cuentas el cambio a la misma base, pero como este cambio no aporta ninguna modificación en la KBM2L no lo tendremos en cuenta.

Vamos a suponer también que los valores recibidos para los valores *proporcionIni* y *resta* son 0.75 y 0.25 respectivamente. Lo que estamos consiguiendo con estos valores, es separar el espacio de búsqueda en 4 fragmentos "del mismo tamaño". Aunque la porción del espacio que contienen es la misma, estos vecindarios no tienen por qué contener necesariamente el mismo número de bases, ya que podemos ver que la distribución de las distancias G no es uniforme, ya que, aunque la distancia máxima es 7, podemos ver que hay 4 valores por encima de 5. Es decir, que las bases con distancia 6 o 7 suponen una representación del 40%, mientras que para que los vecindarios estuviesen equilibrados tendrían que suponer 2/7 del total, o un 29%.

De esta forma, vamos a organizar las bases en vecindarios. Para ello lo primero que hará falta es calcular el 75%, el 50% y el 25% de la máxima distancia, que es 7. Así, este cálculo devuelve los valores de 5.25 , 3.5 , y 1.75 respectivamente.

Estas son las fronteras de decisión, por lo que es trivial separar las bases en los vecindarios en base a G, quedando de la siguiente forma:

- **Vecindario 1**: Todas las bases cuya distancia G esté comprendida entre 5.25 y 7. Lo componen B3, B4, B6 y B7.
- **Vecindario 2**: Todas las bases cuya distancia G esté comprendida entre 3.5 y 5.25. Lo componen B9 y B10.
- **Vecindario 3**: Todas las bases cuya distancia G esté comprendida entre 1.75 y 3.5. Lo componen B2, B5 y B8.
- **Vecindario 3**: Todas las bases cuya distancia G sea menor de 1.75. A este vecindario solo pertenece la base B1.

Así sería la separación en vecindarios para este caso concreto. Como ya habíamos indicado, aunque los vecindarios contienen la misma proporción de la distancia G, no se traduce en vecindarios del mismo tamaño. Esto se debe a dos razones fundamentales:

- Hay más cambios de base con un valor de G elevado que con valor bajo. Debido a la definición de la distancia G, esto se debe a que para conseguir una distancia grande tenemos dos posibilidades, o cambiar dos atributos muy alejados entre sí, es decir, cambiar atributos de poco peso con atributos de mucho peso; o intercambiar dos atributos de poco peso, ya que en la distancia G se tiene en cuenta lo alejados que están de la posición inicial estos atributos. Sin embargo, para obtener una distancia baja, necesitamos intercambiar dos atributos de mucho peso, no hay más alternativas. Por tanto, siempre habrá más cambios con una G alta.
- El último vecindario siempre tendrá menos bases que el resto. Esta razón no es tan importante como la anterior, y conforme crece el número de atributos de la lista el cambio debido a esta razón pierde peso. El último vecindario contiene las bases entre el 0% del valor máximo y el 25%, es decir, entre 0 y 1.75. Cuando lo vemos en proporción, parece que el vecindario es igual de grande que el resto, sin embargo, la mínima base siempre tendrá una G de 1, por lo que en realidad este vecindario contiene a las bases entre 1 y 1.75, es decir, solo a las que tienen un valor de 1, ya que no hay G con decimales. Sin embargo, como hemos dicho, cuando el valor de los atributos suba mucho, al alcanzar distancias máximas mucho más altas, la diferencia de tamaño entre este vecindario y el resto se irá haciendo más pequeño, ya que ese punto que recortamos no representa el mismo porcentaje cuando el vecindario alcanza hasta una distancia más alta.

De este ejemplo, podemos sacar varias conclusiones. La primera es que, aunque las porciones que salgan incluyan distancias G no enteras (que no existen), esto no afecta a los vecindarios, ya que se cogerían las bases con una G que se encuentren dentro del conjunto acotado.

La segunda conclusión es que debido a que estos vecindarios no son del mismo tamaño, pueden utilizarse distintos métodos para explorar estos vecindarios. En este ejemplo, la diferencia absoluta en el número de soluciones en cada vecindario no es excesivamente grande, ya que se ha utilizado un número reducido de atributos para la base. Sin embargo, en problemas del mundo real que haya una gran cantidad de atributos, esta diferencia de tamaño puede hacerse muy grande.

En estos casos, aunque no se ha tenido en cuenta en este trabo, se abre la posibilidad de explorar los vecindarios de forma distinta, ya que en el caso de que el vecindario sea pequeño, es posible que se pudiese explorar este vecindario en su totalidad, mientras que aquellos que sean más grandes, tendrán que utilizar heurísticas o procedimientos para explorar solo un pequeño conjunto de las soluciones.

Una vez recibimos los argumentos, las primeras líneas de código (3, 4 y 5), se hace una copia de la proporción para poder utilizarla más tarde, y se inicializa la solución base, la cual se asigna a la lista recibida.

3.1.1.1.2 Exploración del vecindario

Una vez hemos definido en profundidad las estructuras de los vecindarios y las bases que pertenecen a cada uno de ellos, es el momento de explorar su espacio de soluciones.

Para ello, se ha creado una función auxiliar denominada *getBestSwap*, incluida en el anexo "7.2 Código de getBestSwap". Como su nombre indica, este método devuelve el mejor cambio de base posible de un vecindario específico en el que se obtenga una mejora respecto a la base actual.

Este algoritmo, como es el encargado de explorar los vecindarios, recibe la información necesaria para definir este algoritmo. De esta forma, como argumentos la proporción y el tamaño del vecindario, según lo explicado en el punto anterior: si se recibe un 0.7 y un 0.15 respectivamente, significa que el vecindario irá del 70% al 85%.

Una vez recibe estos datos, inicializa la solución inicial, a la cual le asigna la actual, es decir, la lista recibida y su base. A partir de ahí, mediante un bucle, calculará todas las bases a una distancia de Hamming 2, de cada una de las cuales calcularemos la distancia G, y eliminaremos todas las que no se encuentren en el vecindario. Hay otras formas de hacer este proceso que evitaría tener que hacer los cálculos a todas las posibilidades, sin embargo, como en este caso no estamos trabajando con las KBM2L, sino con simples vectores, aunque haya muchas combinaciones (alrededor de 1000 a partir de los 20 atributos), estos cálculos son simples intercambios de elementos entre vectores, por lo que no debería afectar demasiado al rendimiento.

Una vez tenemos aquellas bases que sí pertenecen al vecindario indicado, para cada una haremos un cambio de base en la KBM2L recibida mediante las funciones *custom.base* y *swap.base.kbm*.

La función *swap.base.kbm* está definida en la infraestructura ya desarrollada, y recibe como argumento dos listas KBM, la original con lase actual, y una nueva KBM, la cual está vacía, es solo un objeto de este tipo con la nueva base con la que quiere realizarse el cambio. Así, este método coge la base de la segunda lista, y devuelve otra KBM transformando la primera lista para que tenga la base de la segunda.

El problema es que para ello necesitamos crear otro objeto R igual de tipo KBM2L, pero que esté vacío y que tenga la base que queremos utilizar. Para ello, y como se ha explicado en el apartado ”2.3.3 Infraestructura”, se ha creado una función llamada *custom.base*, que clona la lista recibida como primer argumento, la vacía, y realiza el cambio de base con la que se proporciona como segundo argumento.

Con estas herramientas ya podemos crear la nueva lista, ya ordenada según la base que queremos probar, y con la función auxiliar definida en la infraestructura denominada *length.kbm*, obtenemos el número de ítems resultantes. Este número de ítems se compara al que tenía la KBM2L original, y si el resultante es menor, se guarda en una lista, tanto el número de ítems como la base a la que se asocia ese resultado.

Este proceso se repite para todas las bases del vecindario, y al acabar todas las iteraciones, se comprobará la lista en la que hemos almacenado las bases que producían mejoras. Si esta lista está vacía, se devolverá la base recibida, es decir la base de la KBM2L original. Si no está vacía, se devolverá como mejor solución aquella con el menor número de ítems producidos.

Aquí podemos ver un gasto de memoria, ya que estamos almacenando todas las bases mejores, mientras que podríamos quedarnos solo con la que tenga el menor número de ítems, e ir sustituyendo esta si encontramos una mejor. Sin embargo, este es un método auxiliar el cuál se invoca en cada iteración, por lo que esa memoria ocupada se va eliminando cada vez, además de que se prepara este método para introducir otras variantes.

Por ejemplo, surgen algunas ideas de cómo se puede mejorar el método, y es estudiando no solo la mejor base, sino unas pocas de las mejores. Así, si hay varias bases que comparten el mismo número de ítems, podría estudiarse si hay alguna mejor que otra mediante otro método rápido, como es el caso de los *switches*. Esta alternativa no se ha estudiado, pero la infraestructura se ha realizado así para soportar otro tipo de variaciones.

Aun así, podemos observar que lo que se está almacenando son las bases (que son simples vectores), en vez de el objeto de la KBM2L, por lo que, aunque esto tiene el problema de que la mejor base habrá que convertirla de nuevo en lista KBM, esta lista ocupa mucho menos espacio que si estuviésemos almacenando todos los objetos.

Por tanto, podemos observar que esta variante destaca por la exploración completa del vecindario. Cuando se estén manejando muchos atributos, esto supondrá que se necesite una cantidad de tiempo muy elevado para obtener una solución, ya que con el número de atributos estos vecindarios adquieren un tamaño muy grande, por lo que la completa exploración de todo el subconjunto será muy costoso.

A cambio, este método debería de obtener bastantes buenos resultados ya que hace una exploración metódica y sistemática a lo largo del espacio de búsqueda.

3.1.1.1.3 Comprobación de la nueva lista

La función *getBestSwap* () devuelve a la función principal una base, aquella con los mejores resultados de todo el vecindario.

Si la solución devuelta es la misma base que teníamos, significa que el método no ha encontrado ninguna base mejor en todo el vecindario, por lo que repetiremos el paso anterior en el siguiente vecindario, para lo cual haremos una reducción en la *proporción* (línea 13 de Ilustración 9 Código del algoritmo de búsqueda de entorno variable**¡Error! No se encuentra el origen de la referencia.**).

En caso de que sí se haya devuelto una base distinta, significa que sí que ha habido una mejora, aunque no sepamos el grado de mejoría. En este caso, actualizaremos la KBM2L con la nueva base, siguiendo el mismo procedimiento de cambio de base explicado en el punto anterior. Además de actualizar la lista, será necesario volver al primer vecindario, por lo que asignaremos de nuevo el valor *proporcionIni* (la proporción introducida como primer vecindario por el usuario) a la variable *proporción*, volviéndose a ejecutar el paso Exploración del vecindario con la nueva lista en el primer vecindario.

En el caso de que la solución devuelta fuese la misma que ya teníamos, y que se hayan acabado los vecindarios, se habrá cumplido el punto de parada, y se considerará la solución actual como mínimo global del problema. Se sabe que se han acabado los vecindarios porque la *proporción* es menor de 0, por lo que ya no tiene sentido seguir bajando de vecindarios, dado que no existen.

3.1.1.1.4 Punto de parada

Como elemento adicional de control del algoritmo, se ha introducido una variable para indicar el número máximo total de vecindarios que se pueden explorar.

Aunque el usuario solo indique 5 vecindarios, como cada vez que se obtiene una solución mejor se retorna al primero, lo normal es explorar muchas veces todos los vecindarios, y no solo 5 vecindarios en total.

Por tanto, todo el proceso descrito anteriormente se ha metido en un bucle *for*, el cual está preparado para detenerse una vez cumplido. Como ya se ha comentado, esto es solo un mecanismo de control para acotar las esperas, y tiene el problema de que el método puede detenerse cuando no se han terminado de explorar todos los vecindarios.

Por ello, este valor se ha establecido inicialmente en 50, que es un número elevado, y a lo largo de las pruebas ha sido suficiente para que no se detuviese el algoritmo debido a esta condición de parada, sino a la interior, ya que es la que nos interesa porque indica que se ha llegado al mínimo global, mientras que esta no asegura haber alcanzado el mínimo, sino que solo asegura que no hemos invertido más "tiempo" del establecido como máximo.

3.1.2 Búsqueda de entorno variable truncado

Esta es una variante del algoritmo de búsqueda de entorno variable, que intenta reducir una de las fallas de la variante anterior, y es el tiempo empleado.

En la alternativa anterior, podíamos ver que se exploraba cada vecindario a fondo, valorando cada una de las posibilidades. Lo que se intenta conseguir con esta alternativa es, mediante una sencilla modificación del algoritmo, obtener un resultado parecido en un tiempo considerablemente más bajo.

Para ello nos hemos centrado en la cantidad de soluciones que se explora de cada vecindario. Por ello, se ha introducido un argumento adicional en la función *getBestSwap*, que permite al usuario introducir la probabilidad de que se pruebe una base. Así, si el usuario introduce un 0.3, cada vez que se vaya a evaluar una base del vecindario, se generará un número aleatorio entre el 1 y el 0, y si este es menor que el recibido por el usuario, se hará el cambio de base y se calcularán los ítems de la nueva lista como en los casos anteriores.

Si por el contrario este número generado es mayor, esa base no se evaluará, con lo cual, aunque seguimos habiendo hecho el cálculo para hallar esa combinación, ya hemos determinado que no se requiere mucho tiempo para generar obtener las bases de cada vecindario, por lo que no supone un gasto excesivo.

Aunque la idea no parece mala, en pruebas iniciales no ha tenido mucho acierto. Aunque sí se ha reducido el tiempo de búsqueda, también se han añadido algunos efectos negativos:

- Numerosas veces el algoritmo termina en lo que parecen mínimos locales.
- Hace falta que el usuario conozca el tamaño de los vecindarios. Para usar esta variación con buenos resultados, es mejor conocer el tamaño en

valores absolutos del número de bases pertenecientes a cada vecindario. Esto se debe a que, si se introduce un porcentaje demasiado bajo en vecindarios extremadamente grandes, podemos estar testeando un número de bases muy pequeño. Debido a esto, si no se introducen suficientes iteraciones del algoritmo, es decir, un número de vecindarios suficientemente grandes puede darse el caso de que hemos recorrido todos los vecindarios sin haber encontrado una de las bases mejores, debido al azar introducido.

De esta forma, aunque esta variación no ha dado todo el éxito que se esperaba, se va a combinar en estudios posteriores del estudio con otras herramientas, para intentar obtener mejores resultados.

3.2 Algoritmo de recocido simulado

La otra metaheurística que utilizaremos será el algoritmo de recocido simulado. Aunque al contrario que en los algoritmos evolutivos, en la búsqueda de entorno variable no hacía falta conocer nada del problema que se iba a evaluar, sin embargo, sí que había que conocer un poco de la posible topología de las soluciones, para poder separar ese espacio de búsqueda en los vecindarios. Por el contrario, en este algoritmo no es necesario ni tener conocimientos sobre el problema ni sobre la distribución del espacio de búsqueda.

Esto hace que sea una de las metaheurísticas más fáciles de implementar a problemas generales, ya que su implementación no depende de a qué problema se está aplicando.

Además de esto, esta metaheurística ha tenido mucho éxito, y se utiliza en múltiples campos de la ingeniería. Debido a esto, parece un algoritmo prometedor para nuestro trabajo, por lo que será el segundo que implementemos.

Por tanto, vamos a ver cómo serían los pasos de esta metaheurística asociada a nuestro problema:

1. *Recibe una KBM2L no optimizada*. Esta KBM2L se asigna como la solución actual, de la que usaremos la base.
2. *Realizaremos un cambio de solución al azar dentro del vecindario actual*. Con este cambio comprobaremos si se ha producido mejora o no.
    a. Se ha producido una mejora: Sustituiremos la solución actual por la recién calculada, y repetiremos este segundo paso.
    b. No se ha producido ninguna mejora: Aceptaremos esta solución como nueva solución en base a una función de probabilidad. Esta función deberá aceptar una cantidad de soluciones mayoreds al principio del algoritmo, y menos conforme se vayan produciendo

más iteraciones. Tanto si reemplazamos la solución como si no, repetiremos el paso 2.

3. *Punto de parada*. Este algoritmo tiene como punto de parada un número de iteraciones máximas fijo el cual se define al comenzar el algoritmo, y cuando se alcanza este número se devuelve la solución actual como solución final. Adicionalmente, podemos definir un número máximo de iteraciones sin que se produzca una mejora con los cambios de base.

Vemos que de la misma forma que en la búsqueda de entorno variable, para este algoritmo también es necesario hacer una definición de vecindario. Aunque el método hace cambios al azar, podría parecer que esto no es necesario, sin embargo, si hiciésemos cambios de forma aleatoria con todas las posibles soluciones, tendríamos una probabilidad despreciable de encontrar el mínimo global en un tiempo acotado.

Lo que asume este algoritmo es que el espacio de soluciones está representado por una curva parecida a la "*Ilustración 6. Función de soluciones de recocido simulado*". Este tipo de funciones tiene subidas y bajadas, pero conforme nos acercamos al mínimo global, todas las soluciones cercanas van siendo menores de forma generalizada.

Debido a esta curva, cuando encontramos una solución mejor y la sustituimos por la actual acercándonos de esta forma al mínimo global, al acotar en un espacio más pequeño las posibilidades para realizar el cambio de base, lo que estamos haciendo es mantener parte del vecindario (ya que habrá muchas bases que sean comunes, normalmente las que estén más alejadas del mínimo) mientras que añadimos otras más cercanas al óptimo de la función. De esta forma nos vamos acercando poco a poco al mínimo.

Hace falta definir un tamaño adecuado para este vecindario, ya que en el peor de los casos si es demasiado grande este sería la totalidad de las soluciones, y si es demasiado pequeño podría haber tan pocas posibilidades que nos quedamos encerrados en un mínimo local. Debido a esto, surge la necesidad de conseguir un vecindario relativamente pequeño, pero que contenga el número suficiente de elementos para que no nos impida avanzar en la solución.

Por otra parte, debido al nombre de estos vecindarios, podemos pensar en una idea parecida al caso de la búsqueda de entorno variable, y utilizar la distancia G entre dos bases para estructurar estos subconjuntos. Sin embargo, esto sería un error, ya que en esta metaheurística siempre se utiliza la misma estructura de vecindario, porque los cambios son siempre los mismos en ese conjunto pero de forma aleatoria. Por tanto, fijarnos una distancia G para realizar todos los cambios nos estaría condenando a realizar siempre los mismos tipos de modificaciones en la lista, con lo cual habría un punto en el que dejaríamos de avanzar, ya que depende de esta distancia esos cambios son mejores al principio o al final.

Lo que haremos por tanto será definir el vecindario mediante la distancia de Hamming. Al fijar una distancia de Hamming, estamos controlando cuántos elementos se intercambian en cada cambio de base, por lo que podemos seguir la pista a estas modificaciones, mientras que la distancia G no está fija ya que no sabemos el peso de estos atributos que se alteran, por lo que en cada iteración se utiliza un tipo distinto de alteración en la tabla.

Aunque el algoritmo se utiliza en muchos problemas con buenos resultados, en este caso el punto anterior es un punto en contra, ya que, al producirse cambios de todo tipo en todo momento, estamos también aplicando cambios con distancia G alta cuando estamos cerca de la solución y cambios de baja G cuando estamos todavía en las primeras iteraciones, lo cual no es lo óptimo.

Se podría modificar para que tuviese al principio más tendencia para seleccionar aquellos cambios con una distancia elevada, mientras que según avanza las iteraciones va disminuyendo la distancia, pero esto se ve como un punto de mejora y escapa al objetivo del trabajo, ya que de forma adicional, esta teoría de los momentos ideales para hacer un cambio de base con determinadas distancias son solo hipótesis, y están lejos de ser demostradas, por lo que también es interesante aplicar una variable aleatoria de este tipo para poder ver la evolución.

De forma parecida al caso anterior utilizaremos una distancia de Hamming con valor de 2, ya que nos permite poder seguir de forma controlada los cambios que se producen, además de que estamos reduciendo el tamaño del vecindario bastante, ya que cuantos más elementos se intercambien más posibilidades hay.

Por tanto, definimos como vecindario para el algoritmo de recocido simulado como todas las bases que se encuentren a una distancia de Hamming 2 (dos elementos cuya posición ha sido intercambiada) de la base actual.

Una vez definido los vecindarios podremos realizar el paso dos del esqueleto que hemos explicado. Cuando realicemos el cambio de base, igual que en la anterior metaheurística, actualizaremos la KBM2L y comprobaremos el nuevo número de ítems de esta. Si hemos encontrado una lista mejor, actualizaremos la solución de inmediato, sin embargo, a diferencia de otras heurísticas, si la nueva solución resulta peor que la actual, todavía hay una oportunidad de aceptarla.

Esta oportunidad se mide mediante una función de probabilidad, la cual al principio tiene que aceptar soluciones peores con más frecuencia que en las últimas iteraciones del algoritmo. El objetivo de esto es escapar de mínimos locales, en los cuales podemos quedarnos atrapados en el comienzo del procedimiento, mientras nos aseguramos de no aceptar muchas soluciones peores al final del algoritmo, ya que nos estaríamos alejando del posible mínimo

global, ya que entendemos que al final de las iteraciones nos estaremos acercando a la mejor solución del problema, o al menos a una aproximada.

Por lo tanto, aunque al principio sí que queremos aceptar soluciones peores, ya que esto nos permite explorar un espacio que quizá no se hubiese explorado de otra manera, lo cual sería un error, seguir aceptando una gran cantidad de soluciones que empeoren el resultado actual conforme avanza el algoritmo comienza a ser peligroso, y puede ralentizar el progreso e incluso desviarnos del mínimo global.

Hay muchas formas de implementar esta función probabilística, aunque de forma generalizada todas dependen del parámetro de la temperatura, haciendo referencia al origen de este método. De esta forma, esta temperatura se reduce en cada iteración, indicando menos saltos entre las moléculas, lo que indica que con cada iteración es menos probable cambiar a una base peor que la actual. Esta reducción se puede hacer de dos formas, de forma lineal o no lineal.

- Una reducción lineal en la temperatura produce una disminución de forma más o menos lineal en la probabilidad de aceptar estas bases peores.
- Por otro lado, esta reducción se puede hacer de forma no lineal, por ejemplo, exponencialmente o como una reducción de un porcentaje de la temperatura actual. Con estas variaciones lo que se busca es aumentar o disminuir la probabilidad de estos cambios de base en determinadas zonas respecto a la alternativa de la reducción lineal, ya que depende del problema es posible que nos interese más un mayor número de cambios al principio que si lo redujésemos de una forma lineal, y menos cambios al final, o viceversa.

Vemos que la estrategia de reducción de la temperatura es importante, así como la fórmula probabilística habrá que definirlo también, pero esta no debe depender solo de la temperatura, ya que no supone el mismo cambio aceptar una base que empeore un poco a alejarnos completamente y aceptar una base mucho peor.

En nuestro caso habrá que definir un grado de empeoramiento, y lo haremos mediante los ítems de las listas, con lo que podremos medir el grado de empeoramiento que se produce.

Por última parte solo faltaría decidir el punto de parada. Este punto de parada tiene una adaptación muy sencilla, y es el número de iteraciones máximas que queremos probar un cambio de base.

Ya tenemos la idea del algoritmo definida, así que vamos a ver su funcionamiento y la forma de codificarla.

3.2.1 Algoritmo recocido simulado básico

Una vez definidos estos conceptos, ya podemos implementar el algoritmo adaptándolo a nuestro problema. La adaptación de esta metaheurística es bastante inmediata, ya que no es necesario definir ningún concepto adicional relativo al espacio de las soluciones. El código se encuentra en el anexo "7.3 Código del algoritmo de recocido simulado".

La función recibe como argumento tres parámetros: el KBM2L que se quiere optimizar, el número de iteraciones y la temperatura inicial.

Esta variable de temperatura es el mecanismo para reducir la probabilidad de aceptar soluciones peores en cada iteración, mientras que el número máximo de iteraciones indica el número de veces que se intentará realizar un cambio de base para encontrar y la solución mejor. En este algoritmo, no tiene sentido utilizar otro punto de parada, ya que es normal que esté numerosas iteraciones seguidas sin encontrar ninguna mejora, debido a la aleatoriedad del cambio, por lo que solo terminará cuando se agoten todas las iteraciones.

Esto hace que el algoritmo dure siempre un tiempo acotado determinado por el usuario. A diferencia de la búsqueda de entorno variable, esto significa que el usuario puede elegir si quiere una solución más aproximada o exacta, ya que aumentando el número de iteraciones que se realicen, se explora el espacio de soluciones de forma más exhaustiva, a cambio de más tiempo de ejecución. Sin embargo, en el algoritmo anterior el espacio siempre se recorre de la misma manera, y es un parámetro que no se puede cambiar, por lo que en el recocido simulado tenemos más flexibilidad en este aspecto.

De la misma forma que en los casos anteriores, el algoritmo comienza asignando la lista de entrada y su base como la solución inicial, la cual se irá sustituyendo cuando se encuentren soluciones mejores, y después comienza el bucle *for* para realizar las iteraciones indicadas.

Lo primero que se hará en cada iteración es calcular la temperatura actual en base a la temperatura inicial. Como hemos explicado en puntos anteriores, aquí surgen distintas alternativas, por lo que después de varias pruebas, se ha optado por la fórmula siguiente:

$$Temperatura\ actual = \frac{temperatura\ inicial}{iteración\ actual}$$

Ecuación 3 Cálculo de la temperatura

Lo que se consigue con esta función es una temperatura elevada en los primeros pasos, la cual decrece rápidamente con la segunda derivada positiva, por lo que cada vez decrece a menos velocidad. El objetivo es por tanto que al principio haya muchas probabilidades de aceptar bases peores, las cuales decrecen

radicalmente en las primeras iteraciones, reduciéndose poco a poco hasta que la temperatura actual sea muy baja.

Para que la temperatura se aproxime al valor 0, será necesario que la temperatura inicia sea inferior al número total de iteraciones. Debido a esto, habría sido suficiente un único argumento del número de iteraciones, y en base a este se calcula la temperatura adecuada, pero al dar la posibilidad de introducir los dos valores se permite más flexibilidad para comprobar la eficiencia y efectividad del método cuando alteras estos argumentos.

El siguiente paso será escoger una base aleatoria del mismo vecindario que la actual. Recordamos que, para este método, hemos definido el vecindario de una base como todas las bases que se encuentren a una distancia de Hamming con valor de 2, es decir, aquellas que han sufrido un cambio de orden en exactamente dos atributos. Para ello se ha creado una función en R (*combinaciones.base*), que recibe como parámetro una base, y devuelve todas las bases del vecindario. Para ello, le pasaremos a este método la base de la lista actual.

Una vez tenemos toda la lista, se elegirá una sola de las alternativas de forma aleatoria, mediante la función *sample* proporcionada por R. En este momento, probaremos la base de la misma forma que en el algoritmo anterior, cambiamos de base la lista actual y comprobaremos el número de ítems que devuelve. Si se ha obtenido una mejora, mantendremos este cambio y asignaremos esta nueva lista a la solución.

Si en caso contrario la solución es peor, ejecutaremos el siguiente fragmento de código:

```
#si es peor, hay probabilidad de cambiarlo
    else{
      prob<-exp(-(items2-items)/temp)

    ## si se da el caso, se acepta
      if(prob>=runif(1, 0, 1)){
        bestkbm<-prueba
        items<-items2
        bestbase<-newBase
      }
    }
```

Ilustración 10 Posibilidad de aceptar soluciones peores

La primera línea dentro del *else* calcula la probabilidad de que aceptemos esa solución, la cual depende de 3 variables: la temperatura, el número de ítems actuales y el número de ítems de la lista con la nueva base. La resta del número de ítems hace que la probabilidad de aceptar la nueva base no solo depende de a iteración en la que nos encontremos, sino tiene en cuenta el grado de

empeoramiento de la lista con la nueva base. Así, aunque haya probabilidades de aceptar bases peores, hay menos probabilidad de aceptar aquellas que producen un empeoramiento muy grande. Para el cálculo de esta probabilidad, nos basaremos en la ecuación "Ecuación 1", quedando de la siguiente forma.

$$Probabilidad = e^{-\frac{items2-items1}{temperatura\ actual}}$$

Ecuación 4 Probabilidad de aceptar bases peores.

Podemos observar que el número *e* está elevado a la negativa, por lo que, a menor temperatura, menor probabilidad, y a menor diferencia entre el número de ítems, mayor probabilidad habrá de aceptar la base, por lo que con esta fórmula se consigue el objetivo propuesto.

Para aplicar el valor obtenido, generaremos un número aleatorio mediante *runif* (método de R), entre el valor 0 y el 1. Con este método, la probabilidad de obtener un número menor a un determinado valor es ese mismo valor. La explicación es que el número de elementos entre el 0 y X, está representado por ese X. Así, la probabilidad de obtener un número entre 0 y 1 es de 1(100%), mientras que si quisiésemos obtener un número menor de 0.8 la probabilidad será del 80%, ya que el 80% de los valores entre 0 y 1 son menores de 0.8

De esta forma, comparando este número generado de forma aleatoria, con el resultado de la ecuación de probabilidad, podemos tomar la decisión de si aceptamos o no la nueva base: si el número generado es menor que la probabilidad calculada, entonces aceptaremos la nueva ase y actualizaremos la solución. Esto se debe a la explicación anterior, ya que si obtenemos una probabilidad de 0.3 y generamos 100 números aleatorios entre 0 y 1, el 30% de estos serán inferiores, que es el número de casos que queremos aceptar, por lo que cuando inferior, es cuando aceptamos el cambio de base que produce un empeoramiento en la solución.

Todo este proceso lo repetiremos tantas veces como las iteraciones indicadas por el usuario, y una vez se terminen, se devolverá la solución actual.

A diferencia de la búsqueda del entorno variable, este método utiliza el componente del azar, por lo que cada ejecución será distinta a la anterior, y no necesariamente se llegará a la misma solución. Además, como ya hemos comentado, la duración de este método depende del número de iteraciones que se realicen, por lo que el usuario será responsable de introducir un número apropiado, ya que con un número demasiado pequeño puede haber todavía margen de mejora, mientras que con uno demasiado grande puede darse la posibilidad de haber encontrado la solución mucho antes de terminar el algoritmo. La solución a esto como ya se ha explicado sería poner un punto de parada tras múltiples iteraciones sin mejorar, pero es peligroso debido a que

con tablas de muchos atributos los vecindarios son muy grandes, por lo que no es raro que haya muchas iteraciones seguidas sin mejoría, aunque haya varias soluciones mejores, por lo que se ha optado por esta opción.

3.2.2 Mezcla VNS-Recocido simulado

Con tres algoritmos distintos ya podemos empezar a ver fortalezas y debilidades de cada uno.

Por una parte, la búsqueda del entorno variable hace una exploración más profunda y metódica del espacio de búsqueda, en las que además aprovecha las propiedades de las KBM2L para aplicar los mejores cambios en cada momento.

A cambio, el recocido simulado tiene la capacidad de ser más rápido ya que no explora el espacio de soluciones de forma tan exhaustiva, aunque debido a la aleatoriedad que se ha introducido, también puede dar peores resultados tanto de tiempo como de efectividad algunas veces. Además, el mecanismo de aceptar bases peores para escapar de los mínimos locales es una buena estrategia que no se ha contemplado para la búsqueda de entorno variable.

Por tanto, se ha desarrollado una última variante que intenta mezclar las partes buenas de cada algoritmo. Esta variante, coge la estructura principal del VNS, y la adapta para introducir algunos de los conceptos clave del algoritmo de recocido simulado.

```
vns2.rafa<-
function(rafa.kbm,nIter=100,proporcionIni=0.8, resta=0.2, temperature=100
){
  proporcion=proporcionIni
  ## inicializamos solucion
  base=rafa.kbm@base
  bestkbm<-rafa.kbm

  ## condicion parada, numero de iteraciones maximo
  for (i in 1:nIter){
    temp <- temperature/i
    bestBase=getBestSwap2(bestkbm, proporcion,resta,temp)

    ##si no mejora, reducimos vecindario
    if(all(bestBase==base)){
      proporcion=proporcion-resta
    }

    #si mejora, volvemos al primer vecindario y actualizamos KBM
    else{
      base=bestBase
      apoyokbm<-custom.base(bestkbm, base)
      bestkbm<-swap.base.kbm(bestkbm,apoyokbm)
      proporcion=proporcionIni
    }

    ## fin de vecindarios, se devuelve la solucion
    if (proporcion<0){
      print('se acabaron los vecindarios')
      break
    }
  }
  return(bestkbm)
}
```

Ilustración 11 Mezcla VNS_Annealing

Podemos observar que el esqueleto es casi idéntico al de la búsqueda del entorno variable, aunque con algunas ligeras modificaciones.

Como argumentos, además de la lista KBM, la proporción, y la resta (para los vecindarios del VNS), recibe también un número máximo de iteraciones para poder acotar el tiempo de búsqueda, y una variable de temperatura, de igual forma que en el recocido simulado.

Después de inicializar la solución inicial, entra en el bucle para controlar las iteraciones máximas, donde calcula el valor de la temperatura actual de la misma forma que en el recocido simulado.

Después llama a la función *getBestSwap2*, que tiene el mismo efecto que el *getBestSwap* de la variante básica del VNS. Es decir, devuelve la mejor base del vecindario indicado, pero le pasa como argumento la temperatura actual de forma adicional.

Una vez se devuelve la base, comprobamos si ha habido una mejora o no: si la ha habido volvemos al primer vecindario con la nueva lista, y si no la ha habido, avanzamos de vecindario hasta el momento en el que se acaben, que será el momento de parada del algoritmo. Estos vecindarios están organizados de la misma forma que la que se ha utilizado en la búsqueda de entorno variable básico, haciendo una separación por la distancia G, y fragmentándolo en bloques del tamaño indicados por el usuario.

Podemos ver por tanto que el funcionamiento es prácticamente idéntico a la búsqueda de entorno variable, pero las mayores diferencias se encuentran en la función *getBestSwap2*.

El script de este método es más largo que el resto, por lo que no se ha incluido en la memoria, sin embargo, se puede ver todo el código en el repositorio de github indicado en “Anexos”.

Este método, como aquel en el que se basa, comienza inicializando la solución con la base actual, y crea las listas donde luego almacenará las soluciones del vecindario mejores que la actual.

En este momento, calcularemos todas las bases a distancia H2, y filtraremos para quedarnos con las que pertenezcan al vecindario actual. En esta variante no se va a explorar el vecindario en su totalidad, sino que se va a seleccionar de forma aleatoria un subconjunto de ellos. Está codificado para que solo se explore el 50%, pero esto se puede modificar de manera sencilla. De esta forma, es posible que obtengamos una solución peor o que tengamos que recorrer más vecindarios debido a que se queden fuera algunas de las mejores alternativas, sin embargo, se espera una gran mejora en la velocidad del algoritmo para encontrar una buena aproximación del mínimo global.

Una vez tiene las bases cuya calidad hay que comprobar, se transforma la lista y se calculan los nuevos ítems de la forma habitual. En este punto se han probado dos aproximaciones:

- Se testean las bases seleccionadas y se selecciona aquella que dé el mejor resultado, como en los casos anteriores.
- La primera base que produzca una mejora respecto a la lista actual es la que se devuelve.

Aunque las dos opciones sean interesantes, después de varias pruebas se ha seleccionado la primera opción, ya que al reducir desde el principio el vecindario consideramos que no es necesario seguir reduciendo más, aunque puede ser interesante utilizar la segunda política con la variante básica del VNS.

Una vez realizado este paso hay dos posibilidades, que se haya realizado una mejora, o que no se haya conseguido. En el primer caso se devuelve esa solución, como en el *getBestSwap*. En caso contrario, se seleccionará la mejor alternativa que se haya encontrado (que es peor que la solución actual), y se realizará el mismo proceso que en el algoritmo del recocido simulado, en el cual se calculará la probabilidad de aceptar esa base mediante la diferencia de ítems y la temperatura. Si al final se acepta, se devolverá esa nueva base, y en caso contrario se devolverá la misma que recibió el método.

De esta forma, se han intentado combinar los puntos fuertes de ambos métodos.

3.3 Optimizaciones generales

Además de estos algoritmos, se han intentado implementar ciertas mejoras sin modificar el funcionamiento de estos. Basándonos en los trabajos previos y en lo que se ha podido observar del funcionamiento de los algoritmos hasta ahora, parece que uno de los cuellos más importantes del algoritmo es el cambio de base de una KBM2L.

Este cambio tarda mucho, y cuando la lista tiene numerosos atributos, parece que un solo cambio de base puede tardar casi un minuto, aunque lo analizaremos más tarde en el capítulo “4 Análisis de los resultados”. Por lo tanto, las aproximaciones que vamos a realizar tratan de mejorar el problema desde este punto de vista.

3.3.1 Switches

La idea que se va a desarrollar está basada en el concepto de los *switches*, explicado en el apartado “2.3.2.5 Testeo de una nueva base”.

Para ello, se ha desarrollado un código que obtiene los supremos de los ítems, y los almacena en una lista. Esta lista contendrá los índices de dichos elementos, así como la política asociada a dicho estos, por lo que es una matriz en la que cada fila es una entrada de la tabla de decisiones, y cada columna el valor de ese atributo, ya sea atributo o política.

No es necesario utilizar únicamente un elemento de cada ítem, de la misma forma que no es necesario que sea el supremo el elemento que represente al ítem, sin embargo, el estudio previo del que proviene la idea [3], parece indicar que es un ítem representativo y que con la inclusión de este debería de ser suficiente.

Una vez tenemos la lista de estos supremos, recordemos que no es una lista KBM, sino que tiene un formato de matriz de R, aplicaremos el cambio de base, intercambiando de sitio las columnas necesarias, y reordenando la lista en base a los nuevos índices. Esta matriz se le pasará como argumento a otro método denominado *getSwitches*, cuyo trabajo consiste en contar el número de saltos de política que existen en esta matriz.

El número devuelto de K no representa el número de ítems que tendrá la KBM2L con esa base, sino que es una cota inferior, ya que, al tener solo un pequeño conjunto de los elementos, el conjunto total tendrá al menos tantos ítems como *switches*, por lo que utilizaremos este valor como aproximación.

En base a esto, hemos implementado este código para el algoritmo de la búsqueda de entorno variable, en el script *vns.switch.R*. Este funciona de la misma forma que el VNS básico desarrollado, el cual recorre los vecindarios, y llama a un método (en este caso *switch.kbm)* que realiza la misma función que *getBestSwap*: hace una exploración completa del vecindario, y devuelve aquella base con los mejores resultados.

Es por tanto dentro de esta función *switch.kbm* donde incluiremos el código que se acaba de explicar, ya que es aquí donde se hace el testeo de las bases del vecindario, y solo consideraremos bases mejores a la actual aquellas que tenga un número de *switches* estrictamente menor a los ítems de la KBM2L actual.

Una vez este método nos devuelve la mejor base, ya se hará la transformación a KBM2L con esta nueva base, sin embargo, antes de aceptar por completo esta base, ahora sí que se calculará el número de ítems reales. Como este cambio de base es obligado para poder testear de nuevo el primer vecindario, esta tarea no consume más tiempo. Sin embargo, hay que tener en cuenta que como los *switches* devuelve una cota mínima en vez del número real de ítems, podemos estar aceptando una base peor, sin saber el grado de empeoramiento. Por tanto, antes de aceptarla comparamos la nueva KBM con la anterior, y si se realiza una mejora, ya podemos continuar con el algoritmo.

En caso de que no exista mejora, rechazaremos esta base, y pasaremos al siguiente vecindario. Vemos por tanto que, aunque la base no se acepte no hemos desperdiciado tiempo, ya que este cambio de base habría que haberlo hecho de todas formas si se hubiese producido una mejora, y solo estamos haciendo como mucho un cambio de base por vecindario, al contrario del resto de algoritmos desarrollados.

El desarrollo realizado para implementar el testeo de las bases consigue una mejora del tiempo en dos frentes:

- Solo se realiza a lo sumo un cambio de base con la KBM2L entera por cada vecindario.

- La comprobación del número de *switches* no se hace en la lista KBM, sino que los datos de esta se insertan en una matriz de R. De esta forma, aunque estamos perdiendo mucha información, estamos conservando la que necesitamos (los índices las políticas), por lo que todas las operaciones sobre esta matriz son mucho más rápidas, ya que es un objeto que consume mucha menos memoria, y no tiene todos los mecanismos de control que sí que tienen definidos los objetos KBM, por lo que los cambios de base y de orden en las filas son mucho más inmediatos.

De esta forma, se espera que con esta nueva adaptación a la heurística de VNS, aunque puedan conseguirse resultados peores, se obtenga una gran mejora en el tiempo de ejecución.

4 Análisis de los resultados

Una vez desarrollado el código, solo queda probarlo para comprobar su funcionamiento. Para ello, van a hacerse pruebas donde mediremos el tiempo total de ejecución, así como la efectividad de los métodos para reducir el número de ítems.

Estas pruebas de ejecución van a contener dos fases distintas:

- Pruebas con KBM2L creadas artificialmente
- Pruebas con tablas de problemas reales.

Separarlo en estas dos fases nos va a permitir poder fijarnos en distintas cosas, y empezar por ver si los algoritmos funcionan, hasta en la segunda fase poder ver si son aplicables al mundo real

4.1 Pruebas con listas artificiales

El crear nuestras propias KBM2L de forma artificial, nos permite tener un mayor control sobre el ejemplo que estamos estudiando. El plan de acción en este caso va a ser crear una KBM2L compacta L_0, con un número de ítems reducido, y luego cambiar esta lista de base, para desordenarla creando la lista L_1, donde intentaremos que haya un número elevado de ítems.

Es sobre esta lista desordenada L_1 donde aplicaremos las heurísticas desarrolladas. De esta forma, tenemos conocimiento de hasta qué punto es posible reducir el número de ítems, ya que la hemos creado nosotros.

De forma adicional, vamos a medir el tiempo de ejecución de cada algoritmo en cada caso. El tiempo de ejecución se medirá mediante la función *system.time* proporcionada por R [19]. Este método, devuelve el tiempo de ejecución de CPU, con la posibilidad de devolver otras medidas indicadas por el usuario en los argumentos, pero en nuestro caso solo nos importa ese tiempo de ejecución.

Debido al componente aleatorio introducido en algunas heurísticas, será necesario realizar las pruebas varias veces y con casos distintos, ya que las medidas pueden cambiar de una ejecución a la siguiente.

4.1.1 Creación de las listas sintéticas

Para la creación de estas KBM2L personalizadas, la infraestructura ofrece un método que nos permite crear tablas aleatorias (*random.kbm*). Esta función, no solo crea listas aleatorias, si no que permite introducir los parámetros que definan esa lista: el número y nombre de atributos, el cardinal de los atributos, el número de columnas de decisión, y el cardinal de estas, además del parámetro más importante en nuestro caso, el número de ítems de esta lista.

Vamos a ver una ejecución de prueba para ver su funcionamiento mediante la ejecución del siguiente fragmento de código:

```
inst.kbm <- random.kbm( rl= c("D1","D2"), rcl=c(2,2),
            al=c("A1","A2","A3","A4"), acl=c(2,2,2,2), n=4)
```

Ilustración 12 Creación de un KBM2L aleatorio

Estamos creando un objeto de tipo KBM, mediante la función indicada previamente, la cual recibe varios argumentos:

- Rl: *Response list*, representa la lista de respuestas, es decir, el número de columnas que contienen la información de la decisión.
- Rcl: *Response cardinal list*, la lista de los cardinales de cada atributo de respuesta.
- Al: *Atribute list*, el nombre de la lista de los atributos, con lo que estamos indicando también su número
- Acl: *Atribute cardinal list*, de forma análoga que, con las respuestas, indicamos en el orden especificado por la lista de atributos el cardinal de cada uno de ellos.
- N: El número de ítems que deberá tener la lista.

Así, al invocar este método, nos devolverá una el objeto KBM, el cual podemos imprimir en formato de *offset* para ver su estructura de forma sencilla:

```
kbm.offset mode:
1 <5, 1, 0.120146890636533| 6
2 <6, 3, 0.508260161615908| 1
3 <14, 1, 0.639347029617056| 8
4 <15, 0, 0.00219431379809976| 1
```

Ilustración 13 KBM2L sintético

Aunque es una estructura parecida a la explicada en el apartado "2.3 Knowledge Based Matrix To List", vemos que hay algunas diferencias, ya que la explicada anteriormente, de cada ítem solo era necesario indicar el *offset* del supremo y la política del ítem, y vemos que, en este caso, de cada ítem (representados por cada línea), existen 5 valores.

Esto se debe a que, al almacenar esta estructura de forma informática, hay otros datos que pueden ser de valor:

- El primer valor representa únicamente la posición numérica del ítem respecto a los demás, por lo que un 1 significa que es el primer ítem.
- El segundo y el tercer valor son los explicados en el apartado anterior, el *offset* del supremo y la política del ítem. Son estos dos valores los más

importantes, y los que representan por completo a la lista, mientras que los otros son auxiliares.

- El tercer valor es la utilidad de la decisión, que indica con un valor del 0 al 1, el beneficio de realizar esta operación. Es un valor comúnmente utilizado en sistemas de apoyo a la decisión, pero que no tendrá ninguna relevancia para el trabajo actual.
- El cuarto valor es el número de elementos contenidos en ese ítem. Es un valor redundante, ya que se puede sacar este valor con una resta entre los *offset* de los ítems, pero verla de forma inmediata puede ayudar a interpretar las listas de forma más rápida.

Podemos observar que la política "2" no se encuentra en la lista, esto se debe a que, debido a la generación aleatoria, no tienen por qué darse todas las casuísticas.

4.1.2 Realización de las pruebas

Con la herramienta del apartado anterior, ya somos capaces de generar las listas que necesitamos para hacer las pruebas.

En ellas, ejecutaremos los métodos desarrollados, sin embargo, excluiremos la búsqueda de entorno variable truncado, debido a que la mezcla de VNS-recocido simulado ofrece las mismas características más la ventaja de escapar de mínimos locales, por lo que se ha considerado innecesario probar ese algoritmo. Por lo que probaremos el VNS básico, el recocido simulado, la mezcla de VNS con el recocido, y el VNS utilizando los *switches* para el testeo de las bases.

Se crearán listas con distinto número de atributos (7,8 y 9), todos binarios, las cuales inicialmente tendrán siempre 8 ítems. Por tanto, estas listas tendrán una cantidad de entradas con valor de 2 elevado al número de atributos, es decir, 128, 256 y 512 elementos respectivamente. A estas listas, se les aplicará un cambio de base al azar, el cual originará otra KBM con un número mayor de ítems, aunque este variará en cada caso.

Después, a cada una de las listas generadas, se le aplicarán todos los algoritmos para poder comparar la eficiencia y efectividad de los métodos utilizando las mismas listas. Después de realizar la ejecución, compararemos los tiempos y los resultados obtenidos, teniendo en cuenta que el mejor resultado de ítems posibles a obtener es 8.

Esta es la tabla de los resultados:

Propiedades de la lista		**Tiempo de ejecución en segundos / ítems obtenidos**			
Atributos	**Items iniciales**	**VNS**	**Annealing**	**Mezcla annealing-VNS**	**Vns-Switches**
7	**88**	44 / 8	41 / 10	53 / 18	12 / 22
7	**78**	70 / 8	49 / 17	38 / 33	14 / 14
7	**55**	44 / 8	42 / 15	62 / 33	4 / 55
7	**76**	25 / 08	35 / 08	41 / 18	9 / 16
7	**48**	50 / 08	66 / 10	11 / 30	7 / 16
8	**99**	476 / 8	497 / 16	189 / 15	46 / 11
8	**49**	265 / 8	201 / 8	108 / 21	27 / 23
8	**153**	338 / 8	350 / 8	270 / 24	56 / 20
8	**135**	332 / 8	273 / 8	390 / 12	44 / 17
8	**123**	471 / 8	218 / 11	145 / 38	47 / 113
9	**98**	2928/16	4841 / 21	1145 / 18	224 / 22

Tabla 5 Resultados de las pruebas con tablas sintéticas

Cada fila representa una lista distinta, la cual tiene el número de atributos e ítems iniciales que se indica en sus respectivas columnas. En las columnas de las heurísticas, se encuentran los resultados obtenidos, como dos valores separados por una barra. El primer valor representa el tiempo de ejecución en segundos del algoritmo hasta conseguir esa solución, y el segundo valor indica el número de ítems de la solución obtenida.

De forma general, podemos ver que como ya habíamos supuesto, el tiempo de ejecución aumenta de forma drástica con el aumento del número de atributos. También se puede apreciar que la búsqueda de entorno variable siempre ha encontrado la solución óptima menos en el último caso, mientras que el resto de métodos varían, parece que el algoritmo de recocido es el segundo con mejores resultados. Vamos a analizar las medias de tiempos y de ítems en base al número de atributos de las bases:

Propiedades de la lista	**Tiempo medio de ejecución en segundos / media de ítems obtenidos**			
Atributos	**VNS**	**Annealing**	**Mezcla annealing-VNS**	**VNS-Switches**
7	46.6 / 8	46.7 / 12	41 / 26.4	9.2 / 24.6
8	376.4 / 8	307.8 / 10.2	220.4 / 22	44 / 36.8
9	2928/16	4841 / 21	1145 / 18	224 / 22

Tabla 6 Resultados medios de las pruebas con tablas sintéticas

De estos resultados pueden obtenerse las siguientes conclusiones:

- El VNS es el más efectivo, obteniendo unas listas mucho más optimizadas que el resto de los métodos en general, aunque seguido de cerca por el algoritmo de recocido simulado.
- La utilización de *switches* disminuye de una manera radical el tiempo empleado, aunque consigue unos resultados peores al resto de métodos. Así, aunque no parece el algoritmo ideal, en listas que sean excesivamente grandes es posible que sea útil para poder obtener unos primeros resultados aproximados.
- Aunque la mezcla del recocido simulado con la búsqueda de entorno variable obtiene peores resultados que los dos métodos independientes, parece que conforme crece el tamaño de la tabla va mejorando su efectividad, así como el tiempo de ejecución, y con nueve atributos obtiene un número de ítems similar a estos dos, pero en mucho menos tiempo (alrededor de cuatro veces más rápido). Esta mejora puede ser debido a que se está escogiendo una proporción del vecindario para testear, por lo que conforme crecen los vecindarios, hay más opciones que permiten hacer una mejor búsqueda por el espacio de soluciones.

De esta forma, podemos afirmar que el mejor de los algoritmos desarrollados es el VNS aquel con los mejores resultados, los cuales son muy parecidos al recocido simulado tanto en tiempo como en número de ítems. Sin embargo, este método tiene un tiempo de ejecución bastante elevado comparado con la mezcla VNS-Annealing y comparado también con los *switches*, por lo que, con tablas muy grandes, puede ser interesante hacer una primera iteración con estos dos últimos métodos, ya que la mezcla es tres veces más rápida que el VNS, mientras que con los *switches* logramos una ejecución trece veces más rápida en el caso de nueve atributos. Además, esta diferencia de tiempos se verá cada vez más incrementada conforme crezca la base que se está probando.

De forma adicional, la mezcla de Annealing-VNS obtiene un número de ítems cada vez mejor, hasta que consigue unos resultados parecidos en las tablas de 9 atributos, por lo que nos hace preguntarnos si cuanto mayor sea la tabla, más se aproxima a unos resultados tan buenos de la búsqueda de entorno variable.

Es en estos resultados donde se puede ver por fin la ventaja de la construcción de este tipo de listas, ya que el usuario no necesitará navegar por cientos de registros, sino que la nueva lista solo tiene unas pocas decenas, en las cuales podrán verse los atributos comunes para las decisiones, ya que al ser entradas reducidas será más fácil extraer patrones, pudiendo asociar determinados valores de los atributos a la toma de determinadas soluciones.

4.2 Pruebas con ejemplos reales

Para los ejemplos reales, se ha utilizado un dataset hecho público en 2007 [20], en el cual se incluye información de las variables detectadas en los pacientes, asociado a la decisión que hay que tomar.

Este dataset contiene una gran cantidad de información, por lo que se ha filtrado para probar con un ejemplo de tamaño reducido, eliminando columnas y filas. Para ello, se han hecho cuatro archivos *csv* de distinto tamaño en base a este dataset:

- 10 atributos y 21 filas
- 20 atributos y 26 filas
- 30 atributos y 47 filas
- 40 atributos y 49 filas

En todos los casos, todos los atributos son binarios, se tiene un único atributo de decisión, el cual puede tomar 4 valores, pero debido a un error en la infraestructura para la creación de KBM2L en base a ficheros *csv*, esta columna se ha dividido en 2, las dos de valores binarios, pero, aunque se ha cambiado la representación, sigue manteniendo la misma información.

Estas tablas son muy dispersas, ya que la cantidad de entradas que se tiene de ellas son muy reducidas. El número de posibilidades que hay con 10 atributos son 1024, por lo que deberíamos tener 1024 entradas en vez de 21, por lo que todas las que no tenemos, se les asignará la política -1. Así, aunque se podría pensar que el tiempo de ejecución va a ser extremadamente alto, esto no será así, ya que al ser una matriz tan dispersa no hay tantos elementos para ordenar.

Estos ficheros se convierten de forma inmediata a objetos KBM mediante la función proporcionada por la infraestructura llamado *df.kbm*, el cual recibe como argumentos el nombre del fichero, el número de atributos, y el número de atributos de decisiones.

Estas tablas no se van a ejecutar con todas las heurísticas desarrolladas, sino que, dados los resultados de la prueba anterior, hemos seleccionado únicamente la búsqueda de entorno variable por ser el método con mejores resultados, y el VNS con los *switches,* por ser aquel que con mayor velocidad.

Una vez realizamos las pruebas obtenemos los siguientes resultados:

Propiedades de la lista		**Tiempo de ejecución en segundos / ítems obtenidos**	
Items iniciales	**Atributos**	**VNS**	**VNS-Switches**
36	10	7.20 / 24	3.96 / 36
47	20	48.60 / 44	45.69 / 45
91	30	383.3 / 69	460 / 91
96	40	1067.91 / 88	1223 / 96

Tabla 7 Resultados de las pruebas con ejemplos reales

Como habíamos supuesto, no consumen tanto tiempo como podría esperarse por el número de atributos. En el caso de los switches, no estamos consiguiendo ninguna mejora, excepto en el segundo caso, que reducimos en 2 el número de ítems de la lista, además de que el tiempo de ejecución es bastante similar al del VNS, en algunos casos siendo incluso superior a este.

El motivo por el que se cree que se están obteniendo estos malos resultados con el algoritmo de los *switches* es por la dispersión de la tabla. En el método de los switches, estamos utilizando los supremos de los ítems, con lo cual estamos ahorrando mucho tiempo al utilizar menos entradas. Sin embargo, en tablas con entradas tan dispersas, cada ítem tiene prácticamente un solo elemento, por lo que tiene el mismo resultado aplicar utilizar estos *switches* que los ítems enteros, ya que no estamos haciendo una reducción significativa.

Por otra parte, aunque el VNS sí que obtiene mejores resultados, hay una reducción muy pequeña del número de ítems, sin embargo, no son unos buenos resultados y no sirven para utilizarse ni para obtener conocimiento.

Aunque los resultados obtenidos en estos ejemplos son malos, no significa que sea inútil el trabajo realizado, ya que estos pueden deberse a varios factores:

- Puede ser que justo estas tablas no puedan reducirse mucho más, por lo que no sería un problema de los algoritmos sino de que esas bases de conocimiento no admiten una mayor compresión.

- Es posible que el dataset completo sí que pueda reducirse en gran medida, pero como hemos escogido unos elementos arbitrarios de la tabla creando una lista con una gran dispersión, estos no puedan reordenarse mejor, mientras que si hubiésemos cogido el dataset entero sí. El problema es que esta opción era inviable debido a que los algoritmos desarrollados todavía son demasiado lentos.

Por tanto, aunque estos resultados no hayan sido buenos, tenemos que recordar que con las tablas sintéticas fueron bastante prometedores, por lo que antes de descartarlos convendría realizar más pruebas o mejorar el código ya hecho.

5 Conclusiones y líneas futuras

Los sistemas de ayuda para la toma de decisiones, así como las herramientas para representar sus soluciones han tenido un auge en los últimos años, y cada vez son más usados por profesionales para apoyarse al realizar su trabajo. En este trabajo, en concreto nos hemos centrado en las tablas de decisiones, aunque muchos de los problemas que tiene este sistema son extrapolables al resto de las alternativas.

Cuando estas tablas son extremadamente grandes y tienen muchos atributos, surgen problemas en dos ámbitos distintos: por una parte, la memoria ocupada y el tiempo de operación al navegar estas tablas mediante equipos informáticos pueden suponer un trabajo extra y una disminución al rendimiento; y por otra, debido al gran tamaño es imposible que un usuario pueda interpretar la tabla ni extraer conocimiento de ella, teniendo que limitarse a obtener la mejor solución que indique el sistema.

Para solucionar este problema se proponen las KBM2L, que permiten reducir estas tablas de decisión en gran medida, agrupando las entradas por la decisión tomada, por lo que estamos solucionando los dos problemas a la vez, al ser más pequeñas podrán ser utilizadas más fácilmente por los equipos informáticos, además de que el usuario podrá verla en su totalidad, y además como está agrupada por decisiones puede extraerse conocimiento en base a la obtención de patrones para tomar cada decisión.

El objetivo de este trabajo ha sido la implementación de un software capaz de generar estas KBM2L y reducirlas lo máximo posible mediante metaheurísticas de optimización combinatoria.

En este apartado trataremos las conclusiones obtenidas en el proyecto, así como las próximas acciones que se proponen para continuarlo.

5.1 Conclusiones

Una tabla de decisiones es una herramienta extensamente utilizada en problemas de toma de decisiones, sin embargo, cuando estas son muy grandes, el tamaño no permite a los usuarios utilizarlas para todo su potencial, siendo imposible interpretarlas o extraer conocimiento de ellas, además de que pueden utilizar muchos recursos del ordenador.

Por ello, surge la propuesta de los KBM2L, la cual se basa en la propuesta de reordenar estas tablas mediante la permutación de sus atributos, y agrupar las entradas de la tabla en base a las decisiones tomadas, con lo que la lista se convierte en más pequeña y por tanto manejable e interpretable por un usuario.

Aunque existe una infraestructura desarrollada en lenguaje R que permite la creación de estas listas KBM2L, no existe un software capaz de realizar esta

ordenación para obtener una lista compacta. Por ello, el objetivo de este trabajo ha sido desarrollar e implementar métodos que consigan reducir estas listas lo máximo posible.

Como la reordenación de estas tablas de decisión se basa en la permutación de los atributos para encontrar la configuración óptima, nos encontramos ante un problema de optimización combinatoria, el cual, debido a la cantidad posible de atributos existentes, necesitará el uso de metaheurísticas de optimización combinatoria a causa del gran espacio de búsqueda resultante.

Por tanto, en este trabajo se han implementado estas técnicas de optimización para el caso concreto de las KBM2L, en concreto, se han desarrollado los métodos de la búsqueda de entorno variable y el algoritmo de recocido simulado, además de algunas variaciones de estos.

Después de desarrollar estos métodos para la creación de las listas, se han realizado una serie de pruebas de rendimiento, para comprobar la efectividad y eficiencia de estos. En ellas, hemos obtenido unos resultados prometedores en los cuales se ha demostrado la posibilidad de utilizar estas listas y las ventajas que proporcionan.

Sin embargo, todavía nos encontramos en un punto alejado de la posibilidad de aplicar estos métodos en problemas del mundo real, ya que para problemas grandes estos métodos todavía consumen mucho tiempo de ejecución.

5.2 Líneas futuras

De este trabajo surgen varias líneas de proyectos futuros posibles. La primera es la implementación de otras metaheurísticas para la optimización de las listas, ya que, aunque con distintas variaciones, solo se han aplicado dos algoritmos distintos. Además, aunque las heurísticas basadas en poblaciones han sido descartadas para este proyecto, sería interesante comprobar si de verdad son o no aplicables a este problema, y qué resultados pueden aportar.

La segunda línea es la más importante y aquella que merece más atención, y es la optimización de los algoritmos desarrollados para que operen en un tiempo más reducido. Aunque hay muchas posibilidades para realizar estas optimizaciones, en este trabajo se proponen dos ideas: la ejecución paralela de código y la aplicación del multicomienzo.

Lo que se intenta obtener con el multicomienzo es evitar puntos de partida que puedan conducir a una solución errónea. Es posible que haya un gran grupo de soluciones iniciales que, por la topología del espacio de soluciones de un problema concreto, el método alcance siempre un mínimo local, donde se queda atrapado. Al empezar desde varias soluciones iniciales, tenemos más posibilidades de evitar ese mínimo local, por lo que sería ideal que estas soluciones iniciales estuviesen alejadas entre sí.

Como el multicomienzo sería lo mismo que ejecutar el algoritmo varias veces seguidas con una solución inicial distinta, en la que escogemos bases aleatorias, el objetivo es implementar en R una forma de ejecutar todo el algoritmo de forma paralela para ahorrar tiempo de ejecución.

La otra razón para intentar implementar la ejecución paralela de código es el cuello de botella detectado en la búsqueda de entorno variable básica. Aquí, la mayor parte del tiempo de ejecución consumido se dedica al cambio de base de las KBM2L, para testear una nueva base en ese mismo vecindario. La idea para ahorrar el tiempo es, en el VNS, paralelizar el testeo de estas nuevas bases. Ya que no dependen unas de otras, y consumen un tiempo razonable para poder paralelizarlo, al conseguir esto se conseguiría una mejora radical en el tiempo de ejecución del algoritmo, ya que, si esta tarea fuese paralelizable al 100%, estaríamos realizando de forma efectiva un único cambio de base por vecindario.

Uno de los puntos clave para aplicar el paralelismo en R es decidir el grano de ejecución con el que se realiza, ya que crear los procesos necesarios para este entorno es una tarea costosa. Debido a esto, las tareas que se paralelicen tienen que ser costosas en lo relativo a tiempo de ejecución para que merezca la pena esta implementación. Por ello, el paralelismo aplicado al multicomienzo sería adecuado, mientras que la implementación para la exploración del vecindario requeriría un estudio, ya que a lo mejor es ineficiente si el cambio de base para esa lista es rápido.

Sin embargo, no se ha conseguido realizar esta tarea. Debido a las dificultades de R y R estudio a la hora de implementar procesos paralelos en el sistema operativo de Windows, que es donde se ha estado realizando todo el proyecto, a pesar de los esfuerzos, no ha sido posible implementarlo.

Por otra parte, como ya se ha dicho sigue siendo posible utilizar el multicomienzo, aunque debido a la ausencia de paralelismo es como ejecutar el algoritmo varias veces seguidas con bases iniciales distintas. Sin embargo, se recomienda esta ejecución por las ventajas que se han discutido.

Por último, también puede realizarse un análisis más exhaustivo en problemas reales, ya que, aunque con las tablas sintéticas se han obtenido buenos resultados, los obtenidos con los dataset reales no han sido concluyentes.

De esta forma, aunque el proyecto ha dado sus frutos, aún existen varios puntos de mejora que pueden aplicarse para optimizar el proceso y alcanzar mejores soluciones.

6 Bibliografía

[1] U. Kjærulff and A. Madsen, *Bayesian Networks and Influence Diagrams*. 2008. doi: 10.1007/978-0-387-74101-7.

[2] S. R. Insúa, C. B. Lozoya, and A. M. Caballero, "Fundamentos de los sistemas de ayuda a la decisión," 2002.

[3] J. A. Fernández Del Pozo, C. Bielza, and M. Gómez, "A list-based compact representation for large decision tables management," in *European Journal of Operational Research*, Feb. 2005, vol. 160, no. 3 SPEC. ISS., pp. 638–662. doi: 10.1016/j.ejor.2003.10.005.

[4] Wikipedia contributors, "Decision table — Wikipedia, The Free Encyclopedia." 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Decision_table&oldid=1010597104

[5] R. Howard and J. Matheson, "Influence Diagrams," *Decis. Anal.*, vol. 2, pp. 127–143, 2005.

[6] R. Shachter, "Evaluating Influence Diagrams," *Oper. Res.*, vol. 34, pp. 871–882, 1986.

[7] S. Consoli and K. Darby-Dowman, "Combinatorial Optimization And Metaheuristics." May 2006.

[8] D. G. Luenberger and Y. Ye, *Linear and Nonlinear Programming*, vol. 116. New York, NY: Springer US, 2008. doi: 10.1007/978-0-387-74503-9.

[9] P. Hansen and N. Mladenović, "Variable neighborhood search: Principles and applications," *European Journal of Operational Research*, vol. 130, no. 3, pp. 449–467, May 2001, doi: 10.1016/S0377-2217(00)00100-4.

[10] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, May 1983, doi: 10.1126/science.220.4598.671.

[11] "Simulated Annealing in NetLogo," *http://www.cs.us.es/~fsancho/?e=206*, Oct. 19, 2019.

[12] T. Weise, M. Zapf, R. Chiong, and A. J. Nebro, "Why Is Optimization Difficult?," 2009. doi: 10.1007/978-3-642-00267-0_1.

[13] M. Dorigo, M. Birattari, and T. Stützle, "Ant Colony Optimization," *Computational Intelligence Magazine, IEEE*, vol. 1, pp. 28–39, May 2006, doi: 10.1109/MCI.2006.329691.

[14] H. Ahmed and J. Glasgow, "Swarm Intelligence: Concepts, Models and Applications," May 2012. doi: 10.13140/2.1.1320.2568.

[15] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. Wiley, 2005. doi: 10.1002/047174882X.

[16] R. W. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technical Journal*, vol. 29, no. 2, Apr. 1950, doi: 10.1002/j.1538-7305.1950.tb00463.x.

[17] P. Larrañaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic, "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators," *Artificial Intelligence Review*, vol. 13, no. 2, pp. 129–170, 1999, doi: 10.1023/A:1006529012972.

[18] J. Chambers, *Software for Data Analysis*. New York, NY: Springer New York, 2008. doi: 10.1007/978-0-387-75936-4.

[19] "RDocumentation," *https://www.rdocumentation.org/*.

[20] J. P. Pestian *et al.*, "A Shared Task Involving Multi-Label Classification of Clinical Free Text," in *Proceedings of the Workshop on BioNLP 2007: Biological, Translational, and Clinical Language Processing*, 2007, pp. 97–104.

7 Anexos

En esta sección se ofrece información extra de interés. Aunque hay varios fragmentos de código, no está incluido todo debido a su extensión, pero este es accesible desde github: https://github.com/Rafata8/TFM

7.1 Código de custom.base

Función en R encargada de crear un objeto KBM vacío, clon del primer argumento, pero con la base definida en el array pasado como segundo argumento.

```
custom.base <-
function (x,sambas)
{
    sambas<-match(sambas,x@base)
    y <- empty.kbm(x)
    lbas <- length(x@base)

    y@AttributeList <- x@AttributeList[sambas]
    y@AttributeCardinalList <- x@AttributeCardinalList[sambas]
    y@base <- x@base[sambas]
    j <- sambas[1]
    if (j == 1)
        shift <- 1
    else shift <- sum(x@AttributeCardinalList[1:(j - 1)]) + 1
    range <- (shift):(shift + x@AttributeCardinalList[j] - 1)
    y@AttributeDomainList <- x@AttributeDomainList[range]
    y@xbase <- x@xbase[range]
    for (i in 2:lbas) {
        j <- sambas[i]
        if (j == 1)
            shift <- 1
        else shift <- sum(x@AttributeCardinalList[1:(j - 1)]) +
            1
        range <- (shift):(shift + x@AttributeCardinalList[j] -
            1)
        y@AttributeDomainList <- c(y@AttributeDomainList, x@AttributeDoma
inList[range])
        y@xbase <- c(y@xbase, x@xbase[range])
    }
    y@WeightsAttribute <- WEIGHTS(y@AttributeCardinalList)
    return(y)
}
```

Ilustración 14 Código de custom.base

7.2 Código de getBestSwap

Función encargada de obtener la mejor base en el vecindario dado en el algoritmo básico de la búsqueda de entorno variable.

```
getBestSwap<-function(rafakbm, proporcion=0.8, resta=0.2){
  ## se inicializan los items iniciales y la solucion actual como la base
 inicial
  items=length.kbm(rafakbm)
  bestbase=rafakbm@base

  ## listas donde se almacenan las bases que produzcan mejoras
  ## en lista bases se almacena la base, y en la otra lista se almacena e
l número de items de laKBM2L asociado a esa lista
  listaBases<-list()
  listaItems<-c()

  j<-1
  b<-length(bestbase)

  ## doble bucle para iterar por todos los posibles cambios de base con H
=2
  for(it1 in 1:((b)-1)){
      for (it2 in (it1+1):(b)){

      ## calculamos la distancia G según la fórmula
      g2= (it1-1)+(it2-it1-1)+2-1

      ## filtramos las que estén dentro del vecindario
      if(g2>((b)-1)*proporcion & g2<= ((b)-1)*(proporcion + resta)){

        ## calculamos los items de la KBM2L con la nueva base
        newBase<-vector.swap(bestbase,it1,it2)
        prueba<-custom.base(rafakbm, newBase)
        prueba<-swap.base.kbm(rafakbm,prueba)
        items2<-length.kbm(prueba)

        ## comprobamos si se ha producido una mejora
        if(items2<items){

          ## si es así, lo añadimos a la lista
          listaBases[[j]]<-newBase
          listaItems[[j]]<-items2
          j=j+1
        }
      }
    }
  }
```

```
  ## si hay mejoras, cogemos la mejor
  if(length(listaItems)>0){
    bestbase<-listaBases[[which.min(listaItems)]]
  }

  ## si no hay mejoras, se devuelve la base inicial
  return (bestbase)

}
```

Ilustración 15 Código de getBestSwap

7.3 Código del algoritmo de recocido simulado

Codificación del algoritmo de recocido simulado.

```
simulated_annealing <- function(rafakbm,nIter,temperature=50){

## inicializamos solucion
  bestbase=rafakbm@base
  bestkbm<-rafakbm
  items=length.kbm(rafakbm)

  ## comenzamos iteraciones
  for (i in 1:nIter){
    temp <- temperature/i

    # cogemos una solucion al azar del vecindario
    combinaciones<-combinaciones.base(bestbase)
    bases<-combinaciones$base
    newBase<-sample(bases, 1)[[1]]

    #lo hacemos kbm y comparamos numero de items
    prueba<-custom.base(rafakbm, newBase)
    prueba<-swap.base.kbm(rafakbm,prueba)
    items2<-length.kbm(prueba)

    # si es menor, lo cambiamos
    if(items2<items){
      bestkbm<-prueba
      items<-items2
      bestbase<-newBase
    }

    #si es peor, hay probabilidad de cambiarlo
    else{
      prob<-exp(-(items2-items)/temp)

    ## si se da el caso, se acepta
      if(prob>=runif(1, 0, 1)){
        bestkbm<-prueba
        items<-items2
        bestbase<-newBase
      }
    }
  }

  return(bestkbm)
}
```

Ilustración 16 Código del algoritmo de recocido simulado

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Wed Jun 16 19:54:30 CEST 2021
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)