Universidad Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática.

Departamento de Ciencias de la Computación.

GRASP con VNS para el Problema de la Selección de Características

Trabajo Fin de Carrera

Autor
Ricardo Bacelo Polo

Tutores
Abraham Duarte Muñoz
Alfonso Fernández Timón

Viernes 16 de Julio 2010

Agradecimientos

No somos nada sin las personas que día a día comparten nuestros buenos y malos momentos. Agradecer a estas personas el que haya llegado este momento de presentar este proyecto y dar por finalizada la carrera. En especial destacar a algunos de ellos:

A Elena, que sin ella, nada de esto sería posible.

A mis tutores de proyecto, Alfonso Fernández y Abraham Duarte, por hacer posible cerrar una etapa y enseñarme a abrirme paso hacia la programación orientada a objetos.

A mis amigos, por dedicar tiempo a aguantarme.

A mi familia.

Resumen

Dada la cantidad de volumen de datos que se maneja hoy en día, se hace necesaria la aplicación de técnicas de preprocesamiento sobre los conjunto de datos. De las técnicas que se pueden desarrollar para la preparación de los datos, centraremos la atención en la reducción de datos. De las múltiples vías que se pueden seguir para obtener esta reducción, profundizaremos en la selección de características.

En este proyecto fin de carrera se han desarrollado tres algoritmos para resolver el problema de la selección de características. El primero de ellos consiste en, a partir de una base de datos vacía, ir añadiendo características de la base de datos mientras mejore la calidad de ésta. El segundo consiste en, a partir de una base de datos completa, ir quitando características mientras mejore la calidad. Por último se ha desarrollado un algoritmo meta-heurístico, GRASP. Este tipo de procedimientos se caracterizan por utilizar una construcción *greedy,* de la cual se obtiene una solución, que posteriormente es mejorada.

Los resultados que se obtienen tras los experimentos indican, que no es posible determinar a priori si una tabla obtendrá mejores porcentajes si aplicamos un constructivo de añadir características o un constructivo de eliminar características. Esto vendrá condicionado por los datos y las clases que contenga la base de datos.

Sí podemos determinar que, cuánto mayor es el volumen de las características que contienen las bases de datos, mayor es el tiempo de ejecución que se tarda en evaluar cada una de ellas.

ÍNDICE

Agradecimientos I

Resumen III

Capítulo 1
- Introducción 1
- 1.1 Descripción del problema 1
- 1.2 Aplicación del problema 4
- 1.3 Métodos de resolución 6

Capítulo 2
- Descripción algorítmica 13
- 2.1 Planteamiento del problema 15
- 2.2 Algoritmos constructivos 17
- 2.3 Búsquedas locales 19
- 2.4 GRASP 22
- 2.5 VNS 24
- 2.6 Particularizaciones a nuestro problema 25

Capítulo 3
- Objetivos 29

Capítulo 4
- Descripción informática 31
- 4.1 Requisitos funcionales 31
- 4.2 Requisitos no funcionales 32
- 4.3 Ciclo de vida 33
- 4.4 Diagrama de Gantt 34
- 4.5 Descripción del código 35

Capítulo 5
- 5.1 Hardware 51
- 5.2 Tecnología utilizada 51
- 5.3 Descripción de los ficheros 53
- 5.4 Experimentos 55

Capítulo 6
- Conclusiones y trabajos futuros 61

Bibliografía 63

Índice de Figuras

Figura 1.1 Gráfico esquemático de las ventajas de la minería de datos 3
Figura 1.2 Esquema de resolución 11
Figura 2.1 Algoritmos del proyecto 14
Figura 2.2 Pseudocódigo del proyecto 16
Figura 2.3 Pseudocódigo del constructivo que añade columnas 17
Figura 2.4 Pseudocódigo del constructivo que borra columnas 18
Figura 2.5 Pseudocódigo del método de mejora de borrar columnas 20
Figura 2.6 Pseudocódigo del método de mejora de añadir columnas 21
Figura 2.7 Pseudocódigo del método de mejora del intercambio de columnas 22
Figura 2.8 Pseudocódigo del método GRASP 23
Figura 2.9 Pseudocódigo del método VNS 24
Figura 2.10 Cálculo del vecino más cercano 26
Figura 4.1 Ciclo de vida 33
Figura 4.2 Diagrama de Gantt 34
Figura 4.3 Diagrama de clases 36
Figura 4.4 Pseudocódigo crea_test_entre() 42
Figura 4.5 Pseudocódigo del método mejora_vns() 45
Figura 4.6 Pseudocódigo del método evaluación_constr() 47
Figura 4.7 Tabla de valores de ejemplo de *fitness* 48
Figura 4.8 Pseudocódigo del cogerCandidatos() 49
Figura 5.1 Esquema de entrada y salida 53
Figura 5.2 Formato de entrada de los archivos 54
Figura 5.3 Formato de salida del archivo de soluciones 54
Figura 5.4 Datosde las tablas comparativas 55
Figura 5.5. Resultados del archivo *monks-1.test* 56
Figura 5.6 Resultados del archivo *lymphography.data* 57
Figura 5.7 Resultados del archivo *glass.data* 58
Figura 5.8 Resultados del *pima-indians-diabetes.data* 59

Capítulo 1

Introducción

En este capítulo se introducirán los temas que se van a desarrollar en este proyecto, tanto terminología como conceptos básicos. En el primer apartado se introducirá el problema de la **selección de características**, una explicación de cuáles son sus usos y en qué consiste. Además, se explicará que es la minería de datos.

En el segundo apartado, **aplicación del problema**, se describirán las posibilidades que nos presenta este planteamiento teórico, de la **selección de características**, así como las aplicaciones de uso del *Data Minning* dentro del mundo real.

Por último, en el apartado de **métodos de resolución**, se detallarán los métodos y cuestiones teóricas que guiaron este proyecto en su realización. Se explicará cuales han sido los algoritmos **heurísticos** y **meta-heurísticos** utilizados en la realización del proyecto.

1.1 Descripción del problema

1.1.1 ¿Qué es la selección de características?

La *selección de características* es imprescindible en cualquier aplicación de minería de datos. La razón es que, cuando se genera un modelo de minería de datos, a menudo el conjunto de datos contiene más información de la necesaria para generar el modelo. Por ejemplo, un conjunto de datos puede contener 500 columnas que describen las características de los clientes, pero, tal vez, sólo 50 de esas columnas se usan para generar un determinado modelo. Si mantiene las columnas innecesarias durante la generación del modelo, se necesitarán más CPU y memoria durante el proceso de entrenamiento, así como más espacio de almacenamiento para el modelo completado.

Aunque los recursos no sean un problema, normalmente se deben quitar las columnas innecesarias porque pueden degradar la calidad de los patrones detectados por las razones siguientes:

- Algunas columnas tienen ruido o son redundantes. Este ruido dificulta la detección de patrones significativos a partir de los datos.
- Para detectar patrones de calidad, la mayoría de los algoritmos de minería de datos requieren un conjunto de datos de entrenamiento mucho más grande en un conjunto de datos multidimensional. Sin embargo, en algunas aplicaciones de minería de datos se dispone de muy pocos datos de aprendizaje.

La selección de características ayuda a resolver el problema de tener demasiados datos de escaso valor o pocos datos de mucho valor.

1.1.2 ¿Qué es y qué no es la minería de datos?

La minería de datos puede definirse como la **extracción no trivial de información implícita, previamente desconocida y potencialmente útil, a partir de los datos**. Para conseguirlo, hace uso de diferentes tecnologías que resuelven problemas típicos de agrupamiento automático, clasificación, asociación de atributos y detección de patrones secuenciales. La minería de datos es, en principio, una fase dentro de un proceso global denominado **descubrimiento de conocimiento en bases de datos** (*Knowledge Discovery in Databases* o *KDD*), aunque finalmente haya adquirido el significado de todo el proceso en lugar de la fase de extracción de conocimiento.

Es habitual que los expertos en estadística confundan la minería de datos con un análisis estadístico de éstos (afirmaciones de este tipo pueden encontrarse en documentación de empresas dedicadas al procesamiento estadístico que *venden* sus productos como herramientas de minería de datos). La diferencia fundamental entre ambas técnicas es muy clara: para conseguir una afirmación utilizando un paquete estadístico, es necesario conocer a priori que existe una relación entre dos conceptos y

lo que realizamos con nuestro entorno estadístico es una cuantificación de dicha relación.

En el caso de la minería de datos el proceso es muy distinto: la consulta que se realiza a la base de datos (al *Data Warehouse*) busca relaciones, por ejemplo, entre parejas de productos que son adquiridos por una misma persona en una misma compra. De esa información, el sistema deduce, junto a otras muchas, la afirmación anterior. Como podemos ver, en este proceso se realiza un acto de descubrimiento de conocimiento real, puesto que no es necesario ni siquiera sospechar la existencia de una relación entre estos dos productos para encontrarla.

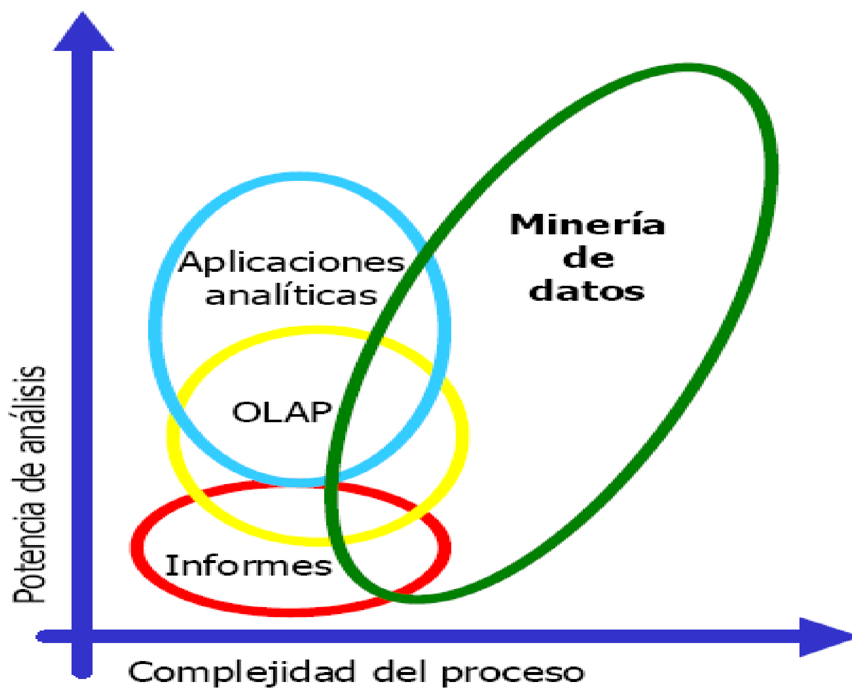

Figura 1.1 Gráfico esquemático de las ventajas de la minería de datos

En la **Figura 1.1** se detallan las ventajas de la minería de datos frente a otros métodos estadísticos, como los informes o las aplicaciones analíticas. Se puede observar de la figura que la minería de datos se vuelve más eficiente frente al resto de métodos según aumenta la complejidad del proceso.

1.2 Aplicación del problema

El nombre de *Data Mining* deriva de las similitudes entre buscar información valiosa de negocios en grandes bases de datos. Por ejemplo: encontrar información de la venta de un producto entre grandes cantidades de datos almacenados o minar una montaña para encontrar una veta de metales valiosos. Ambos procesos requieren examinar una inmensa cantidad de material, o investigar inteligentemente hasta encontrar exactamente dónde residen los valores. Dadas bases de datos de suficiente tamaño y calidad, la tecnología de *Data Mining* puede generar nuevas oportunidades de negocios al proveer estas capacidades:

• Predicción automatizada de tendencias y comportamientos. *Data Mining* automatiza el proceso de encontrar información predecible en grandes bases de datos. Preguntas que tradicionalmente requerían un intenso análisis manual, ahora pueden ser contestadas directa y rápidamente desde los datos. Un típico ejemplo de problema predecible es el marketing orientado a objetivos (*targeted marketing*). *Data Mining* usa datos en *mailing* promocionales anteriores para identificar posibles objetivos para maximizar los resultados de la inversión en futuros *mailings*. Otros problemas predecibles incluyen pronósticos de problemas financieros futuros y otras formas de incumplimiento, e identificar segmentos de población que probablemente respondan similarmente a eventos dados.

• Descubrimiento automatizado de modelos previamente desconocidos. Las herramientas de *Data Mining* barren las bases de datos e identifican modelos previamente escondidos en un sólo paso. Otros problemas de descubrimiento de modelos incluye detectar transacciones fraudulentas de tarjetas de crédito e identificar datos anormales que pueden representar errores de tipeado en la carga de datos.

Las técnicas de *Data Mining* pueden redituar los beneficios de automatización en las plataformas de *hardware* y *software* existentes y pueden ser implementadas en sistemas nuevos a medida que las plataformas existentes se actualicen y sean desarrollados nuevos productos. Cuando las herramientas de *Data Mining* son implementadas en sistemas de procesamiento paralelo de alta prestaciones, pueden analizar bases de datos masivas en minutos. El procesamiento más rápido significa que los usuarios pueden automáticamente experimentar con más modelos para entender datos complejos. La alta velocidad hace que sea práctico para los usuarios analizar inmensas cantidades de datos. Las grandes bases de datos, a su vez, producen mejores predicciones.

Algunos ejemplos que muestran la eficacia de estas técnicas son, por ejemplo los siguientes casos de uso:

- **Detección de fraudes en las tarjetas de crédito.** En 2001, las instituciones financieras a escala mundial perdieron más de 2.000 millones de dólares estadounidenses en fraudes con tarjetas de crédito y débito. El *Falcon Fraud Manager* es un sistema inteligente que examina transacciones, propietarios de tarjetas y datos financieros para detectar y mitigar fraudes. En un principio estaba pensado, en instituciones financieras de Norteamérica, para detectar fraudes en tarjetas de crédito. Sin embargo, actualmente se le han incorporado funcionalidades de análisis en las tarjetas comerciales, de combustibles y de débito. El sistema *Falcon* ha permitido ahorrar más de seiscientos millones de dólares estadounidenses cada año y protege aproximadamente más de cuatrocientos cincuenta millones de pagos con tarjeta en todo el mundo, aproximadamente el sesenta y cinco por ciento de todas las transacciones con tarjeta de crédito.

- **Descubriendo el porqué de la deserción de clientes de una compañía operadora de telefonía móvil.** Este estudio fue desarrollado en una operadora española que básicamente situó sus objetivos en dos puntos: el análisis del perfil de los clientes que se dan de baja y la predicción del comportamiento de sus nuevos clientes. Se analizaron los diferentes históricos de clientes que habían abandonado la operadora (12,6%) y de clientes que continuaban con su servicio (87,4%). También se analizaron las variables personales de cada cliente (estado civil, edad, sexo, nacionalidad, etc.). De igual forma se estudiaron, para cada cliente, la morosidad, la frecuencia y el horario de uso del servicio, los descuentos y el porcentaje de llamadas locales, interprovinciales, internacionales y gratuitas. Al contrario de lo que se podría pensar, los clientes que abandonaban la operadora generaban ganancias para la empresa; sin embargo, una de las conclusiones más importantes radicó en el hecho de que los clientes que se daban de baja recibían pocas promociones y registraban un mayor número de incidencias respecto a la media. De esta forma se recomendó a la operadora hacer un estudio sobre sus ofertas y analizar profundamente las incidencias recibidas por esos clientes. Al descubrir el perfil que presentaban, la operadora tuvo que diseñar un trato más personalizado para sus clientes actuales con esas características. Para poder predecir el comportamiento de sus nuevos clientes se diseñó un sistema de predicción basado en la cantidad de datos que se podía obtener de los nuevos clientes comparados con el comportamiento de clientes anteriores.

1.3 Métodos de resolución

En este apartado se dará una explicación teórica acerca de los algoritmos y procesos que hemos utilizado para resolver el problema que nos plantea el proyecto.

1.3.1 ¿Que son las heurísticas y las meta-heurísticas?

En computación, dos objetivos fundamentales son encontrar algoritmos con buenos tiempos de ejecución y buenas soluciones, usualmente las óptimas. Una **heurística** es un algoritmo que abandona uno o ambos objetivos; por ejemplo, normalmente encuentran buenas soluciones, aunque no hay pruebas de que la solución no pueda ser arbitrariamente errónea en algunos casos; o se ejecuta razonablemente rápido, aunque no existe tampoco prueba de que siempre será así. Las heurísticas generalmente son usadas cuando no existe una solución óptima bajo las restricciones dadas (tiempo, espacio, etc.), o cuando no existe del todo.

A menudo, pueden encontrarse instancias concretas del problema donde la heurística producirá resultados muy malos o se ejecutará muy lentamente. Aún así, estas instancias concretas pueden ser ignoradas porque no deberían ocurrir nunca en la práctica por ser de origen teórico. Por tanto, el uso de heurísticas es muy común en el mundo real.

Una **meta-heurística** es un método heurístico para resolver un tipo de problema computacional general, usando los parámetros dados por el usuario sobre unos procedimientos genéricos y abstractos de una manera que se espera eficiente. Normalmente, estos procedimientos son heurísticos. El nombre combina el prefijo griego "meta" ("más allá", aquí con el sentido de "nivel superior") y "heurístico" (de ευρισκειν, *heuriskein*, "encontrar").

Las meta-heurísticas, generalmente, se aplican a problemas que no tienen un algoritmo o heurística específica que dé una solución satisfactoria; o bien cuando no es posible implementar ese método óptimo. La mayoría de las meta-heurísticas tienen como objetivo los problemas de optimización combinatoria, pero por supuesto, se pueden aplicar a cualquier problema que se pueda reformular en términos heurísticos, por ejemplo en resolución de ecuaciones booleanas.

Las meta-heurísticas que utilizaremos serán VNS y GRASP.

1.3.2 GRASP

Una meta-heurística **GRASP** (*Greedy Randomized Adaptive Search Procedure*) es un algoritmo comúnmente aplicado a problemas de optimización combinatoria. Como diversos métodos constructivos, la aplicación del GRASP consiste en crear una solución inicial y después efectuar una búsqueda local para mejorar la calidad de la solución. Se diferencia de otros métodos en la generación de esa solución inicial, basada en las tres primeras iniciales de sus siglas en inglés: voraz (*Greedy*), aleatoria (*Randomized*) y adaptativa (*Adaptive*).

Mientras otros algoritmos como la búsqueda tabú y los algoritmos genéticos utilizan estrategias de gran énfasis en la búsqueda local, GRASP usa un constructivo para generar la mejor solución y, después, utilizar la búsqueda local sólo para pequeñas mejoras.

La estrategia de construcción de una solución en GRASP consiste en la definición de un criterio de evaluación de los elementos que pueden ser insertados en un conjunto que, al final del proceso, será una solución para el problema de optimización que se pretende resolver. Ese criterio se adapta a la solución ya construida, de forma que la valoración de los elementos cambia durante la construcción de la solución. Sin embargo, ese criterio no es tomado como referencia absoluta para la decisión del próximo elemento a ser insertado, habiendo una elección aleatoria entre los mejores elementos cada iteración.

Una solución para un problema de optimización combinatoria es visto en GRASP como un conjunto con elementos que atiendan a todas las restricciones existentes en el problema. Se comienza con un conjunto vacío, y son insertados elementos en ese conjunto hasta que represente una solución viable para el problema.

Cada iteración, todos los elementos **candidatos** son evaluados según una función voraz que mide el beneficio de la inserción de ese elemento para la construcción de la solución. La medida de ese beneficio es miope, por ser una evaluación imprecisa de cómo la inserción de un elemento puede intentar obtener una

solución de mejor calidad, sea en número de elementos necesarios, en el impacto en la función objetivo del problema u otra métrica cualquiera.

Una vez realizada esa valoración, se construye la **RCL** (*Restricted Candidate List*): una lista conteniendo los elementos con mejor valor en la función voraz, pudiendo su tamaño ser definido por un parámetro absoluto como un número de elementos que deban existir en la lista, o por un porcentaje de tolerancia entre el valor del mejor elemento encontrado y el valor de un elemento que puede estar en la lista. De esa lista, se escoge un elemento aleatoriamente para ser insertado en la solución.

Al final de cada iteración, todos los elementos candidatos dentro de la RCL son reevaluados por una función de evaluación definida para escoger uno de los candidatos, tomándose como referencia el último elemento tomado de la anterior iteración.

1.3.3 Búsqueda Local

Los algoritmos de búsqueda local parten de una solución inicial, y, aplicándole operadores de movimiento, la van alterando; si la solución alterada es mejor que la original, se acepta, si no lo es, se vuelve a la inicial. El procedimiento se repite hasta que no se consigue mejora en la solución.

Los algoritmos que se usarán para intentar mejorar la solución proporcionada por los constructivos serán de añadir, quitar o intercambiar las columnas que no hayan sido seleccionadas en la solución constructiva.

1.3.4 Regla de *los k-vecinos más cercanos*

El método de selección que utiliza el constructivo para elegir qué columna es mejor en cada momento para ser seleccionada se define como la regla de *los k-vecinos más cercanos*. Esta regla consiste en clasificar una instancia de la que conocemos su clase, dentro de un conjunto de instancias previamente clasificadas, y calculando a qué instancia del conjunto clasificado, más se acerca o se parece, la instancia que queremos clasificar.

La regla de *los k-vecinos más cercanos* puede ser usada como referencia para otros clasificadores, ya que siempre provee un rendimiento razonable en la mayoría de las aplicaciones. La regla se puede describir como sigue:

1. Definir una medida de distancia entre puntos.

2. Calcular las distancias del punto a clasificar, $\mathbf{x}_0$, a todos los demás puntos de entrenamiento.

3. Seleccionar los *k* puntos muestrales más próximos al que se pretende clasificar.

4. Calcular la proporción en que los *k* puntos pertenecen a cada una de las poblaciones.

5. Clasificar el punto $\mathbf{x}_0$ en la población con mayor frecuencia entre los *k* puntos.

El desarrollo de la regla supone que un punto de observación debe estar situado cerca de los demás puntos muestrales que pertenecen a la misma clase en el espacio de entrenamiento.

La **Figura 1.2** describe el proceso que se sigue al evaluar el conjunto original de las características. Como indica la figura, las características son introducidas en un árbol de decisión para ser después evaluadas por un algoritmo genético. Este da paso al uso de la regla de *los k-vecinos más cercanos*. Si, tras esta evaluación, hemos conseguido encontrar la solución que buscamos, paramos y validamos la solución. En cambio si no ocurre esta condición de parada, volvemos a evaluar las características con el algoritmo genético.

Podemos hacer un esquema de los métodos de resolución utilizados:

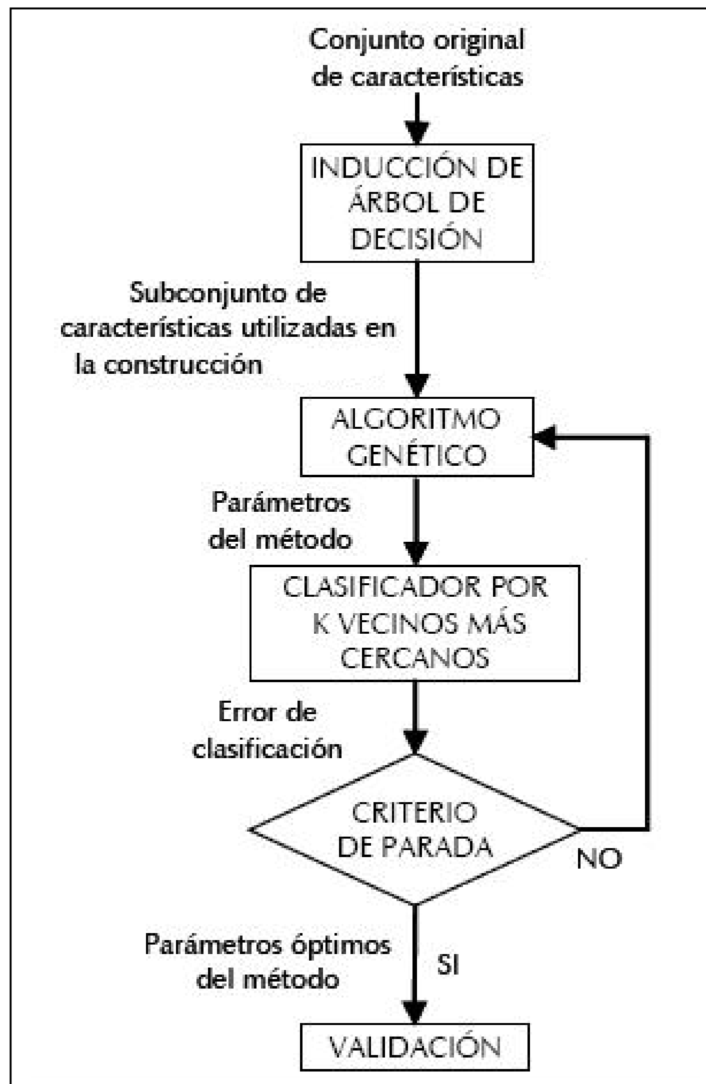

Figura 1.2 Esquema de resolución

Capitulo 2

Descripción algorítmica

En este capítulo, se describen los algoritmos que se han desarrollado para resolver el problema de *Selección de características* planteado, así como una descripción del proceso de construcción de estos algoritmos.

La vida real nos descubre que manejamos mucha cantidad de información que se almacena en bases de datos. Esta gran cantidad de información dificulta la construcción, en un tiempo razonable, de un modelo para la toma de decisiones. Es así, que surge la idea de intentar reducir estas bases de datos para, tanto facilitar la interpretación de cualquier persona, como para minimizar el tiempo de procesamiento.

A través de la selección de características, podemos llegar a conseguir el objetivo que nos planteamos. Es decir, la reducción de datos dentro de un tiempo razonable. Para llegar a conseguir este objetivo, planteamos dentro del proyecto dos posibles alternativas:

- Partiendo de una base de datos sin ninguna característica, vamos añadiendo posibles de éstas que podrían mejorar la calidad de la información que contiene la base de datos.
- Partiendo de una base de datos completa, vamos eliminando características que mejoren la calidad de información que contiene la base de datos.

Estas dos soluciones serán ejecutadas dentro de un algoritmo meta-heurístico GRASP, que ejecutará 100 veces, estas dos construcciones de solución y dará como resultado la mejor de ellas.

A estas dos posibles soluciones de reducción de datos que planteamos, podemos aplicar una serie de métodos heurísticos y meta-heurísticos que posiblemente optimicen la solución inicial obtenida.

Los métodos *Local Search* son algoritmos heurísticos que se utilizan para resolver problemas de optimización. En la ejecución del proyecto llevaremos a cabo tres de estos algoritmos de optimización que son detallados en la figura de más abajo: *borraC, aniadeC e Intercambio.*

Por último, se utiliza un algoritmo meta-heurístico, método más agresivo para intentar conseguir una mejor solución. En este caso usaremos *VNS (Variable Neighborhood Search).*

	Algoritmo	Descripción
Constructivos	Constructivo	Construye la mejor solución a partir de una tabla vacía.
	Destructivo	Construye la mejor solución a partir de una tabla con todos los datos.
Mejoras	borraC (*LocalSearch, LS1*)	Borra columnas de la tabla para intentar mejorar los resultados.
	aniadeC (*LocalSearch, LS2*)	Añade columnas a la tabla para intentar mejorar los resultados.
	Intercambio (*LocalSearch, LSIntercambio*)	Intercambia columnas entre las que están puestas y otras que están sin poner.
Meta-heurísticos	GRASP	Se construyen soluciones mediante Constructivo y Destructivo, devolviendo la mejor encontrada.
	VNS	Se busca mejorar la solución mediante el intercambio simultáneo y aleatorio de varias columnas.

Figura 2.1 Algoritmos del proyecto.

La **Figura 2.1** muestra los algoritmos que se han implementado en el proyecto. Los constructivos, que son los métodos que van a construir la solución inicial. Las mejoras, que son los métodos que tras haber aplicado un constructivo mejoran la solución. Por último, los algoritmos meta-heurísticos más agresivos a la hora de buscar una solución o mejorarla.

2.1 Planteamiento del problema

En este apartado se explicarán los algoritmos implementados para realizar la construcción de una tabla reducida. Para construir una tabla reducida a partir de una tabla con muchas instancias de distintas clases, podemos seguir dos estrategias:

· A partir de una copia de la tabla original, vamos quitando columnas de la misma para tratar de mejorar los resultados.

· Construimos una tabla directamente seleccionando las columnas que mejoren el resultado.

Dentro de cada construcción de la solución, debemos de tener una técnica de elección que vaya, de alguna manera, seleccionándonos qué columnas son mejores o peores para construir una solución localmente óptima. Para ello la otra técnica que se ha utilizado para la selección de columnas consiste en buscar las columnas de la tabla original que mejoren el porcentaje de calidad o *fitness* de la tabla reducida. Para añadir una columna nueva a la tabla reducida con esta técnica, seleccionaremos aleatoriamente una de las instancias que mejoren el *fitness*, pero con restricciones. Para ello realizaremos una lista restringida de candidatos (*Restricted Candidate List*, RCL), en la que introduciremos estas columnas que restringiremos por calidad, es decir, sólo nos quedaremos con las instancias que superen un umbral de calidad determinado, que definiremos de la siguiente manera:

$$Umbral = peorfitness + \beta * (mejorfitness - peorfitness)$$

El valor β será un número real entre 0 y 1, que dependiendo del valor que le demos, se pueden dar diferentes interpretaciones a la fórmula:

- Si elegimos que $\beta = 0$ el umbral será el peor *fitness*, con lo que realizaríamos una selección totalmente aleatoria.

- Si $\beta = 1$, solo podríamos elegir la instancia con el mejor *fitness* (que sería totalmente voraz y determinista). La fórmula quedaría así:

En nuestro algoritmo elegiremos el valor de β a 0'5, para que el umbral sea un valor medio entre la elección aleatoria y voraz.

Una vez que conocemos la forma que tenemos de resolver el problema que se nos ha planteado, podemos describir, de forma global, cuál será el pseudocódigo con el que trataremos de encontrar la mejor solución para nuestro problema.

```
Abre archivo;
Lee num columnas y num filas
Lee datos;
Construye matriz;

Aplico metodo grasp{

        Aplico metodo_constructivo;

}

Presento resultados para ese archivo;
```

Figura 2.2 Pseudocódigo del proyecto.

La **Figura 2.2** describe el pseudocódigo del proyecto. El primer paso es abrir el archivo y procesar los datos que éste contiene. Leemos las columnas y las filas que la forman para poder construir una matriz de esas dimensiones. Después aplicamos un método GRASP y un método constructivo. Cuando el proceso ha terminado presentamos los resultados.

2.2 Algoritmos constructivos

2.2.1 Añadir Columnas

En este primer planteamiento, la solución se va ir generando a partir de de una matriz vacía a la que vamos a ir añadiendo columnas de la base de datos original y comprobando con cuál de estas columnas mejora el *fitness* tras aplicar a este proceso un método *greedy*.

Este método *greedy* va a seleccionar, entre todos los posibles candidatos que mejoren los resultados del *fitness*, uno de ellos al azar.

Cuando se ha elegido un candidato se vuelve a repetir el proceso de añadir columnas hasta que no existe ningún candidato que satisfaga la mejora del *fitness*.

```
SiCondicion{

        RecorreColumnas{
                añadePosibleColumna();
                CalculaFitnessPoniendola();
                GuardaFitnessYBorraColumna();
        }
        SelecciónDeColumnaMedianteMetodoGreedy();
        SiMejoraFitness -> CondicionTrue y AñadeColumnaASolucion();

        }
```

Figura 2.3. Pseudocódigo del constructivo que añade columnas

La **Figura 2.3** muestra el pseudocódigo del constructivo que se basa en añadir columnas a la matriz inicial de características. Mientras la condición de salida no se cumpla, se va recorriendo las columnas de la matriz y se van evaluando cada posible columna. Para cada columna se calcula su *fitness*. Cuando se han evaluado todas las columnas, se selecciona una de ellas a través de un método *greedy*. Después, se evalúa si ha mejorado el *fitness,* se añade la columna a la solución y se continua o no con la ejecución.

2.2.2 Borrar Columnas

En un segundo planteamiento, la solución va a generarse a partir de una matriz completa con todos los datos de la base de datos. A esta tabla vamos a irle quitando columnas en función de una mejora del *fitness*. Igual que en el planteamiento primero, vamos a aplicar un método *greedy* a los candidatos que mejoren nuestro *fitness*.

```
SiCondicion{

        RecorreColumnas{
                eliminaPosibleColumna();
                CalculaFitnessQuitandola();
                GuardaFitnessYReponeColumna();
        }
        SelecciónDeColumnaMedianteMetodoGreedy();
        SiMejoraFitness -> CondicionTrue y AñadeColumnaASolucion();

        }
```

Figura 2.4. Pseudocódigo del constructivo que borra columnas

La **Figura 2.4** muestra el pseudocódigo del algoritmo constructivo que va eliminando columnas para encontrar una solución. Mientras no se cumpla la condición, el algoritmo va recorriendo las columnas y quitándolas de la matriz de características. Todas las columnas son evaluadas y se obtiene su *fitness*. Después se evalúan a través de un método *greedy* que elige una de ellas. Si tras esta ejecución el *fitness* mejora, añadimos la columna a la solución y continuamos con la evaluación del resto de columnas.

2.3 Búsquedas locales

Las búsquedas locales son algoritmos que parten de una solución ya construida, en este caso nuestro mediante constructivos, y la van mejorando progresivamente. Los procedimientos realizan en cada paso un avance que mejora la solución. El algoritmo finaliza cuando, para una solución, que ha podido ser modificada a partir de la inicial, no existe una solución que lo mejore.

En este caso hemos llevado a cabo tres tipos de búsquedas locales. En el caso de que hayamos partido de un método constructivo que haya ido añadiendo columnas a una matriz vacía, aplicaremos un método de mejora que irá borrando columnas para intentar mejorar la solución propuesta.

Por otro lado si hemos aplicado un método constructivo que ha ido quitando columnas a una matriz que originalmente tenía todos los datos, aplicaremos un método de mejora de ir añadiendo columnas para intentar mejorar la solución propuesta por el algoritmo constructivo.

El último método de búsqueda local que hemos aplicado a ambos algoritmos constructivos, es el de intercambio. Este algoritmo intenta ir intercambiando columnas que estén en la tabla de datos, por otras que se hayan descartado en la solución inicial de los constructivos, para así intentar mejorar el *fitness*.

2.3.1 Borrar Columnas

La finalidad de los algoritmos *Local Search* es mejorar la solución propuesta por los constructivos. Es así, que esta mejora intentará borrar o deseleccionar columnas que estén dentro de la solución inicial mientras la calidad de la base de datos mejore. Este proceso se ejecuta siempre que la solución se haya construido a partir de la inserción de columnas en la base de datos. No tiene sentido aplicarlo a una base de datos que se haya construido borrando columnas pues sería un proceso redundante.

El proceso se muestra en pseudocódigo en la **Figura 2.5**:

```
MientrasMejoreFitness{

        RecorreColumnasQueNoEstenEnSolucion{
                eliminaPosibleColumna();
                CalculaFitnessQuitandola();
                GuardaFitnessYReponeColumna();
        }
        BuscaMejorFitnessAlQuitarColunma();
        SiMejoraFitness -> CondicionTrue y AñadeColumnaASolucion();

        }
```

Figura 2.5. Pseudocódigo del método de mejora de borrar columnas

La **Figura 2.5** muestra el pseudocódigo del método de mejora de borrar columnas. El algoritmo recorre las columnas que no estén en la solución y las elimina de esta. Con cada eliminación temporal de cada columna, evalúa si mejora el *fitness*. Después de evaluar todas, comprueba si con alguna de ellas, mejora el *fitness*. Si mejora, añadirá la columna a la solución y volverá a evaluar el resto de columnas. Este proceso se ejecutará mientras mejore el *fitness*.

2.3.2 Añadir Columnas

Al contrario que en algoritmo que se describió en el apartado anterior, éste parte de la solución propuesta por el algoritmo constructivo de borrar columnas para avanzar un paso en la solución propuesta y encontrar una mejor. Este algoritmo va intentando añadir columnas para mejorar la solución propuesta.

El proceso se muestra en pseudocódigo en la **Figura 2.6**:

```
MientrasMejoreFitness{

        RecorreColumnasQueNoEstenEnSolucion{
                añadePosibleColumna();
                CalculaFitnessPoniendola();
                GuardaFitnessYReponeColumna();
        }
        BuscaMejorFitnessAlPonerColunma();
        SiMejoraFitness -> CondicionTrue y AñadeColumnaASolucion();

        }
```

Figura 2.6. Pseudocódigo del método de mejora de añadir columnas

La **Figura 2.6** muestra el método de mejora de añadir columnas. Este método recorrer las columnas que no están en la solución y las añade temporalmente a la matriz. Con cada una de ellas comprueba el *fitness* que se obtiene. Después de evaluar todas, comprueba si con alguna de ellas mejora la el *fitness*. Si mejora, añade la columna a la solución y vuelve a evaluar el resto de columnas de la matriz.

2.3.3 Intercambio de Columnas

Este algoritmo de búsqueda local, parte de una solución propuesta, tanto por el algoritmo constructivo de ir añadiendo columnas a la matriz o bien de la solución propuesta del algoritmo constructivo de ir borrando columnas de la matriz. En ambos casos, este algoritmo de intercambio de columnas, intenta mejorar la solución propuesta intercambiando columnas que están en la matriz de la solución con columnas que se hayan descartado de la solución.

El proceso se muestra en pseudocódigo en la **Figura 2.7**:

```
RecorreColumnasQueNoEstenEnSolucion{

        SeleccionaColumnaNoEnSolucion();
        CambiaPorOtraColumnaEnSolucion();
        CompruebaSiMejoraFitness();


    SiMejoraFitness -> CondicionTrue e IntercambiaColumnasEnSolucion();

    }
```

Figura 2.7. Pseudocódigo del método de mejora de intercambio de columnas

La **Figura 2.7** muestra el pseudocódigo de mejora al ejecutar el método de intercambio de columnas. El algoritmo recorre cada una de las columnas que no están en la solución y la intercambia por una que si esta. Comprueba el *fitness* que se obtiene por el intercambio de una columna por otra y compara si mejora. Si mejora, añade la columna a la solución y borra la columna por la que se hizo el intercambio.

2.4 GRASP

Tal y como se explico en el Capítulo 1, GRASP es un método basado en la aplicación iterativa de un constructivo y una mejora con los que se construye una solución, pudiendo ser catalogado como método multi-arranque. Puesto que los constructivos y mejoras que tenemos tienen tiempos de computo razonables, este sistema será aplicado repetidas veces (en nuestro caso para los resultados experimentales, 100 veces), construyendo varias soluciones y quedándonos finalmente con la mejor de todas ellas.

A ambos método constructivos hemos empleado este método GRASP, al igual que a sus posibles mejoras. Es decir, como se observará en el capítulo de resultados experimentales, aplicaremos, por una parte el método GRASP a una construcción de ambos tipos (añadir y quitar columnas). Por otro lado, utilizaremos el método GRASP para aplicar un constructivo y sus posibles mejoras. Todo esto se verá con más detalle en el apartado de resultados experimentales.

El pseudocódigo del método GRASP se detalla en la **Figura 2.8**:

```
Mientras (condición de parada no sea satisfecha){
  solución = cree aleatoriamente una solución de forma constructiva();
 solución = búsqueda local(solución);
  si solución es la mejor solución hasta entonces conocida {
    guarda(solución);
            }
}
```

Figura 2.8. Pseudocódigo del método GRASP

La **Figura 2.8** muestra el pseudocódigo del método GRASP. El método crea una solución de forma constructiva y la almacena. Después, a esta solución, le aplica un método de *búsqueda local (Local Search).* Si la solución que se obtiene mejora a una que se tenga almacenada de otras ejecuciones, se almacenará. Este proceso se realizará hasta que se cumpla la condición de parada que hayamos definido.

2.5 VNS

Como se definió en el **Capítulo 1**, VNS es una meta-heurística relativamente nueva que intenta evitar quedar atrapada en óptimos locales cambiando la estructura de la vecindad donde se realiza la búsqueda. Podemos resumir el principio de la operación en lo siguiente:

VNS se basa en un cambio sistemático de las estructuras de vecindades dentro de un procedimiento de búsqueda local.

En el caso de nuestro algoritmo, el algoritmo VNS recibirá una matriz que ha dado como solución uno de los algoritmos constructivos e intentará mejorar la solución propuesta. Para ello, el algoritmo elegirá columnas al azar para quitar e insertar alguna que no esté seleccionada y comprobará que la solución propuesta en ese momento mejora o no el resultado. Si es así, volverá a empezar desde el principio y si la solución no ha mejorado entonces probará a quitar más columnas intentando conseguir una mejor solución.

El pseudocódigo del método VNS se detalla en la **Figura 2.9**:

```
almacenaSolucionInicial()
RecorreVecinos{

        MientrasSolucionInicialnoMejore o FindeVecinos{

                BuscaVecinoAleatorio();
                EncuentraSolucionMejor();
                SiMejora->NuevsSolucionInicial;

        }
}
```

Figura 2.9. Pseudocódigo del método VNS

La **Figura 2.9** muestra el pseudocódigo del método VNS. El algoritmo parte de una solución inicial. El algoritmo toma el primer elemento y busca un vecino para intentar mejorar la solución a través de *búsquedas locales (Local Search)*. Si la solución mejora, añade el vecino y toma la nueva solución como inicial. Este proceso se ejecutará hasta que la solución mejore o el primer elemento no tenga más vecinos. Después repetirá el proceso hasta que no queden más elementos para recorrer.

2.6 Particularizaciones a nuestro problema

La reducción de datos, como se explicó en la introducción, se puede realizar para diferentes tipos de estudios. La idea que subyace siempre es la misma: a partir de una tabla dada, construir una tabla con algunas de sus características más representativas, de manera que la tabla reducida tenga un alto porcentaje de calidad.

En nuestro caso, nos basaremos en la reducción de datos para resolver problemas de *selección de características*. Dependerá de tres valores que indicaran la calidad de la tabla reducida. Estos valores son el porcentaje de acierto, el porcentaje de reducción y el *fitness* o porcentaje de calidad, los cuales se explican a continuación:

- **Porcentaje de aciertos:** El porcentaje de aciertos indica la cantidad de instancias de la tabla original que se han clasificado correctamente a partir de la tabla reducida.

Supongamos que tenemos una tabla con n instancias, y que a partir de esta construimos una tabla tomando m de sus instancias, de manera que $m<<n$. Diremos que el porcentaje de aciertos de la tabla reducida será $(k * 100) / n$, donde k será el número de instancias de la tabla original a las que se les predice correctamente la clase a la que pertenecen comparándolas con las instancias que tengamos en la tabla reducida. Por tanto, el valor de k puede ser como mucho el mismo valor de n.

Cada una de las *n* instancias tendrán un atributo con un valor que indicarán la clase a la que pertenecen. Para tratar de predecir esas clases a partir de las instancias de la tabla reducida, utilizaremos la regla del *vecino más cercano*, es decir, diremos que es de la clase de la instancia que tenga más cerca.

Para calcular la distancia que hay entre dos instancias utilizaremos la distancia euclídea. La **distancia** expresa, en términos generales, la proximidad o lejanía entre dos objetos. Si, supongamos, tenemos puntos en el espacio, la mejor forma de saber cuál de ellos es más cercano a uno tomado como referencia, es a través de la distancia euclídea que los separa.

Supongamos que tenemos una base de datos de entrenamiento con cinco columnas y cuatro filas de la siguiente manera:

Atributo 1	*Atributo 2*	*Atributo 3*	*Atributo 4*	*Clase*
5	*2*	*3*	*4*	*1*
2	*4*	*8*	*2*	*2*
3	*4*	*3*	*2*	*2*
1	*3*	*1*	*3*	*1*

Donde la última columna representa la clase. Al igual tenemos una base de datos de test de dos filas y cinco columnas.

Atributo 1	*Atributo 2*	*Atributo 3*	*Atributo 4*	*Clase*
2	*1*	*3*	*3*	*1*
3	*1*	*4*	*2*	*2*

Cogemos la primera fila de test y la primera fila de entrenamiento y calculamos:

$(C_{1,1}Test-C_{1,1}Entren)^2+(C_{1,2}Test-C_{1,2}Entren)^2+ (C_{1,3}Test-C_{1,3}Entren)^2+(C_{1,3}Test-C_{1,3}Entren)^2$

Al resultado obtenido se le aplica una raíz cuadrada. El resultado se almacena si es la primera comparación, sino lo es, se compara con el que tengamos almacenado y si es menor, se almacena el nuevo resultado junto a la fila de entrenamiento a la que pertenece.
Una vez recorrido con la primera fila de test, todas las filas de la matriz de entrenamiento, mediante la operación explicada más arriba se obtendrán una fila que es la que menor distancia se ha obtenido. Con la fila de entrenamiento y la fila de test se comparan las dos clases. Si coinciden, es un acierto. Si no es un fallo.

Este proceso se repite hasta terminar con todas las filas de test. Este resultado nos dará una cantidad de aciertos.

Figura 2.10. Calculo del vecino más cercano

La **Figura 2.10** muestra un ejemplo de cómo se procede a la hora de calcular la distancia euclídea entre dos instancias. Cada una de las filas de test busca la instancia de la matriz de entrenamiento que menos distancia las separa. Comparan sus clases y si coinciden suman un acierto, sino un fallo.

Calcularemos entonces la distancia de cada instancia de la tabla original con cada instancia de la tabla reducida, y clasificaremos cada una de las instancias de la tabla original con la clase de la instancia de la tabla reducida más cercana. Si esta clase realmente es la clase de la instancia, se acumulará un acierto, en caso contrario no se acumula. Cuando se haya realizado este proceso con todas las instancias, se calcula el porcentaje con la formula expuesta anteriormente.

- **Porcentaje de Reducción:** Este valor representa la cantidad de datos que no se han incluido en la tabla reducida con respecto a la tabla original. Tomando el ejemplo anterior, en el que teníamos la tabla original con n, donde n es la cantidad total de datos (filas por columnas) y la tabla reducida con m, donde m es la cantidad de datos que hay en la tabla como resultado de seleccionar las columnas y no estar presentes en la solución, tal que $m << n$, diremos que el porcentaje de reducción de m con respecto a n será el resultado de:

$$((n - m) / n) * 100.$$

- ***Fitness*** **o Porcentaje de calidad:** Este valor representa el grado de calidad de la tabla reducida. Este valor depende de los otros 2 valores calculados anteriormente, y se resuelve con la siguiente fórmula:

$$Fitness = \alpha * porcAciertos + (1 - \alpha) * porcReduccion$$

Donde α es un valor entre 0 y 1. Con este valor daremos más peso al resultado que más nos interese, ya sea los aciertos o la reducción. En este proyecto se han tenido en cuenta ambos valores por igual, por tanto $\alpha = 0'5$.

Con estos 3 valores determinaremos el grado de calidad de las tablas reducidas que construyamos.

Capítulo 3

Objetivos

El objetivo principal del proyecto es la implementación y comparación de algoritmos que basados en métodos heurísticos y meta-heurísticos resuelvan el problema de **la selección de características**. Aparte de este objetivo global, se ha tenido otra serie de objetivos secundarios:

- **Ampliación de conocimientos:** Familiarizarse con los términos de *heurística* y *meta-heurística* y las estrategias que definen. Además, investigar y conocer el algoritmo GRASP para adaptarlo al problema e implementar un método constructivo y de mejora por este algoritmo. Desarrollar un método constructivo de añadir características y otro constructivo de ir eliminando características. Por último, familiarizarse con el algoritmo VNS y desarrollarlo para el proyecto.

- **Profundizar en el paradigma de la programación orientada a objetos**: Durante el desarrollo de la titulación técnica, se profundiza en mayor medida en el desarrollo de algoritmos estructurados, pero actualmente, gran parte del desarrollo software se realiza con lenguajes orientados a objetos como C++ o Java, así como en el mundo laboral. Con este proyecto se pretende profundizar en el diseño y programación de algoritmos utilizando Java, complementando así la formación adquirida.

- **Código limpio:** Como programadores, este objetivo es fundamental. Durante la codificación del proyecto debe tratarse siempre de tener un código limpio. Para ello, éste debe de estar comentado e indexado correctamente, sin código redundante.

- **Calidad de las abstracciones:** El diseño de las clases debe ser adecuado, de forma que estas sean reutilizables y fácilmente ampliables.

Capítulo 4

Descripción Informática

En este capítulo se explicará el desarrollo del proyecto. Por un lado se describirá el diagrama de Gantt, el ciclo de vida, los requisitos funcionales y no funcionales y por último la descripción del código implementado, así como, sus procedimientos auxiliares que han ayudado a llevar a encontrar una solución óptima para el problema.

4.1 Requisitos funcionales

Los requisitos funcionales son aquellos que describen lo que debe realizar el programa. Para nuestro proyecto han sido los siguientes:

- Nuestro programa debe ser capaz de reducir un conjunto de datos elevado a otro conjunto inferior de datos, de manera que el nuevo conjunto mantenga una buena calidad para clasificar.

- El programa debe ser capaz de clasificar instancias con respecto a un conjunto de datos, de calcular el porcentaje de reducción entre dos conjuntos de diferentes tamaños y el porcentaje de calidad o *fitness* del conjunto reducido en el menor tiempo posible.

- Debe informar del tiempo de ejecución del algoritmo y del mejor resultado encontrado en dicha ejecución en un fichero de informe y por pantalla.

- Debe mostrar información por pantalla al inicio de la ejecución, detallando el número de instancias del fichero, el método de construcción utilizado, el número de columnas que tienen las instancias así como si se ignoró alguna columna al inicio del procesamiento de los datos.

- Los ficheros de la aplicación contendrán una serie de instancias y columnas que puedan ser leídas y almacenadas.

4.2 Requisitos no funcionales

Estos requisitos son aquellos que dicen qué se espera de la ejecución del programa y cómo debe reaccionar:

- Los datos de los ficheros pueden venir separados por distintos caracteres que habrá que reconocer, aunque el formato general de los ficheros es el mismo para todos.

- Debe ser capaz de construir tablas de diferentes tamaños y número de atributos, para que se pueda utilizar para estudios que no sean relacionados con la clasificación de instancias.

- Los algoritmos estarán basados en métodos *heurísticos* y *meta-heurísticos*.

- Los tiempos de ejecución para las instancias propuestas con el enunciado del proyecto deben ser razonables (del orden de los minutos), teniendo en cuenta que, lógicamente, a instancias más complejas mayor tiempo de cómputo.

4.3 Ciclo de vida

El ciclo de vida es un marco de referencia que contiene todos los procesos, actividades y tareas que se realizan durante el desarrollo, la explotación y el mantenimiento del software, desde que éste se define hasta que se deja de utilizar. Para realizar este desarrollo se ha seguido el modelo en espiral. Las actividades de este modelo se conforman en una espiral, en la que cada bucle o iteración representa un conjunto de actividades. Las actividades no están fijadas a priori, sino que las siguientes se eligen en función del análisis de riesgo, comenzando por el bucle interior.

A continuación se detalla en qué consiste cada etapa del ciclo:

A - **Análisis de requisitos:** Se analizan los objetivos que se deben de resolver para dar el siguiente paso o terminar el desarrollo.

B - **Discusión de posibles alternativas:** Aquí se debaten las distintas alternativas para implementar la solución al problema.

C - **Desarrollo:** Se implementa lo decidido en la etapa anterior.

D - **Validación:** Se comprueban los resultados obtenidos con el nuevo desarrollo para comprobar que cumplan los requisitos fijados anteriormente.

E - **Planificación:** Se revisa el proyecto y se planifica el siguiente ciclo.

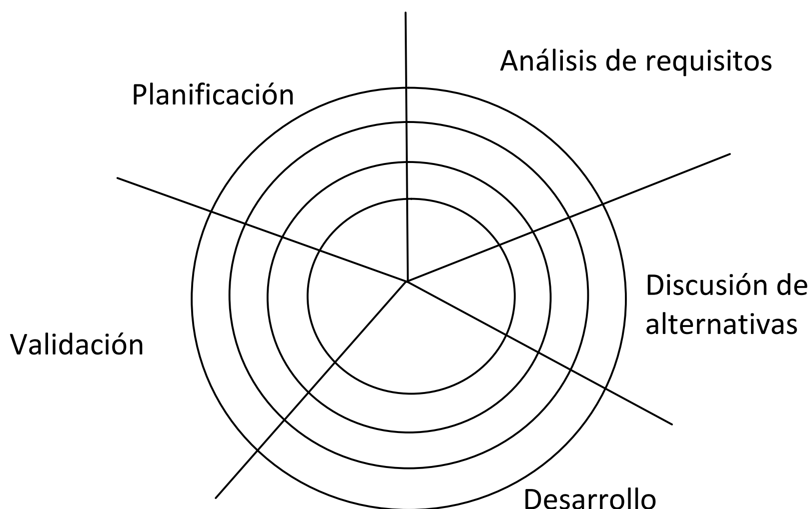

Figura 4.1 Ciclo de vida

La **Figura 4.1** muestra el ciclo de vida del proyecto. Podemos distinguir en él, las cinco etapas por las que ha pasado cada fase del proyecto. Una mayor abertura del ángulo que abarca la etapa, conlleva que el tiempo que se ha tardado en desarrollar es mayor.

4.4 Diagrama de Gantt

El objetivo del diagrama de Gantt es mostrar el tiempo de dedicación previsto para diferentes tareas o actividades a lo largo de un tiempo total determinado. A pesar de no indicar las relaciones existentes entre actividades, la posición de cada tarea a lo largo del tiempo hace que se puedan identificar dichas relaciones e interdependencias. En el diagrama las líneas verticales representan el mes de trabajo:

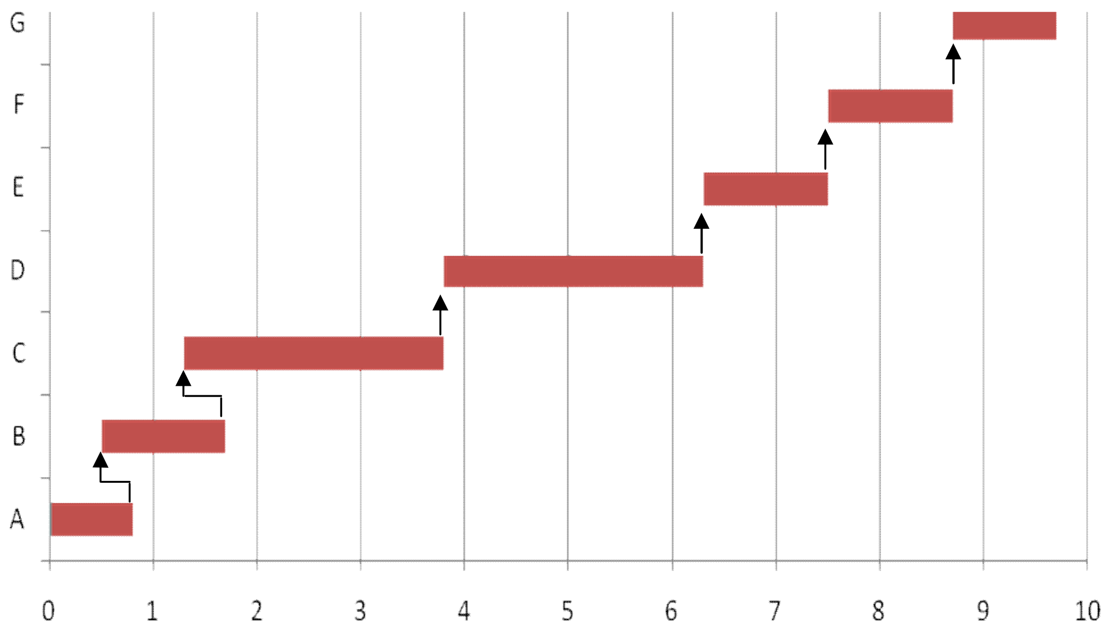

Figura 4.2 Diagrama de Gantt

La **Figura 4.2** muestra el diagrama de Gantt del proyecto. Las lineas granates identifican el tiempo que se ha tardado en desarrollar cada una de las tareas en el tiempo. El eje de abscisas representa el mes de trabajo, durante el cúal, se ha ido desarrollando el proyecto. El eje de ordenadas indica la fase de trabajo, la cual se explica a continuación:

A=Familiarizarse con el problema de la reducción de datos y la *meta-heurística*.

B=Repasar conceptos Java y aprender nuevos conceptos que nos ayudarán en el desarrollo.

C=Recoger datos de los ficheros y familiarizarse con los datos a reducir.

D=Implementar algoritmos constructivos de borrado e inserción de columnas.

E=Implementar el método GRASP a la solución.

F=Implementar métodos de mejora, como insertar, borrado e intercambio de columnas.

G=Implementar algoritmo VNS para la solución.

4.5 Descripción del código

La versión final del proyecto se compone de cinco clases Java que son las que contienen los métodos más importantes de la aplicación. Incluye también otra clase Java donde está el método *main* que lanza el constructor que se quiera usar, en cada caso. Las clases interactúan entre sí para manejar las tablas de datos e intentar reducir y resolver el problema planteado.

La **Figura 4.3** representa el diagrama de clases. Cada uno de los cuadros de algoritmos representa una clase del proyecto. La cabecera de los cuadros indica la clase. El cuerpo de cada cuadro representa los métodos más importantes que podemos encontrar dentro de la clase.

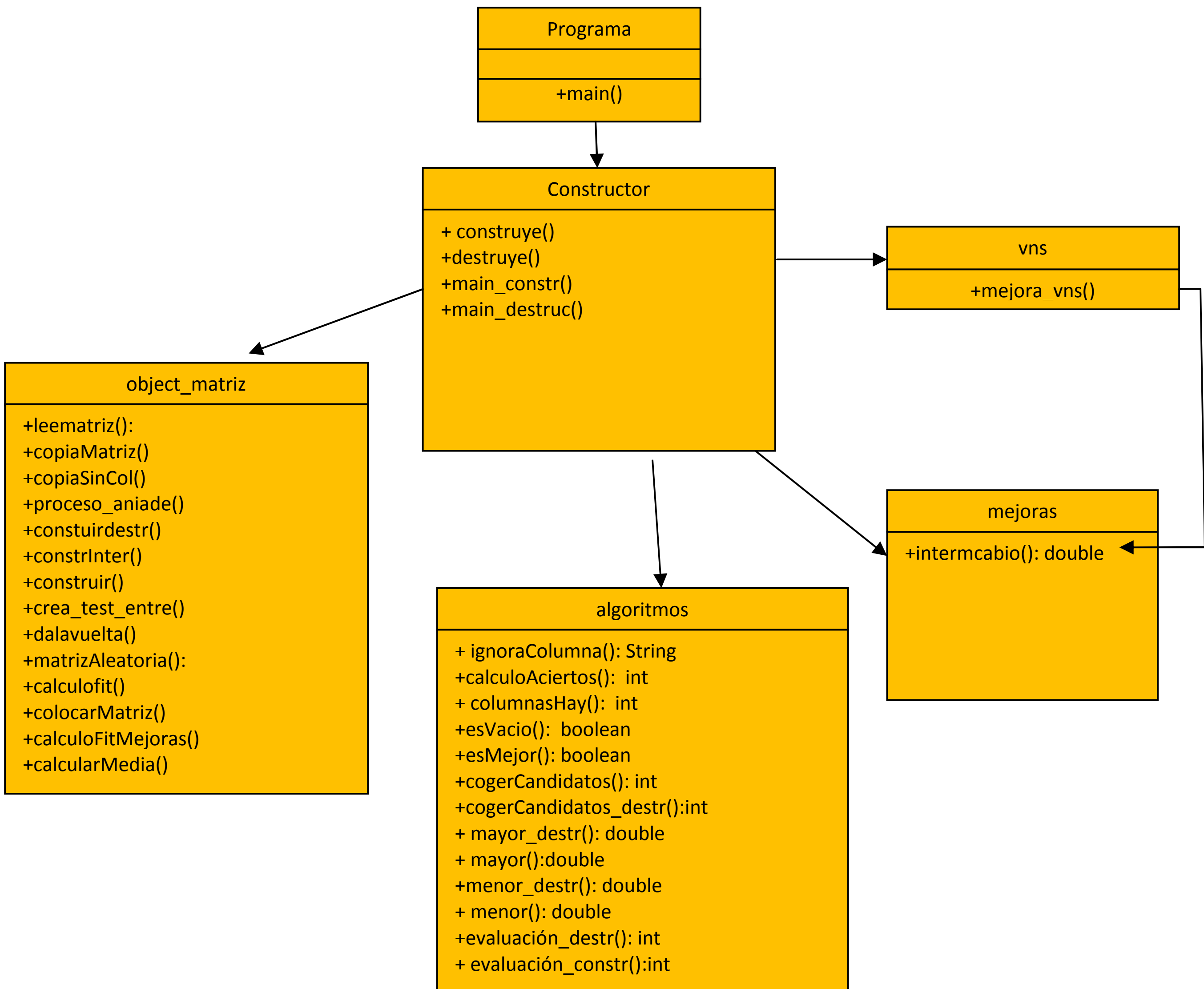

Figura 4.3 Diagrama de clases

A continuación se explica cada una de las clases:

- ***Programa***: En esta clase encontramos el método *main* desde donde se deberá lanzar el proyecto. En esta clase podemos encontrar los dos métodos constructores que podremos ejecutar. Si queremos que se lance el constructor que parte de una matriz vacía y va añadiendo columnas lanzaremos el método `main_constr()`, mientras que si queremos el otro constructor, el cuál parte

de una matriz completa y va eliminando columnas, lanzaremos el `main_destruc()`. Aparte, en este método, se lee la base de datos que se va a utilizar.

- ***Object_matriz:*** Esta clase contiene todos los métodos relacionados con la matriz y sus variables. Así como los métodos que controlan el *array* que contiene las columnas seleccionadas, y todos los métodos para manejar la matriz de datos.

- ***Constructor:*** Contiene el método GRASP y el proceso de construcción de la solución.

- ***Mejoras:*** Contiene el método de intercambio de columnas.

- ***VNS:*** Contiene el método VNS que podemos aplicar a la solución que nos proporciona el constructor.

- ***Algoritmos:*** Contiene todos los métodos auxiliares que no se encuadran dentro de ninguna otra clase que nos ayudan, por ejemplo, a buscar el menor valor dentro de un *array*.

FASE 1. Diseño y puesta en marcha

En la primera fase del desarrollo, era previsible plantear la solución a la lectura de los archivos de datos que se van a reducir, mediante la selección de características que nos hemos planteado. Para ello, la lectura de las instancias y su almacenamiento se realiza dentro de la clase *object_matriz*. La clase *object_matriz*, como se ha indicado anteriormente, tratará todos los métodos relacionados con la matriz de datos y sus atributos relacionados directamente con ella, así como, la parte más importante, el *array* de enteros, que indica si una columna ha sido seleccionada o descartada. En este procedimiento de lectura se hará una lectura del archivo, comprobando las características que éste nos indica (por ejemplo, una columna de la base de datos debe ser ignorada o cuál es la clase) y se almacenará en un *array* de doble dimensión.

Fase 1.1 Lectura y almacenamiento de datos

Como ya se citó anteriormente, empezaremos creando la clase object_matriz, que a la larga será la clase más importante, pues contendrá todos los métodos relacionados con el tratamiento de la matriz de datos que usaremos para encontrar la solución a nuestro problema. Poco a poco iremos añadiendo nuevos métodos a esta clase según vayamos necesitando métodos que nos ayuden.

Object_matriz:

- Propiedades: En esta primera fase contiene los atributos públicos de la matriz de datos, donde almacenaremos la información, que será un *array* de doble dimensión (matriz a partir de ahora para simplificar) de *String*.
- Métodos:
    - *leer_matriz().* Lee el fichero de entrada y almacena los datos en la matriz.
    - *matrizAleatoria().* Normalmente, las bases de datos vienen ordenadas por su clase. Si esto ocurre, nuestras pruebas no tendrían mucha validez pues las comprobaciones las hacemos en grupos de test y de entrenamiento que explicaremos más adelante. Este método reorganiza las instancias de la tabla con aleatoriedad.
    - *colocarMatriz().* En algunos casos, la clase de la matriz no vendrá dado por la última columna de la base de datos, sino que ésta será la primera. En ese caso usamos este método para recolocar la matriz para la correcta ejecución.

- *copiaMatriz()/copiaSinCol()/colocarMatriz().* Estos tres procesos auxiliares, ayudan a recolocar la base de datos en el caso de que, como se indico más arriba, la matriz venga con una columna que nos indica que debe de ser ignorada o la clase está colocada en otro lugar.

Aparte de esta clase, se crea la clase *main*, en la cual irá el método `main()`. Este método `main()` será el encargado de llamar al método `lee_matriz()` para la carga de los datos y a los constructores `main_constr()` y `main_destr()`, que se explicarán más adelante.

Fase 1.2 Elegir la mejor columna para ser eliminada/insertada

Tras el almacenamiento de los datos en la matriz, llega el momento de elegir que columna debemos eliminar o insertar. Para llevar este control, se usa un *array* de enteros donde indicaremos con ‘0’ ó ‘1’ que columnas están seleccionadas. Este *array* de enteros debe tener una dimensión. Esta dimensión viene dada en la lectura del archivo. Cada archivo nos indica el número de columnas que posee cada base de datos y si alguna de ellas debe ser ignorada.

Para la elección de que columna debemos de seleccionar utilizaremos, como ya indicamos, la regla del vecino más cercano. Por ello creamos un método que selecciona la mejor columna en función de cuál de ellas al ser insertada o eliminada conlleva un mejor *fitness*.

Para conseguir este resultado añadimos a la clase *object_matriz*, propiedades y métodos nuevos que necesitaremos para el tratamiento de la matriz, así como una nueva clase *Algoritmos* donde pondremos aquellos métodos que nos devuelven un valor específico sobre el tratamiento de partes de la matriz.

Object_matriz

- Propiedades: *columnas* y *filas*, de tipo *Integer* que, como su propio nombre indica, contiene el número de columnas e instancias que contiene la base de datos.

- Métodos:

  - *proceso_aniade().* En este método se llama a `calculoAciertos()` de la clase *Algoritmos* para calcular cual es el número de aciertos que se consigue mediante la regla del vecino más cercano eliminando o insertando una columna determinada.

  - *main_constr()/main_destr().* Estos métodos se encargan de construir la solución, haciendo las llamadas al método *`proceso_aniade()`*, y va comprobando si el *fitness* mejora. Si mejora, selecciona esa columna y lo indica en el *array* de enteros.

  - *calculoFit():* Se lleva a cabo el cálculo del *fitness* que se ha tenido dependiendo del número de aciertos por cada columna seleccionada. Es decir devolverá un *array* de *double* donde indica en cada una de sus posiciones que *fitness* se ha conseguido con cada columna.

Algoritmos:

- Propiedades: Esta clase no contiene atributos, pues son procesos auxiliares.

- Métodos:

    - *calculoAciertos():* Aquí es donde se implementa directamente el algoritmo del vecino más cercano y devuelve, como su nombre indica, el numero de aciertos que se ha tenido seleccionando una columna determinada.

Fase 2. El tratamiento de las bases de datos

En una primera fase, el resultado de que columna era mejor para ser seleccionada, se hacía comparando dos ficheros de base de datos. Uno de estos ficheros contenía los datos de entrenamiento y otros los de test. Estas instancias de cada uno de los ficheros se almacenaban en una matriz test y una matriz de entrenamiento. Estas dos BD se comparan, cada una de las instancias de la parte de entrenamiento con todas las de test y se busca su vecino más cercano, como ya se explicó en los capítulos anteriores de esta memoria.

Esto supone un cálculo enorme, lo que conlleva un tiempo destacable debido a la cantidad de instancias de cada uno de estos ficheros. Por ello se eligen bases de datos de menor tamaño que se comparan de la siguiente manera: Se dividen el conjunto en 10 partes iguales, luego se realizan 10 iteraciones. En cada una de ellas uno de los subconjuntos ejerce de test y los 9 restantes se unen para formar el conjunto de entrenamiento. El resultado final de los aciertos y *fitness* es la media de lo conseguido en las 10 iteraciones. Este proceso hay que adaptarlo a nuestro proyecto. Lo que conlleva a incluir nuevos métodos y recolocar algunos de los ya existentes.

Object_matriz

- *Métodos:*

    - *crea_test_entre(arrayList).* En este proceso realizaremos la construcción de las dos matrices de test y de entrenamiento. Será un *arrayList* quien indique cuáles son las filas que harán de test y cuáles de entrenamiento.

```
DesdeInicioMatriz hasta FinalMatriz{
        seleccionaInstancia();
If (pertenece aIntervalodeTest){
        añadeATestInstancia();
        }
else{
        añadeAentrenamientoInstancia()
}
```

Figura 4.4 Pseudocódigo crea_test_entre()

- *calcularMedia()* Calcula la media de las iteraciones que hemos ido realizando con las comprobaciones de cada una de las partes de la base de datos que hacían de test y de entrenamiento.
- *construir()/construirdestr()/constrInter().* Estos procesos, serán los encargados, dependiendo del tipo de constructivo que apliquemos, de orientar a la matriz a construir de una determinada forma. Es decir, si es un proceso constructivo de añadir columnas, le dirá que columnas están de momento agregadas y cuáles no.

La **Figura 4.4** muestra el pseudocódigo del algoritmo `crea_test_entre()`. Este método, recorre la matriz de datos y va seleccionando instancias. Si la instancia pertenece al intervalo de valores que engloban la matriz de test, se añade la instancia a la matriz de test, sino a la de entrenamiento.

Llegados a este punto, vamos a simplificar y separar los métodos que construyen la solución para una mayor compresión y lectura y poder reutilizar el código para la implementación de las mejoras. Las mejoras que implementaremos a continuación, no serán sino ejecutar el constructivo contrario al que se utilizó para la primera solución. Para ello dentro de la clase *object_matriz* crearemos dos nuevos métodos:

Object_matriz.java

- *construye(). Este método lleva a cabo la posible selección de una nueva columna en la matriz. En este caso la añadiría a la solución.*
- *destruye(). Igual que el método anterior, construye(), selecciona una posible nueva columna para la matriz . En este caso la eliminaría de la matriz.*

Fase 3. Creación de mejoras

Una vez que están implementados los constructivos llega la hora de crear aquellas mejoras que encontramos más apropiadas para intentar optimizar la solución. Estas tres mejoras, como ya se describieron en los capítulos anteriores, serán: borrar, añadir e intercambiar columnas. Estos procesos se ejecutaran mientras mejoren los resultados. Para ello crearemos una nueva clase denominada *mejoras.java.* En ella encontramos únicamente el método de intercambio, pues al final y al cabo, los métodos de añadir y borrar serán los mismos que construye y destruye respectivamente del proceso constructivo inicial.

- *construye(object_matriz):* El concepto de borrar será el mismo que el de seleccionar columnas e ir eliminándolas de la matriz. Este método será aplicado cuando hayamos ejecutado un constructivo que fuera añadiendo columnas. No tiene mucho sentido aplicarlo a un constructivo que ha ido eliminando columnas pues sería redundante.

- *destruye(objetc_matriz):* Igual que en el caso de borrar. Intenta mejorar la solución propuesta por el constructor, añadiendo columnas a una matriz que ha sido creada a partir del constructivo de eliminar columnas.

- *Intercambio(object_matriz):* Este método se encargara de intentar mejorar la solución intercambiando columnas que se hayan desechado en la primera construcción. Es decir, coge una columna que esté seleccionada y la elimina. Después elige otra que no esté en la matriz y comprueba si con ello mejora o empeora la solución. Si esta mejora, la columna que se selecciono como probable se introduce en la matriz y la que se eliminó queda descartada automáticamente.

Fase 4. Implementación de VNS

Tras el proceso de creación de los métodos anteriores de mejora, podemos implementar un algoritmo meta-heurístico. Para la implementación de este método, y por comprensión del proyecto, se ha creado una clase exclusivamente que contiene este método para no mezclarlo con las mejoras. Este algoritmo, puede decirse, que es una ampliación del método de mejora de intercambio, pues su base se verá sustentada por él. El método VNS implementado recoge las columnas que quedan por insertar en la solución de los constructivos, y en vez de probar a intercambiar sólo una de las columnas seleccionadas por otra de ellas, lo va a intentar por dos. Si esto no mejora la solución, lo intentará volviendo a deseleccionar tres, e introducir una. Este proceso continua hasta que lo ha intentado con la mitad de las columnas restantes que no se han seleccionado originalmente.

Si en algún momento, cuando se ha ejecutado este proceso, la solución mejora, el algoritmo intentará volver a ejecutar un método de *mejora* y después volver a ejecutar el algoritmo VNS. En el siguiente pseudocódigo podemos ver cómo funciona:

```
columnasRestantes=ColSinSeleccionar();
Mientras columnasPruebas < columnasRestantes ó mejoraSolucion{
            DeseleccionoColumnas(numColunmasAQuitar);
            LlamadaIntercambio();
            ComprueboSimejoraSolucion();
            SiMejora  -> mejoraSol=true;
            NoMejora ->RestauroValoresOriginales();
            numColumnasAquitar++;
      }
      Si MejoraSolucion ->LlamadaMejoraCorrespondiente();
```

Figura 4.5 Pseudocódigo del método mejora_vns()

La **Figura 4.5** muestra el pseudocódigo del método `mejora_vns()`. Este método cuenta, en primer lugar, las columnas que quedan sin seleccionar. Mientras que las columnas de prueba, que se van a ir tomando, sean menor que el numero de columnas que están sin seleccionar o no mejore la solución se irá ejecutando el algoritmo. Éste deselecciona tantas columnas como se le indique la variable. Después hace una llamada al método *intercambio*() y comprueba si mejora la solución. Si no mejora restaura los valores originales y aumenta el número de columnas a deseleccionar. Si, en cambio, la solución mejora, el método termina y llama a un método de mejora.

Por lo tanto en la clase *vns* se introduce el siguiente método:

- *mejora_vns(object_matriz):* Como se explico más arriba, contendrá las llamadas y el tratamiento para ejecutar este algoritmo.

Y para la clase *Algoritmos* el nuevo método:

- *columnasHay(object_matriz).* Devolverá el número de columnas restantes que no se han incluido en la solución dada por los constructivos, con o sin haber aplicado métodos de mejora.

Fase 5. Creación del método GRASP

Como ya se comentó en capítulos anteriores, para dar una mayor fiabilidad al compendio de los datos, debemos de introducir un poco de aleatoriedad en la construcción de la solución.

Hasta ahora, para seleccionar que columna era más óptima para ser seleccionada, ya fuera introducida en la matriz o bien eliminada de la matriz, cogíamos la que más beneficiaba a la solución, es decir, la que su *fitness* era mayor que ninguna otra. Al introducir el método GRASP, procederemos de la siguiente manera:

Introducimos todas las columnas que mejoran el *fitness* dentro de una lista. A esta lista la llamaremos lista de candidatos. A esta lista de candidatos, le introduciremos una restricción para incluir a cada uno de estos candidatos en una lista restringida (*RCL*). Esta restricción que impondremos será la siguiente:

$RCL = \{ e \in CL \ / \ eval(e) >= th\}$

$Th = min(eval(e)) + \alpha (max(eval(e)-min(eval(e))$

eval(e) = fitness que se obtiene al seleccionar una columna

Una vez que hemos obtenido esta lista restringida de candidatos, según las condiciones que hemos impuesto, seleccionaremos una de ellas al azar de la lista. Para la implementación de estos métodos, incluiremos dentro de la clase Java, *Algoritmos.java,* una serie de métodos estáticos que nos devolverán la columna elegida de la lista restringida de candidatos. Todos estos métodos se explican a continuación:

Algoritmos.java

- *evaluacion_constrc(listaColumnas,valoresFitness,mejofitnesactual,columnastotales).* Este método será el encargado de llevar el peso y de hacer las llamadas a otros métodos menores, como calcular el mayor de los *fitness* o el menor. Devolverá al final de su ejecución, la columna que se ha seleccionada o si no existe ninguna que mejore la construcción de la matriz. El método `evaluación_destrs()` es igual que este método, salvo que se utiliza para cuando la construcción de la matriz es a base de eliminar columnas.

```
evaluaSiHayColumnasCandidatosAun();
SiHay{
        Elige fitnessMayor();
        Elige fitnessMenor();
        calculaEvalucion();
        EscogeCandidato();

retorna Candidatos;
        }
```

Figura 4.6 Pseudocódigo de evalución_constr()

La **Figura 4.6** muestra el pseudocódigo de `evaluación_constr()`. Este método evalúa si aún quedan columnas por seleccionar. Si existen, elige entre todos los posibles candidatos el mayor *fitness*. Después, elige, de nuevo entre los candidatos, el menor *fitness*. Con estos datos, escoge un candidato que satisfaga las condiciones y lo retorna

.

- *Mayor_constr()/Mayor_destruc():* Devuelve el mayor *fitness* de todos los posibles candidatos que superen el *fitness* actual.

- *Menor_constr()/Menor_destruc():* Devuelve el menor *fitness* de todos los posibles candidatos que superen el *fitness*.

Para una mayor claridad de estos dos métodos podemos explicarlos con un claro ejemplo de su funcionamiento:

C1	C2	C3	C4	C5	C6	C7
42,3	**45,7**	**48,4**			**48,2**	**46,6**

Figura 4.7 Tabla de valores de ejemplo de fitness

La **Figura 4.7** representa una posible tabla de valores de *fitness*. La primera fila de la tabla identifica la columna que se ha seleccionado en este caso para ser tratada, bien añadida o eliminada, mientras que la segunda fila de la tabla representa el *fitness* que se obtiene al seleccionar esa columna. En el caso de las columnas cuatro y cinco que no aparece ningún valor, es que son las columnas que actualmente están seleccionadas.

Estos valores son evaluados primero por el método `mayor_constr()` o `mayor_destruc()`, dependiendo del constructivo que estemos evaluando. Supongamos que el *fitness* actual que tenemos es de 46,1. El método devolverá un valor de 48,4 referente a la columna 3. Es el mayor valor de todos aquellos que superan el *fitness*.

Después se hace la llamada al método de `menor_constr()` o `menor_destruc()` como se indicó arriba. En este caso, al pasarle estos valores, y suponiendo que el *fitness* actual es de 46,1, devolverá el valor de 46,6. Es el menor valor de todos aquellos que superan el *fitness* actual.

Con estos valores se hace la evaluación que se describió más arriba para la lista restringida de candidatos y en ella se introduce los candidatos que cumplan esa condición. Con ello se crea el método para su evaluación.

Algoritmos:

- *Métodos:*
    - *cogerCandidatos().* Devuelve la columna, seleccionada al azar, que está dentro de la lista de candidatos. En caso de que no haya ninguna que cumpla la condición, devuelve un digito de control.

```
RecorreListaCandidatos{
        Si CumpleCondicionRCL{
                AñadeALista();
                        }
                }
EscogeAlAzarEntreListaRCL();
DevuelveColumnaSeleccionada;
```

Figura 4.8 Pseudocódigo de cogerCandidatos()

La **Figura 4.8** muestra el pseudocódigo del método `cogerCandidatos()`. El algoritmo recorre la lista de candidatos. Si encuentra uno que cumple la condición para están dentro de la *RCL*, se añade. Cuando ha comprobado todos, escoge uno al azar y devuelve el seleccionado.

Una vez llevado a cabo esta selección de columna, comprobamos que no sea el dígito de control, y si no lo es, añadimos la columna seleccionada a la matriz y modificamos el valor del *fitness* mayor que tenemos.

Capítulo 5

Durante este capítulo, se explicará cual es el hardware y software utilizado para realizar el proyecto. Además, se mostrará cual es el formato de entrada y salida de los archivos de texto, usados para obtener los datos. Por último, se presentan los resultados experimentales de la ejecución del proyecto.

5.1 Hardware

Los datos presentados a continuación son las características hardware del equipo donde se ha ejecutado el proyecto.

Estos datos son importantes a la hora de comparar los tiempos obtenidos en el apartado de **Experimentos**, las tablas comparativas, si los quisiéramos comparar con otros resultados experimentales, otros enfoques a la hora de resolver el problema o trabajos futuros.

- Intel Core 2 CPU
- 2,66 GHz
- 3,5 GB de RAM

5.2 Tecnología utilizada

Para la realización del proyecto, se ha necesitado dar uso de dos tecnologías que se explican a continuación.

5.2.1 Java

Java es un lenguaje de programación orientado a objetos desarrollado por *Sun Microsystems* a principios de los años 90. El lenguaje en sí mismo toma mucha de su sintaxis de *C* y *C++*, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria.

La implementación original y de referencia del compilador, la máquina virtual y las bibliotecas de clases de Java fueron desarrolladas por *Sun Microsystems* en 1995. Desde entonces, Sun ha controlado las especificaciones, el desarrollo y evolución del lenguaje a través del *Java Community Process*, si bien otros han desarrollado también implementaciones alternativas de estas tecnologías de Sun, algunas incluso bajo licencias de software libre.

Entre noviembre de 2006 y mayo de 2007, *Sun Microsystems* liberó la mayor parte de sus tecnologías Java bajo la licencia *GNU GPL*, de acuerdo con las especificaciones del *Java Community Process*, de tal forma que prácticamente todo el Java de Sun es ahora software libre (aunque la biblioteca de clases de *Sun* que se requiere para ejecutar los programas *Java* aún no lo es).

La elección de *Java* frente a otros lenguajes de programación fue elegida por las siguientes razones:

- La independencia de la plataforma que nos proporciona la programación en Java.
- Profundizar en los conocimientos adquiridos sobre la programación orientada a objetos y desarrollarlos.
- Permite crear programas modulares y códigos reutilizables.

5.2.2 Eclipse

Eclipse es un entorno de desarrollo integrado de código abierto multiplataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Esta plataforma, típicamente ha sido usada para desarrollar entornos de desarrollo integrados (del inglés IDE), como el IDE de *Java* llamado *Java Development Toolkit* (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse (y que son usados también para desarrollar el mismo Eclipse).

5.3 Descripción de los ficheros

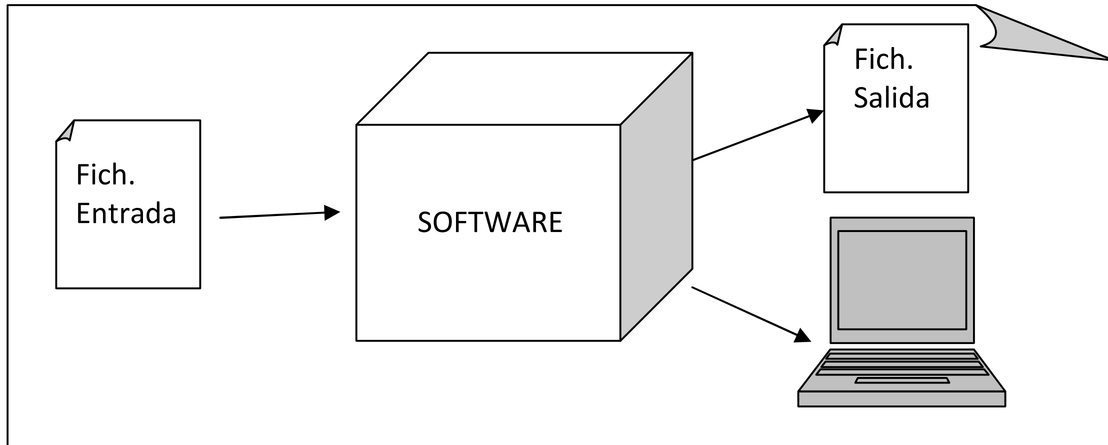

Figura 5.1 Esquema de entrada y salida

La **Figura 5.1** muestra las entradas y salidas del programa. Para el correcto funcionamiento, deben existir los ficheros en el mismo directorio del proyecto, de donde la aplicación lee los nombres de los ficheros que contienen cada una de las tablas. A partir de las tablas leídas, el software genera un fichero *'resultados.txt'* con los resultados, siempre que dentro de **Eclipse** se lo indiquemos con este nombre, además de ir mostrándolos por la consola.

Los archivos con los que se ha trabajado y realizado los experimentos, son bases de datos recopiladas de diferentes hospitales del mundo que han llevado a cabo sobre estudios linfáticos, cáncer, etc. Estos archivos se encuentran en esta página web: *http://archive.ics.uci.edu/ml/index.html*

En concreto dos de los archivos utilizados, ***lymphography.data***, fueron recopilados en el *University Medical Centre, Institute of Oncology, Ljubljana, Yugoslavia*, y los datos fueron recopilados por *M. Zwitter and M. Soklic*, mientras que el archivo ***glass.data,*** fue recopilado por *B. German* perteneciente al *Home Office Forensic Sciencie Servic.*

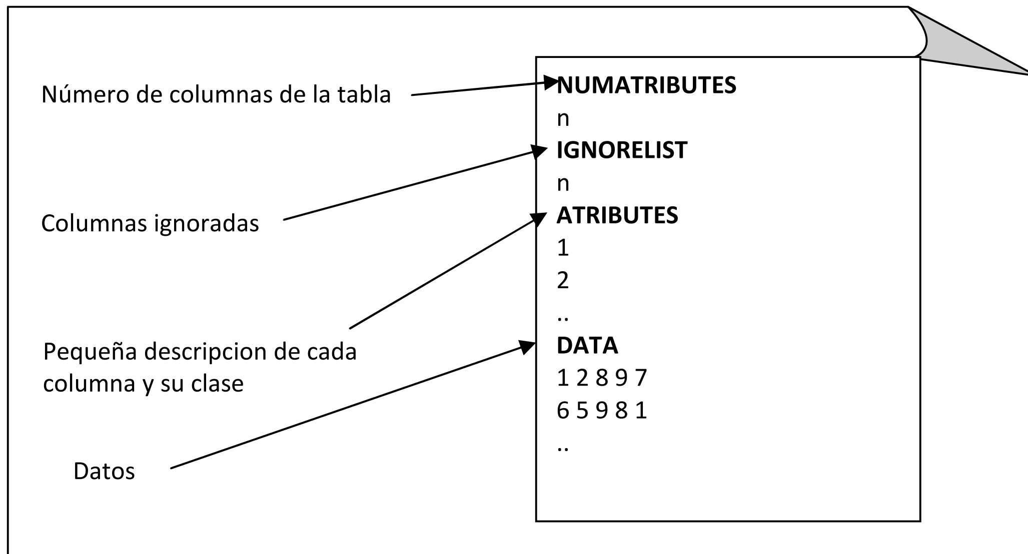

Figura 5.2 Formato de entrada de los archivos

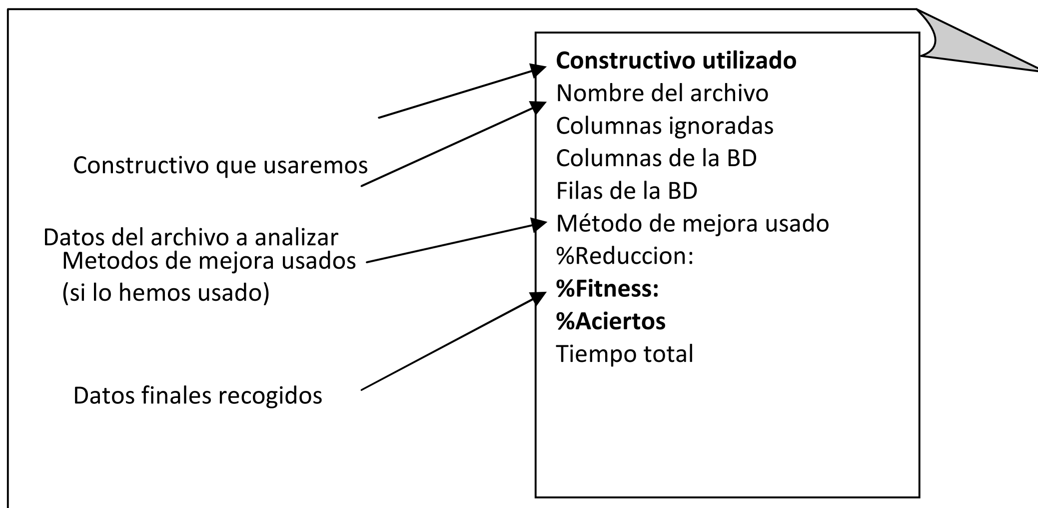

Figura 5.3 Formato de salida del archivo de soluciones

La **Figura 5.2** representa el formato de los archivos leídos por el proyecto. Estos archivos se mantiene constantes en su formato pero muchos de ellos contienen una descripción de cada una de las columnas en la parte de ***ATRIBUTES.*** Los datos que incluiremos en la matriz vienen en la parte denominada ***DATA.***

La **Figura 5.3** representa el formato de salida tanto de la consola como del archivo de soluciones si hemos decidido obtenerlo. Los datos que se recogen, son los que luego se utilizarán para mostrarlos en las tablas de resultados experimentales.

Cabe destacar en este punto, el formato de las tablas a analizar. Las bases de datos en muchas ocasiones, viene ordenadas por su clase. A la hora de clasificar, según nuestro método de clasificación, no daría buenos resultados. Es por ello que en **cada ejecución**, reorganizamos la tabla aleatoriamente sus instancias para que la clasificación sea mejor.

Para cerrar este apartado, es conveniente explicar el formato de las tablas comparativas. La **Figura 5.4** muestra los datos recogidos que tendrán estas tablas. Cada uno de estos datos recogidos representa una característica para tener en cuenta al determinar la calidad de la tabla y de su reducción.

Datos de las tablas comparativas.	
%Aciertos	Porcentaje de aciertos que se obtuvo de la mejor solución.
%Reducción	Porcentaje de la reducción de columnas de la mejor solución.
%Fitness	Porcentaje del mejor *fitness* que se obtuvo
CPUTime (s):	Tiempo que tardo en ejecutarse el total de los archivos.

Figura 5.4. Datos de las tablas comparativas

Cada una de las tablas presentara junto al constructivo utilizado los métodos que ha utilizado para mejorar la solución. Para su mayor comprensión y simplificación de la tabla se ha denominado de la siguiente forma:

- LS1 = Mejora de borrar columnas
- LS2 = Mejora de añadir columnas
- Inter = Intercambio de columnas
- VNS = Método VNS

5.4 Experimentos

En esta parte se presentan los resultados experimentales obtenidos tras el proceso de los algoritmos constructivos.

Se destaca que el proceso de construcción 1, hace referencia a ir añadiendo columnas a la tabla de datos vacía mientras que la construcción 2, la referencia el ir quitando columnas a la tabla completa (destructivo).

Al evaluar las tablas, es importante destacar algunos detalles para comprender los datos. Cuando se ejecuta un constructivo 1, el porcentaje de reducción que se indica, es el referido a los datos que no se han incluido en la tabla. Es decir, un porcentaje alto significará que se han incluido pocas columnas, mientras que uno bajo significara lo contrario. En cambio, si ejecutamos un constructivo 2 y el porcentaje de reducción es bajo, significara que al ejecutarse, se han borrado pocas columnas de la base de datos. Mientras que un porcentaje alto significara que se han borrado bastantes columnas.

monks-1.test

Base de datos compuesta por 8 columnas, de las cuales ignoramos una de ellas, y con 433 filas.

	% Aciertos	% Reducción	% Fitness	CPUTime (m' s''):
Constructivo 1	97,67	85,681	55,98	5' 12''
Constructivo 1 + LS1	96,49	71,594	55,864	7' 36''
Constructivo 1 + Inter	97,44	71,594	55,864	24' 10''
Constructivo 1 + VNS	97,44	71,594	55,864	12' 2''
Constructivo 2	93,02	14,319	89,368	5' 8''
Constructivo 2 + LS2	93,95	28,406	89,834	9' 4''
Constructivo 2 + Inter	94,17	28,406	89,95	7' 20''
Constructivo 2 + VNS	90,47	28,406	88,09	7' 25''

Figura 5.5. Resultados del archivo *monks-1.test*

De la tabla de la **Figura 5.5**, podemos observar dos circunstancias totalmente opuestas. Si partimos de un **constructivo 1**, es decir, ir añadiendo columnas, el algoritmo incluirá pocas columnas en la solución final y las mejoras que hemos aplicado no consiguen ningún cambio efectivo. Por otro lado, al aplicar el **constructivo 2**, el algoritmo no quitará apenas columnas de la base de datos consiguiendo un alto *fitness* pero menos aciertos que añadiendo columnas.

lymphography.data

Base de datos compuesta por 19 columnas y 149 filas.

	% Aciertos	% Reducción	% Fitness	CPUTime (m' s''):
Constructivo 1	88,57	79,481	46,917	18' 58''
Constructivo 1 + LS1	71,43	78,805	69,925	14' 50''
Constructivo 1 + Inter	74	79,481	68,233	65' 10''
Constructivo 1 + VNS	60,66	79,481	69,342	18' 2''
Constructivo 2	60,71	94,71	77,726	4' 42''
Constructivo 2 + LS2	60	90,078	77,368	6' 26''
Constructivo 2 + Inter	64,29	84,78	76,88	7' 20''
Constructivo 2 + VNS	56,43	90,078	75,583	18' 52''

Figura 5.6. Resultados del archivo *lymphography.data*

De la tabla de la **Figura 5.6,** podemos observar bien cómo influyen las mejoras cuando se aplican a la solución dada por los constructivos. En este caso por el **constructivo 1.** Al aplicar tanto la mejora de ***LS1*** como ***Intercambio*** mejoramos el *fitness* pero bajamos en el número de aciertos. Es importante destacar que, en ambos constructivos, a pesar de tener un porcentaje de reducción bajo en el **constructivo 1** y, alto en el **constructivo 2**, se añaden y se quitan bastantes columnas. No se aprecia tanto este dato, pues como se indica arriba, la base de datos está compuesta por 19 columnas. Esto hace que la inclusión de 5, por ejemplo en el **constructivo 1**, no sea tan notorio como que se incluyan 5 en una base de datos de tan solo 7 columnas.

glass.data

Base de datos compuesta por 214 filas y 11 columnas, de las cuales, hay que ignorar una de ellas.

	% Aciertos	% Reducción	% Fitness	CPUTime (m' s''):
Constructivo 1	92,381	89,953	51,19	4' 15''
Constructivo 1 + LS1	95,24	80,38	52,619	4' 30''
Constructivo 1 + Inter	93,81	89,953	51,904	10' 28''
Constructivo 1 + VNS	93,81	89,953	51,904	4' 20''
Constructivo 2	91,9	80,375	90,95	3' 15''
Constructivo 2 +LS2	93,33	80, 375	91,67	5' 10''
Constructivo 2 + Inter	92,38	80,37	91,19	5' 20''
Constructivo 2 + VNS	92,38	80,375	91,19	8' 32''

Figura 5.7. Resultados del archivo *glass.data*

De la tabla de la **Figura 5.7,** podemos observar que si ejecutamos un **constructivo 1**, la calidad de la base de datos reducida no es muy bueno: no se consigue un alto *fitness* y la reducción de la tabla no es alta. En cambio, si partimos de un **constructivo 2,** el *fitness* es muy alto borrando bastantes columnas, seleccionando aquellas que son mejores entre todas.

pima-indians-diabetes.data

Base de datos compuesta por 769 filas y 9 columnas

	% Aciertos	**% Reducción**	**% Fitness**	**CPUTime (m' s''):**
Constructivo 1	98,29	88,87	54,7	29' 30''
Constructivo 1 + LS1	96,05	77,88	53,582	42' 10''
Constructivo 1 + Inter	97,5	77,88	54,31	60' 20''
Constructivo 1 + VNS	86,48	88,87	48,78	51' 20''
Constructivo 2	68,68	11,12	78,787	33' 20''
Constructivo 2 +LS2	66,18	77,87	77,54	55' 40''
Constructivo 2 + Inter	67,89	77,88	78,39	62' 30''
Constructivo 2 + VNS	69,87	77,88	77,99	64' 23''

Figura 5.8. Resultados del archivo *pima-indians-diabetes.data*

En esta tabla de la **Figura 5.8,** podemos encontrar datos muy curiosos. Por una parte al ejecutar el **constructivo 1,** observamos que el *fitness* que conseguimos es muy bajo. A partir de una matriz vacía no conseguimos clasificar muy bien a pesar de que el número de aciertos es muy elevado.

Por otro lado, al ejecutar un **constructivo 2**, podemos llegar a los mismos resultados de calidad de tabla en *fitness,* a través de quitar solo unas pocas columnas o quitando muchas de ellas como se puede apreciar en la tabla. En cambio en esta ejecución conseguimos un mayor *fitness* aunque el número de aciertos disminuye.

Capítulo 6

Conclusiones y futuros trabajos

En este capítulo expondremos las conclusiones derivadas del trabajo realizado y las posibles mejoras y futuras líneas de trabajo que pueden derivar de la realización del proyecto.

6.1 Conclusiones

Durante la realización del proyecto, la finalidad de éste ha sido la reducción de datos mediante la selección de características dentro del área de la minería de datos. Esta reducción la hemos elegido para resolver problemas de clasificación de instancias e intentar ignorar características de base de datos que pueden ser de poca relevancia. Para su implementación hemos utilizado el lenguaje de programación Java, lenguaje orientado en objetos y muy usado en el mundo laboral hoy.

Para la realización del proyecto, hemos ido dividendo el proceso en distintas fases. Estas fases han ido subiendo de dificultad a la hora de plantearlas como de implementarlas. Así, en una primera fase, implementábamos la manera de leer archivos de texto que contenían base de datos para ser almacenadas en una matriz, para más tarde calcular, el grado de calidad de las instancias de esa matriz, seleccionando las características que mejoraban la calidad de esta matriz (porcentaje de aciertos, de reducción y de *fitness*). De esta forma, seleccionábamos las mejores características de la matriz con las que podíamos clasificar las instancias de esta.

Para la selección de que columnas son mejores para nuestra matriz reducida, usamos el algoritmo *GRASP*. Mediante este método buscamos aquellas características que mejoran la calidad de la tabla.

Tras la recogida de los datos por parte de ambos constructivos implementados, añadir características a una matriz vacía y borrar características de una matriz completa, podemos obtener las siguientes conclusiones:

- Cuanto mayor es el número de instancias de una base de datos, mayor es el tiempo que tarda en clasificar instancias y encontrar una característica a seleccionar.

- Una base de datos puede ser muy buena clasificando, mientras que otras pueden llegar a dar una baja tasa de aciertos en comparación con el número de instancias que contienen.

- La mayoría de las bases de datos están muy optimizadas y es complicado obtener más de cuatro o cinco características que no sean importantes.

- Los métodos de mejora usados en este proyecto, no mejoran prácticamente en nada la solución obtenida por los constructivos salvo en casos excepcionales.

6.2 Futuros trabajos

El problema de la reducción de datos a través de la selección de características tiene multitud de variantes para ser resuelta. A primera vista, podemos comprobar que algunas de las bases de datos utilizadas dan un muy buen porcentaje de clasificación, mientras que otras no clasifican casi nada. Podemos concluir tras la observación de estos datos en las tablas que, cuanto más heterogéneos son los datos, menor porcentaje de calidad se obtiene, y, más tarda en clasificar cuanto mayor es el volumen de datos. Se podría tratar de que las bases de datos sean más homogéneas para comprobar que el sistema de selección de características, mejora la calidad de la tabla reducida.

Al igual, como se habrá podido comprobar, al usar los métodos de mejora, no se ha obtenido una mayor optimización de la solución propuesta por los constructivos, excepto en algún caso excepcional. Se podría tratar de mejorar la solución con otros métodos de mejora existentes.

Bibliografía:

- **Datos:**
  - http://archive.ics.uci.edu/ml/

- **Libros**
  - "Piensa en Java", Eckel, 2ª Edición, Addison-Wesley, 2002
  - T.A. Feo and M.G.C. Resende (1989) A probabilistic heuristic sea la computationally difficult set covering problem. *Operations Research Letters*, 8:67–71, 1989.
  - M.G.C. Resende and C.C. Ribeiro (2003) Greedy randomized adaptive search procedures. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pp. 219–249, Kluwer Academic Publishers, 2003.
  - FLESZAR, K. y HINDI, K. S. "Solving the resource-constrained project scheduling problem by a variable neighbourhood search". European Journal of perational Research, forthcoming. 2004.
  - Metaheurísticas. Abraham Duarte Muñoz, Juan José Pantrigo Fernández, Micael Gallego Carrillo. (2006)

- **Páginas Web:**
  - http://es.wikipedia.org/wiki/Heur%C3%ADstica_(inform%C3%A1tica)
  - http://es.wikipedia.org/wiki/Metaheur%C3%ADstica
  - http://msdn.microsoft.com/es-es/library/ms175382.aspx
  - http://www.daedalus.es/fileadmin/daedalus/doc/MineriaDeDatos/DAEDALUS-WP-Mineria_Datos.pdf
  - http://exa.exa.unne.edu.ar/depar/areas/informatica/SistemasOperativos/Mineria_Datos_Vallejos.pdf