WEB DE ESQUEMAS ALGORÍTMICOS

PROYECTO FINAL DE CARRERA DE INGENIERÍA INFORMÁTICA

Autores:

José Víctor Jiménez Corbalán.
jvictorjimenez@gmail.com

Andrés Palazón Jiménez.
andrespalazon@gmail.com

Directores:

Francisco Javier Bermúdez Ruiz.
fjavier@um.es

Domingo Giménez Cánovas.
domingo@dif.um.es

Septiembre de 2008

Universidad de Murcia

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE CÓDIGOS

1. INTRODUCCIÓN Y MOTIVACIÓN

Tradicionalmente, las prácticas de los primeros cursos de enseñanza de los algoritmos se han realizado confiando en el cumplimiento por parte de los alumnos de los esquemas algorítmicos con los que se trabaja. Esto, en la práctica, se encamina a que cada alumno realice la implementación de los problemas reestructurando el esquema, lo que conlleva una carencia en la enseñanza de los mismos en términos de pérdida de tiempo y conceptos por parte del alumno.

Además, el alumno que comienza a realizar las prácticas de una asignatura desde su propio ordenador debe instalar y configurar el software adecuado para la creación de programas (editores, compiladores, etc…). Esto conlleva un gasto de tiempo que no está ligado directamente al aprendizaje relacionado con la asignatura. Igualmente, el alumno debe diseñar una estructura de clases y ficheros básica para el problema implementado. Esta estructura puede ser común en todos los problemas que se han implementado utilizando la misma técnica de diseño de algoritmos.

Para evitar la pérdida de tiempo en la gestión docente por parte del profesorado es necesario también un entorno de diseño del problema. En la praxis, cada alumno crea un diseño del esquema algorítmico diferente y un diseño de la estructura de su programa también diferente, lo que supondrá una pérdida de tiempo por parte del docente en la evaluación de la práctica. Esto se debe a que el profesor debe llegar a comprender en profundidad el diseño del programa y del esquema algorítmico creado, evaluando también si el alumno ha entendido los conceptos básicos de la técnica algorítmica usada (a pesar de la reestructuración de los esquemas básicos estudiados).

Los problemas comentados pueden afectar asimismo a programadores a nivel profesional que se dediquen a la resolución de problemas utilizando las técnicas de diseño de algoritmos, ya sean informáticos o de otro campo científico donde se utilicen estos algoritmos, como robótica, química, biología, etc.

1.1 Objetivos del proyecto

El objetivo principal del proyecto es facilitar la realización y corrección de programas sencillos diseñados con la ayuda de una técnica de diseño de algoritmos, de forma que se guíe a los alumnos y se les fuerce al diseño de los algoritmos siguiendo de manera sistemática los esquemas clásicos. Esto se conseguirá atendiendo a los siguientes hitos:

- Introducir un método didáctico e interactivo para la enseñanza y el estudio del comportamiento de algoritmos.
- Aplicación del esquema algorítmico de una manera correcta y sin perder de vista sus principios de funcionamiento.
- Estructura "bien conocida" del diseño de los programas que se han creado utilizando una misma técnica de diseño de algoritmos.
- Necesidad por parte del usuario de obviar las acciones de distribución y configuración del entorno para la programación de su problema.
- Utilización de un lenguaje de programación potente en la realización de la implementación de los problemas, que recoja varios paradigmas de programación y sea de utilidad en las asignaturas de algoritmos de los primeros años de las carreras de informática.

- Necesidad de un lugar común donde se puedan encontrar ejemplos de utilización de una técnica de diseño de algoritmos y su coexistencia con la propia implementación de otros problemas, su ejecución y su análisis.

Teniendo en cuenta todo lo comentado, y en vista de los propósitos mencionados en el párrafo anterior, en la siguiente sección se comentarán las soluciones que se darán para el problema propuesto.

1.2 Propuesta de solución

Debido a las dificultades que entraña el aprendizaje inicial de las asignaturas de algoritmos en las carreras de Informática, se ha pensado en una aplicación que guíe y mejore el aprendizaje y la resolución de problemas mediante el uso de esquemas algorítmicos. Esto dará soporte a un lugar de desarrollo común para los alumnos y profesores, en el que se seguirá una estructura definida previamente en la utilización de los algoritmos correspondientes y en la estructura propia de la aplicación desarrollada que contiene la solución de implementación del problema.

En las prácticas de las asignaturas de Algoritmos y Estructuras de Datos de las carreras de Informática en la Universidad de Murcia se utiliza el lenguaje de programación C++. Este lenguaje de propósito general está muy extendido en el ámbito científico. C++ es un lenguaje de programación de carácter general, diseñado a mediados de los años 1980, por Bjarne Stroustrup, como extensión del lenguaje de programación C. C++ es un lenguaje que abarca tres paradigmas de la programación: la programación estructurada, la programación genérica y la programación orientada a objetos. Su potencia reside en que puede trabajar tanto a bajo nivel como a alto nivel, aunque esto conlleva una dificultad mayor en su aprendizaje. Debido a las características comentadas, la implementación de los problemas a través de nuestra aplicación se realizará en este lenguaje de programación.

El auge que las aplicaciones Web están teniendo en los últimos años, así como las ventajas que proporcionan este tipo de aplicaciones frente a los desarrollos de escritorio ha impulsado la creación de la aplicación que aquí se presenta para su utilización en un entorno Web. No obstante, también interesa que se pueda tener una versión para utilizar desconectado de la red global, para usuarios que no puedan disponer de conexión permanente al servidor que aloja la herramienta. Por ello se ha implementado una adaptación como aplicación de escritorio de modo que en la pantalla principal de la aplicación se da la opción al usuario de descargársela junto con un manual de instrucciones para su configuración como aplicación local. Se trata de una configuración local de un servidor TOMCAT con la aplicación desarrollada integrada.

Entre las ventajas conseguidas por una aplicación Web frente a una aplicación de escritorio se encuentran:

- No es necesaria la instalación y configuración de nada que no sea el navegador y la red para el acceso a la aplicación.
- Se puede utilizar la aplicación desde cualquier máquina con acceso a la red.
- El usuario no tiene que reinstalar actualizaciones y parches de la aplicación.
- El mantenimiento de la aplicación se realiza únicamente en el servidor.
- Pocos requisitos hardware/software para los puestos clientes.
- La aplicación es multiplataforma.

Por otro lado, las desventajas de la aplicación distribuida frente a una aplicación de escritorio son:

- No se integra con el escritorio, por lo tanto se desaprovechan todas las ventajas que se puedan obtener de esta integración.
- Interfaz muy limitado respecto a una aplicación de escritorio.
- Dependencia de la ocupación y disponibilidad del servidor para la ejecución eficiente de los programas del usuario.
- Diferencias de presentación entre plataformas y navegadores. La falta de estándares ampliamente soportados dificulta el desarrollo de las aplicaciones.
- La imposibilidad de acceso a la red implica que no se pueda utilizar la aplicación distribuida.

Así este proyecto presenta la herramienta para la programación de esquemas algorítmicos con C++ WEA (Web de Esquemas Algorítmicos), como realización de las ideas propuestas. El objetivo principal ha sido la creación de una aplicación en un entorno Web para que los alumnos de las asignaturas de iniciación a los algoritmos en las carreras de informática dispongan de una herramienta de ayuda didáctica.

WEA es una herramienta desarrollada con tecnologías Java 2 [9]. Las tecnologías y técnicas utilizadas son JSP, servlets, modelo vista-controlador, CSS, patrón *Action,* utilización de ficheros de propiedades para simplificar la configuración de la aplicación en diferentes entornos y eliminar dependencias de aplicación, etc.

La herramienta permite la edición básica, compilación y análisis de problemas programados en C++. Estos algoritmos se deben implementar utilizando los esquemas algorítmicos ofrecidos en la Web. Los esquemas algorítmicos dados no son modificables y están ya implementados. En este sentido, la etapa de diseño de la aplicación ya ha sido resuelta gracias a la herramienta desarrollada. El alumno únicamente se tiene que centrar en rellenar las funciones propias para la técnica de diseño de algoritmos escogida. Obligatoriamente siempre se cumple la correcta y exhaustiva utilización del esquema algorítmico.

Los esquemas algorítmicos estandarizados tienen las siguientes características:

- Son escritos, y por lo tanto fácilmente accesibles para su consulta.
- Están estandarizados, por lo que es posible comprobar que todo el mundo hace lo mismo.
- Son abstractos y generales.
- Son simbólicos. Uno hace sus operaciones enteramente por medio de símbolos de manipulación, sin referencias al mundo real o cualquier otro modelo.
- Son métodos activos, en el sentido que el usuario hace una selección definida y tiene el control de sus propias operaciones.

La herramienta WEA se basa en la premisa de que los usuarios tendrán un nivel de comprensión de la algorítmica fundamental, y que por tanto han seguido un primer curso de iniciación a la programación. Los desarrollos podrán ir desde ejemplos básicos hasta el nivel máximo de expresión que ofrece C++. Además, la herramienta es un apoyo en la enseñanza de la programación orientada a objetos en C++ y la utilización de algunos patrones de diseño. Adicionalmente, se incluyen ejemplos guiados, tutoriales de utilización y ayudas para la correcta comprensión del

funcionamiento de la técnica algorítmica por un lado y de la herramienta creada por otro.

1.3 Antecedentes y situación actual

La necesidad de buscar nuevas formas de enseñanza se hace patente debido a los cambios acontecidos en la sociedad. Los nuevos alumnos se han desarrollado en un entorno eminentemente tecnológico, con nuevas formas de comunicación y percepción, mayoritariamente audiovisuales. Además, los conceptos relacionados con las técnicas de diseño de algoritmos son muchas veces complejos y alejados de las técnicas mentales utilizadas para la resolución de problemas. Con la ayuda de herramientas que permitan simplificar el desarrollo de los algoritmos se puede mejorar el aprendizaje del alumno, haciendo que adquiera los conceptos con menos esfuerzo y tiempo.

El esfuerzo en la búsqueda de nuevas formas de enseñar algoritmos se está llevando a cabo orientado a la animación de algoritmos básicos de programación [1,2,3,4]. Sin embargo, estas herramientas suelen orientarse a un tipo concreto de algoritmos (por ejemplo: algunas permiten la visualización de esquemas de ordenación únicamente), enfocando su radio de actuación a esquemas básicos de programación, no estando los esquemas más complejos recogidos. Asimismo, la animación permite la visualización únicamente de estructuras básicas, como arrays y en algún caso matrices, árboles o listas.

Otro proyecto que orienta la enseñanza de algoritmos es Animación y Simulación de Algoritmos Paralelos de Exploración de Grafos, que utiliza la técnica de diseño de algoritmos Ramificación y Poda para recorrer grafos de forma paralela [5].

Siguiendo con herramientas que ayudan en el aprendizaje de algoritmos, se pueden observar otros programas centrados en la instrucción en un solo algoritmo. Siguiendo este camino nos encontramos con una "herramienta gráfica para el aprendizaje del algoritmo simplex" [6]. Este algoritmo se utiliza en Investigación Operativa para la resolución de problemas de programación lineal.

Por otro lado se pueden encontrar otras herramientas, como DFD, que es un editor e intérprete de algoritmos representados en diagramas de flujo [7]. Esta herramienta permite crear diagramas de flujo, y a partir de ellos ejecutarlos o realizar otras funciones como ejecutar paso a paso, ejecutar hasta un punto de ruptura, etc.

Con respecto a programas que ayudan a la realización de prácticas de programación referentes a problemas concretos implementados con técnicas de diseño de algoritmos, se puede encontrar Mooshak [8]. Es una herramienta online que evalúa de forma automática si cierto código fuente es una solución correcta para un problema dado. Esta herramienta se ha estado utilizando durante este curso lectivo en la asignatura Algoritmos y Estructuras de Datos de la Universidad de Murcia, y en varios concursos de programación.

No hemos encontrado ningún desarrollo orientado a la creación de un lugar común de trabajo en el entorno de una asignatura, más allá de los campus virtuales que ofrecen la mayoría de las universidades. Las herramientas desarrolladas en ese sentido son muy específicas, muy ligadas a la asignatura relacionada, ya que tienden a crear redes sociales en torno a una materia muy concreta con necesidades diferentes. En este sentido, el proyecto es innovador, satisfaciendo algunas de las

necesidades que tienen los desarrolladores de código para la resolución de problemas en el entorno de la docencia de las asignaturas de algoritmos y para entornos científicos de resolución de problemas para una materia específica.

1.4 Metodología

1.4.1 Planificación temporal

El proyecto comienza en septiembre de 2007 con la reunión entre el director Domingo Giménez con los dos miembros del grupo de trabajo. En este punto se definen los objetivos a alcanzar en el proyecto y debido a la escasa formación en programación Web de los miembros se establece la necesidad de tener un director para el desarrollo de la aplicación (F. Javier Bermúdez Ruiz). Con esto, el proyecto toma **dos ámbitos de formación para los miembros, por un lado la parte de esquemas algorítmicos y por otro el aprendizaje de desarrollo de aplicaciones Web**. Así desde principios de Octubre de 2007 hasta pasado Diciembre los dos miembros del proyecto se dedican a realizar, sin estar matriculados en la asignatura del profesor Bermúdez, las prácticas de la asignatura de Desarrollo de Aplicaciones Distribuidas para tomar una formación base antes de empezar a programar.

En torno a Febrero de 2008 comienza el desarrollo de la aplicación Web, el cual se extiende hasta la primera semana de agosto de 2008.

La aplicación se monta inicialmente con dos esquemas algoritmos, Divide y Vencerás y Ramificación y Poda (uno por miembro del equipo de trabajo), terminando todas las funcionalidades de la aplicación de modo que el paso de añadir nuevos algoritmos sea algo trivial y solo conlleve el trabajo del diseño de los algoritmos. Así desde Febrero hasta junio de 2008 se desarrollo la aplicación con estos dos algoritmos y con todas las funcionalidades operativas: compilación de código, ejecución de código, realización de pruebas sobre funciones, realización de pruebas sobre entradas, redirección de la entrada estándar al ejecutable ,generación de graficas, subidas de ficheros al servidor, gestión de usuarios, seguridad de usuarios con encriptación MD5, obtención del código generado por los usuarios en un fichero zip y ayudas y ejemplos que guían el uso de la aplicación. Desde junio a agosto de 2008 se añadieron los dos siguientes esquemas algoritmos, Bactracking y Programación Dinámica, y se adaptaron las funcionalidades ya implementadas: pruebas sobre funciones, pruebas sobre entradas, ayudas y ejemplos, y todo lo demás se ha reutilizado.

1.4.2 División del trabajo

La aplicación consta de cuatro esquemas algorítmicos: Divide y Vencerás y Programación Dinámica diseñados y desarrolladas por Andrés Palazón y Ramificación y Poda y Backtracking realizados por José Víctor Jiménez.

En cuanto al desarrollo de la aplicación Web se ha realizado en paralelo por ambos miembros y prácticamente la totalidad de funcionalidades implementadas han sido realizadas por los dos. La búsqueda y la utilización de librerías para gráficas y subida de ficheros al servidor se han realizado de manera conjunta. Ha habido una división de trabajo en cuanto a la gestión de usuarios que ha sido realizada por Andrés Palazón, mientras que José Víctor se ha encargado de la construcción de ficheros zip para que el usuario descargue el código generado y la realización de un paquete para descargarlo como aplicación de escritorio.

Por otro lado las funcionales de compilación, ejecución de pruebas sobre funciones, ejecución de pruebas sobre entradas y redirección de la entrada estándar al ejecutable, etc, han sido desarrollados por ambos miembros de forma paralela.

1.5 Resumen de apartados posteriores

La división de la memoria se ha realizado de la siguiente manera: En el presente capítulo (Capítulo 1) se ha introducido la motivación existente para la realización del proyecto, los objetivos perseguidos con el proyecto, la propuesta de soluciones para la problemática planteada.

Los restantes capítulos se estructuran como sigue: El capítulo 2 describe la aplicación desarrollada dando una visión general, y detallando las tareas que el usuario puede realizar con la aplicación: edición, compilación, ejecución, pruebas y análisis de los resultados obtenidos. El capítulo 3 describe el diseño realizado para crear la aplicación, arquitectura, patrones de diseño, etc. El capítulo 4 detalla cada uno de los esquemas implementados y puestos a disposición de los usuarios. A continuación se describen las perspectivas futuras de la aplicación en el capítulo 5. Por último se muestra la bibliografía utilizada y el apartado de anexos.

2. DESCRIPCIÓN DE LA APLICACIÓN

Esta sección explica el funcionamiento de WEA, comentando los aspectos más destacables divididos en las áreas y funciones más importantes de la aplicación.

La aplicación, inicialmente, incluye cuatro esquemas implementados: Divide y Vencerás, Backtracking, Ramificación y Poda y Programación Dinámica.

2.1 Visión general

Los usuarios deben ingresar en la Web con su cuenta. Si el **login** es correcto verán una página donde dispondrán de enlaces para elegir el algoritmo sobre el que deseen trabajar, tal como se muestra en la Figura 1.

Figura 1. Página principal de la aplicación

Tras eso se pasa a la **implementación del problema**. El usuario tiene que programar los algoritmos en C++. Así, para cada uno de ellos se han creado las estructuras de clases (programación orientada a objetos) correspondientes. En cada una de las clases se dejan las cabeceras de las funciones necesarias a rellenar, y se da la opción de añadir nuevas funciones auxiliares para la resolución del problema. En cualquier caso, como mínimo el usuario debe implementar las funciones especificadas por la aplicación. Una vez rellenas podrá compilar su algoritmo y comprobar si el resultado es el que esperaba.

La aplicación Web se puede dividir en 3 áreas que agrupan varias páginas y funciones relacionadas:

- Edición de código: Zona para la introducción del código por parte del usuario y la creación de la estructura de aplicación necesaria para la compilación.
- Compilación y ejecución: Zona para la compilación y ejecución del código

creado en la zona de edición.
- Análisis de algoritmos: Zona para el análisis de algoritmos, incluyendo visualización de gráficas y pruebas.

En todas las zonas se dispone de un menú principal para facilitar la navegabilidad y funcionalidad ofrecida a los usuarios. En este menú se dispondrá de un área de gestión de usuario, un árbol de navegación por los algoritmos, una serie de enlaces de interés y una zona para activar las ayudas y ejemplos.

2.2 Edición de código

La edición puede realizarse de dos maneras, introduciendo el código a través de teclado o cargando un fichero con el código de cada función de la clase correspondiente. La pantalla de edición es similar sea cual sea el esquema escogido. Las pantallas cambian según sea el fichero de la estructura de clases elegido para editar en ese momento. La página principal de cada esquema muestra la plantilla del propio esquema y una zona de edición donde se puede introducir el código correspondiente al *main*, tal como se muestra en la Figura 2.

El esquema dispone de enlaces a las funciones que deben ser implementadas de las clases auxiliares. Estos enlaces llevan a otras páginas que contienen áreas de edición de código para cada clase auxiliar. En estas páginas se debe completar la declaración de la clase correspondiente, introducir la implementación de los métodos obligatorios y las posibles funciones auxiliares, métodos auxiliares y sobrecarga de operadores, en caso de que los haya.

El menú de edición contiene botones para:

- Cargar el código introducido en la Web en el fichero correspondiente. Genera el fichero de la estructura de compilación correspondiente a la página en la que se encuentra.
- Volver a la "página principal" del algoritmo, en caso de que no se encuentre en ella el usuario.
- Cargar en la página Web el código proveniente de un fichero. En este caso, el archivo debe estar formateado correctamente, tal como se indica en la Web.

Además, el menú de edición de la página principal del algoritmo incluirá botones para compilar el problema implementado y para cargar el fichero de entrada por defecto para la ejecución del algoritmo.

2.3 Compilación y ejecución

Para la correcta **compilación** del algoritmo, se debe cargar cada uno de los ficheros necesarios de la estructura de la aplicación a crear. La página de compilación muestra los errores generados por el compilador para la implementación del problema; si no hay errores indica que la compilación es correcta. Esta página se muestra en la Figura 3.

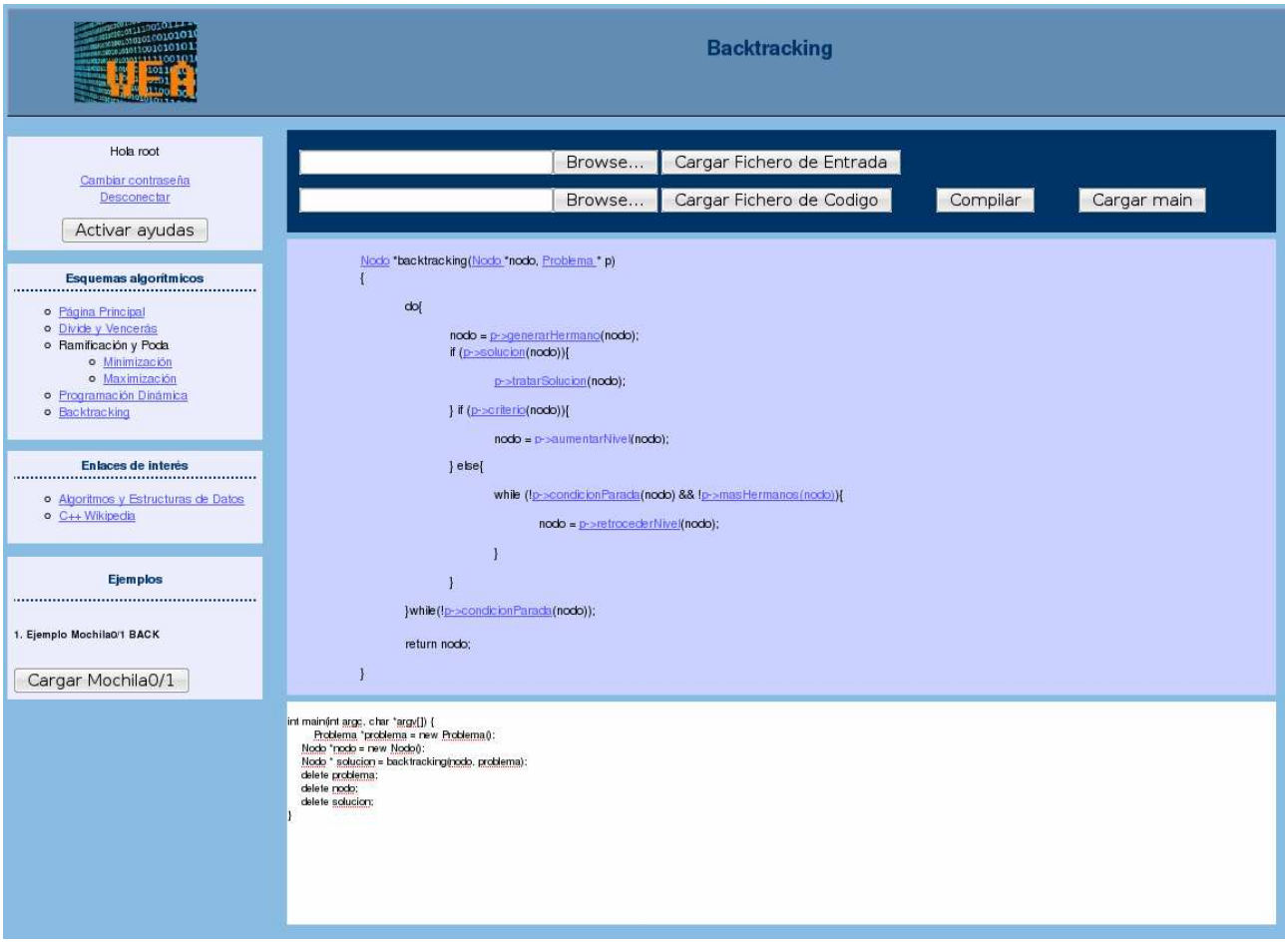

Figura 2. Ejemplo de un esquema

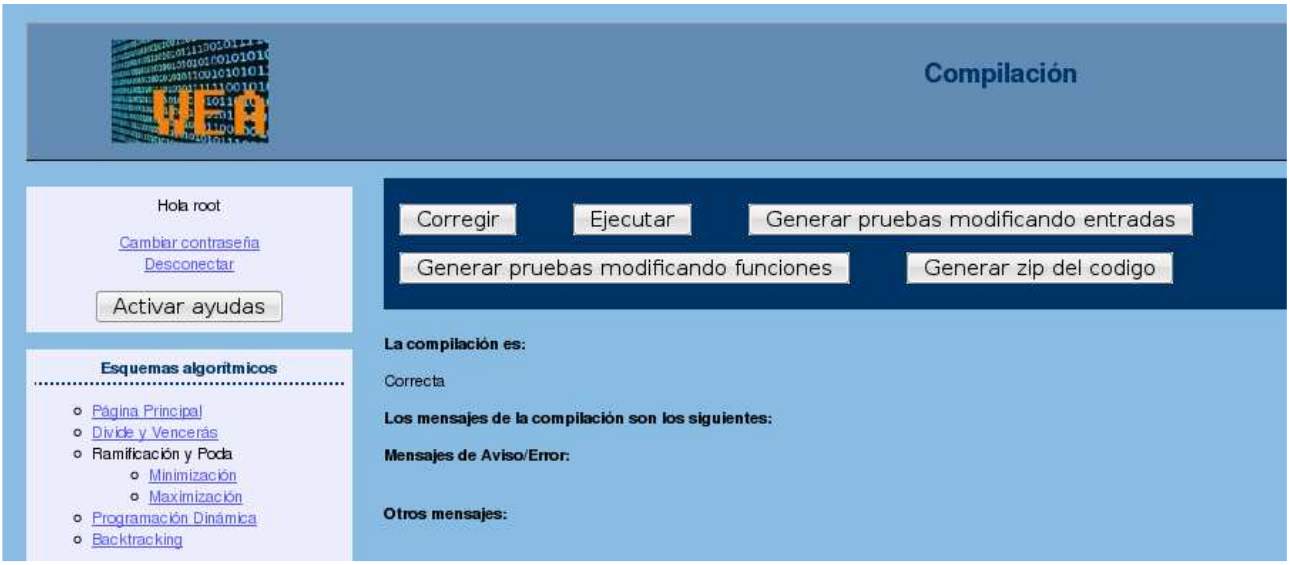

Figura 3. Pantalla resultante de una compilación correcta

A partir de una compilación correcta se puede **ejecutar** el resultado, o realizar pruebas sobre los programas compilados. La ejecución siempre se realiza redireccionando el fichero de entrada por defecto a la entrada estándar. Se mostrará el tiempo de ejecución y los valores que el usuario desee que aparezcan por la salida estándar. La ejecución de las pruebas muestra los valores que desee mostrar por

salida estándar el usuario.

Con una compilación correcta también se pueden obtener, en un fichero .zip, los archivos con el código que ha sido compilado. Estos archivos pueden ser modificados por el usuario con un editor de código y después cargarlos otra vez en el editor de la herramienta WEA.

2.4 Análisis de algoritmos

A partir de una compilación correcta, se puede hacer un análisis de algoritmos sobre el soporte de la plataforma Web gracias a varias funciones:

■ **Pruebas sobre funciones**: Se pueden realizar pruebas sobre funciones interesantes a modificar para el estudio del algoritmo. Para cada prueba se modifica la función correspondiente del esquema escogido. Se puede elegir el número de pruebas a realizar, y eso dará lugar a espacio para la edición de las funciones a modificar, como se muestra en la Figura 4. La entrada estándar para la ejecución será el fichero de entrada por defecto. Los resultados obtenidos se muestran en una tabla, y opcionalmente, si la salida estándar para cada ejecución es un número únicamente, una gráfica de barras con los valores correspondientes. También muestra una tabla con los tiempos de ejecución para cada prueba y una gráfica de barras con los valores de tiempo de ejecución frente al número de prueba.

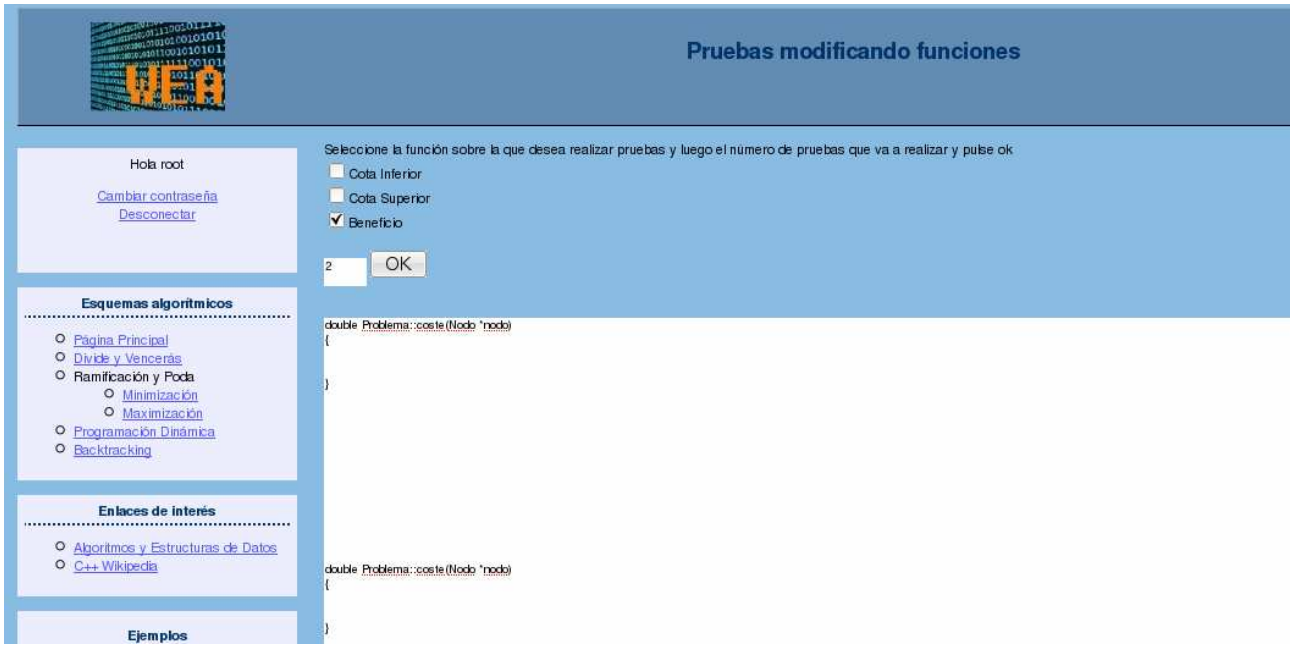

Figura 4. Pruebas modificando funciones

■ **Pruebas sobre entradas**: Se realiza una prueba por cada fichero de entrada cargado. La pantalla de carga de ficheros se muestra en la Figura 5. Esta funcionalidad recoge a la anterior, pero aquella se ha incluido porque hace más sencillo el análisis para usuarios noveles. Con la entrada a un programa se puede realizar cualquier análisis deseado. Para simular el análisis sobre funciones por ejemplo, se puede pasar por entrada estándar un valor entero que haga elegir entre varias implementaciones de la función a modificar para las pruebas. Cada salida de las pruebas se numera con un índice. La ejecución muestra lo mismo que la prueba anterior.

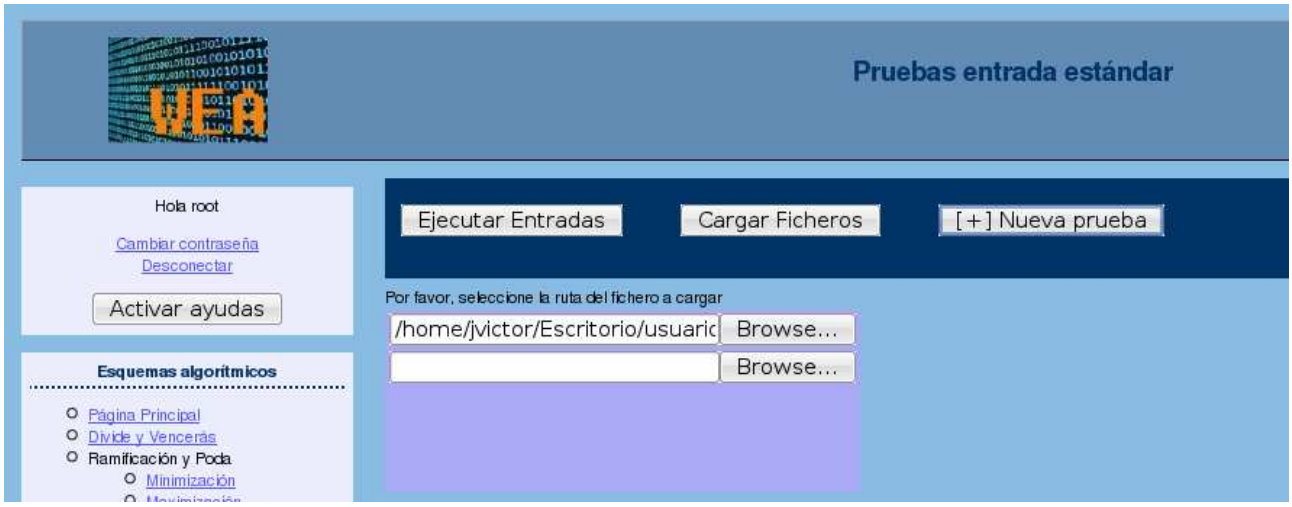

Figura 5. Pruebas eligiendo distintos ficheros de entrada

Una vez realizadas las pruebas y compiladas se pueden obtener los resultados y analizarlos a través de las gráficas que ofrece la aplicación, como se muestra en la Figura 6. Como mínimo se muestra una gráfica con los tiempos de ejecución de cada prueba pero además si la salida de las aplicaciones es un número, se muestran otras gráficas con la evolución numérica del resultado.

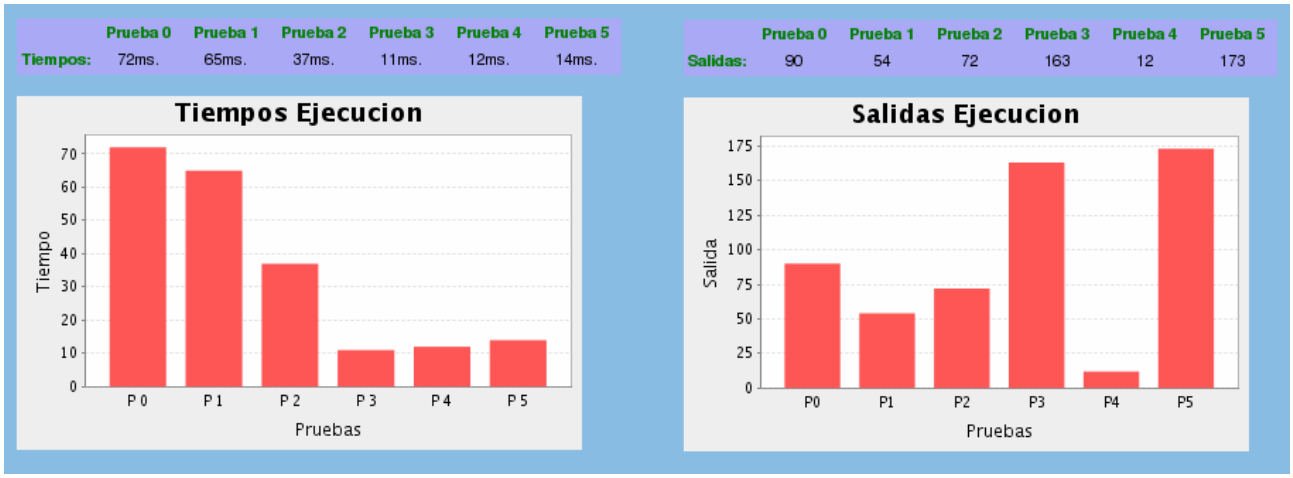

Figura 6. Gráficas generadas tras las pruebas.

2.5 Ayuda y ejemplos

En esta sección se mostrará la implementación de las ayudas y ejemplos, implementados con la finalidad meta de hacer la aplicación autoexplicativa. Los ejemplos serán muy sencillos, con el objetivo de que un usuario entienda como funciona la aplicación y la implementación de sus problemas a través de ella. Los ejemplos están hechos a modos de tutoriales, pudiendo con ellos aprender como se debe utilizar la herramienta. En la Figura 7 se puede observar donde se encuentran las ayudas y ejemplos dentro del menú de la aplicación.

Figura 7. Lateral izquierdo. Gestión de usuario, ayudas, ejemplos, mapa de navegación y enlaces

Los **botones de ayuda** se mostrarán únicamente cuando el usuario desee. En el menú principal se dispone de un botón para activar las ayudas de todos los botones. Inicialmente, cuando se accede a la página Web no se mostrarán los botones de ayuda, pero los usuarios noveles pueden activar las ayudas pulsando el botón del menú principal. Una vez activado aparecerá un botón de ayuda al lado de cada botón de la aplicación. Estos botones abrirán un pop-up con un texto explicativo sobre el botón del que se quiere obtener la ayuda, tal y como se muestra en la Figura 8.

Los **ejemplos** se deberán activar en la "página principal" de cada esquema algorítmico. La activación cargará el código correspondiente al ejemplo en la Web. Además, se abrirá un pop-up con un texto explicativo del ejemplo y las opciones correspondientes que hay. El usuario puede modificar el código cargado según su propio criterio y compilar. Una vez que se ha activado el ejemplo se puede continuar con sus indicaciones o no. En caso de que no se haya cargado el código del ejemplo no se pueden activar las opciones posteriores. A continuación, se puede seguir el ejemplo o no, según el usuario desee. Si se desea seguir el ejemplo, éste va dando explicaciones a modo de tutorial, como se ha comentado antes. Para el análisis de algoritmos, si se sigue el ejemplo, se proponen varias funciones características del algoritmo correspondiente. Para el análisis sobre entradas también se proponen varios ficheros con entradas para la ejecución.

Figura 8. Ayudas.

Los ejemplos para cada uno de los algoritmos son:

• Divide y Vencerás: Problema de quicksort implementado con divide y vencerás.El problema implementado es un quicksort para diez números enteros, donde la base y el número de subproblemas en los que se divide el problema inicial son 2. La función de solución directa del problema será un algoritmo de ordenación de inserción directa.

• Programación dinámica: Problema de sucesión de Fibonacci implementado con Programación Dinámica. El ejemplo calcula el término de la sucesión correspondiente al índice pasado por entrada estándar. El fichero de ejemplo contiene el número 11, con lo que el resultado esperado sería 89.

• Backtracking: Problema de la mochila01 mediante backtracking. Se trata de una mochila de 5 elementos y con una capacidad de mochila de 8 unidades. Los datos de entrada donde se encuentran tanto el costo de los elementos como las unidades que ocupan se introducirán por entrada estándar, con redirección a través de un fichero. El fichero con los datos de entrada iniciales contendrá la línea: 5 7 2 25 16 2 1 4 6 1. Los 5 primeros datos corresponden a los costes de los elementos, y los 5 siguiente al peso de los mismos. Con esta entrada la salida esperada del programa será 41.

• Ramificación y poda, maximización: Problema de la mochila01 mediante ramificación y poda en un contexto de maximización. Se trata de una mochila de 5 elementos y con una capacidad de mochila de 8 unidades. Los datos de entrada donde se encuentran tanto el costo de los elementos como las unidades que ocupan se introducirán por entrada estándar, con redirección a través de un fichero. El fichero con los datos de entrada iniciales contendrá la línea: 5 7 2 25 16 2 1 4 6 1. Los 5 primeros datos corresponden a los costes de los elementos, y los 5 siguiente al peso de los mismos. Con esta entrada la salida esperada del programa será 173, ya que con estos criterios de poda (heurísticos) se elimina la solución óptima.

• Ramificación y poda, minimización: Problema de las Ntareas mediante ramificación y poda en un contexto de minimización. El problema de las Ntareas,

consiste en asignar a un serie de procesadores N tareas a ejecutar de manera que esta asignación sea mínima en tiempo de ejecución. En nuestro ejemplo el número de procesadores será de 3, y el número de tareas de 7. El problema recibe como entrada en primer lugar la velocidad de los procesadores, en instrucciones por segundo, y seguidamente el número de instrucciones de cada tarea. Se supone que las tareas se ejecutan de manera indivisible. Así por ejemplo, en nuestro problema la entrada será la siguiente: 4 5 2 7 8 6 9 3 1 10. Los primeros tres valores corresponden a las velocidades de cada uno de los tres procesadores respectivamente. Y los siguiente 7 datos al número de instrucciones de cada una de las 7 tareas. El resultado esperado es de 65.

Todos los ejemplos excepto el último se han realizado siguiendo las indicaciones y propuesta del texto-guía de las asignaturas de Algoritmos y Estructuras de Datos de la Universidad de Murcia [10]. El último ejemplo, n-tareas, ha sido sacado de la asignatura de 5º curso de Ingeniería en Informática de Algoritmos y Programación Paralela, donde este año ha sido propuesto como práctica de la misma, y debido a que José Víctor Jiménez lo implementó de manera paralela con un algoritmo de Ramificación y Poda se ha considerado interesante adaptarlo.

2.6 Gestión de usuarios

Para hacer más amigable la interacción con la herramienta WEA, se ha incluido una zona de gestión de usuarios para un identificador de usuario administrador, *root.* Este usuario puede realizar el ingreso de nuevas cuentas de usuario, cambiar las contraseñas de un usuario existente o eliminar usuarios existentes, tal como se muestra en la Figura 9. Por seguridad, no se permite que un usuario se registre en la aplicación ni elimine su cuenta. La única gestión permitida a un usuario que no sea administrador es el cambio de su contraseña.

Figura 9. Gestión de usuarios, visible sólo para el usuario Root

3. DISEÑO GENERAL DEL PROYECTO

La arquitectura de un proyecto Web se suele dividir en diferentes capas lógicas que tienen las siguientes características:

- Cada capa es independiente.
- Cada capa tiene una interfaz bien definida.
- Cada una de las capas puede modificarse sin afectar al resto.
- Cada capa abarca un aspecto del desarrollo.

Tradicionalmente, la arquitectura del desarrollo de una aplicación cliente/servidor se hacía siguiendo un modelo de 2 capas, pero esto conlleva problemas de dificultad de mantenimiento, escalabilidad limitada, etc. Para solucionar estos problemas, la tendencia actual es que la arquitectura de un sistema distribuido se desarrolle en n-capas. Habitualmente una aplicación se estructura en las siguientes capas:

- La capa de presentación corresponde con la interfaz de usuario. Es la encargada de presentar la información y permitir la interacción del usuario con el sistema. También incluye la lógica de la presentación, donde se genera la interfaz de usuario de manera dinámica.
- La capa de negocio que incluye la lógica de negocio, donde se procesan los comandos del usuario y se realizan los cálculos necesarios. A su vez transporta información entre las capas adyacentes.
- La capa de datos es la encargada de almacenar y recuperar la información del soporte físico de almacenamiento. Realiza las conversiones necesarias para que la capa de negocio reciba los datos en un formato comprensible para ella.

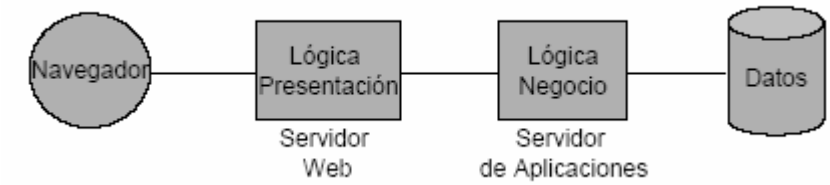

Figura 10. Arquitectura de la aplicación

Esta arquitectura mostrada en la Figura 10 es la utilizada en la herramienta desarrollada. La capa de presentación se ha realizado por medio de JSP, tecnología utilizada para generar contenido dinámico en forma de páginas HTML. La capa de la lógica de negocio se ha implementado con ayuda del patrón de diseño *Action* y el diseño de clases necesario para cada función. Para la capa de datos se ha optado por la gestión de los datos (cuentas de usuario) a través de ficheros para facilitar la posibilidad de crear una distribución de la herramienta en modo offline. En un principio se comenzó a utilizar un Sistema Gestor de Base de Datos, en concreto MySQL, pero se desechó en favor de la utilización de ficheros, para hacer más sencilla la instalación de la herramienta cuando no se está conectado a Internet (dadas las pocas necesidades de acceso a datos que requiere la aplicación).

Para la interacción entre las capas se ha elegido el uso del patrón de diseño *Modelo-Vista-Control*. Este patrón es aplicado a sistemas con un alto grado de interacción con el usuario a través de una interfaz gráfica y se ha convertido en uno de los patrones centrales en el diseño de aplicaciones Web. Éste patrón divide el

sistema en tres estratos:

- Vista: gestiona la presentación de información a través de la interfaz al usuario. Capa de presentación.
- Control: recoge los comandos del usuario y actúa sobre el modelo, la vista o ambos conforme a lo ordenado por el usuario.
- Modelo: gestiona los datos que conforman el estado del sistema, enviando información a la vista y actualizándose a petición del control. Aquí se incluye la capa de lógica de negocio y la capa de datos.

Este patrón funciona en un entorno Web tal como aparece en la Figura 11.

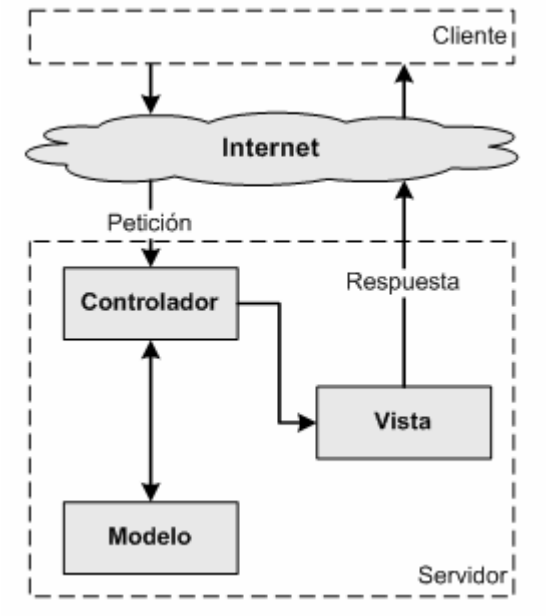

Figura 11. Patrón Modelo-Vista-Control

Para la aplicación del patrón MVC se podría haber optado por la utilización de un framework que incluya el uso de este patrón, como puede ser Struts, WebWorks o Spring. Sin embargo se ha optado por la utilización del modelo propuesto por el profesor de la Universidad de Murcia Francisco Javier Bermúdez para la realización de las prácticas de su asignatura, Desarrollo de Aplicaciones Distribuidas [11]. Se ha optado por esta decisión ya que esta propuesta es muy sencilla comparada con los otros frameworks, que ofrecen más opciones pero aumentando la complejidad de utilización. Debido a los conocimientos escuetos que poseemos sobre estos frameworks, las desventajas al usar alguno de ellos al desarrollar nuestra aplicación serían mucho mayores que las ventajas ya que llevaría mucho tiempo aplicarlo correctamente y se utilizarían muy pocas opciones de las ofrecidas por los mismos.

El patrón utilizado hace uso de otros patrones, tales como controlador de Fachada y Action. Concretamente, se utiliza una clase FrontController que hace de controlador de fachada. Este patrón proporciona una única interfaz a un conjunto de interfaces de un sistema. Esta clase FrontController se apoya en otra (PetitionHelper) para las gestión de las peticiones HTTP. Está última clase utiliza el patrón *Action* porque analiza la URL e instancia (por metaclases) un objeto *Command* para ejecutar las peticiones que le han llegado.

Por cada fichero que se creará de la estructura de compilación existirá una página JSP correspondiente. El código que escribirá el usuario para la resolución de un

problema se recogerá gracias al uso de textarea de HTML. En el caso más general se incluirá un textarea por cada método o función, otro para las funciones auxiliares que pueda implementar el usuario y otro para la declaración de clase. Para la página principal de cada algoritmo, se incluirá un textarea solamente para la inclusión de la función main de C++.

La herramienta Web recogerá el código introducido por el usuario en su navegador y creará la estructura de compilación. Esta estructura hace referencia a la estructura del diseño de clases y también a la organización del diseño de inclusiones de ficheros. El código introducido se debe haber escrito en C++, según sus normas léxicas, sintácticas y semánticas. Con esta estructura de clases del programa C++, se llamará al compilador en el servidor y si no hay ningún error, se generará un ejecutable como se puede observar en la Figura 12.

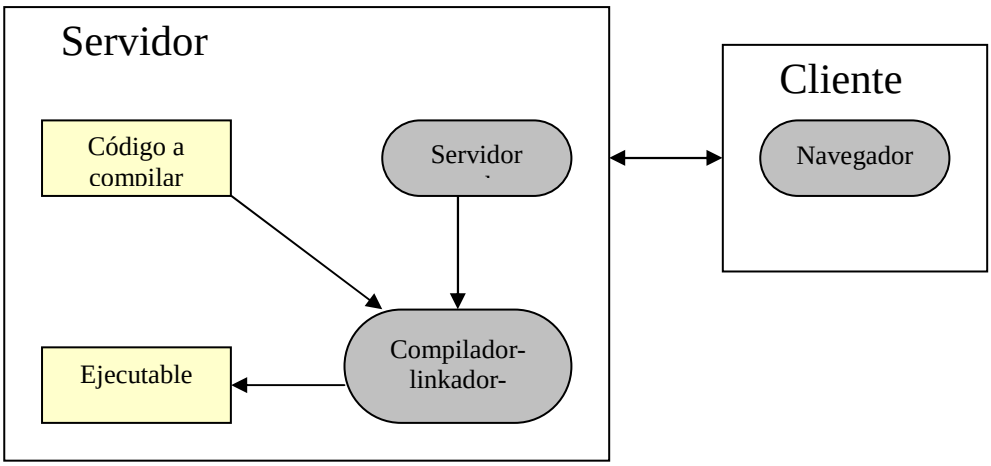

Figura 12. Estructura de compilación del proyecto

La entrada/salida permitida para los programas compilados será únicamente la estándar. Para ello, se permite que el usuario introduzca ficheros que serán redireccionados a la entrada estándar del programa que se ejecutará en el servidor. La salida estándar será recogida y preparada para mostrarla en el navegador a través de páginas dinámicas. Este funcionamiento se muestra en la Figura 13.

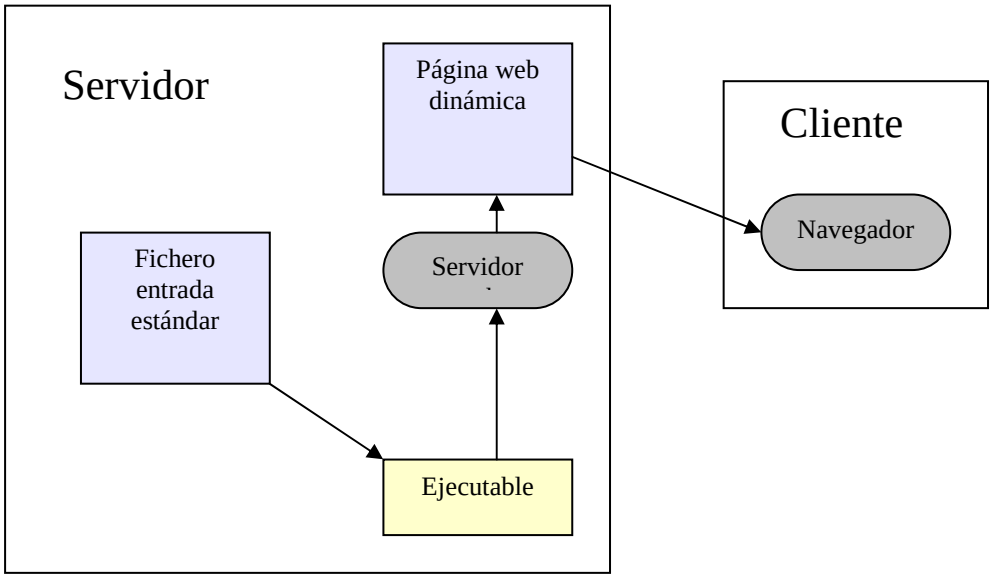

Figura 13. Estructura de ejecución de las aplicaciones generadas

Todos los archivos residirán en el servidor, haciendo necesaria la subida de archivos al servidor. Los archivos subidos tendrán diferentes cometidos:

- Ficheros que contienen el código a compilar.
- Ficheros con entradas para el programa.

Como todos los procesos relacionados con la compilación y ejecución se realizan en el servidor, se consigue separar la resolución de problemas de programación de la distribución/configuración de un entorno adecuado para realizar programas.

Para la compilación se ha elegido el compilador g++, que es el nombre del compilador para C++ de GNU. Para el despliegue del compilador se ha utilizado un fichero de propiedades para facilitar la configuración del compilador específico utilizado en el entorno donde se instale la aplicación Web, ya sea un servidor o un PC de escritorio.

Además, como se ha comentado anteriormente, también se utiliza un fichero para el almacenamiento de los usuarios. En este fichero se almacenan los identificadores de usuario y sus claves asociadas. Por motivos de seguridad, las claves almacenadas estarán encriptadas con MD5.

4. DISEÑO DE LOS ESQUEMAS ALGORÍTMICOS

La fijación de un esquema algorítmico concreto para un lenguaje de programación es una tarea ardua debido a que debe cumplir ciertos requisitos:

- Generalidad: Esto sirve para que pueda aplicarse el esquema algorítmico a todos los problemas que se pueden resolver mediante el algoritmo concreto.
- Utilidad: El diseño de un esquema algorítmico demasiado general puede derivar en un planteamiento demasiado estricto y que haga perder de vista los principios del algoritmo.
- Sencillez: El esquema debe ser fácil de comprender y utilizar.

Las técnicas de diseño de algoritmos estudiadas tienen características muy diferentes:

- Recursiva o secuencial.
- Posibilidad de encontrar 1 solución o todas.
- Posibilidad de devolver la mejor de las soluciones o la primera que se encuentre.
- Existencia de una función interesante de variar para el análisis de un problema.

Teniendo en cuenta las características y los requisitos de diseño de los esquemas, hemos primado la generalidad sobre la sencillez en ciertos casos, para que el esquema pueda recoger todos los ejemplos posibles para la técnica de diseño de algoritmos. A pesar de ello, en todos los casos el esquema posee bastante sencillez y utilidad para la comprensión de los algoritmos. Las peculiaridades de cada diseño se verán en el apartado correspondiente del esquema algorítmico.

El diseño de los esquemas se ha realizado teniendo como referencia principal el libro *Algoritmos y Estructuras de Datos, Volumen II* [10], que es el libro medular en la asignatura Algoritmos y Estructuras de Datos de 2º de las carreras de Informática en la Universidad de Murcia. La necesidad de generalidad en la búsqueda de los esquemas nos ha llevado, por ejemplo, al estudio de los diferentes esquemas existentes para integrarlos en uno solo, su adaptación a problemas más generales o su creación, ya que en las referencias utilizadas no se daba un esquema como tal en pseudocódigo o en algún lenguaje formal, sino una mera explicación sobre su funcionamiento. Además, estudiamos distintos problemas resueltos con la técnica escogida para establecer las ideas de generalidad de cada esquema.

Las funciones que contienen los esquemas algorítmicos serán inmutables. El programador deberá respetar totalmente las interfaces de llamada de los métodos implicados en el esquema algorítmico. A no ser que el diseño concreto de una técnica obligue a otra cosa, la función que implementa el esquema algorítmico se incluirá en un archivo con nombre *main.cpp*. Éste también contendrá la función *main* de C++. A partir de éste fichero se introducirán las clases auxiliares de la estructura a compilar, con la directiva *#include*.

Las clases complementarias contendrán un boceto para su implementación. Como mínimo en cada clase se define una declaración básica de la clase, sus atributos y métodos. Asimismo, se define un prototipo de las interfaces del constructor y destructor de la clase. Además, se prevé que el programador deberá incluir sobrecarga de constructores y otros métodos y funciones auxiliares. Las clases que

deben implementar métodos del esquema algorítmico también contendrán la interfaz de esos métodos.

```
class Clase {
//Declaraciones privadas de la clase
private:

//Declaraciones pÃºblicas de la clase
public:
Clase();
~Clase();
};

Clase::Clase() {

}

Clase::~Clase() {

}

//Otras funciones auxiliares
```

Código 1. Estructura de una clase

Cada una de las clases correspondientes a un esquema algorítmico se encontrará en un fichero que tendrá por nombre el de la clase (por ejemplo para la clase Problema, *problema.cpp*) y cuyo diseño es totalmente análogo al Código 1.

A continuación se describe el diseño pormenorizado de cada uno de los esquemas diseñados. Como hemos comentando en el apartado de División del Trabajo, los esquemas algorítmicos Divide y Vencerás y Programación Dinámica han sido desarrollados por A. Palazón y los esquemas de Ramificación y Poda y Backtracking por J. V. Jiménez.

4.1 Divide y vencerás

Divide y vencerás puede ser considerada una filosofía general de resolución de problemas aplicada en muchos campos, no solo en el entorno informático. Algunos ejemplos de otros ámbitos de aplicación son el marketing, la administración, la estrategia militar, etc. En nuestro contexto, la técnica de diseño de algoritmos Divide y Vencerás intenta resolver un problema dividiéndolo en varios subproblemas más sencillos de resolver, solucionarlos, y combinar los resultados para obtener la solución del problema original. El proceso de división continúa hasta que los subproblemas obtenidos son los suficientemente pequeños como para obtener una solución directa. Se llaman casos base a esos subproblemas más sencillos que se pueden resolver de manera directa.

El esquema implementado es recursivo, pudiendo aplicarse a todo tipo de problemas que se puedan resolver por descomposición. El diseño ha tenido como principio básico la generalidad, ya que los esquemas consultados se centraban en casos específicos de división del problema únicamente en un número fijo de subproblemas, con lo que el número de llamadas recursivas se establecían de forma fija también. El estudio de diversos problemas de aplicación de la técnica de diseño de algoritmos sacó a la luz la necesidad de realizar ciertas acciones para dividir, que tendrán su consecuencia directa en el número de subproblemas obtenidos, y por tanto

en las llamadas recursivas a realizar.

Las clases que debe implementar el programador serán:

- Solucion: Contiene la solución y un método para mostrarse por salida estándar.
- ConjuntoSoluciones: Contiene un conjunto de soluciones de ciertos subproblemas.
- Base: Contiene la definición de una base para un problema concreto.
- Problema: Contiene la definición del problema y los siguiente métodos implicados en el esquema:
  - pequeno: Comprueba si un problema es lo suficientemente pequeño como para resolverlo de forma directa. Recibe como entrada una base. Devuelve un booleano.
  - solucion: Devuelve la solución directa de un problema. Devuelve un objeto solución.
  - dividir_recursivo: Este método contendrá la funcionalidad referente a la división del problema en subproblemas y las llamadas recursivas correspondientes a la función divide_y_vencerás. Recibe como entrada una base. Devuelve un conjunto de soluciones. Esta función es necesaria ya que a priori no se puede saber en cuantos subproblemas se va a dividir el problema actual y de qué manera.
  - problema->combinar: Recibe el conjunto de soluciones obtenido anteriormente y los combina en una sola solución. Devuelve un objeto solución.

Los ficheros de la estructura a compilar son:

- main.cpp: Contiene la función de C main.
- problema.cpp: Contiene la clase Problema
- solucion.cpp : Contiene la clase Solucion
- conjuntosoluciones.cpp : Contiene la clase ConjuntoSoluciones.
- base.cpp: Contiene la clase Base.

La función que contendrá el esquema algorítmico se llamará *divide_venceras*. Ésta se hallará en el fichero *problema.cpp*, y no en *main.cpp*. Ésto se debe a que en el método *dividir_recursivo* se hacen llamadas recursivas a *divide_venceras*. Si esta función se incluye en *main.cpp*, también habría que llamarla desde *problema.cpp*, ya que contiene la definición del método *dividir_recursivo*. Ésto se lograría con una directiva *#include "main.cpp"*, que incluiría la definición de la función. Pero *main.cpp* también necesita la definición de problema. Así, debería incluir una directiva *#include "problema.cpp"* en *main.cpp*. Pero eso nos llevaría a recursividad en las inclusiones de ficheros, por lo que se ha optado por la inserción de la función *divide_venceras* en el fichero *problema.cpp*, correspondiente a la clase Problema. De todas maneras, ésto se realiza de manera transparente al usuario y no influye en la codificación de los problemas por parte del programador.

A continuación se explica el funcionamiento de la técnica algorítmica. Inicialmente el algoritmo comprueba si el problema es lo suficientemente pequeño como para obtener una solución directa. Si es así, devuelve esa solución directa. En caso de que el problema no sea lo suficientemente pequeño como para obtener la solución directa, se divide el problema en un conjunto de subproblemas que se resolverán llamando recursivamente a *divide_venceras*. Esta llamada se realizará en

el método *dividir_recursivo* de la clase problema. Esto retornará un conjunto de soluciones a los subproblemas, que se combinarán para obtener la solución que será devuelta a la función que ha llamado al método *divide_venceras*.

El esquema utilizado es el mostrado en Código 2.

```
Solucion * divide_venceras( Problema * problema, Base * b)
{
   ConjuntoSoluciones *s;
   if (problema->pequeno(b)) {
      return problema->solucion();
   }
     else {
      s=problema->dividir_recursivo(b);
      Solucion * solucion = problema->combinar(s);
      delete s;
      return solucion;
   }
}
```

Código 2. Esquema algorítmico Divide y Vencerás

Esta función deber ser llamada desde *main* o desde otra desde la que sea visible. El prototipo de *main* que se ofrece es un esquema básico en el que se crean las clases necesarias para la ejecución del algoritmo. Estas clases (Problema y Base) serán las que recibe el algoritmo. Además, se muestra la solución devuelta por la función *divide_venceras* y se destruyen las clases creadas.

Igualmente se ofrece una estructura de inclusiones de ficheros que el usuario podría cambiar según sus necesidades. El fichero *main.cpp* incluye a *problema.cpp*. En este último se incluye la función *divide_venceras* que contiene el esquema algorítmico, que es perfectamente visible desde *main*. El archivo *problema.cpp* incluye a su vez a *base.cpp* y *conjuntosoluciones.cpp*, que serán utilizados por los métodos de Problema y la función *divide_venceras* para la búsqueda de la solución. El fichero *conjuntosoluciones.cpp* (que contiene la clase ConjuntoSoluciones) incluye a su vez a *solucion.cpp*, ya que Solucion es el tipo de elemento del conjunto. Finalmente, *solucion.cpp* incluye estas dos directivas:

```
#include <iostream>
using namespace std;
```

para el uso de la librería estándar de C++ *iostream* y la facilidad en el uso del espacio de nombres *std*.

La estructura de clases se puede ver en la Figura 14.

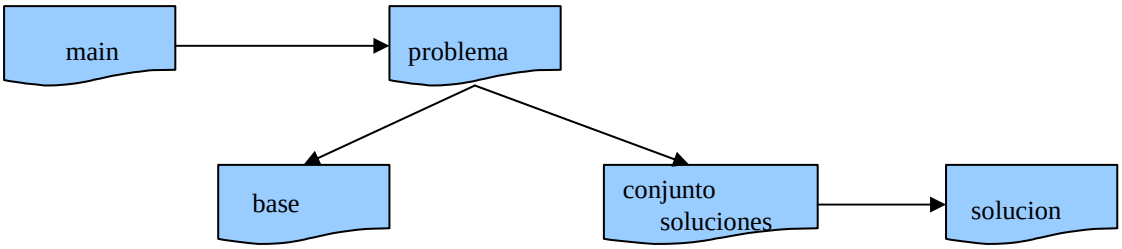

Figura 14. Diagrama de clases. Divide y Vencerás

4.2 Programación Dinámica

Es una técnica para obtener la solución óptima en problemas en los que la solución se logra con una serie de decisiones. Es una técnica ascendente, en la que se parte de la solución de los problemas de menor tamaño hasta la de más grande. Como almacena la solución de los problemas más pequeños, se evita tener que repetir cálculos en la resolución de los subproblemas.

No es usual que en la enseñanza de esta técnica se den los esquemas escritos como tales en lenguaje formal. Es más usual explicar su filosofía y funcionamiento, para que cada programador lo escriba como mejor le convenga. En base al estudio de ejemplos diversos y los esquemas adaptados a cada problema, se ha llegado a un esquema iterativo generalizado para todo tipo de problemas que se pueden resolver por descomposición.

Las clases que debe implementar el programador serán:

- Solucion: Contiene la solución final y un método para mostrar esa solución por salida estándar.
- DatosIniciales: Contiene los datos preliminares para inicializar los datos almacenados del problema.
- Iterador: Clase para recorrer los datos del problema. Contiene dos métodos para ésto:
  - haySiguiente: Devuelve un booleano, evaluando si en la estructura recorrida hay elemento siguiente.
  - siguiente: Devuelve el dato actual, y pone el cursor en el dato siguiente, si lo hay.
- Dato: Contiene un elemento individual de los datos almacenados por el problema.
- Datos: Contiene el espacio de memoria relacionado con los elementos almacenados por el problema. Los métodos de esta clase implicados en el esquema son:
  - inicializar: Recibe como entrada unos DatosIniciales, e inicializa con ellos los Datos del problema.
  - iterador: Devuelve un Iterador para los Datos del problema.
  - calcularActual: Recibe el Dato que actualmente se está leyendo. Con esta información calcula la solución para problemas de mayor tamaño.

- componerSolucion: Devuelve la solución final.

Los ficheros de la estructura a compilar son:

- main.cpp: Contiene la función de C main y el esquema algorítmico.
- solucion.cpp: Contiene la clase Solucion.
- datosiniciales.cpp: Contiene la clase DatosIniciales.
- iterador.cpp: Contiene la clase Iterador.
- dato.cpp: Contiene la clase Dato.
- datos.cpp: Contiene la clase Datos.

La función que contendrá el esquema algorítmico se llamará *PD*, y recibe como entrada Datos y Datos Iniciales y devuelve una clase Solucion. Para el recorrido de los elementos se ha optado por el patrón Iterator, que proporciona recorrido secuencial de una serie de elementos agregados sin tener que exponer su representación interna. Ésto hace que el alumno se acostumbre al uso de este patrón, muy extendido, a pesar de no estudiarlo hasta cursos posteriores. De hecho, este patrón es muy sencillo de entender y puede ser una buena introducción para el estudio posterior de patrones de diseño, o una base para la introducción de su uso en programas posteriores que realice el alumno.

A continuación se presenta el funcionamiento del esquema algorítmico. De manera preliminar, el algoritmo inicializa los Datos de su problema con los DatosIniciales que le han llegado en la llamada a la función. A continuación crea un Iterador para recorrer su estructura de Datos. Seguidamente pasa a recorrer la estructura, calculando en cada paso la solución para el problema actual en base a problemas más pequeños resueltos anteriormente. Por último, se retorna la Solucion al problema original tratando las soluciones encontradas.

El esquema utilizado es el mostrado en el Código 3.

```
Solucion * PD(Datos * datos,DatosIniciales * datosIniciales)
{
 datos->inicializar(datosIniciales);
 Iterador * it =datos->iterador();
 while (it->haySiguiente()) {
         Dato * d=it->siguiente();
         datos->calcularActual(d);
 }
 return datos->componerSolucion();
}
```

Código 3. Esquema algorítmico Programación Dinámica.

Esta función deber ser llamada desde *main* o desde otra desde la que sea visible. El prototipo de *main* que se ofrece es un esquema básico en el que se crean las clases necesarias para la ejecución del algoritmo. Éstas clases (Datos y DatosIniciales) serán las que recibe el algoritmo, al llamarse a través de la función *PD*. Además, se muestra la solución devuelta por la función *PD* y se destruyen las clases creadas.

La estructura de inclusión de los ficheros para este esquema será la siguiente. El fichero *main.cpp* incluye a *datos.cpp*, ya que necesita la clase Datos para la ejecución del algoritmo. A su vez, es presumible que Datos necesite a todas las demás clases auxiliares para el cálculo del problema, así que deberá incluir a todas las que

faltan. De esa manera, en el fichero *datos.cpp* habrá directivas para la inclusión de *datosiniciales.cpp*, *solucion.cpp*, *dato.cpp* e *iterador.cpp*. Por último, *solucion.cpp* incluye estas dos directivas:

```
#include <iostream>
using namespace std;
```

para el uso de la librería estándar de C++ *iostream* y la facilidad en el uso del espacio de nombres *std*, al igual que en el diseño de la técnica divide y vencerás.

La estructura de clases se puede ver en la Figura 15.

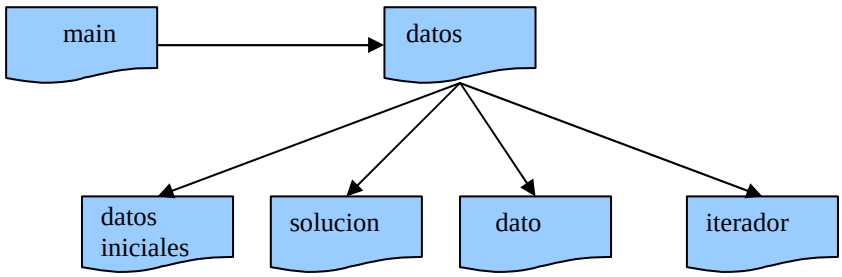

Figura 15. Diagrama de clases. Programación Dinámica

4.3 Backtracking

El Backtracking (método de retroceso o vuelta atrás) es una técnica general de resolución de problemas, aplicable a problemas de optimización, a problemas en los que se quiere encontrar una única solución o todas las soluciones, tanto si se sabe que hay solución como si no se sabe o incluso si no la hay.

El Backtracking realiza una búsqueda exhaustiva y sistemática en el espacio de soluciones. Por ello, suele resultar ineficiente.

Dependiendo del tipo de problema que se está resolviendo, el algoritmo continúa hasta que se obtiene una solución completa del problema, o hasta que no queden más posibles soluciones a evaluar. El resultado es equivalente a hacer un recorrido en profundidad en el árbol de soluciones. Sin embargo, este árbol es normalmente implícito, y no se almacena en memoria.

Las clases que debe implementar el programador serán:

- Problema: Contiene la definición del problema y los siguiente métodos implicados en el esquema:
  - problema->generarHermano(nodo): Para un nodo dado genera el hermano correspondiente.
  - problema->aumentarNivel(nodo): Aumenta el nivel del nodo actual para comenzar a estudiar los nodos del siguiente nivel del árbol.
  - problema->retrocederNivel(nodo): Es el paso de Backtracking, se trata de volver atrás en el nivel actual, para así volver buscando por otra rama, por otro hermano.
  - problema->tratarSolucion(nodo): Cuando encontramos una solución tenemos que tratarla, bien podemos acabar, o almacenar y quedarnos con todos, o simplemente quedarnos con la mejor solución encontrada.
  - problema->criterio(nodo): Esta función nos devuelve verdadero o

falso en función de si se cumple el criterio para pasar al siguiente nivel en el nodo actual.
  - problema->masHermanos(nodo): Nos devuelve verdadero o falso en función de si el nodo tiene mas hermanos o no.
  - problema->condiciónParada(nodo): Es la condición por la cual el algoritmo deja de correr.
  - problema->solucion(nodo): Nos devuelve verdadero o falso en función de si el nodo es una solución o no.

- Nodo: Contiene la definición de un nodo, que será la estructura de datos de la solución del problema.

Los ficheros de la estructura a compilar son:
- main.cpp: Contiene la función de C *main*.
- problema.cpp: Contiene la clase Problema.
- nodo.cpp : Contiene la clase Nodo.

Existe un fichero *lista.cpp*, pero es transparente al usuario. Se compila siempre y se detalla en las ayudas el conjunto de funciones posibles que se pueden utilizar para si el usuario así lo desea poder utilizar esta clase. Por ejemplo cuando se quieren encontrar todas las soluciones y queremos ir almacenándolas en una lista.

La función que contendrá el esquema algorítmico se llamará Backtracking.

A continuación se explica el funcionamiento de la técnica algorítmica. Se ha utilizado un esquema iterativo que no ha podido ser sacado del libro de la asignatura sino que se ha diseñado un esquema abstracto que engloba todos los casos contemplados por el libro, de modo que en él se pueden considerar los casos en los que no hay solución, hay varias, se sabe e incluso no se sabe si hay solución. El esquema itera mientras no se cumpla la condición de parada, implementada por el usuario. Mientras va generando nodos, y si es solución se trata. Este tratamiento también lo implementa el usuario. Si no es solución y se cumple el criterio se aumenta de nivel y se siguen generando los nodos de ese nivel y repitiendo el proceso. Si se llega a un punto en el que no se cumple el criterio, mientras no se cumpla la condición de parada y no haya más hermanos para el nodo se retrocede.

El esquema utilizado se muestra en Código 4 en la función Backtracking.

Esta función deber ser llamada desde *main* normalmente. El prototipo de *main* que se ofrece es un esquema básico en el que se crean las clases necesarias para la ejecución del algoritmo. Se crean los nodos y el problema y se llama a la función de Backtracking.

Igualmente se ofrece una estructura de inclusiones de ficheros que el usuario podría cambiar según sus necesidades. El fichero *main.cpp* incluye a *problema.cpp*. El archivo *problema.cpp* incluye a su vez a *lista.cpp* (a la cual el usuario no tiene acceso) y *lista.cpp* a su vez incluye al nodo, ya que en el caso de utilizar sería una lista de nodos. En muchas ocasiones el usuario necesitara mostrar los nodos, para ello tendrá que incluir en la clase nodo las directivas:

```
#include <iostream>
using namespace std;
```

```
#include "problema.cpp"

Nodo *backtracking(Nodo *nodo, Problema * p)
{
	do{
		nodo = p->generarHermano(nodo);
		if (p->solucion(nodo)){
			p->tratarSolucion(nodo);
		}
		if (p->criterio(nodo)){
			nodo = p->aumentarNivel(nodo);
		}
		else{
		while (!p->condicionParada(nodo) && !p->masHermanos(nodo)){
			nodo = p->retrocederNivel(nodo);
			}
		}
	}while(!p->condicionParada(nodo));
	return nodo;
}
```

Código 4. Esquema algorítmico Backtracking

para el uso de la librería estándar de C++ *iostream* y la facilidad en el uso del espacio de nombres *std*.

La estructura de clases se puede ver en la Figura 16.

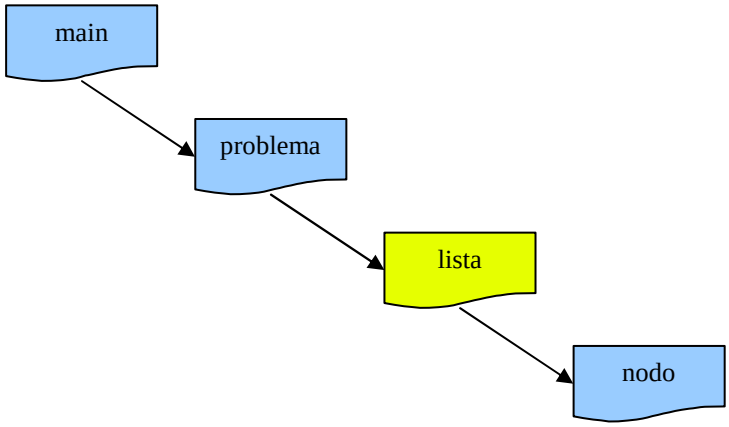

Figura 16. Diagrama de clases. Backtracking y Ramificación y Poda.

4.4 Ramificación y Poda

El método de diseño de algoritmos Ramificación y poda (también llamado *Ramificación y Acotación* o del término inglés, *Branch and Bound*) se aplica mayoritariamente para resolver cuestiones o problemas de optimización.

La técnica de Ramificación y poda recorre un árbol de soluciones, donde cada rama nos lleva a una posible solución posterior a la actual. La característica de esta técnica con respecto a otras anteriores (y a la que debe su nombre) es que el

algoritmo se encarga de detectar en qué ramificación las soluciones dadas ya no están siendo óptimas, para podar esa rama del árbol y no continuar malgastando recursos y procesos en casos que se alejan de la solución óptima.

El objetivo de la poda es eliminar aquellos nodos que no lleven a soluciones buenas. Para ello el algoritmo lleva en cuenta las cotas de coste estimado inferior y superior para cada rama.

Las clases que debe implementar el programador serán:

- Problema: Contiene la definición del problema y los siguientes métodos implicados en el esquema:
  - problema->CS(nodo): Estima cual es la cota superior asociada al nodo en cuestión.
  - problema->CI(nodo): Estima cual es la cota inferior asociada al nodo en cuestión.
  - problema->coste(nodo): Estima el coste asociado al nodo en cuestión.
  - problema->obtenerHijos(nodo): Genera una lista con los hijos del nodo en cuestión.
  - problema->mejorSolucionFinal(nodo): Devuelve verdadero si el nodo en cuestión es la mejor solución final encontrada hasta el momento.
  - problema->mejor(nodo1,nodo2): Devuelve verdadero si el nodo1 es mejor que el nodo2. El criterio de mejor dependerá del tipo de problema. Por ejemplo en minimización mejor sería menor coste, mientras que en maximización mejor sería el mayor coste.
  - problema->solucionFinal(nodo): Devuelve verdadero si el nodo es una solución final del problema.

- Nodo: Contiene la definición de un nodo, que será la estructura de datos de la solución del problema.

Los ficheros de la estructura a compilar son:

- main.cpp: Contiene la función de C *main*.
- problema.cpp: Contiene la clase Problema
- nodo.cpp : Contiene la clase Nodo

Existe un fichero *lista.cpp*, pero es transparente al usuario. Al usuario sólo se le obliga a escoger entre tres métodos de inserción: LIFO, FIFO, LC-LIFO. Una vez escogido el tipo de inserción la aplicación genera el fichero *lista.cpp* correspondiente, evitándose así la implementación de toda una lista por parte del usuario.

La función que contendrá el esquema algorítmico se llamará *ramificaciónYpoda*.

A continuación se explica el funcionamiento de la técnica algorítmica. El algoritmo utiliza una lista de nodos vivos (LNV) donde inicialmente se encuentra el nodo raíz. En esta lista se encuentran los nodos de los que todavía no se han generado los hijos y que no se han podado. La generación de nuevos nodos se realiza tomando un nodo de la lista y generando todos sus hijos (calculando su beneficio estimado, sus cotas inferior y superior, viendo si se puede podar e incluyendo en la lista de nodos vivos si no se ha podado). Hay distintas estrategias de elección de un nodo de la lista de nodos vivos para generar sus hijos, en la aplicación ofrecemos las siguientes:

- Estrategia FIFO: Se generan los descendientes de los nodos en el mismo orden en que han sido generados. Por tanto la lista de nodos vivos se implementa como una cola.
- Estrategia LIFO: En este caso se implementa como una pila.
- Estrategia LC-FIFO: Cuando se dispone de una estimación del beneficio se elige entre todos los nodos vivos, como nodo para generar sus hijos, el nodo de mayor beneficio estimado.

El esquema utilizado es el mostrado en el Código 5.

```
#include "problema.cpp"
Nodo* ramificacionYpoda(Nodo *raiz, Nodo *s, Problema * p)
{
        Lista *LNV = new Lista();
        Lista *hijos = new Lista();
        Nodo *x,*y;
        double C;
        LNV->Insertar(raiz);
        C = p->CS(raiz);
        while (!LNV->ListaVacia()) {
                LNV->Primero();
                x = LNV->ValorActual();
                LNV->Borrar();
                if (p->CI(x) < C){
                     hijos = p->obtenerHijos(x);
                     while (!hijos->ListaVacia()){
                            hijos->Primero();
                            y = hijos->ValorActual();
                            hijos->Borrar();
                            if (p->mejorSolucionFinal(y,s)){
                                  s=y;
                                  C = p->mejor(C,p->coste(y));
                            }
                           else if (!p->solucionFinal(y) && p->CI(y)<C){
                                  LNV->Insertar(y);
                                  C=p->mejor(C,p->CS(y));
                            }
                      }
                 }
        }
        LNV->~Lista();
        hijos->~Lista();
        return s;
}
```

Código 5. Esquema algorítmico Ramificación y Poda.

Esta función deber ser llamada desde *main* normalmente. El prototipo de *main* que se ofrece es un esquema básico en el que se crean las clases necesarias para la ejecución del algoritmo. Se crean los nodos y el problema y se llama a la función de ramificación y poda.

Igualmente se ofrece una estructura de inclusiones de ficheros que el usuario podría cambiar según sus necesidades. El fichero *main.cpp* incluye a *problema.cpp*. El archivo *problema.cpp* incluye a su vez a *lista.cpp* (a la cual el usuario no tiene acceso) y *lista.cpp* a su vez incluye al nodo, ya que se trata de una lista de nodos. En muchas ocasiones el usuario necesitará mostrar los nodos, para lo que tendrá que incluir en la clase nodo las directivas:

```
#include <iostream>
using namespace std;
```

para el uso de la librería estándar de C++ *iostream* y la facilidad en el uso del espacio de nombres *std*.

El estructura de clases se muestra en la Figura 16.

5. CONCLUSIONES Y TRABAJOS FUTUROS

Este trabajo presenta una herramienta para la enseñanza de esquemas algorítmicos. Se está considerando su utilización para el próximo curso de manera experimental en algunos grupos de prácticas de la asignatura de Algoritmos y Estructuras de Datos de segundo curso. Además, puede ser usada de manera autónoma, tanto conectándose a la página de la aplicación como descargándola de ella.

En cuanto a las conclusiones obtenidas del proyecto, nos ha supuesto un esfuerzo extra al no haber cursado ninguna asignatura relacionada con el desarrollo de aplicaciones web a lo largo de nuestra carrera. Debido a esto, nos ha aportado un aprendizaje complementario ya que habíamos orientado nuestra enseñanza en la carrera en otras áreas. Además, el trabajo realizado nos ha satisfecho debido a la introducción a los pormenores del desarrollo de este tipo de aplicaciones, conjugándolo con nuestro interés por el área de estudio de la algoritmia.

En paralelo al desarrollo de la memoria de este proyecto, se realizó un trabajo para el primer Simposio Docente Sobre el Aprendizaje y Enseñanza de Algoritmos, un foro de exposición de ideas sobre la enseñanza de esta materia en las carreras universitarias de Informática, a celebrar en octubre de 2008 en Granada. Este trabajo está pendiente de aprobación por parte de los correctores del Simposio.

El abanico de posibilidades de desarrollo futuro que presenta esta aplicación es muy amplio. Las líneas de este trabajo se pueden dividir en 2: Desarrollo de líneas docentes y relacionadas con la algoritmia; y líneas de investigación en la mejora de la aplicación distribuida.

Con respecto al desarrollo de líneas nuevas funcionalidades docentes y relacionadas con la algoritmia, podemos enumerar:

- Se pueden incluir nuevos esquemas algorítmicos de tipo secuencial de manera sencilla. La infraestructura para el desarrollo de nuevos algoritmos ya está creada y su inclusión es bastante simple.
- También se está trabajando en la inclusión de esquemas algorítmicos paralelos, lo que servirá para utilizarse en una asignatura de introducción a la programación paralela.
- Adicionalmente, la herramienta podrá adaptarse para ser usada en entornos científicos (no sólo didácticos) para resolver de forma cercana a la óptima, problemas enviados por usuarios y que siguieran uno de los esquemas en la aplicación.

Con respecto a las posibles mejoras de la aplicación, se pueden observar las siguientes direcciones:

- La herramienta utiliza como compilador g++. Esto es fácilmente modificable gracias al fichero de propiedades, que elimina las dependencias del código. No obstante, existen otras dependencias con el compilador algo mas complejas que no han sido subsanadas. Por ejemplo, la lectura del fichero que nos informa del resultado del proceso de compilación. Se podría definir un formato independiente de información de este resultado y que la aplicación programada leyese datos de este formato. Deberíamos entonces crear una indirección con el

formato propio del compilador g++ a través de adaptadores para tratar dicha salida (es decir, adaptadores encargados de aplicar una transformación del formato propio del compilador en cuestión al formato estándar definido para la aplicación).

- Ya que la inclusión de nuevos compiladores es bastante sencilla, la aplicación podría seguir la línea de introducir nuevos lenguajes de programación, modificando los prototipos de funciones propuestos para el lenguaje C++ e insertando los propios del lenguaje a desarrollar.
- La aplicación se podría ampliar introduciendo un analizador sintáctico para el código de la compilación, ya que existen librerías que facilitan mucho esta labor.
- La aplicación aplicación podría evolucionar hacia un Framework que haga más sencilla la inclusión de nuevos esquemas a través de una interfaz bien definida con el usuario.

6. BIBLIOGRAFIA

[1] Biliana, K.; Thibaut, D.; *Sorting Algorithms*. 1997

http://maven.smith.edu/ thiebaut/java/sort/demo.html

[2] Gosling, J.; Harrison, J.; Boritz, J. *Sorting Algorithms Demo*. 2001

http://www.cs.ubc.ca/ harrison/Java/sortingdemo.html

[3] Laza, R.; Glez-Peña, D.; Méndez, J. R.; Fdez-Riverola, F.; Baltasar Garca, J.; Reboiro, M.; *ALT: Algorithm Learning Tool*. 2007

http://trevinca.ei.uvigo.es/ rlaza/teoria.htm

[4] Rosales, I; *Animadorde Algoritmos*. Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla.

http://www.lsi.us.es/docencia/asignaturas/ip1/trabajos/IndiceAnimador.htm

[5] Cachairo González, J. F.; Díaz Rodríguez, M.; Vallecillo Moreno, A.; *Animación y Simulación de Algoritmos Paralelos de Exploración de Grafos*. Departamento de Lenguajes y Ciencias de la Computación. E.T.S.I. Informática. Universidad de Málaga.

http://www.lcc.uma.es/~av/Proyectos/grafos/articulo.pdf

[6] Fernández, V.; Urdangarin, I.; Zelaia, A.; *Herramienta gráfica para el aprendizaje del algoritmo simplex*. Dpto. de Ciencia de la Computación e IA Facultad de Informática Universidad del País Vasco.

http://www.sc.ehu.es/ccwikera/docs/simplex_jenui2006.pdf

[7] Cárdenas Varela, F.; Castillo Izquierdo, N.; Daza Castillo, E.; *Editor e Intérprete de Algoritmos Representados en Diagramas de Flujo – DFD*. Universidad del Magdalena Santa Marta – Colombia.

http://mx.geocities.com/ikky_fenix/proy_dfd.html

[8] *System for managing programming contests on the Web*.

http://mooshak.dcc.fc.up.pt/~zp/mooshak/

[9] Sun Microsystems, Inc. *Developer Resources for Java Technology.*

http://java.sun.com/

[10] Giménez Cánovas, D.; Cervera López, J.; García Mateos, G.; Marín Pérez, N. *Algoritmos y Estructuras de Datos. Volumen II*. Diego Marín, 2003.

[11] Bermúdez Ruiz, F. J. *Desarrollo de Aplicaciones Distribuidas.*

http://dis.um.es/~jbermudez/dad/

7. ANEXOS

Anexo I. Estructura de ficheros y clases para el esquema algorítmico Divide y Vencerás

En todas las clases se incluye: la declaración e implementación del constructor y del destructor de la clase, y una zona para la inclusión de otros métodos y funciones auxiliares por parte del usuario.

Para la compilación del algoritmo se crean 5 ficheros:

- *main.cpp* (función de C *main*): Se incluye el fichero *problema.cpp* que contiene la clase Problema y se ofrece un boceto para el *main*.

```
#include "problema.cpp"
 int main(int argc, char *argv[]) {
    Base * b = new Base();
    Problema *problema = new Problema();
    Solucion * solucion =divide_venceras(problema,b);
    solucion->mostrar();
    delete problema;
    delete solucion;
    delete b;
}
```

- *problema.cpp*(clase Problema): Se incluyen los ficheros *base.cpp* y *conjuntosoluciones.cpp* que contienen las clases Base y ConjuntoSoluciones respectivamente. En este fichero se encuentra la función *divide_venceras*, que contiene el esquema algorítmico. Además se ofrece una implementación vacía y la interfaz de los métodos implicados en el esquema.

```
#include "base.cpp"
#include "conjuntosoluciones.cpp"
class Problema {
//Declaraciones privadas de la clase
private:

//Declaraciones publicas de la clase
public:
        Problema();
        bool pequeno(Base *base);
        Solucion * solucion();
        ConjuntoSoluciones * dividir_recursivo(Base  * b);
        Solucion * combinar(ConjuntoSoluciones * s);
        ~Problema();
};


Solucion * divide_venceras( Problema * problema, Base * b) {
   ConjuntoSoluciones *s;
   if (problema->pequeno(b)) {

      return problema->solucion();
   }
     else {
      s=problema->dividir_recursivo(b);
      Solucion * solucion = problema->combinar(s);
      delete s;
      return solucion;
   }
}
```

• *solucion.cpp* (clase Solucion): En este archivo se incluye la librería estándar de C++ *iostream*. Además se incluye la directiva *using* para tener disponible para su uso directo el espacio de nombres *std*. El método mostrar es el más destacable del que se ofrece su interfaz y su implementación vacía. Este método será usado por el usuario para mostrar por salida estándar la solución al problema.

```
#include <iostream>
using namespace std;

class Solucion {
//Declaraciones privadas de la clase
private:

//Declaraciones publicas de la clase
public:
        Solucion();
        void mostrar();
        ~Solucion();
};

Solucion::Solucion() {

}

void Solucion::mostrar() {

}

Solucion::~Solucion() {

}

//Otras funciones auxiliares
```

• *base.cpp* (clase Base): Boceto básico en el que se incluyen, como en las demás clases, las declaraciones e implementaciones vacías de constructor y destructor de la clase y un área donde se prevé que se incluirán otros métodos y funciones.

```
class Base {
//Declaraciones privadas de la clase
private:

//Declaraciones publicas de la clase
public:
        Base();
        ~Base();
};

Base::Base() {

}

Base::~Base() {

}

//Otras funciones auxiliares
```

- *conjuntosoluciones.cpp* (clase ConjuntoSoluciones): Se incluye el fichero *solucion.cpp* que contiene la clase Solucion. Al igual que la clase Base, sólo contiene un boceto básico de una clase en C++.

```
#include "solucion.cpp"
class ConjuntoSoluciones {
//Declaraciones privadas de la clase
private:

//Declaraciones publicas de la clase
public:
        ConjuntoSoluciones();
        ~ConjuntoSoluciones();
};

ConjuntoSoluciones::ConjuntoSoluciones() {

}

ConjuntoSoluciones::~ConjuntoSoluciones() {

}

//Otras funciones auxiliares
```

Anexo II. Estructura de ficheros y clases para el esquema algorítmico Programación Dinámica

En este caso para la compilación del algoritmo se crean 6 ficheros:

• *main.cpp* (función de C *main*): Se incluye el fichero *datos.cpp* que contiene la clase Datos, se inserta el esquema en la función PD y se ofrece un boceto para el *main*.

```
#include "datos.cpp"
Solucion * PD(Datos * datos,DatosIniciales * datosIniciales) {
 datos->inicializar(datosIniciales);
 Iterador * it =datos->iterador();
 while (it->haySiguiente()) {
        Dato * d=it->siguiente();
        datos->calcularActual(d);
 }
 return datos->componerSolucion();
}

int main(int argc, char *argv[]) {
        Datos * datos = new Datos();
        DatosIniciales * datosIni = new DatosIniciales();
        Solucion * sol =PD(datos,datosIni);
        sol->mostrar();
        delete datos;
        delete datosIni;
        delete sol;
}
```

- *datos.cpp* (clase Dato): Se incluyen los ficheros *solucion.cpp*, *datosiniciales.cpp*, *dato.cpp* e *iterador.cpp* que contienen las clases Solucion, DatosIniciales, Dato e Iterador respectivamente. Además se ofrece una implementación vacía y la interfaz de los métodos implicados en el esquema.

```
#include "solucion.cpp"
#include "datosiniciales.cpp"
#include "dato.cpp"
#include "iterador.cpp"

class Datos {
//Declaraciones privadas de la clase
private:

//Declaraciones publicas de la clase
public:
        Datos();
        void inicializar(DatosIniciales * datosIniciales);
        Iterador * iterador();
        void calcularActual(Dato * d);
        Solucion * componerSolucion();
        ~Datos();
};

Datos::Datos() {

}

void Datos::inicializar(DatosIniciales * datosIniciales)
{

}

Iterador * Datos::iterador()  {

}

void Datos::calcularActual(Dato * d) {

}

Solucion * Datos::componerSolucion() {

}

Datos::~Datos() {

}

//Otras funciones auxiliares
```

• *solucion.cpp* (clase Solucion): En este archivo se incluye la librería estándar de C++ *iostream*. Además se incluye la directiva *using* para tener disponible para su uso directo el espacio de nombres *std*. El método mostrar es el más destacable del que se ofrece su interfaz y su implementación vacía. Este método será usado por el usuario para mostrar por salida estándar la solución al problema.

```
#include <iostream>
using namespace std;

class Solucion {
//Declaraciones privadas de la clase
private:


//Declaraciones publicas de la clase
public:
        Solucion();
        void mostrar();
        ~Solucion();
};

Solucion::Solucion() {

}

void Solucion::mostrar() {

}

Solucion::~Solucion() {

}

//Otras funciones auxiliares
```

• *datosinciales.cpp* (clase DatosIniciales): Boceto básico en el que se incluyen, como en las demás clases, las declaraciones e implementaciones vacías de constructor y destructor de la clase y un área donde se prevé que se incluirán otros métodos y funciones.

```
class DatosIniciales {
//Declaraciones privadas de la clase
private:

//Declaraciones publicas de la clase
public:
        DatosIniciales();
        ~DatosIniciales();
};

DatosIniciales::DatosIniciales()  {

}

DatosIniciales::~DatosIniciales()  {

}

//Otras funciones auxiliares
```

• *iterador.cpp (*clase Iterador): Esta clase contiene, como métodos más destacables, *haySiguiente* y *siguiente*, usados para realizar el recorrido con el iterador.

```
class Iterador {
//Declaraciones privadas de la clase
private:

//Declaraciones publicas de la clase
public:
        Iterador();
        bool haySiguiente();
        Dato * siguiente();
        ~Iterador();

};

Iterador::Iterador() {

}

bool Iterador::haySiguiente() {

}

Dato * Iterador::siguiente()  {

}

Iterador::~Iterador()  {

}

//Otras funciones auxiliares
```

• *dato.cpp* (clase Dato): Boceto básico en el que se incluyen, como en las demás clases, las declaraciones e implementaciones vacías de constructor y destructor de la clase y un área donde se prevé que se incluirán otros métodos y funciones.

```
class Dato {
//Declaraciones privadas de la clase
private:

//Declaraciones publicas de la clase
public:
        Dato();
        ~Dato();
};

Dato::Dato(){

}

Dato::~Dato()  {

}

//Otras funciones auxiliares
```

Anexo III. Estructura de ficheros y clases para el esquema algorítmico Backtracking

En este caso para la compilación del algoritmo se crean 4 ficheros:

• *main*.cpp (función de C *main*): Se incluye el fichero *problema.cpp* que contiene la clase Problema, se inserta el esquema en la función *backtracking* y se ofrece un boceto para el *main*. Hemos considerado dos esquemas. Uno para el caso de maximización y otro para minimización.

```
#include "problema.cpp"

Nodo *backtracking(Nodo *nodo, Problema * p)
{
        do{


                nodo = p->generarHermano(nodo);

                if (p->solucion(nodo)){
                        p->tratarSolucion(nodo);
                }

                if (p->criterio(nodo)){
                        nodo = p->aumentarNivel(nodo);

                }
                else{

                        while (!p->condicionParada(nodo) && !p->masHermanos(nodo)){
                                nodo = p->retrocederNivel(nodo);
                        }

                }



        }while(!p->condicionParada(nodo));

        return nodo;
}

int main(int argc, char *argv[]) {
        Problema *problema = new Problema();
   Nodo *nodo = new Nodo();
   Nodo * solucion = backtracking(nodo, problema);
   delete problema;
   delete nodo;
   delete solucion;
}
```

• *problema.cpp* (clase Problema): Se incluye el fichero *lista.cpp*, que contienen la clase Lista. Además se ofrece una implementación vacía y la interfaz de los métodos implicados en el esquema.

En ningún caso el usuario tiene que implementar dicha clase. Esta clase ya viene implementada por defecto. Se le ofrece una ayuda a través de la cual se le muestran los métodos para la utilización de la clase lista por si le es necesaria, ya que tampoco es obligatoria su utilización. Está pensada sobre todo para el caso en que al tratar las soluciones quiera ir almacenándolas todas, de ahí que sea una lista de nodos.

```
#include "lista.cpp"


class Problema {
//Declaraciones privadas de la clase
private:
double *p;
double *b;
int tam;
double M;
Nodo *sol;


//Declaraciones publicas de la clase
public:

Problema();
Nodo *generarHermano(Nodo *nodo);
Nodo *aumentarNivel(Nodo *nodo);
Nodo *retrocederNivel(Nodo *nodo);
void tratarSolucion(Nodo *nodo);
bool criterio(Nodo *nodo);
bool masHermanos(Nodo *nodo);
bool condicionParada(Nodo *nodo);
bool solucion(Nodo *nodo);
~Problema();

};

Problema::Problema() {

}


Nodo *Problema::generarHermano(Nodo *nodo){


}


Nodo *Problema::aumentarNivel(Nodo *nodo) {

}

Nodo *Problema::retrocederNivel(Nodo *nodo) {

}
```

```
void Problema::tratarSolucion(Nodo *nodo) {
/*******************************************************/
/*************PULSE SOBRE EL BOTÓN*************/
/*********DE COMO UTILIZAR LAS LISTAS**********/
/***************SITUADO ARRIBA.*******************
/*******************************************************/
}

bool Problema::criterio(Nodo *nodo){

}
bool Problema::masHermanos(Nodo *nodo) {

}
bool Problema::condicionParada(Nodo *nodo){

}
bool Problema::solucion(Nodo *nodo){

}

Problema::~Problema() {

}

//Otras funciones auxiliares
```

• *nodo.cpp* (clase Nodo): En esta clase nodo, se dejan a implementar tanto el constructor como el destructor del objeto.

En alguna ocasión puede que el usuario quiera mostrar un nodo. Para ello, y ya que no siempre tendrá por qué ser así, habrá que incluir la librería *iostream* y la directiva *using namespace std*;

```
class Nodo {
//Declaraciones privadas de la clase

private:

//Declaraciones publicas de la clase
public:


Nodo();
~Nodo();
};

Nodo::Nodo() {

}

Nodo::~Nodo() {

}

//Otras funciones auxiliares
```

- *lista.cpp* (clase Lista): La clase lista que se utiliza implícitamente, del mismo modo que en ramificación y poda, tiene la siguiente estructura e incluye a la clase *nodo.cpp* ya que se trata de una lista de nodos.

```
#include "nodo.cpp"
class Celda {
  public:
  Nodo valor;
  Celda *siguiente;
  Celda(Nodo nodo, Celda *sig = NULL)
  {

        valor = nodo;
        siguiente = sig;
   }

   friend class lista;

};

typedef Celda *pcelda;

class Lista {
  public:
   Lista() { anterior = primero = actual = NULL; }
   ~Lista();

   void Insertar(Nodo *nodo);
   void InsertarLNV(Nodo *nodo);
   void Borrar();
   bool ListaVacia() { return (primero== NULL || longitud==0); }
   void Siguiente();
   void Primero();
   bool Actual() { return actual != NULL; }
   Nodo* ValorActual() { return &actual->valor; }
   int Longitud() { return longitud; }


  private:
   pcelda primero;
   pcelda anterior;
   pcelda actual;
   int longitud ;
};

Lista::~Lista()
{
  pcelda aux;
  int cont = longitud;
  while(cont!=0) {
     Primero();
     Borrar();
     cont--;
}

primero = NULL;
actual = NULL;
anterior = NULL;
}
```

```
void Lista::Insertar(Nodo *nodo)
{
	if(ListaVacia()) {

		primero = new Celda(*nodo, primero);
		if (anterior!=NULL) {
			nodo->mostrar();
			delete anterior;
		}
		anterior = NULL;
		actual = primero;
		longitud = 1;
	}
	else {
		anterior = actual;
		actual->siguiente = new Celda(*nodo,anterior->siguiente);
		actual = actual -> siguiente;
		longitud++;
	}
}

void Lista::Borrar()
{
	pcelda siguiente,aux;

	if (actual->siguiente == NULL) {

		if (actual == primero) {
			delete primero;
			actual = NULL;primero=NULL;

		}
		else{
			delete actual;
			actual = anterior;
			actual->siguiente = NULL;
			aux = primero;
			while (aux!=actual){
				anterior = aux;
				aux = aux->siguiente;
			}
		}
	}
	else{
		if (actual == primero) {
			primero = actual->siguiente;
			anterior = NULL;
			delete actual;
			actual = primero;

		}
		else{
			actual = actual->siguiente;
			delete anterior->siguiente;
			anterior->siguiente = actual;

		}

	}
	longitud--;
}
```

```
void Lista::Siguiente()
{

  anterior = actual;
  actual = actual->siguiente;
}

void Lista::Primero()
{

  anterior = NULL;
  actual = primero;
}

void Lista::InsertarLNV(Nodo *nodo){

Insertar(nodo);
}
```

Como se puede observar en este caso está implementada completamente, simplemente se deja al usuario la opción de utilizarla o no, ya que no es necesaria su utilización. Al igual que en el caso anterior se muestra al usuario mediante una ayuda los métodos disponibles para su utilización.

Anexo IV. Estructura de ficheros y clases para el esquema algorítmico Ramificación y Poda

En este caso para la compilación del algoritmo se crean 4 ficheros:

• *main.cpp* (función de C *main*): Se incluye el fichero *problema.cpp* que contiene la clase Problema, se inserta el esquema en la función *ramificacionYpoda* y se ofrece un boceto para el *main*. Hemos considerado dos esquemas. Uno para el caso de maximización y otro para minimización.

CASO MAXIMIZACIÓN:

```
#include "problema.cpp"

Nodo* ramificacionYpoda(Nodo *raiz, Nodo *s, Problema * p)
{
Lista *LNV = new Lista();
Lista *hijos = new Lista();
Nodo *x,*y;
double C;

LNV->Insertar(raiz);
C = p->CI(raiz);

while (!LNV->ListaVacia()) {
   LNV->Primero();
   x = LNV->ValorActual();
   LNV->Borrar();
   if (p->CS(x) > C){

      hijos = p->obtenerHijos(x);
         while (!hijos->ListaVacia()){
         hijos->Primero();
         y = hijos->ValorActual();
         hijos->Borrar();
         if (p->mejorSolucionFinal(y,s)){

            s=y;
            C = p->mejor(C,p->coste(y));


         }
         else if (!p->solucionFinal(y) && p->CS(y)>C){
            LNV->Insertar(y);
            C=p->mejor(C,p->CI(y));

         }

      }

   }
}

LNV->~Lista();
hijos->~Lista();
return s;

}
```

CASO MINIMIZACIÓN:

```
#include "problema.cpp"

        Nodo* ramificacionYpoda(Nodo *raiz, Nodo *s, Problema * p)
        {
        Lista *LNV = new Lista();
        Lista *hijos = new Lista();
        Nodo *x,*y;
        double C;

        LNV->Insertar(raiz);
        C = p->CS(raiz);

        while (!LNV->ListaVacia()) {
           LNV->Primero();
           x = LNV->ValorActual();



           LNV->Borrar();
           if (p->CI(x) < C){

              hijos = p->obtenerHijos(x);
                  while (!hijos->ListaVacia()){
                 hijos->Primero();
                 y = hijos->ValorActual();
                 hijos->Borrar();
                 if (p->mejorSolucionFinal(y,s)){

                    s=y;
                    C = p->mejor(C,p->coste(y));


                 }
                else if (!p->solucionFinal(y) && p->CI(y)<C){
                    LNV->Insertar(y);
                    C=p->mejor(C,p->CS(y));



                 }

              }

           }
        }

        LNV->~Lista();
        hijos->~Lista();
        return s;

}
```

La función *main* es la misma para los dos tipos de esquemas.

```
int main(int argc, char *argv[]) {
    Problema *problema = new Problema();
    Nodo *raiz = new Nodo();
    Nodo *s = new Nodo();
    Nodo * nodo = ramificacionYpoda(raiz, s, problema);
    delete problema;
    delete nodo;
    delete raiz;
    delete s;
}
```

• *problema.cpp* (clase Problema): Se incluye el fichero *lista.cpp*, que contienen la clase Lista. Además se ofrece una implementación vacía y la interfaz de los métodos implicados en el esquema.

En ningún caso el usuario tiene que implementar dicha clase. Esta clase ya viene implementada por defecto en sus tres versiones: LIFO, FIFO, LC-LIFO. De modo que el usuario según escoja el método desde el esquema se ofrecerá una u otra. Además se le ofrece una ayuda a través de la cual se le muestran los métodos para la utilización de la clase lista, necesaria en el momento de la generación de los hijos de un nodo. Y por si le es necesaria en funciones auxiliares.

```
#include "lista.cpp"

class Problema {
//Declaraciones privadas de la clase
private:


//Declaraciones publicas de la clase
public:
Problema();
Problema();
double CS(Nodo *nodo);
double CI(Nodo *nodo);
double coste(Nodo *nodo);
Lista* obtenerHijos(Nodo *nodo);
bool mejorSolucionFinal(Nodo *nodoY, Nodo *nodoS);
double mejor(double x, double y);
bool solucionFinal(Nodo *nodo);
~Problema();
};

Problema::Problema() {

}


double Problema::CS(Nodo *nodo) {

}


double Problema::CI(Nodo *nodo) {

}
```

```
double Problema::coste(Nodo *nodo) {

}

Lista Problema::obtenerHijos(Nodo *nodo) {
/*********************************************************/
/*************PULSE SOBRE EL BOTÓN**************/
/*********DE COMO UTILIZAR LAS LISTAS**********/
/***************SITUADO ARRIBA.*******************
/*********************************************************/
Lista *hijos = new Lista();
//Construir la lista



return hijos;
}

bool Problema::mejorSolucionFinal(Nodo *nodoY, Nodo *nodoS) {

}

double Problema::mejor(double x, double y) {

}

bool Problema::solucionFinal(Nodo *nodo) {

}
Problema::~Problema() {

}
//Otras funciones auxiliares
```

- *nodo.cpp* (clase Nodo): La clase nodo ya lleva implementados por defecto ciertos métodos que será obligatorios a utilizar en el algoritmo de ramificación y poda. De ahí que se le muestre el mensaje de no tocar lo que ya hay implementado.

En alguna ocasión puede que el usuario quiera mostrar un nodo. Para ello, y ya que no siempre tendrá por qué ser así, habrá que incluir la librería *iostream* y la directiva *using namespace std*;

```
/*SE ACONSEJA NO TOCAR NINGUNA DE LAS DECLARACIONES E
IMPLEMENTACIONES QUE YA VIENEN POR DEFECTO
PARA ASEGURAR EL CORRECTO FUNCIONAMIENTO DE LA APLICACION.
SE PUEDE AÑADIR TODO LO QUE SE QUIERA EN CUALQUIER CAMPO*/

class Nodo {
//Declaraciones privadas de la clase

private:
double cs;
double ci;
double be;



//Declaraciones publicas de la clase
public:


Nodo();
double getCI(){return ci;}
double getCS(){return cs;}
double getBE(){return be;}
void setCI(double valor){ci=valor;}
void setCS(double valor){cs=valor;}
void setBE(double valor){be=valor;}
~Nodo();
};


Nodo::Nodo() {

}


Nodo::~Nodo() {

}

//Otras funciones auxiliares
```

• *lista.cpp* (clase Lista): La clase lista que se utiliza implícitamente, como ya se ha comentado, tiene la siguiente estructura e incluye a la clase *nodo.cpp* ya que se trata de una lista de nodos.

```
#include "nodo.cpp"


class Celda {
  public:
  Nodo valor;
  Celda *siguiente;
  Celda(Nodo nodo, Celda *sig = NULL)
  {

        valor = nodo;
        siguiente = sig;
   }

   friend class lista;

};

typedef Celda *pcelda;

class Lista {
  public:
   Lista() { anterior = primero = actual = NULL; }
   ~Lista();

   void Insertar(Nodo *nodo);
   void InsertarLNV(Nodo *nodo);
   void Borrar();
   bool ListaVacia() { return (primero== NULL || longitud==0); }
   void Siguiente();
   void Primero();
   bool Actual() { return actual != NULL; }
   Nodo* ValorActual() { return &actual->valor; }
   int Longitud() { return longitud; }


  private:
   pcelda primero;
   pcelda anterior;
   pcelda actual;
   int longitud ;
};

Lista::~Lista()
{
  pcelda aux;
  int cont = longitud;
  while(cont!=0) {
     Primero();
     Borrar();
     cont--;

  }

primero = NULL;
actual = NULL;
anterior = NULL;
}
```

```
void Lista::Insertar(Nodo *nodo)
{
	if(ListaVacia()) {

		primero = new Celda(*nodo, primero);
		if (anterior!=NULL) {
			nodo->mostrar();
			delete anterior;
		}
		anterior = NULL;
		actual = primero;
		longitud = 1;
	}
	else {
		anterior = actual;
		actual->siguiente = new Celda(*nodo,anterior->siguiente);
		actual = actual -> siguiente;
		longitud++;
	}
}

void Lista::Borrar()
{
	pcelda siguiente,aux;

	if (actual->siguiente == NULL) {

		if (actual == primero) {
			delete primero;
			actual = NULL;primero=NULL;

		}
		else{
			delete actual;
			actual = anterior;
			actual->siguiente = NULL;
			aux = primero;
			while (aux!=actual){
				anterior = aux;
				aux = aux->siguiente;
			}
		}
	}
	else{
		if (actual == primero) {
			primero = actual->siguiente;
			anterior = NULL;
			delete actual;
			actual = primero;

		}
		else{
			actual = actual->siguiente;
			delete anterior->siguiente;
			anterior->siguiente = actual;

		}

				}
		longitud--;
}
```

```
void Lista::Siguiente()
{

  anterior = actual;
  actual = actual->siguiente;
}

void Lista::Primero()
{

  anterior = NULL;
  actual = primero;
}
```

Como se puede observar falta por implementar el método *insertarLNV*, éste es el que se implementa dinámicamente según la elección del usuario, utilizando una estrategia LIFO, FIFO, LC-LIFO. Así en este nuevo método se posiciona (en función del criterio) el lugar de la lista donde insertar y se llama al método insertar que ya viene implementado.