Escuela Técnica Superior de Ingeniería **Informática**

etsinf

PROYECTO FINAL DE CARRERA

TÍTULO:	**Optimización de rutas de vuelo para un hidroavión en sus tareas de vigilancia de incendios forestales**
TITULACIÓN:	Ingeniería Informática
AUTOR:	Santiago Jiménez Serrano <sanjiser@inf.upv.es>
DIRECTOR UPV:	Juan Carlos Ruiz García <jcruiz@disca.upv.es>
CODIRECTORA UPC:	Cristina Barrado Muxí <cristina.barrado@upc.edu>
FECHA:	Septiembre de 2011

A mi abuela Rosa,
se te echa mucho de menos.

A mi abuela Dolores,
el verdadero motivo para terminar esta carrera.

A mi madre, Virginia,
porque nunca me falla cuando la necesito.

Resumen

En este trabajo se ha desarrollado un conjunto de bibliotecas de clases (API), junto con las interfaces gráficas de usuario que las utilizan, para generar la ruta de vuelo óptima que ha de seguir un hidroavión en sus tareas de vigilancia y extinción de incendios.

Para ello se tiene en cuenta la información extraída del sitio web del EFFIS (*European Forest Fire Information System*), donde se ofrece en tiempo real los *hotspots* (posibles incendios detectados por el instrumento MODIS a bordo de satélites) recientes y antiguos, y los fuegos confirmados por este mismo organismo (*fires*).

Una vez extraída esta información, donde se incluye la geolocalización de estos puntos (latitud, longitud), a los que llamaremos *waypoints*, se ha implementado algunas variantes que solucionan el problema del TSP (*Travelling Salesman Problem*). A través de algoritmos que hacen uso de las técnicas de 'Ramificación y Poda' y 'Algoritmos Genéticos', creamos una ruta óptima en algunos casos, subóptima en otros.

Finalmente, esta ruta se convierte en un archivo XML que sigue las especificaciones de un plan de vuelo descritas en el '*Flight Plan Specification Language'*, utilizado dentro de la plataforma ISIS para visualizar, simular e incluso hacer volar aparatos reales en modo piloto automático.

Una vez obtenido este archivo, lo podremos visualizar directamente en el mapa a través del Flight Plan Monitor que provee la plataforma ISIS.

La aplicación se ha desarrollado en el IDE Visual Studio 2010 bajo el sistema operativo Windows 7 y el .Net Framework 4.0. El lenguaje de programación utilizado ha sido C#

Palabras Clave

FlightPlan, Planes de vuelo, TSP, Travelling Salesman Problem, Problema del Viajante de Comercio, Branch and Bound, Ramificación y Poda, Algoritmos Genéticos, XML, .net, c#, EFFIS, WFS, Web Feature Service.

TÍTOL:	**Optimització de rutes de vol per a un hidroavió en les seues tasques de vigilància d'incendis forestals**
TITULACIÓ:	Enginyeria Informàtica
AUTOR:	Santiago Jiménez Serrano <sanjiser@inf.upv.es>
DIRECTOR UPV:	Juan Carlos Ruiz García <jcruiz@disca.upv.es>
CODIRECTORA UPC:	Cristina Barrado Muxí <cristina.barrado@upc.edu>
DATA:	Setembre de 2011

Resum

A aquest treball s'ha desenvolupat un conjunt de biblioteques de classes (API), juntament amb les interfícies gràfiques d'usuari que les utilitzen, per generar la ruta de vol òptima que ha de seguir un hidroavió en les seues tasques de vigilància i extinció d'incendis.

Per aconseguir-ho es té en compte la informació extreta de l'assetge web de l'EFFIS (*European Forest Fire Information System*), on s'ofereix en temps real els *hotspots* (possibles incendis detectats per l'instrument MODIS a bord de satèl·lits) recents i antics, i els focs confirmats per aquest mateix organisme (*fires*).

Una vegada extreta aquesta informació, on s'inclou la geolocalització d'aquests punts (latitud, longitud), als quals cridarem *waypoints*, s'ha implementat algunes variants que solucionen el problema del TSP (*Travelling Salesman Problem*). A través d'algorismes que fan ús de les tècniques de 'Ramificació i Poda' i 'Algorismes Genètics', creem una ruta òptima en alguns casos, subòptima en uns altres.

Finalment, aquesta ruta es converteix en un fitxer XML que segueix les especificacions d'un pla de vol descrites al '*Flight Plan Specification Language'*, utilitzat dins de la plataforma ISIS per visualitzar, simular i fins i tot fer volar aparells reals en manera pilot automàtic.

Una vegada obtingut aquest arxiu, ho podrem visualitzar directament al mapa mitjançant el *Flight Plan Monitor*, proveït per la plataforma ISIS.

La aplicació s'ha desenvolupat al IDE Visual Studio 2010 sota el sistema operatiu Windows 7 i el .Net Framework 4.0. El llenguatge de programació utilitzat ha sigut C#

Paraules Clau

FlightPlan, Plans de vol, TSP, Travelling Salesman Problem, Problema del Viatjant de Comerç, Branch and Bound, Ramificació y Poda, Algorismes Genètics, XML, .net, c#, EFFIS, WFS, Web Feature Service.

TABLA DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Introducción

En la mayoría de los servicios forestales y administraciones competentes del arco mediterráneo europeo, se realizan predicciones diarias de riesgo de incendio. Se utilizan métodos más o menos desarrollados y efectivos, basados, cada uno de ellos, en las particularidades del territorio y de la gestión forestal de cada país. Así, se identificarían modelos de riesgo con distintos desarrollos matemáticos donde existen diferentes variables de estudio en cada caso. A pesar de esto, la tendencia en el campo es a unificar e implantar estándares suficientemente elaborados para modelizar el riesgo de incendios en el sur de Europa. Un ejemplo de esto es el método denominado *Forêt-Metéo*, que comprende el análisis del estado de sequedad del combustible, el análisis de la situación meteorológica y el cálculo de distintos índices de riesgo con su consecuente interpretación y divulgación.

Los objetivos de este proyecto han sido los que se detallan a continuación: Extraer esta información en tiempo real de la red, gracias a los servicios prestados por el *EFFIS* (*European Forest Fire Information System*). Una vez hecho esto, tratar estos datos para inferir nueva información que le interese al organismo que utilice el programa en función de sus propias necesidades. Y finalmente calcular un plan de vuelo para un avión de vigilancia forestal que pase por los puntos de riesgo más importantes en dicho día. Este *flightplan* ha de optimizar la distancia recorrida por el avión pasando por el mayor número de *riskpoints* posibles. Es decir, generar la ruta de vuelo con menos kilómetros posible que pase una sola vez por todos esos puntos, exceptuando el aeropuerto, donde pasará dos veces, una durante la salida del avión, y otra durante la llegada.

Además, puesto que se deseaba poder visualizar dicho plan de vuelo en una plataforma ya existente, creada por el Departamento de Arquitectura de Computadores de la *Universitat Politècnica de Catalunya*, llamada ISIS, se genera dicho plan de vuelo cumpliendo las especificaciones del *'Flight Plan Specification Language'*, que no es más que un archivo en formato XML cumpliendo ciertos esquemas de definición de datos (DTD) expuestos más adelante.

Finalmente, la información extraída y utilizada del EFFIS han sido los *hotspots* y los *fires*, que no son más que los puntos detectados en las imágenes del satélite donde se piensa que hay un nuevo incendio (*hotspots*) y los fuegos ya confirmados (*fires*). Puesto que estos puntos están geolocalizados, basta con saber por cuáles queremos pasar, calcular las distancias que hay entre ellos, y aplicar un algoritmo que resuelva el problema del TSP *(Travelling Salesman Problem)* para obtener una ruta adecuada.

Una vez entendidos los requisitos, los grupos de tareas más importantes realizados en este proyecto se podrían dividir en las siguientes:

1 - Extracción de la información de riesgo de incendios en Internet donde se publican estos datos. Como se ha descrito anteriormente, *EFFIS* (http://effis.jrc.ec.europa.eu) y su *WFS (Web Feature Service)*[4] ha sido nuestra principal fuente de información, utilizando los *hotspots* de las últimas 24 horas, 7 días, o todos los detectados hasta el momento y los *fire* (incendios confirmados) de los últimos 7 y 30 días, o pudiendo hacer un recorrido por datos anteriores. Además, se proveen algunas funciones de filtrado de estos puntos en base a diferentes criterios, tales como la pertenencia a un país o provincia en concreto, estar situado en un determinado tipo de terreno o dentro de unas coordenadas.

2 - Una vez obtenida toda la información necesaria de la red, y filtrado los puntos por los que queremos que pase el avión, se han creado diferentes implementaciones que abordan el problema del TSP para generar una ruta óptima. Se ha utilizado Ramificación y Poda con distintas funciones de coste que más adelante se detallan. Además, se ha aprovechado y adaptado un Algoritmo Genético que también aborda este problema. Y finalmente se ha utilizado una mezcla de las dos técnicas para poder obtener en algunos casos una solución óptima en un tiempo algo más razonable. Hay que tener en cuenta que generando esta ruta conseguimos que el avión de reconocimiento recorra todos los puntos que le hayamos indicado ahorrando la mayor cantidad posible de tiempo y combustible.

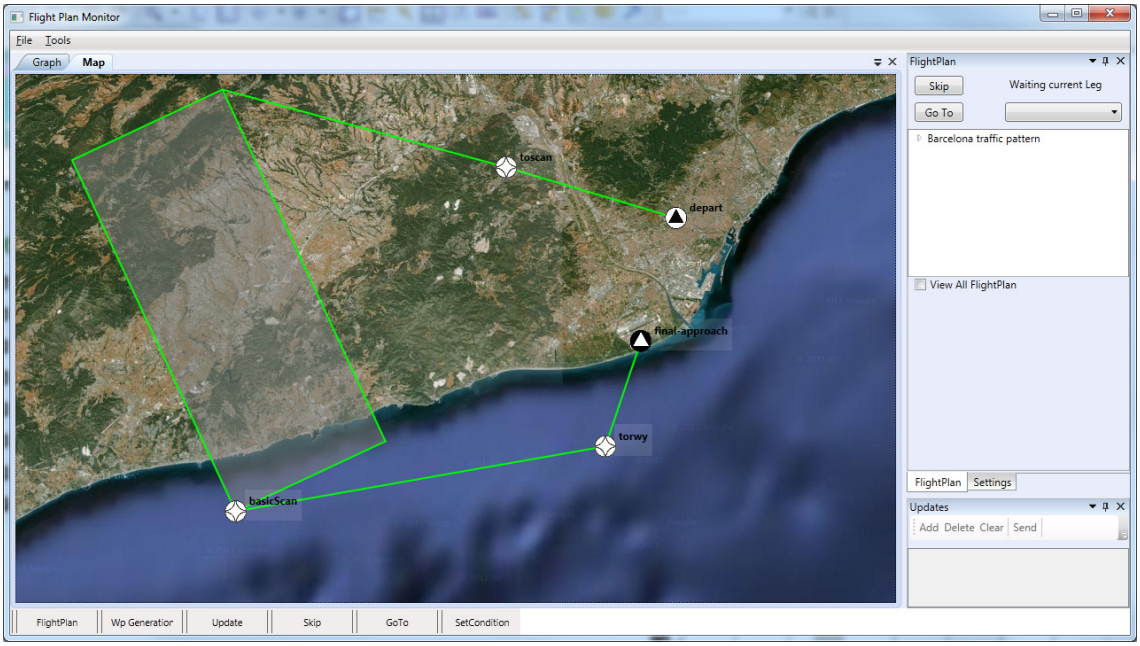

Figura I.1: Ejemplo de visualización de una ruta de vuelo dentro de la plataforma ISIS.

La salida es un fichero XML, cuya estructura, como se ha indicado anteriormente, ya está definida dentro del proyecto *ICARUS* [1][2] en un formato ya existente, llamado *Flight Plan Specification Language*, descrito en la tesis '*Formal Mission Specification and Execution Mechanisms for Unmanned Aircraft Systems*'[3], de Eduard Santamaría Barnadas, formato que es usado en toda la plataforma ISIS por distintos servicios.

3- Creación de una interfaz gráfica de usuario (IGU) que hace uso de todas estas funcionalidades y permite seleccionar el tipo de datos del *EFFIS* con el que queremos trabajar, filtrar los puntos del mapa según diferentes criterios, y a partir de estos puntos utilizar el algoritmo que nos genere la ruta óptima de acuerdo a ciertos parámetros y a la instancia del problema que queramos resolver (no es lo mismo resolver el problema para 15 puntos que para 40, ya que el coste computacional varía mucho de uno a otro tal y como se describe más adelante) con su correspondiente fichero XML para posterior uso en la plataforma ISIS.

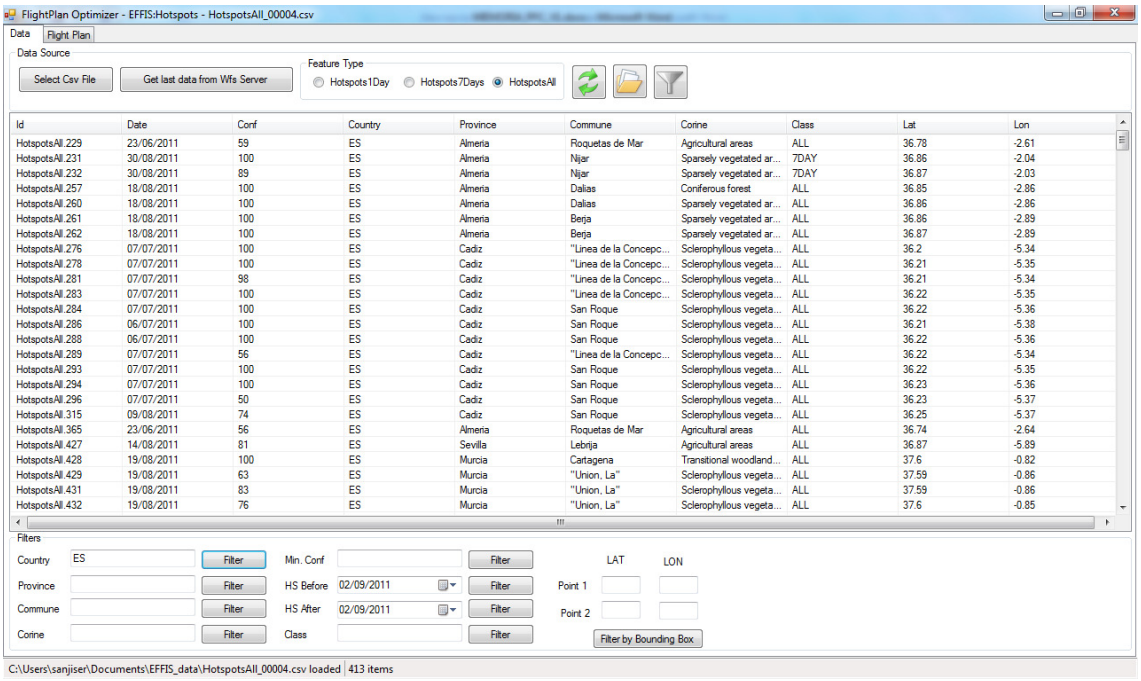

Figura I.2: Captura de pantalla de un formulario EFFIS:Hotspots.

4 – Por último, se ha hecho un estudio de las diferentes prestaciones que ofrecen los algoritmos implementados y utilizados dependiendo del número de nodos que pueda existir en un grafo de estas características, exponiendo sus resultados en distintas gráficas e interpretando en qué situaciones es mejor usar cada tipo y con qué configuración de parámetros.

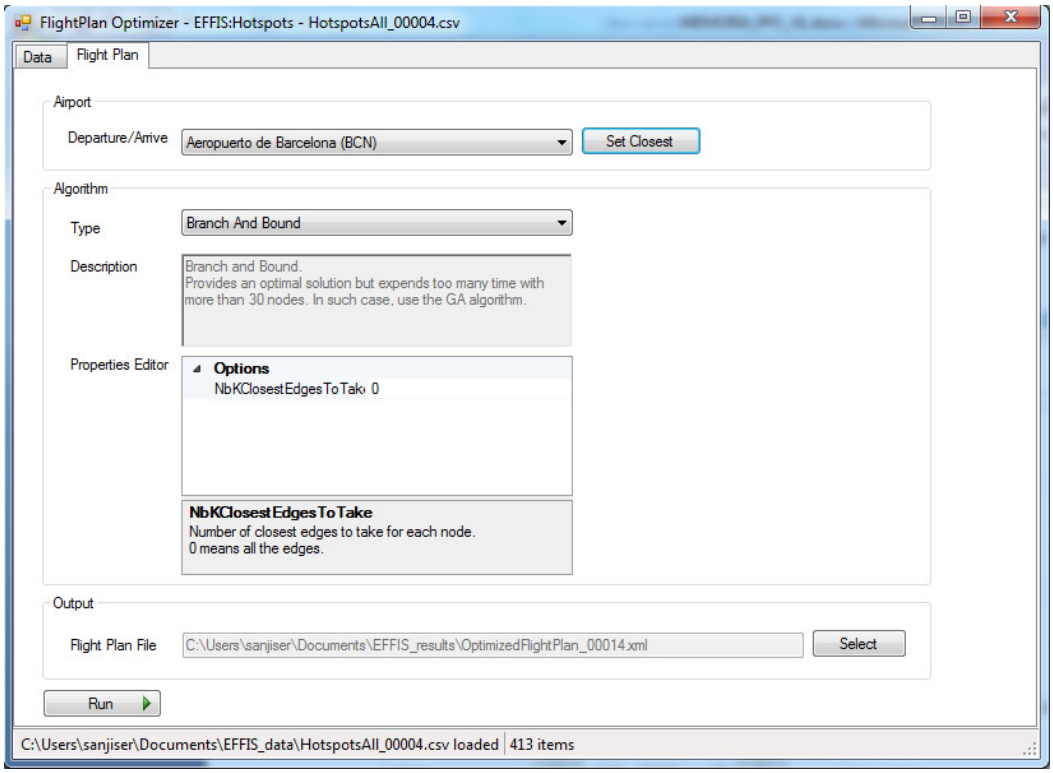

Figura I.3: Captura de pantalla de la GUI implementada – Algoritmo a aplicar a los waypoints filtrados.

Puesto que la plataforma ISIS está implementada en c# en el entorno de desarrollo Visual Studio 2010, el trabajo realizado ha sido bajo el mismo entorno.

Los componentes de gestión del vuelo y de la misión presentados en este proyecto se integran en una arquitectura *hardware/software* más amplia que está siendo desarrollada por el grupo de investigación ICARUS (http://www.icarus.upc.edu).

CAPÍTULO 1 - GLOSARIO

AG	Algoritmo Genético
API	Application Programming Interface
CSV	Comma Separated Values
DTD	Document Type Definition
EFFIS	European Forest Fire Information System
GA	Genetic Algorithm
GPS	Global Positioning System
GUI	Graphic User Interface
FIFO	First In, First Out
FWI	Fire Weather Index
ISIS	Icarus Simulation Integrated Scenario
ISO	International Organization for Standardization
LIFO	Last In, First Out
MAREA	Middleware Architecture for Remote Embedded Applications
MODIS	Moderate Resolution Imaging Spectroradiometer
OGC	Open Geospatial Consortium
RNAV	Area Navigation
RyP	Ramificación y Poda
UAS	Unmanned Aircraft System
URL	Uniform Resource Locator
VAS	Virtual Autopilot System
WFS	Web Feature Service
WMS	Web Map Service
XML	Extensible Markup Language

CAPÍTULO 2 - GENERACIÓN DEL ARCHIVO QUE CONTENIENE UN PLAN DE VUELO

Este capítulo presenta la especificación propuesta por *Eduardo Santamaría Barnadas* en su tesis *"Formal Mission Specification and Execution Mechanisms for Unmanned Aircraft Systems*", para diseñar un plan de vuelo UAS. Esta especificación es la utilizada en la plataforma ISIS, y por lo tanto es la que necesitamos para generar nuestros archivos de salida en los que trazar la ruta a seguir por el avión de vigilancia forestal.[1][2][3]

Un Sistema Aéreo No Tripulado, más conocido como Unmanned Aircraft System (UAS), es un sistema que tiene, como componente central, un avión sin ningún piloto humano a bordo. Ya que se requieren otros componentes para que la nave no tripulada sea capaz de operar en una misión dada, el término UAS se refiere coloquialmente al avión y todos los otros elementos en los que se apoyan sus operaciones.

Habrá que tener en cuenta que, aunque esta especificación está pensada inicialmente para vuelos no tripulados, en nuestro caso no será necesario tomar esto demasiado en cuenta ya que, en principio, nuestros vuelos sí que serán pilotados por personas. Se podría decir que nuestro plan de vuelo generado servirá para trazar una ruta a estos pilotos durante su misión de prevención y/o extinción de incendios. Esta ruta podrá ser visualizada en el *Flight Plan Monitor* que lleven a bordo de la cabina del avión. Incluso, una vez ejecutadas las fases más complicadas de vuelo, como podría ser la de despegue, dicho plan de vuelo podría ser iniciado en modo piloto automático.

Es necesario indicar que esta especificación está basada en el concepto de *leg*, que es la ruta que debe seguir un avión en un tramo determinado. Los tramos están siempre finalizados por *waypoints*, que son, básicamente, una posición geográfica definida en términos de latitud y longitud sobre la que ha de sobrevolar la aeronave.

La sección 2.1 explica la estructura y los diferentes elementos que pueden ser encontrados en un plan de vuelo. En la sección 2.2 se explica la estrategia seguida al crear la API que nos convierte, desde el modelo de objetos, al fichero XML dicho *flightplan*.

2.1 Descripción del 'Flight Plan Specification Language'

La mayoría de los sistemas auto pilotados actuales UAS se basan en listas de *waypoints* como el mecanismo para especificar y ejecutar un plan de vuelo. Estas propuestas tienen algunas limitaciones importantes:

1. Es difícil especificar trayectorias complejas y no soporta construcciones tales como bifurcaciones o iteraciones.

2. No es flexible ya que pequeños cambios pueden implicar tener que lidiar con una considerable cantidad de *waypoints*.

3. No es posible adaptar la misión a las circunstancias.

4. Carece de estructuras para la agrupación y la reutilización de fragmentos del plan de vuelo.

En resumen, los pilotos automáticos actuales se especializan en el control de bajo nivel del vuelo y la navegación se limita a comandos muy básicos para ir a cada *waypoint*. Para mejorar la operatividad de los actuales UAS se necesitan construcciones de más alto nivel, con una mejor semántica, y que permitan adaptarse a posibles circunstancias adversas durante el vuelo. Por esta razón, se propone esta nueva especificación para planes de vuelo.

Algunas de las ideas sobre las que el 'Flight Plan Specification Language' están basadas, provienen de las prácticas actuales en la aviación comercial, descritas en la especificación de procedimientos RNAV (Federal Aviation Administration, 2008). El método de navegación RNAV se aprovecha de la creciente cantidad de ayudas a la navegación, incluida la navegación por satélite, y permite el manejo de aeronaves en cualquier trayectoria de vuelo deseada. Los procedimientos RNAV están compuestos de una serie de partes más pequeñas llamadas *legs*. Para traducir los procedimientos RNAV en un código adecuado para sistemas de navegación, la industria ha desarrollado el concepto "Path and Termination". Códigos "Path Terminator" deben ser usados para definir cada *leg* de un procedimiento RNAV. Los tipos *leg* están identificados por un código de dos letras que describen el tipo de trayectoria (*e.g.*, despegue, en ruta, aterrizaje) y el punto de destino (*e.g.*, la trayectoria finaliza a una altitud, distancia, velocidad, coordenadas). Nuestro mecanismo de especificación hace uso del concepto "Path Terminator" para describir los *legs* básicos. Estamos interesados en un subconjunto de los *legs* RNAV aplicables a la navegación GPS.

Estos elementos son llevados al campo de los UAS y ampliados con construcciones adicionales. Estas nuevas construcciones con *legs* se amplian con nuevos *legs* que

pueden ser de tipo iterativo, de bifurcación, intersección o paramétricas. Conseguimos así una mayor capacidad de adaptación, funcionalidad y expresividad al describir un plan de vuelo. Dicho plan de vuelo se almacena en un documento XML[16][17] que se inyectará en el UAS con el fin de llevar a cabo su ejecución. En las siguientes secciones se describe el contenido del documento XML de especificación del plan de vuelo.

2.1.1 Estructura del nodo *FlightPlan* en el documento XML

El nodo raíz del documento XML que contiene el plan de vuelo para el *UAS* es *FlightPlan*. El listado 2.1 muestra los elementos contenidos en el elemento raíz. Para hacer el contenido más legible algunos elementos han sido reemplazados por puntos suspensivos.

Listado 2.1: Estructura del nodo FlightPlan en el documento XML.

```
<FlightPlan xmlns= 'http://icarus.upc.es/schema/FlightPlan/1.1 '>
      <Locale>         ...</Locale>          <!-- unidades y separadores    -->
      <Fixes>          ...</Fixes>           <!-- waypoints con nombre      -->
      <EmergencyPlans>...</EmergencyPlans> <!-- flightplans de emergencia -->
      <MainFP>         ...</MainFP>          <!-- flightplan principal      -->
</FlightPlan>
```

Locale especifica qué unidades de medida son usadas para la velocidad, altitud y distancias. También especifica cuáles son los separadores decimales y de miles. Los *Fixes* contienen una lista de *waypoints* identificados por nombre, *i.e.* ubicaciones específicas que, por alguna razón, son de especial interés. *EmergencyPlans* contiene un conjunto alternativo de planes en caso de que ocurriese alguna emergencia durante el plan de vuelo principal. Y *MainFP* contiene el plan de vuelo principal, el cual debe ser ejecutado de principio a fin si no sucede ninguna emergencia. Los planes de emergencia y el plan de vuelo principal comparten la misma estructura, presentada en la sección 2.1.4. Recordemos que un *waypoint* es, básicamente, una posición geográfica definida en términos de latitud y longitud que debe sobrevolar la aeronave.

2.1.2 Configuración Regional (Locale Settings)

La configuración regional (*locale settings*) especifica qué unidades son usadas en velocidades, ángulos, altitudes y distancias. Separadores decimales y de listas también se indican. Los posibles valores para cada uno de estos elementos se muestran en la tabla 2.1.

speedUnits		angleUnits		altitudeUnits distanceUnits		decimalSeparator	groupSeparator
ms	m/s	deg	degrees	m	meters	Cualquier string, siendo los más comunes '.' y ','	Como en decimalSeparator además de null
kt	knots	rad	radians	nm	nautical miles		
				ft	feet		

Tabla 2.1: Valores admitidos en la configuración regional de un *fligthplan*.

El listado 2.2 muestra un ejemplo con algunos valores posibles. El ejemplo indica que todos los valores para velocidades incluidos en el plan de vuelo son en metros/segundo, todas las altitudes y distancias estarán indicadas en metros, y el símbolo de separación de decimales será un punto. Un elemento vacío para *groupSeparator* indica que no se usa separador de miles.

Listado 2.2: Ejemplo de configuración regional.

```
<Locale>
       <speedUnits>ms</speedUnits>
       <angleUnits>deg</angleUnits>
       <altitudeUnits>m</altitudeUnits>
       <distanceUnits>m</distanceUnits>
       <decimalSeparator>.</decimalSeparator>
       <groupSeparator />
</Locale>
```

2.1.3 Fixes y Waypoints

Un *fix* describe unas coordenadas específicas en la superficie de la tierra. En la aviación comercial, nos debemos referir a los *fixes* como ayudas a la navegación, intersecciones, aeropuertos, etc. En nuestro caso, se referirán a ubicaciones que, por alguna razón, son de especial interés. Como se muestra en el listado 2.3, un *fix* tiene un identificador, un nombre y una descripción, seguido por sus coordenadas en formato latitud/ longitud.

Listado 2.3: Especificación de la lista de Fixes.

```
<Fixes>
       <Fix id="FIXID">
              <name>Fix de ejemplo</name>
              <description>Algún lugar relevante</description>
              <coordinates>37°38'0.0"N 122°22'0.0 "W</coordinates>
       </Fix>
       <!– Podemos incluir más fixes aquí –->
</Fixes>
```

Los *fixes* están totalmente relacionados a los *waypoints*, los cuales designan una posición geográfica definida en términos de coordenadas latitud/longitud. Existen dos tipos de *waypoints*, los que tienen nombre y los que no. Los primeros corresponden a los *fixes* listados al principio del plan de vuelo, los últimos son posiciones geográficas sin asociación a ninguna ubicación con nombre Por lo tanto, existen dos maneras de especificar *waypoints*. O bien proporcionando sus coordenadas, o indicando el nombre del *fix* al que referencia. Además de su localización, un *waypoint* también contiene un elemento, el cual debe ser *fly-by* o *fly-over*. Para un *waypoint fly-by*, con pasar cerca de él es suficiente. Para un *waypoint fly-over* requiere obligatoriamente pasar por encima de él. Ya que los cambios de velocidad y altitud también aparecen en *waypoints* específicos, opcionalmente un *waypoint* podrá contener estos datos. Si estos valores

estuvieran presentes, indicarían la velocidad y altitud requeridas para el avión en ese *waypoint*. Los *fixes* y *waypoints* se utilizan para especificar el destino de construcciones de tipo *leg* (etapa/fase) de más alto nivel. El listado 2.4 muestra un ejemplo de un *waypoint* que referencia a un *fix*. El elemento *dest* es parte de la especificación de un *leg*, y está descrito en la sección 2.1.6.

Listado 2.4: Descripción XML de un waypoint.

```
<dest>
       <fix>FIXID</fix>
       <fly-over>true</fly-over>
       <altitude>300</altitude>
       <speed>65</speed>
</dest>
```

La tabla 2.2 describe los tipos de datos opcionales y obligatorios que han de mostrar cada instancia de *waypoint*.

2.1.4 Plan de Vuelo Principal *(Main Flight Plan)*

Un plan de vuelo especifica el camino seguido por una aeronave. Como se puede ver en la figura 2.1, cada plan de vuelo está compuesto por una secuencia de etapas (*stages*), tales como despegue, procedimiento de salida y otras. Estas etapas deben seguir un orden concreto. Cada etapa del plan de vuelo está compuesta de una colección estructurada de *legs*. El concepto de *leg* está prestado de RNAV, y es usado para especificar la trayectoria seguida por una aeronave para alcanzar un *waypoint* desde su predecesor. En el caso más simple esta trayectoria estará formada por una línea recta.

Elemento	Tipo de datos	Opcionalidad
`coordinates`	lat lon en dos posibles formatos: dd◦mm'ss.ss"N|S dd◦mm'ss.ss"E|W o dos valores numéricos reales	Obligatorio
`fix`	fix id	Obligatorio
`fly-over`	bool	Opcional (el valor por defecto es `false`)
`altitude`	double	Opcional
`speed`	double	Opcional

Tabla 2.2: Tipos de datos para los elementos de un *waypoint*.

Todos los vuelos requieren un único plan de vuelo, pero otros planes de emergencia adicionales pueden ser incluidos. Los planes de vuelo de emergencia son planes parciales, *i.e.,* carecen de algunas etapas iniciales. Su propósito es ofrecer caminos alternativos cuando ocurre una situación de emergencia. Aparte del número de etapas

incluidas, el plan de vuelo principal y los de emergencia, tienen una estructura idéntica.

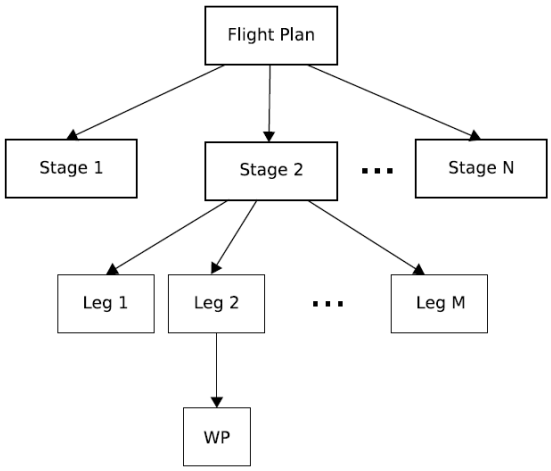

Figura 2.1: Un *flightplan* se compone de *stages*, *legs* y *waypoints*

Todos los planes de vuelo tiene un identificador, un nombre y una descripción (ver listado 2.5). Opcionalmente, para el plan de vuelo principal, se pueden especificar los planes de emergencia.

Listado 2.5: Descripción XML del plan de vuelo principal.

```
<MainFP id="FPID">
       <name>Nombre del flightplan</name>
       <description>Texto describiendo el flightplan</description>
       <!— Lista de stages que forman el plan de vuelo y su ruta —>
       <stages> . . . </stages>
       <emergency>EmergencyFP1 EmergencyFP2 ... </emergency>
</MainFP>
```

2.1.5 Etapas (*Stages*)

Las Etapas, o *Stages*, organizan los *legs* de un plan de vuelo en etapas del vuelo con funcionalidades distintas. Estas *stages* constituyen elementos básicos para la especificación del plan de vuelo, se podría decir que forman su esqueleto. Cada *stage* corresponde conceptualmente a una fase del vuelo, por ejemplo, el despegue o la maniobra de aproximación (ver tabla 2.3). Un conjunto de *stages* junto a sus respectivas listas de *legs* deben cumplir un fin común, que en este caso es indicar las fases de vuelo y los puntos por donde ha de pasar la aeronave. Las *Stages* deben cumplir con las siguientes reglas:

• Cada *stage*, excepto la primera y la última, tiene un único predecesor y un único sucesor.

• Las *Stages* siempre se vuelan en orden secuencial.

• Una *stage* puede tener más de un *leg* de salida. Por ejemplo, un procedimiento de despegue puede terminar en diferentes puntos dependiendo de la dirección de despegue seleccionada.

• Una *stage* puede tener más de un *leg* de entrada. Por ejemplo, un procedimiento de salida, el cual sigue a un despegue, puede comenzar en diferentes posiciones.

• Siempre existirá una correspondencia uno-a-uno entre los *legs* finales de una *stage* determinada y el *leg* inicial de la siguiente. Así se consigue proporcionar una correcta transición entre *stages*. Existen construcciones que permiten al diseñador del plan de vuelo proveer esta correspondencia si fuera necesario.

• Los planes de vuelo de emergencia son una excepción a la regla anterior. La primera *stage* de un plan de emergencia puede tener más de un *leg* inicial. El *leg* seleccionado para entrar en dicho plan de emergencia será aquel cuyo destino sea más cercano a la posición actual de la aeronave.

• La correspondencia entre el *leg* de salida de una *stage* y el de entrada de la siguiente será determinada por su posición en los respectivos listas *finalLegs* e *initialLegs*.

• Todos los legs alcanzables deben tener un *leg* siguiente en la misma *stage* donde aparezcan, o deben aparecer en la lista *finalLegs*. En otras palabras, se debe asegurar que la aeronave no llegue a un punto muerto, sin siguiente punto al que ir y sin poder aterrizar.

Listado 2.6: Descripción XML de una *stage* en un *flightplan*.

```
<stage id="STID" type="Departure" manualOnly="false">
   <name>Nombre de la stage</name>
   <description>Texto describiendo la stage</description>
   <legs> ... </legs>                    <!-- Legs que pertenecen a esta stage -->
   <initialLegs>LegStartId</initialLegs><!-- Lista separada por espacios con leg ids -->
   <finalLegs>LegEndId</finalLegs>       <!-- Lista separada por espacios con leg ids -->
   <emergency> ... </emergency>          <!-- Planes de vuelo de emergencia -->
</stage>
```

El listado 2.6 muestra qué elementos pueden ser encontrados dentro de una *stage*. Hay que destacar que esta especificación está especializada sobre todo en navegación in-flight, es decir, cuando el avión ya ha tomado cierta altura. Etapas como Taxi, Takeoff y Land (ver Tabla 2.3) pueden ser vistas como marcadores de fases del vuelo, que en un futuro contendrán la información que sea requerida por el VAS (*Virtual Autopilot System*) para llevarlas a cabo en un modo automático, una vez que se disponga de estas capacidades.

Cada *stage* tiene un identificador, un nombre, y opcionalmente una descripción. Su propósito se especifica usando el atributo `type`. El atributo `manualOnly` tendrá valor `true` si la ejecución automática de esta *stage* no es posible, por ejemplo, en la *stage taxi* (desplazamiento hasta la pista). Cuando una *stage* es marcada de esta manera, es responsabilidad de un piloto, a bordo o en tierra, el controlar la aeronave. Los valores válidos para los distintos tipos de atributos se pueden ver en la tabla 2.3.

El elemento `legs` lista todos los *legs* que son parte de esta *stage*. Además, los elementos `initialLegs` y `finalLegs`, los cuales son listas separadas por espacios, indican cuál es el conjunto de *legs* iniciales y finales respectivamente para esta *stage*.

Las *stages* tienen un elemento opcional indicando qué planes de emergencia serán llevados a cabo en caso que ocurriese alguna emergencia. Éstos llevarán el aparato al área más cercana donde sea posible un aterrizaje.

Taxi	Ir a la pista de despegue, o salir de la de aterrizaje.
TakeOff	Los *legs* en esta etapa serán usados durante el procedimiento de despegue.
Departure	Estos *legs* se deben sobrevolar después de un despegue a fin de alcanzar el primer punto de la siguiente etapa.
EnRoute	Navega desde un punto inicial a un destino. Puede aparecer más de una vez desde la etapa Departure hasta *Mission* (si *Mission* existiese).
Mission	Serie de *legs* que se deben sobrevolar durante las operaciones de la misión.
Arrival	*Legs* a sobrevolar después de dejar la ruta y antes de iniciar un procedimiento de *Approach*.
Approach	Preparación para el aterrizaje.
Land	Operación de aterrizaje

Tabla 2.3: Tipos de etapas (*stages*).

2.1.6 Legs

Un *leg* especifica la trayectoria de vuelo para alcanzar un *waypoint* específico. En general, los *legs* contienen un *waypoint* de destino y una referencia al siguiente *leg*. El elemento `dest` especifica cual es el *waypoint* de destino. El siguiente *leg*, al igual que el previo, se indican respectivamente por los elementos `next` y `prev`. Sólo los *legs* de intersección, que marcan puntos donde tomar una decisión, son capaces de especificar más de un *leg* anterior o posterior. Es necesario indicar que un *leg*, además de contener un *waypoint* al que llegar, y el tipo de ruta a seguir, también puede contener la altura y la velocidad a la que ha de hacerlo.

Existen cuatro tipos básicos de *legs*:

• ***Legs* básicos:** Son, como su nombre indica, los tipos de *legs* básicos del plan de vuelo. Especifican primitivas como 'Direct to a Fix', 'Track to a Fix'.

• ***Legs* iterativos:** Permiten especificar secuencias repetitivas.

• ***Legs* de intersección:** Indica un punto de unión o cruce para *legs* que finalizan en el mismo *waypoint*. También pueden indicar una bifurcación en un punto, donde se ha de tomar una decisión sobre qué *leg* será el siguiente al que volar.

• ***Legs* paramétricos**: Son *legs* cuya trayectoria puede ser calculada dados los parámetros de un determinado algoritmo. Por ejemplo, un patrón de escaneado (*scan pattern*), donde el avión realiza una serie *zig zags* sobre un área determinada.

Los *legs* de intersección difieren del resto en que pueden alcanzar a más de un sucesor. Todos los *legs* pueden incluir un parámetro opcional indicando cuáles son los planes de emergencia disponibles cuando sucede una contingencia durante la ejecución de dicho *leg*.

A continuación, se detallarán los tipos de *legs* que han sido utilizados durante la realización de este proyecto, que en todos los casos han sido de tipo básico.

2.1.6.1 *Legs* básicos

Esta sección describe los *legs* básicos disponibles para diseñar un plan de vuelo. Nos referimos a ellos como básicos para diferenciarlos de los que contienen estructuras de control como los *legs* iterativos, de intersección o paramétricos. Una visión esquemática de los diferentes tipos de *leg* básicos existentes se muestra en la figura 2.2.

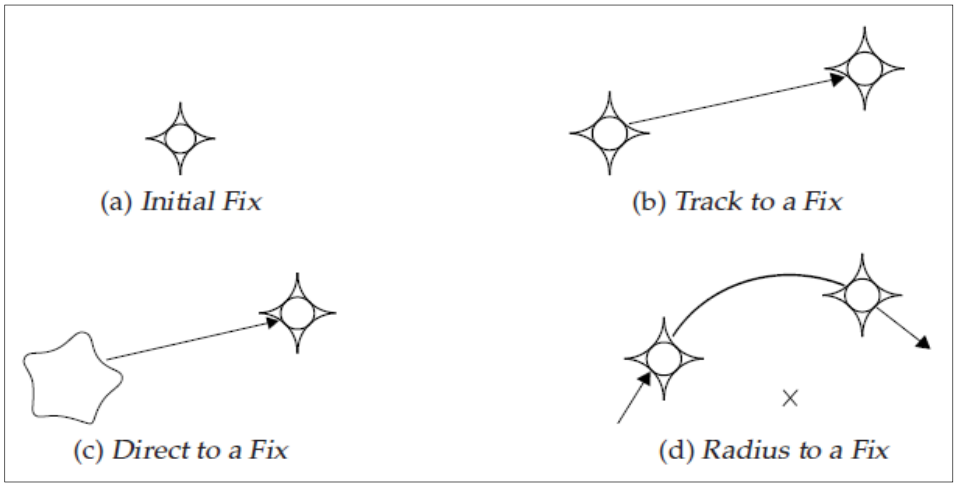

Figura 2.2: Tipos de *leg* básicos disponibles

Initial Fix (IFLeg)

Un 'Initial Fix' indica un punto inicial. Es usado junto con otro tipo de *leg*, por ejemplo TFLeg, para definir la trayectoria deseada.

Track to a Fix (TFLeg)

Un *leg* de tipo 'Track to a Fix' corresponde a una trayectoria en línea recta desde un waypoint a otro. El *waypoint* inicial es el destino del anterior *leg*. El listado 2.7 muestra el aspecto de este tipo de *leg* en un plan de vuelo. El atributo `xsi:type` identifica el tipo de *leg*, mientras que `dest` es el *waypoint* de destino que debe ser alcanzado a la velocidad indicada.

Listado 2.7: Descripción XML de un *leg* 'Track to a Fix'.

```
<leg id="L1" xsi:type ="TFLeg">
       <dest>
              <coordinates>41°17'38.38"N 2°4'35.82"E</coordinates>
              <fly-over>true</fly-over> <!-- Pasar por encima -->
              <speed>60</speed> <!-- Velocidad deseada después del wp -->
       </dest>
       <next>L2</next> <!-- Identificador del siguiente leg -->
</leg>
```

Direct to a Fix (DFLeg)

Un *leg* de tipo '*Direct to a Fix*' es la trayectoria en línea recta seguida por una aeronave desde un área inicial hasta el siguiente *waypoint*. Esto significa que ha de volar al *waypoint* de destino en línea recta, cualquiera que sea su posición actual.

Radius to a Fix (RFLeg)

Un *leg* de tipo '*Radius to a Fix*' se define como un camino constante circular alrededor de un centro de giro que finaliza en un *waypoint*. Se caracteriza por dicho centro de giro y su dirección (*Left* o *Right*).

Listado 2.8: Leg 'Radius to a Fix'.

```
<leg id="L2" xsi:type="RFLeg">
       <dest> ... </dest>
       <next>L3</next>
       <center>41°17'38.38"N 2°5'27.49"E</center>
       <direction>Right</direction>
</leg>
```

2.2 Generación de la API para producir el fichero XML de un FlightPlan

En la sección anterior se ha mostrado la necesidad de seguir un estándar para representar un plan de vuelo. Además, se han mostrado los detalles de la especificación '*Flight Plan Specification Language*', que es la que utilizaremos en nuestro proyecto para generar la ruta de vuelo que ha de seguir el avión. Es por tanto necesaria la creación de la capa de software que genere automáticamente un archivo XML siguiendo dicho estándar, a partir de un modelo de objetos.

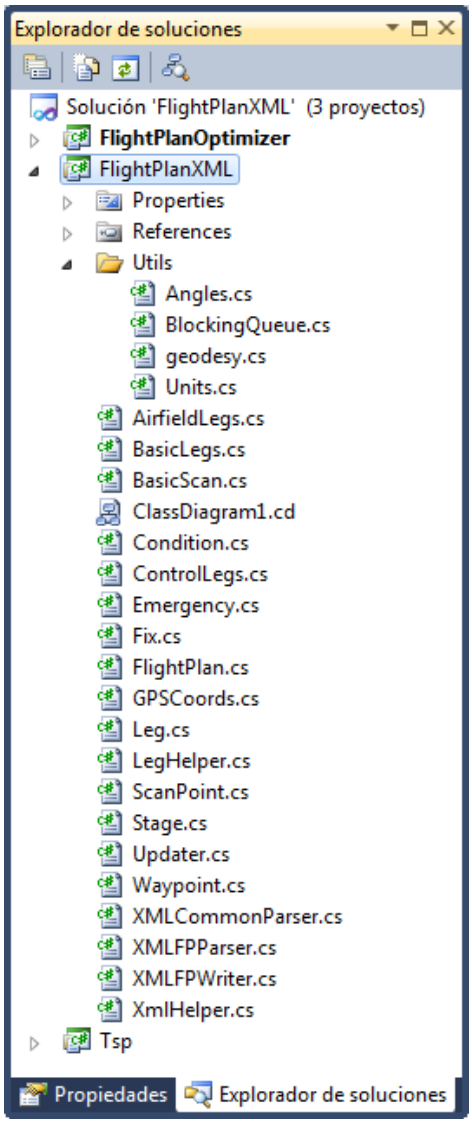

Figura 2.3: Vista en el explorador de soluciones de la librería *FlightPlanXML*

En la plataforma ISIS ya existía una librería software donde este modelo de objetos estaba implementado, llamada *FlightPlanXML* (ver figura 2.3). Sin embargo, la función inicial que cumplía dicha librería era la de leer un archivo XML que contuviera un plan de vuelo ya creado, generalmente por un editor de textos u otras aplicaciones fuera del ámbito de ISIS, y nos generase el modelo de objetos. Es decir, esta librería nos hacía la transformación desde XML al modelo de objetos.

En nuestro caso era necesaria justo la operación inversa, es decir, a partir de la instancia de un *FlightPlan* en el modelo de objetos, generar su archivo XML correspondiente. De esta manera, se aumenta la funcionalidad de la API existente, a la par que aprovechamos dicho modelo de objetos.

Durante esta tarea, en cada una de las clases que forman parte de un plan de vuelo se han añadido distintas operaciones para generar su representación en XML siguiendo los estándares del lenguaje c# y su librería de clases *System.Xml*. A dichos métodos se les ha llamado *GetXmlElement* o *AppendXml*, dependiendo de si una instancia formaba parte o no de una lista de elementos o representaba un elemento único por sí sola.

Además de añadir funcionalidad a las clases ya existentes, se han implementado otras dos nuevas cuyo objetivo es crear el archivo XML a partir de una instancia de tipo *FlightPlan*. Éstas son las clases XMLFPWriter y XMLHelper. En la figura 2.4 se puede ver la representación en el diagrama de clases de aquellas que han sido más importantes durante nuestro trabajo.

Hay que hacer mención especial al método *Distance* en la clase *Waypoint*, el cual nos proporcionará la distancia entre dos *waypoints*. Esta distancia es la base fundamental para más adelante poder calcular las rutas óptimas.

Aunque las tareas aquí desempeñadas pueden parecer sencillas, sirvieron como primera toma de contacto con los planes de vuelo durante la creación del proyecto. De este modo se pudo entender en más profundidad cómo se iban a poder desarrollar las siguientes etapas del mismo.

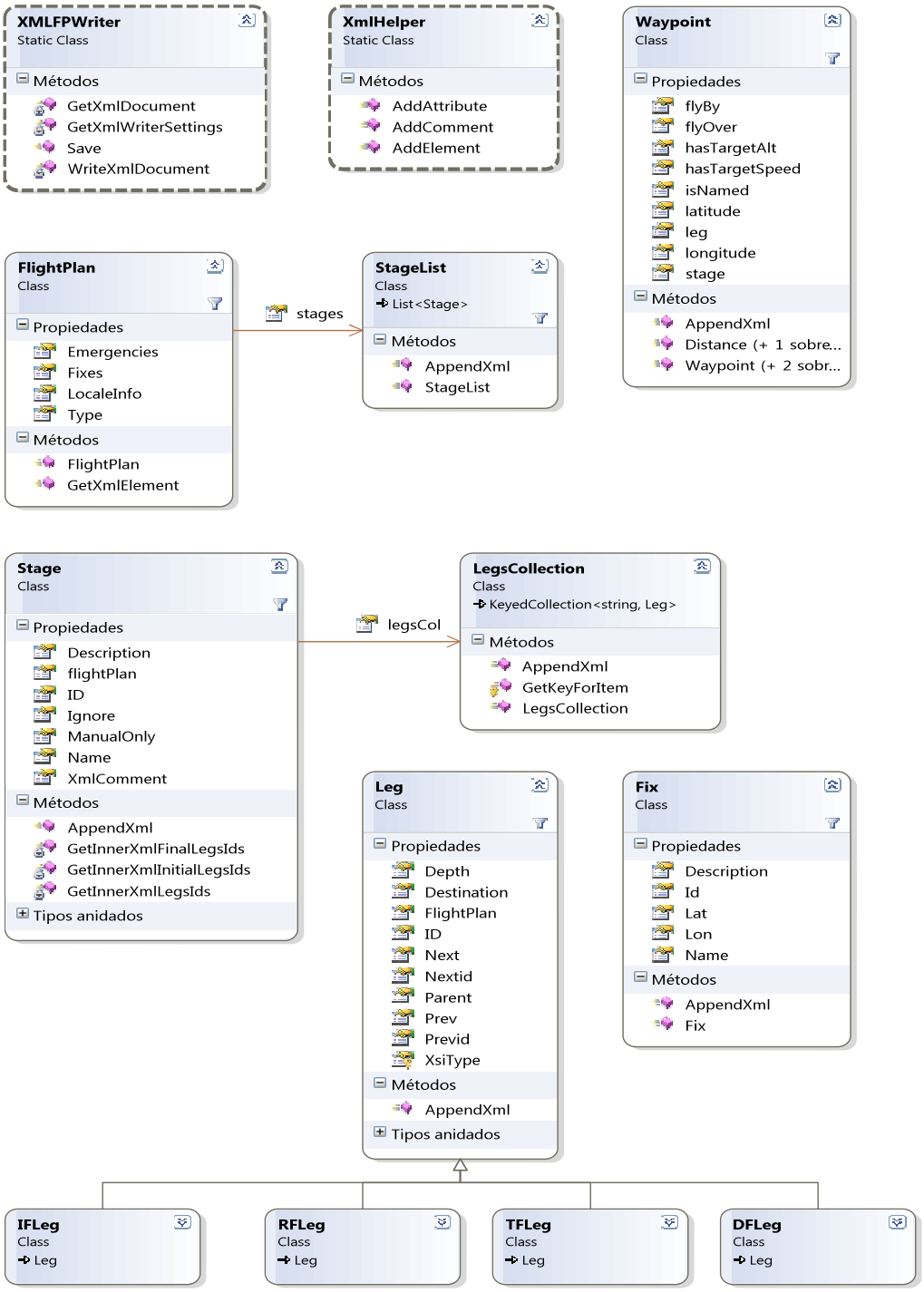

Figura 2.4: Diagrama de clases de la librería FlightPlanXML.

CAPÍTULO 3 - OBTENCIÓN DE COORDENADAS DE INCENDIOS DESDE EL WFS DEL EFFIS

En este capítulo se pasa a describir, en la sección 3.1, qué es un Web Feature Service (WFS), y qué papel juega en la información que queremos obtener referente a incendios para generar nuestro plan de vuelo. Dicha información será extraída del WFS que ofrece el EFFIS (*European Forest Fire Information System*) y que responde a las *features* o características *Hotspot* y *Fire*, explicadas en la sección 3.2. Además, en la sección 3.3 se explica la API generada para descargar esta información de internet, y su posterior modelización y tratamiento mediante el diagrama de clases.

3.1 Descripción de un servidor WFS (Web Feature Service).

Un *Web Features Service* (WFS) representa una opción en la manera en que la información geográfica es creada, modificada e intercambiada en Internet. Más bien que compartir dicha información geográfica a nivel de archivo usando 'File Transfer Protocol' (FTP), por ejemplo, el WFS ofrece acceso directo a información geográfica detallada en el nivel de características (*feature* en inglés) y propiedades de estas características. WFS permite a los clientes obtener o modificar los datos que están buscando, antes que obtener un fichero que contiene estos datos y posiblemente, muchos más a los que finalmente no les dará uso. Estos datos pueden ser utilizados para una amplia variedad de propósitos, incluyendo fines distintos a los que sus productores tenían previsto. [4]

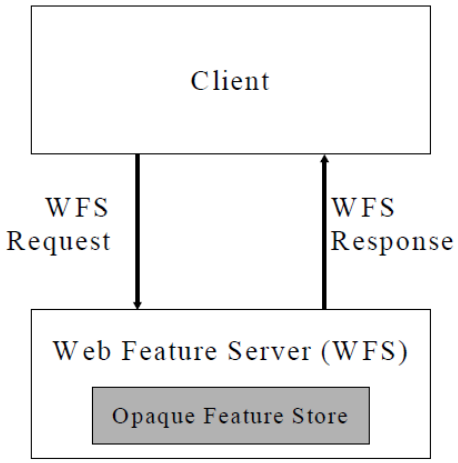

Figura 3.1: Interfaz de un WFS.

En la taxonomía de servicios definida en la ISO 19119, el WFS es básicamente un servicio de acceso a características, pero también incluye elementos de uno de conversión/transformación de coordenadas y formatos geográficos.

3.1.1 Descripción del estándar WFS y su utilización en el EFFIS

Este estándar internacional especifica el comportamiento de un servicio que provee transacciones y acceso a características de tipo geográfico, de una manera independiente a la que utilice la capa de base de datos subyacente. Especifica operaciones de los siguientes tipos: *discovery*, *query*, *locking* y *transaction*.

Las operaciones de tipo '*Discovery*' permiten interrogar al servicio para determinar sus capacidades y obtener el esquema o estructura que la aplicación define sobre los tipos de características que ofrece. Es decir, nos dice qué funciones ofrece el servicio, y la estructura de datos que utiliza en sus '*features*'.

Las operaciones de tipo '*Query*' permiten obtener de la base de datos subyacente las características o valores de estas características, con ciertas restricciones sobre los datos. Es decir, permiten al cliente efectuar una consulta sobre estos datos.

Las operaciones de tipo '*Locking*' permiten acceso exclusivo a características con el propósito de modificarlas o eliminarlas.

Las operaciones de tipo '*Transaction*' permiten crear nuevas características, modificarlas, reemplazarlas y borrarlas de la base de datos subyacente.

Hay que destacar que este estándar internacional no se ocupa de los problemas del control de acceso a la información.

De entre las operaciones definidas en el estándar nos interesan las tres siguientes:

GetCapabilities (operación de tipo '*Discovery*').

Un WFS debe ser capaz de describir sus capacidades o operaciones que es capaz de realizar. Específicamente, debe indicar qué tipos de características (*features*) puede servir, y qué operaciones están soportadas para cada tipo de característica.

DescribeFeatureType (operación de tipo '*Discovery*').

Un WFS debe ser capaz de describir la estructura de cualquier tipo de características que pueda servir. Es decir, ha de poder mostrar el modelo de datos que está utilizando para cada tipo de datos que ofrece. Esta operación

nos ha sido muy útil a la hora de crear las clases *Hotspot* y *Fire* en nuestro código, ya que son las *features* que obtenemos de nuestro servidor WFS alojado en el EFFIS. Éstas se detallan en la sección 3.3.

GetFeature (operación de tipo '*Query*').

Un WFS debe ser capaz de ofrecer, bajo demanda, las instancias de las características (*features*) que un cliente le pida. Además, el cliente ha de ser capaz de especificar qué *feature* de todas las existentes, quiere obtener, pudiendo incluir restricciones propias de una consulta tanto con parámetros espaciales como no espaciales. En la práctica, esta es la operación que utilizamos para obtener las instancias de los datos referentes a *Hotspots* y *Fires* en el WFS del EFFIS.

En nuestro caso concreto, si echamos un vistazo a los resultados que nos dan las siguientes URLs, encontraremos las capacidades (operaciones disponibles) que tiene el WFS del EFFIS, y las estructuras de los datos que maneja para *Hotspots* y *Fires*.

http://geohub.jrc.ec.europa.eu/effis/wfs?request=GetCapabilities

http://geohub.jrc.ec.europa.eu/effis/wfs?request=DescribeFeatureType

Los ficheros XML que obtenemos de estos enlaces también los podemos consultar en la carpeta *WFS_Schemas* de los anexos que contiene este proyecto.

Resumiendo el contenido de la petición `GetCapabilities` encontramos que las operaciones que soporta nuestro WFS son las siguientes:

- `GetCapabilities`
- `DescribeFeatureType`
- `GetFeature`
- `GetGmlObject`
- `LockFeature`
- `GetFeatureWithLock`
- `Transaction`

De las operaciones anteriores, la más importante para nosotros es `GetFeature` ya que es la que nos ofrecerá las instancias de tipo *Hotspot* o *Fire* con la información que nos interesa. Además, los tipos de características que podemos pedir a través de una llamada a `GetFeature` son las siguientes:

- `EFFIS:FireNews`
- `EFFIS:Fires30Days`
- `EFFIS:Fires7Days`
- `EFFIS:FiresAll`
- `EFFIS:Hotspots1Day`
- `EFFIS:Hotspots7Days`

- `EFFIS:HotspotsAll`

Las *features* de tipo `FireNews` no nos interesan, ya que hacen referencia a noticias de incendios aparecidas en medios de comunicación. Puesto que estos incendios ya aparecerán en cualquier otra *feature* de tipo *Fire*, es innecesario su tratamiento.

En el fichero XML obtenido a través de la llamada a `DescribeFeatureTypes` nos detalla la estructura que tendrán todas estas *features*. La tabla 3.1 nos muestra qué información podemos extraer de cada instancia que obtengamos posteriormente con una llamada a `GetFeature`, para cada tipo de *feature*.

FeatureType	Campos del FeatureType
Hotspots1Day	FID, the_geom, HS_Date, HS_Time, Conf, Country, Province, Commune, Corine, Lat,Lon
Hotspots7Days	FID, the_geom, HS_Date, HS_Time, Conf, Country, Province, Commune, Corine, CLASS, Lat, Lon
HotspotsAll	FID, the_geom, HS_Date, HS_Time, Conf, Country, Province, Commune, Corine, CLASS, Lat, Lon
Fires7Days	FID, the_geom, FireDate, Country, Province, Commune, Area_HA, CountryFul, X, Y, LastUpdate
Fires30Days	FID, the_geom, FireDate, Country, Province, Commune, Area_HA, CountryFul, Class, X, Y, LastUpdate
FiresAll	FID, the_geom, FireDate, Country, Province, Commune, Area_HA, CountryFul, Class, X, Y, LastUpdate

Tabla 3.1: Listado de los FeatureTypes ofrecidos por el WFS del EFFIS y sus respectivos campos.

Es necesario indicar que todas estas llamadas a los distintos métodos de un WFS se hacen mediante peticiones GET o POST del protocolo HTTP. En nuestro caso siempre serán de tipo GET.

En el capítulo 5 se detalla el significado de los campos mostrados en la tabla anterior, donde son de gran utilidad a la hora de filtrar puntos en nuestro plan de vuelo a través de la API o la GUI. De momento, lo único que nos interesará para crear este plan de vuelo serán las coordenadas en latitud/longitud para cada *Hotspot* o *Fire*.

Hemos mostrado cuál es el funcionamiento de un servidor WFS y cómo hemos obtenido la información respecto a los datos que vamos a utilizar. La siguiente sección nos da una visión clara de qué significan esos datos.

3.2 Definición y diferencias entre hotspot y fire

Como hemos podido ver en la sección anterior, un WFS (*Web Feature Service*) publica funciones con datos de tipo geoespacial en la red. Esto significa que, en vez de devolver una imagen tal y como tradicionalmente lo ha hecho un *MapServer*, el cliente obtiene información detallada sobre características específicas de los datos con los que se trabaja, junto con un conjunto de niveles basados en geometrías y atributos. Tal como sucede con otras especificaciones OGC (Open Geospatial Consortium)[7], esta interfaz usa XML sobre HTTP como mecanismo de distribución, más concretamente,

GML (Geography Markup Language), el cual es un subconjunto del XML. Además, también es capaz de ofrecer estos datos en formato CSV, formato que finalmente hemos utilizado para leer la información desde nuestra aplicación.

En nuestro caso, las características (*features*), que necesitamos extraer de nuestro servidor WFS son las siguientes:

Nombre de la Característica (Feature)	Descripción
EFFIS:Fires30Days	Fuegos en los últimos 30 días.
EFFIS:Fires7Days	Fuegos en los últimos 7 días.
EFFIS:FiresAll	Fuegos desde el 01-01-2010
EFFIS:Hotspots1Day	Hotspots del MODIS en las últimas 24 horas.
EFFIS:Hotspots7Days	Hotspots del MODIS en los últimos 7 días.
EFFIS:HotspotsAll	Hotspots del MODIS desde el 01-01-2010

Tabla 3.2: Tipos de características (*features*) disponibles en el WFS del EFFIS.

En primer lugar, hay que saber que, tanto un 'hotspot' como un 'fire' son dos características o '*features*' que podemos consultar al WFS del EFFIS. A continuación se detalla qué significado tiene cada una y cuáles son las diferencias que presentan entre ellas.

MODIS (Moderate Resolution Imaging Spectroradiometer) es un instrumento clave a bordo de los satélites Terra (EOS AM) y Aqua (EOS PM). *Terra* MODIS y *Aqua* MODIS observan la totalidad de la superficie de la Tierra cada 1-2 días, adquiriendo datos en 36 bandas espectrales, o grupos de longitudes de onda. Estos datos mejoran nuestra comprensión de las dinámicas globales y los procesos que suceden en tierra, océanos y en la parte baja de la atmósfera. MODIS juega un papel vital en el desarrollo de modelos válidos e interactivos sobre la Tierra, capaz de predecir cambios globales con suficiente precisión y antelación para asistir a la toma de decisiones concernientes a la protección de nuestro entorno. [5]

Los incendios activos son localizados en base a los llamados '***hotspots***' (puntos calientes en su traducción al castellano) producto de los sensores del MODIS, los cuales identifican áreas en el suelo claramente más calientes que sus alrededores. La diferencia de temperatura entre las áreas que están ardiendo respecto sus áreas vecinas permite la identificación y mapeo de fuegos activos. Este modulo estuvo disponible por primera vez en 2007, como manera de dar soporte al 'Monitoring and Information Center (MIC)' de la Comisión Europea en sus actividades de coordinación en la lucha contra incendios forestales entre los Estados Miembros. [6]

La creación de mapas de incendios activos se realiza para proveer una visión general de fuegos actuales en Europa y además ayuda a la creación de mapas de perímetros quemados. La información sobre fuegos activos se actualiza a diario y, cuando es

necesario, está disponible en *EFFIS* dentro de las 2-3 horas siguientes a la adquisición de las imágenes del *MODIS*.

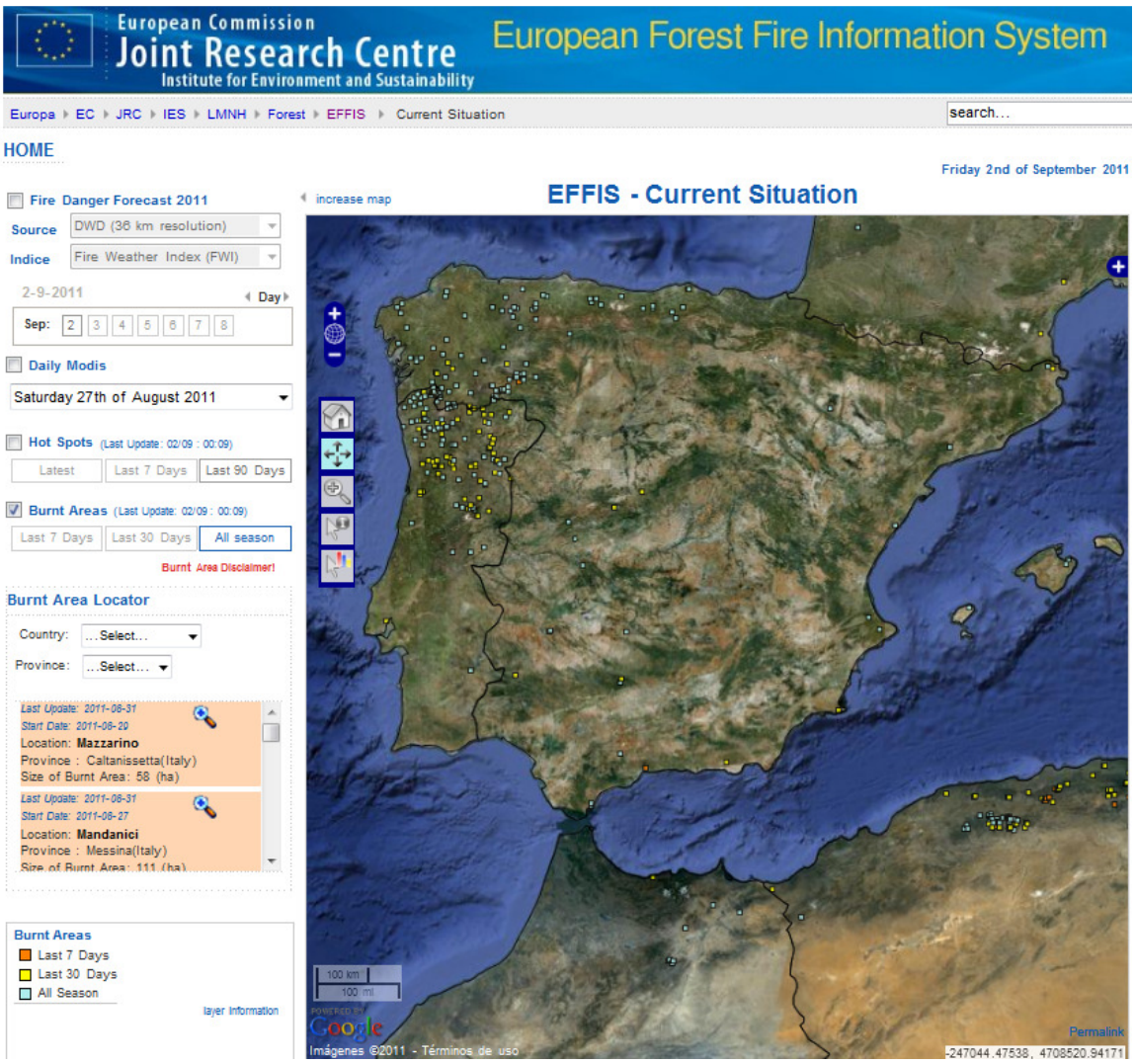

Figura 3.2: Fuegos declarados (‘*fires*’) durante toda la temporada 2011, indicados en la web del EFFIS.

Cuando se interpretan los hotspots mostrados en la aplicación ‘EFFIS Current Situation’, se han de tener en cuenta las siguientes consideraciones:

- La precisión de la situación de los Hotspots en el mapa es de 1.5 kms.
- Algunos fuegos podrían ser pequeños o verse ocultados por humo o nubes, permaneciendo sin detectar.
- Los satélites también detectan otras fuentes de calor (no todos los *hotspots* son fuegos).

Para minimizar falsas alarmas y filtrar fuegos activos no calificados como ‘incendios forestales’ (e.g. quemas agrícolas), el sistema solo muestra un subconjunto seleccionado de *hotspots* detectados por el MODIS Rapid Response System. Con este

fin se aplica un algoritmo que tiene en cuenta la extensión y categoría del terreno cercano al posible incendio, la distancia a áreas urbanas y superficies artificiales, etc, datos de los que se extrae el nivel de 'confianza' (campo *confidence*) del *hotspot*.

Con la herramienta de identificación de incendios, se adjunta información relevante a cada fuego activo, como puede ser las coordenadas geográficas, distrito administrativo (comunidad y provincia) y categoría del tipo de terreno afectado (agrícola, forestal, urbano…).

La figura 3.2 muestra los incendios ya confirmados durante toda la temporada en la Península Ibérica. Se puede apreciar que la mayoría de ellos se concentran en el norte de Portugal y Galicia. Hay que decir que en España, 2011 no ha sido un año especialmente conflictivo en cuanto incendios se refiere. Esta captura de pantalla se obtuvo de la página web donde cualquier usuario puede ver la situación actual de riesgo de incendios de toda Europa, ofrecida por el *EFFIS* (http://effis.jrc.ec.europa.eu/current-situation), el 2 de septiembre de 2011.

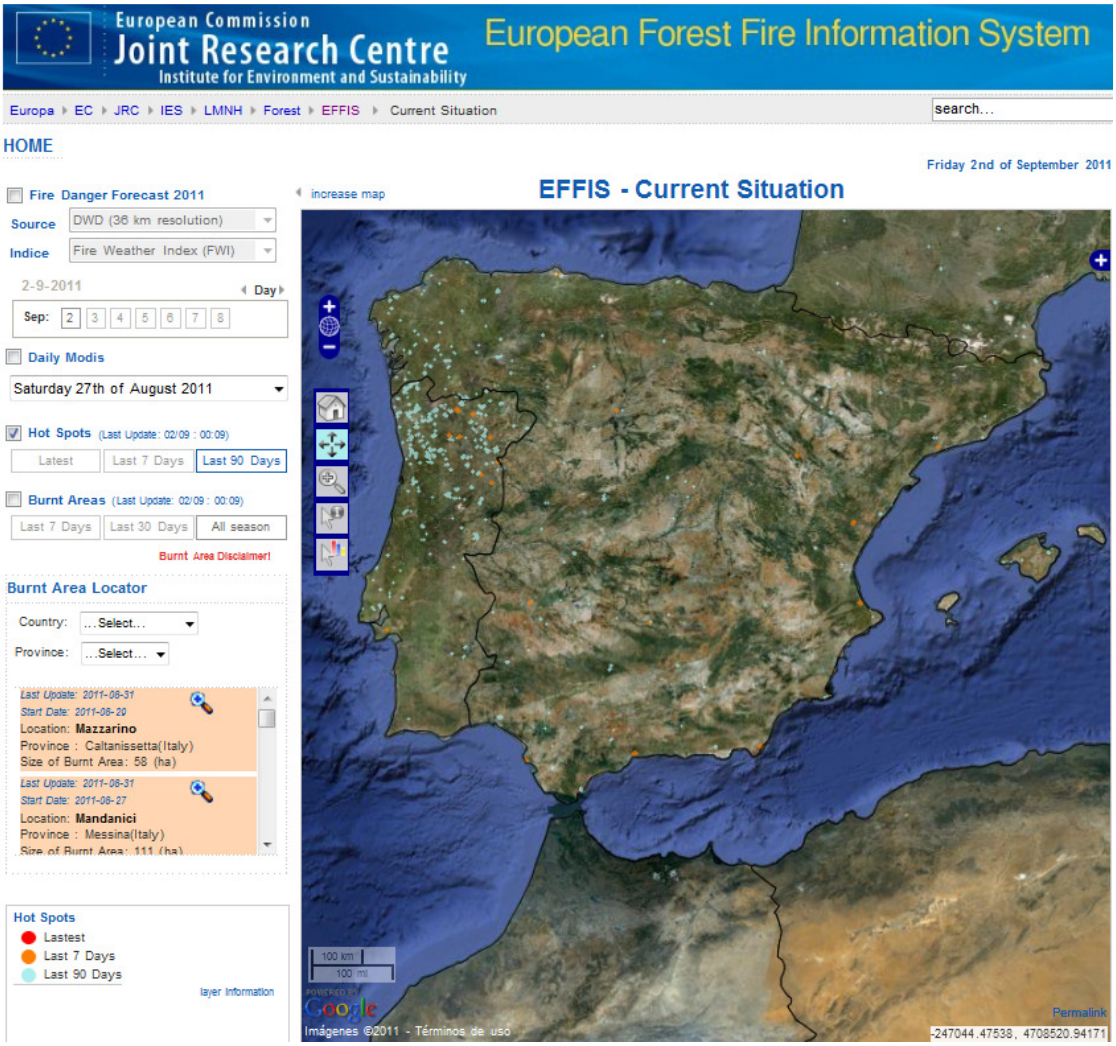

Figura 3.3: *Hotspots* detectados durante los 90 días anteriores al 02/09/2011, indicados en la web del EFFIS.

La figura 3.3 muestra los *hotspots* detectados por el *MODIS* en los 90 días anteriores al 1 de septiembre de 2011. Se aprecia como este mapa es prácticamente igual que el anterior, donde se podía ver la superficie quemada confirmada. Hay que tener en cuenta que, tanto en esta interfaz web, como en los datos ofrecidos a través del servidor *WFS*, también tendríamos accesos a los *hotspots* identificados en las últimas 24 horas, o 7 días, además de a la totalidad de los datos. Por lo tanto se puede afirmar que una manera relativamente rápida de detectar un incendio nos la ofrece este sistema, pudiendo avisar de uno en una zona de posible difícil acceso y despoblada, donde de otro modo, la alerta saltaría cuando los daños hubieran sido mayores. En este caso es posible enviar antes un avión de reconocimiento e incluso un hidroavión, para, ante un comienzo de incendio, poder extinguirlo y dar la voz de alarma con anterioridad.

En la sección 7.1, 'Posibles Ampliaciones', de este mismo documento se da una explicación de lo que es el FWI (Fire Weather Index) y cómo lo podríamos usar para incrementar la funcionalidad de nuestra aplicación.

3.3 API para la descarga y tratamiento de datos del WFS

En las secciones anteriores de este capítulo se ha mostrado una visión general de lo que es un WFS. También se ha detallado el significado de los datos concretos con los que queremos trabajar dado el WFS del EFFIS. Esta sección presenta el modelo de objetos, o API, que hemos creado durante la realización de este proyecto para poder hacer consultas a este servicio, además de poder tratar los datos que nos ofrece de una forma automática.

Se podría decir que el software implementado en esta sección consta de dos partes bien diferenciadas. La primera sería la encargada de hacer la petición web, descargando un archivo que contenga los datos que cumplan con los parámetros de dicha petición. La segunda sería la carga de estos datos en un modelo de objetos que represente fielmente toda esta información.

Centrándonos en la primera tarea de las descritas en el parráfo anterior, encontramos que tendremos que generar una URL que represente una petición HTTP de tipo GET. Su formato será el siguiente:

```
"http://geohub.jrc.ec.europa.eu:80/effis/wfs?
              Request     =GetFeature&
              typeName    ={TYPE_NAME}&
              outputFormat={OUTPUT_FORMAT}&
              resultType  ={RESULT_TYPE}"
```

En la tabla 3.3 encontramos los posibles valores para cada parámetro de la URL a generar.

TYPE_NAME	OUTPUT_FORMAT	RESULT_TYPE
"EFFIS:Hotspots1Day" "EFFIS:Hotspots7Days" "EFFIS:HotspotsAll" "EFFIS:Fires7Days" "EFFIS:Fires30Days" "EFFIS:FiresAll"	"text/xml subtype=gml/3.1.1" "GML2" "GML2-GZIP" "SHAPE-ZIP" **"csv"** "gml3" "gml32" "json" "text/xml; subtype=gml/2.1.2" "text/xml; subtype=gml/3.2"	**"results"** "hits"

Tabla 3.3: Posibles valores de los parámetros de una URL para efectuar una petición al WFS del EFFIS.

En nuestro caso, el formato de salida que necesitamos es el CSV, y además usaremos como RESULT_TYPE los de tipo **"result"**. Sin enbargo, lo que sí que será necesario indicar expresamente es el tipo de información que queremos obtener, y que se especifica mediante el valor del parámetro TYPE_NAME. El significado de la información que nos devuelve con cada una de ellas está detallado en la tabla 3.2.

Para facilitar la creación de esta URL se crea la clase *WfsUrlFactory*, con la que nos bastará crear una instancia de un objeto de este tipo, indicándole los parámetros anteriores, para obtener la dirección HTTP que ha de devolvernos la información deseada. Una vez tengamos esta URL, y pasándosela a otra instancia de la clase *WfsDataDownloader*, si todo va bien, descargaremos el fichero CSV al directorio que le hayamos indicado.

En la figura 3.4 podemos ver el diagrama de clases que implementa las funciones indicadas en los párrafos anteriores. Además, también incluye las clases que pertenecen a la segunda parte del problema, que era modelizar la información obtenida. *ListOfHotspots* y *ListOfFires* representan listas de *Hotspots* y *Fires* respectivamente. Estas listas son capaces de leer un fichero CSV y crear tantas instancias de estos objetos, como contenga dicho fichero. Además, se incluyen funciones de filtrado, y algunas otras utilidades, como crear un *ListViewItem* de cada elemento, que nos ayudarán en la etapa de la creación de una GUI. Este diseño responde a la arquitectura de tres capas, que se explica más detalladamente en el capítulo 6.

Quedan cubiertas de este modo, las necesidades de obtener información en tiempo real sobre riesgos de incendio, e incendios ya detectados, para generar nuestro plan de vuelo. Hay que destacar que en cada instancia de las clases *Hotspot* y *Fire* vienen incluidas sus coordenadas geográficas en términos de latitud/longitud. Cada una de estas coordenadas simbolizará un vértice en nuestro grafo de representación del mapa. Puesto que podemos calcular la distancia existente entre cada una de estas coordenadas, tal y como se explicó en el capítulo anterior, mediante la clase *Waypoint*

y su método *Distance*, dicho valor será el mismo para las aristas que simbolicen la conexión entre vértices. En principio, diremos que este grafo será un grafo totalmente conectado, ya que con el avión somos capaces de ir de una coordenada concreta a cualquier otra de entre las que incluyamos en nuestro grafo.

En el siguiente capítulo se muestran los algoritmos utilizados para generar esta ruta óptima de vuelo.

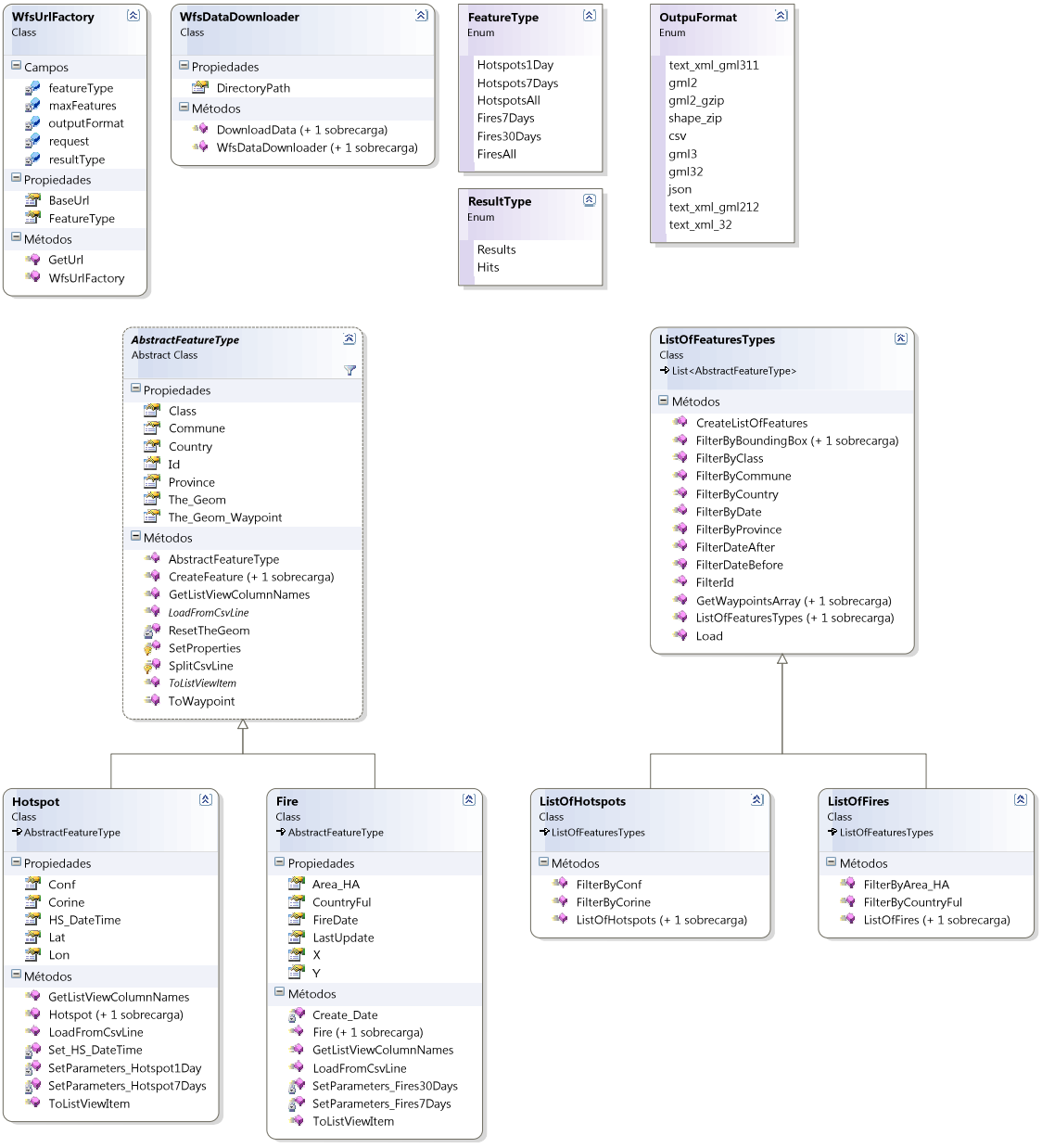

Figura 3.4: Diagrama de clases que modelan los datos y la interacción con el WFS.

CAPÍTULO 4 - ALGORITMOS UTILIZADOS PARA OPTIMIZAR EL PLAN DE VUELO

El capítulo actual presenta los diferentes tipos de algoritmos utilizados para optimizar la ruta de vuelo de nuestro avión en su tarea de prevención o extinción de incendios. Se han utilizado dos técnicas distintas. La primera es 'Ramificación y Poda'. Se muestran para ella las distintas cotas implementadas y los resultados obtenidos para diferentes configuraciones con la implementación realizada. La segunda técnica hace uso de los 'Algoritmos Genéticos'. También se indicarán los resultados obtenidos. Seguidamente, se explicará el funcionamiento de la implementación que une las dos técnicas para obtener mejores resultados en determinadas circunstancias. Y finalmente se indicarán unas conclusiones finales sobre los resultados obtenidos con cada una de las técnicas.

4.1 RAMIFICACIÓN Y PODA

4.1.1 - Introducción

Esta técnica de diseño, cuyo nombre en castellano proviene del término inglés *Branch and Bound,* se aplica normalmente para resolver problemas de optimización. Ramificación y Poda, al igual que el diseño Vuelta Atrás (*backtracking*), realiza una enumeración parcial del espacio de soluciones basándose en la generación de un árbol de expansión.[8][9]

Una característica que le hace diferente al diseño de Vuelta Atrás es la posibilidad de generar nodos siguiendo distintas estrategias. Recordemos que el diseño Vuelta Atrás realiza la generación de descendientes de una manera sistemática y de la misma forma para todos los problemas, haciendo un recorrido en profundidad del árbol que representa el espacio de soluciones. Es decir, *backtracking* explora TODO el árbol de posibles soluciones.

El diseño Ramificación y Poda en su versión más sencilla puede seguir un recorrido en anchura (estrategia LIFO) o en profundidad (estrategia FIFO), o utilizando el cálculo de funciones de coste para seleccionar el nodo que en principio parece más prometedor (estrategia de mínimo coste o LC).

Como veremos a continuación, además de estas estrategias, la técnica de Ramificación y Poda utiliza cotas para podar aquellas ramas del árbol que no conducen a la solución óptima. Para ello calcula en cada nodo una cota del posible valor de aquellas soluciones alcanzables desde ése. Si la cota muestra que cualquiera de estas

soluciones tiene que ser necesariamente peor que la mejor solución hallada hasta el momento no necesitamos seguir explorando por esa rama del árbol, lo que permite realizar el proceso de poda.

En consecuencia, y a la vista de todo esto, podemos afirmar que lo que le da valor a esta técnica es la posibilidad de disponer de distintas estrategias de exploración del árbol y de acotar la búsqueda de la solución, que en definitiva se traduce en eficiencia. La dificultad está en encontrar una buena función de coste para el problema, buena en el sentido de que garantice la poda y que su cálculo no sea muy costoso. Si es demasiado simple probablemente pocas ramas puedan ser excluidas. Dependiendo de cómo ajustemos la función de coste mejor algoritmo obtendremos.

4.1.2 - Conceptos básicos

El algoritmo de Ramificación y Poda (*branch and bound*) es una técnica basada en el recorrido de un árbol de soluciones que:

- Realiza un recorrido sistemático en un árbol de soluciones.

- El recorrido no tiene por qué ser necesariamente en profundidad sino que seguirá una estrategia de ramificación, guiada por estimaciones del beneficio, que se realizarán para cada nodo.

- Se usan técnicas de poda para eliminar nodos que no lleven a la solución óptima.

- La poda se realiza estimando en cada nodo las cotas de beneficio que se pueden obtener a partir del mismo.

- Un concepto fundamental para entender el algoritmo de ramificación y poda es el de nodo vivo, también llamado estado. Nodo vivo del árbol de expansión es un nodo con posibilidades de ser ramificado, es decir, un nodo que no ha sido podado. Para determinar en cada momento qué nodo va a ser expandido y dependiendo de la estrategia de búsqueda seleccionada, necesitaremos almacenar todos los nodos vivos/estados en alguna estructura que podamos recorrer.

4.1.3 - Metodología de resolución

El método general a aplicar para la resolución de un problema mediante este algoritmo es el siguiente:

1. **Para cada nodo `i`**, tendremos: **Cota Superior** `(CS(i))` y **Cota Inferior** `(CI(i))` del beneficio (o coste) óptimo que se podrían alcanzar a partir de ese nodo. Estas cotas determinarán el momento en el que se podrá realizar una poda. Estimación del beneficio (o coste) óptimo que se puede obtener a partir de ese nodo. Esta estimación se puede calcular en función del problema concreto que estemos tratando y su diseño suele ser la parte más importante a la hora de obtener un buen rendimiento del algoritmo. Además puede ayudar a decidir qué parte del árbol se explorará primero.

2. **Estrategia de poda**: Supongamos un problema de minimización de la función objetivo. En un punto intermedio del recorrido del árbol de soluciones nos podemos encontrar algún nodo a partir del cual no se pueda llegar a ninguna solución válida, o que no mejorará la mejor solución que tenemos hasta el momento. Es decir, se podrá podar un nodo `i`, si:

```
CS(i) >= Mejor_Solución_Encontrada_Antes
```

En tal caso, sabríamos que es imposible mejorar la solución actual que tenemos, y sería inútil recorrer esa rama del árbol, con lo que la podamos.

3. **Estrategia de ramificación**: De forma implícita, se tendrá en cuenta a la hora de llevar a cabo la ramificación del árbol de soluciones, la lista de nodos vivos en cada momento. La lista de nodos vivos contiene la lista de nodos que ya han sido generados pero que todavía no han sido explorados, para cada variable del problema a resolver. Son los nodos pendientes de tratar por el algoritmo.

Las posibles estrategias de ramificación y exploración del árbol de estados a emplear son las siguientes:

- Estrategia FIFO (*first in first out*): La lista de nodos vivos es una cola y, por lo tanto, el recorrido del árbol se realiza en anchura.

- Estrategia LIFO (*last in first out*): La lista de nodos vivos es una pila y, por lo tanto, el recorrido del árbol se realiza en profundidad.

- Estrategia LC (*least cost*): Se selecciona de toda la lista de nodos vivos aquél con el que se obtenga un mayor beneficio (o menor coste) para explorar a continuación.

Dadas las características del problema a resolver en nuestro caso, la estrategia de ramificación empleada ha sido la LC, utilizando un *heap* o cola de prioridad como estructura para almacenar estos nodos vivos pendientes de ramificar.

4.1.4 - Algoritmo General

Básicamente, en un algoritmo de Ramificación y Poda básico, se realizan tres etapas:

La primera de ellas, denominada de *Selección*, se encarga de extraer un nodo de entre el conjunto de los nodos vivos. La forma de escogerlo va a depender directamente de la estrategia de búsqueda que decidamos para el algoritmo.

En la segunda etapa, la *Ramificación*, se construyen los posibles nodos hijos del nodo seleccionado en el paso anterior.

Por último se realiza la tercera etapa, la *Poda*, en la que se eliminan algunos de los nodos creados en la etapa anterior. Esto contribuye a disminuir en lo posible el espacio de búsqueda y así atenuar la complejidad de estos algoritmos basados en la exploración de un árbol de posibilidades. Aquellos nodos no podados pasan a formar parte del conjunto de nodos vivos, y se comienza de nuevo por el proceso de selección.

A partir de ahora, a este conjunto de nodos vivos o estados activos lo denotaremos con la letra **A**.

Dependiendo de qué tipo de solución estemos buscando, la condición de finalización del algoritmo variará:

- Si simplemente buscamos una solución de entre todas las posibles, terminará cuando haya encontrado un nodo vivo o estado que sea solución factible.
- Si por el contrario lo que queremos es encontrar la mejor solución de entre todas las posibles, en las estrategias con FIFO y LIFO se tendrá que esperar a que el algoritmo vacíe todo el conjunto de estados activos A.
- En la estrategia LC, puede finalizar con la misma condición que la anterior, o en caso de extraer un nodo cuya cota no vaya a mejorar una ya encontrada anteriormente, también podremos detenerlo. Esto es debido a que, puesto que estamos extrayendo primero siempre el mejor nodo de todas los que tenemos activos, y que éste no mejora una solución factible ya encontrada, es fácil entender que ninguna de las que se encuentra en A tampoco lo hará.

En todos los casos anteriores, si se agota el conjunto de nodos activos sin haber encontrado ninguna solución, significa que no existe para esa instancia del problema.

4.1.5 - Pseudocódigo del algoritmo y ejemplo sencillo

A continuación se muestra pseudocódigo del algoritmo de Ramificación y Poda. Esta sería una versión simplificada donde, dependiendo del problema concreto, tendríamos una manera distinta de calcular las funciones de coste de los nodos vivos, pero, aparte de eso, el algoritmo funcionaría de forma general para cualquier tipo de problema de optimización. Sean:

`G(x)`	Función de estimación de coste para un nodo vivo x.
`A`	Conjunto de nodos o estados vivos (posibles soluciones).
`esFactible`	Función que considera si la propuesta es válida.
`esSolución`	Función que comprueba si se satisface el objetivo.
`óptimo`	Valor de la función a optimizar evaluado sobre la mejor solución encontrada hasta el momento.

El pseudocódigo será el siguiente:

```
// Variables globales
Estructura_LIFO_FIFO_o_LC A;
Real optimo;

Funcion RyP_Inicial()
{
    optimo = peor_valor_posible();
    A      = new Estructura_LIFO_FIFO_o_LC();
    State nodo_inicial = GetNodoInicial();

    // Añadimos el nodo inicial y ejecutamos el algoritmo de RyP
    A.add(nodo_inicial);
    RyP(A);
}


Funcion RyP (A)
{
    WHILE (no_vacio(A))
    {
        // Dependiendo del tipo de estructura que sea A
        // haremos una selección por LIFO, FIFO o LC
        State s = seleccionar(A);

        // Creamos los nuevos hijos resultantes del estado seleccionado
        List<State> l = ramificar(s);

        // Podamos: No añadimos los que sepamos que no
        // mejorarán la solución actual. A esto se le llama poda implícita
        FOREACH(State hijo in l)
        {
            IF (esFactible(hijo) AND G(hijo) < optimo)
                IF esSolucion(x)
                    optimo = x; // Hemos encontrado una solución mejor
                ELSE
                    A.add(hijo); // Podemos encontrar una solución mejor
        }
    }
}
```

Notar que usaremos un menor que (<) para los problemas de minimización y un mayor que (>) para problemas de maximización.

Para ilustrar el funcionamiento del algoritmo y los estados que se generan y podan tomaremos como ejemplo la figura 4.1. El problema que se trata aquí es buscar un subconjunto que sume 10 de entre los elementos {1, 2, 3, 4, 5}. Dado un número, si optamos por utilizarlo en la suma lo representaremos como si fuéramos por la rama derecha, y en caso de no hacerlo, por la izquierda. Así, por ejemplo, la solución {2, 3, 5} es una solución válida ya que sus elementos suman 10. Como cota inferior usaremos el número máximo que podemos sumar con todos los elementos restantes (la suma total posible). Si esta suma es inferior a 10 en algún estado hijo creado, podremos podarlo ya que nunca llegaremos a una solución a partir de él.

Lo importante de este ejemplo es ver cómo, con un algoritmo de *backtracking* hubiéramos tenido que generar 2^6-1 (63) estados, mientras que con esta estrategia hemos generado sólo 47. Notar que el número de nodos que podaremos, y en consecuencia, el tamaño total del problema dependerá en gran medida de la función de coste asociada y de la instancia del problema en concreto.

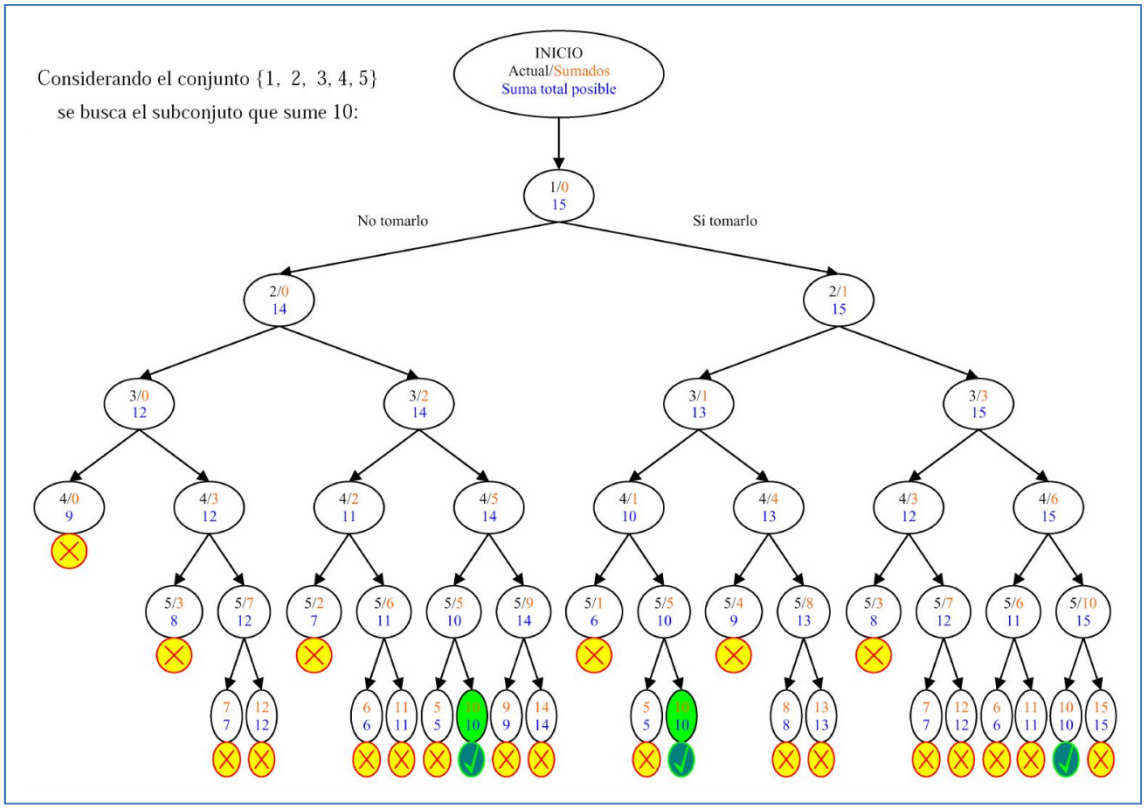

Figura 4.1: Ejemplo de estados generados en ramificación y poda. (Fuente: *http://es.wikipedia.org/wiki/Ramificación_y_poda*)

4.1.6 - Visión detallada de Ramificación y Poda

Ramificación y poda guarda relación con el método de búsqueda con retroceso (*backtracking*) en tanto que propone la búsqueda de una solución factible (óptima) explorando ordenadamente un árbol de estados y eliminando los nodos que no pueden conducir a la solución buscada.
La estrategia de ramificación y poda se distingue de la búsqueda con retroceso tanto en el orden de recorrido del árbol de estados como en la poda de estados. Quizá sea más apropiado denominar al método "*selección, ramificación y poda*", pues éstos son sus pasos básicos:

Selección. En cada instante hay una serie de estados no explorados a los que denominamos "**estados activos**" y se selecciona uno de ellos para explorar: el que parezca más prometedor. Este criterio conduce a una exploración por "primero el mejor". (La búsqueda con retroceso, por contra, sigue un criterio "último en entrar, primero en salir", propio de la pila con que se gestiona la exploración por primero en profundidad).

Ramificación. Es el proceso por el que un estado, que representa un conjunto de soluciones, se divide y genera otros estados que representan a subconjuntos suyos. Recordemos que un estado representa una solución incompleta de nuestro problema, y los nuevos estados hijos generados representan una solución con más detalle que la anterior, que nos acerca a una solución completa.

Poda. Es la supresión de aquellos estados activos que no pueden contener la solución factible óptima. Existen diferentes criterios de poda: **poda por factibilidad** (que ya conocemos por la búsqueda con retroceso), **poda por cota optimista**, **poda por cota pesimista** y **poda basada en la memorización de resultados intermedios** (esta última se puede aplicar a problemas abordables por la técnica de programación dinámica). En nuestro caso concreto, usaremos diferentes cotas optimistas con las que podar los estados que vayamos ramificando.

La técnica de **Ramificación y Poda** se suele interpretar como 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.

Es frecuente aplicar esta técnica de búsqueda a problemas para los que no se conoce un algoritmo eficiente. En tales casos, la estrategia de ramificación y poda presenta un

coste computacional exponencial para el peor de los casos, pero en la práctica, con un diseño adecuado de sus elementos y la debida atención a los detalles de implementación, el tiempo de ejecución puede resultar permisible para problemas de talla moderada. Es más, ciertas variantes de la técnica permiten sacrificar controladamente la optimalidad del resultado para acelerar el cálculo, es decir, proporcionan algoritmos de aproximación. Pero también es posible aplicar la técnica de "ramificación y poda" a problemas para los que ya se conoce un algoritmo eficiente: un diseño apropiado puede asegurar un coste para el peor de los casos igual al de los métodos ya conocidos y, en la práctica, acelerar notablemente el cálculo.

4.1.6.1 - Esquema de ramificación y poda

En general, notaremos por X el conjunto de ***soluciones factibles***. Los elementos de X suelen ser elementos que satisfacen ciertas restricciones y que pertenecen a un conjunto más amplio X'. Los elementos de X' son las **soluciones** y los de $X' - X$ son las **soluciones no factibles**. Buscamos un elemento de X que haga óptimo (mínimo o máximo) el valor de cierta **función objetivo** $f: X \to \mathbb{R}$:

$$\hat{x} = \arg \underset{x \in X}{opt} f(x)$$

Supondremos en lo sucesivo que $X \neq \emptyset$ y, por tanto, existe siempre una solución óptima.

4.1.6.2 - Notación

Al diseñar un algoritmo de ramificación y poda empezamos por definir el concepto de **estado**, que es una representación de un conjunto de soluciones (factibles o no).

$$\textit{Un estado representa un elemento de } P(X')$$

Usaremos los símbolos s, s', s'', etc. para denotar estados y los símbolos x, x', etc. para denotar soluciones factibles. La talla de un estado es el número de soluciones que contiene, por lo que $|s| = 1$ indica que el estado representa un conjunto con una sola solución. Las soluciones factibles de un estado s son $s \cap X$. Es frecuente que los estados representen conjuntos de soluciones que comparten un prefijo. Usamos una notación como ésta para indicar un estado con un prefijo de talla k común a todas sus soluciones:

$$(s1, s2, \ldots, sk, \quad ?)$$

Reservamos las letras mayúsculas para los conjuntos de estados, que son conjuntos de subconjuntos de X', es decir, elementos de $P(P(X'))$. En particular, usaremos la letra A para denotar el **conjunto de estados activos**.

$$A = conjunto\ de\ estados\ activos$$

El conjunto A presenta un contenido diferente tras cada iteración del algoritmo. Denotaremos con $A^{(0)}$ su valor inicial y con $A^{(i)}$ su valor tras la i-ésima iteración.
Los estados han de permitir diseñar cómodamente un proceso de **ramificación**. Dicho proceso se aplica sobre un estado (subconjunto de X') y devuelve un conjunto de estados (conjunto de subconjuntos de X'). En general puede haber un número arbitrario de nuevos estados.

Representamos el proceso de ramificación por medio de la función

$$branch : P(X') \to P(P(X'))$$

Hemos de seguir un **criterio de selección** del estado activo que se ramifica en cada iteración. Hay tres criterios de selección: primero en profundidad, primero en anchura y primero el mejor (aunque veremos que las dos primeras pueden reducirse a casos particulares del tercero). Cualquiera que sea el método escogido lo representaremos por medio de una función $select$ que escoge un estado de A y, por tanto, tiene por perfil

$$select : P\big(P(X')\big) \to P(X')$$

Finalmente, el proceso de **poda** puede ser descrito por una función de perfil

$$prune : P(P(X')) \to P(P(X'))$$

La función objetivo sólo puede aplicarse sobre soluciones factibles, pero un estado terminal no es un elemento, sino un conjunto con un solo elemento. No obstante, abusaremos de la notación y cuando $s = \{x\}$, asumiremos que $f(s)$ se define como $f(x)$.

4.1.6.3 -Esquema con cola de prioridad para el conjunto de estados activos

El criterio de selección afecta a la estructura de datos con que hemos de representar A. Consideremos tres criterios: primero en profundidad, primero en anchura y primero el mejor. Los dos primeros criterios conducen a **estrategias de búsqueda ciega** o **no informadas**, mientras que la última permite diseñar una **estrategia de búsqueda heurística** o **informada**, pues se usa información propia del problema para orientar la búsqueda. Cada estrategia presenta ciertas ventajas:

Primero en profundidad: conduce rápidamente a estados que representan conjuntos de talla unitaria y, por tanto, encuentra rápidamente soluciones factibles, pero que no tienen porqué estar "próximas" a la solución óptima.

Primero en anchura: conduce muy lentamente a estados de talla unitaria pero, al ser toda solución factible de talla finita, presenta la ventaja de encontrar con seguridad la solución óptima si el factor de ramaje es finito, aunque X sea de talla infinita.

Primero el mejor: sus prestaciones dependen de la calidad de la estimación del valor de la función objetivo sobre la mejor solución factible de un estado. Si ésta es buena, se dirige rápidamente hacia una solución factible cuyo valor de la función objetivo está próximo al óptimo.

El último criterio es el más interesante de todos, y es el que utilizaremos en nuestra implementación, pues suele conducir a los procedimientos más eficientes. La determinación de qué estado de A es el mejor en cada instante nos obliga a estimar por medio de cierta **función de puntuación** de estados qué valor de la función objetivo es óptimo en cada estado $s \in A$. Notaremos con $score: P(X') \rightarrow \mathbb{R}$ a dicha función. En cualquier caso, los dos primeros criterios de selección (primero en profundidad y primero en anchura) pueden verse como casos particulares de la selección por primero el mejor. Si puntuamos los estados con un número que indique su profundidad en el árbol y el instante en que se ha generado el estado, la selección del estado de menor puntuación conduce a una búsqueda en anchura. Con una ligera variante de esta función de puntuación, la búsqueda por primero el de mayor puntuación es equivalente a una búsqueda por primero en profundidad. Nos ahorraremos, pues, hablar en el futuro de los diferentes criterios de selección suponiendo siempre una búsqueda por primero el mejor.

Esta estrategia de búsqueda requiere el empleo de una estructura de datos que, en principio, asegure eficiencia en tres operaciones: inserción de estados puntuados, extracción del estado con mejor puntuación y eliminación de los estados no

prometedores. En cada iteración del algoritmo se extrae el estado más prometedor, se realizan b inserciones en A y se podan p estados. La tabla X.X muestra la complejidad con que estas operaciones se pueden efectuar empleando diferentes estructuras de datos (ordenación con respecto a la función de puntuación).

	Extracción	b Inserciones	p Borrados
Lista	$O(\|A\|)$	$O(b)$	$O(\|A\|)$
Lista ordenada	$O(1)$	$O(b \lg b + \|A\|)$	$O(\|A\|)$
Heap	$O(lg\|A\|)$	$O(b\ lg\|A\|)$	$O(\|A\|)$

Tabla 4.1: Coste de una extracción, b inserciones y p borrados en función de la estructura de datos con que se implementa A, el conjunto de estados activos.

A la vista de la tabla, resulta evidente que el heap ofrece un coste aceptable tanto para inserciones como para la extracción del óptimo. El borrado de p elementos es una operación costosa en cualquiera de las estructuras. Hasta que no encontremos una solución satisfactoria para la ejecución eficiente de esta operación, no podemos inclinarnos definitivamente por ninguna estructura de datos (al menos no en términos asintóticos).

Hay un último comentario que hacer acerca de la eficiencia: algunas funciones de puntuación son más interesante que otras en función de la eficiencia computacional con la que puede efectuarse su cálculo. Una propiedad interesante de algunas funciones de puntuación es la *incrementalidad del cálculo*, esto es, la posibilidad de efectuar un cálculo rápido de la función cuando se genera un estado por ramificación de otro aprovechando información de la función de puntuación del segundo. Otra propiedad que observan algunas funciones de puntuación es la coincidencia de su valor con el de $f(x)$ cuando el estado sobre el que se calcula es un estado unitario $s = \{x\}$.

4.1.7 - Comentarios acerca de la eficiencia en Ramificación y Poda

El procedimiento de búsqueda exhaustiva es, en gran cantidad de problemas, extremadamente costoso, ya que la talla del espacio de búsqueda es exponencial con parámetros que miden la talla del problema. Los algoritmos de ramificación y poda sirven como estrategia de reducción de complejidad en algunos de dichos problemas: cuando se poda una rama del árbol de estados, se está eliminando un número de soluciones que es (potencialmente) exponencial. Pero eliminar un número exponencial de elementos de un conjunto de talla exponencial no tiene por qué reducir la talla del conjunto a una cantidad polinómica de elementos. Así pues, el propio algoritmo de ramificación y poda puede resultar un mecanismo de resolución no aplicable en la práctica. Hay que decir, no obstante, que la talla de numerosos problemas de interés práctico puede ser suficientemente reducida como para que la ramificación y poda resuelva en un tiempo razonable el problema sin que ocurra lo mismo con la búsqueda exhaustiva.

Es posible determinar la complejidad computacional de cada una de las iteraciones del bucle principal del esquema. En general, sin embargo, no es posible determinar la complejidad del proceso completo, que depende enormemente de la instancia del problema a resolver y de la *calidad* de la función de puntuación y cota optimista.

En una iteración dada, el coste temporal vendrá dado por el proceso de selección, el proceso de ramificación y el cálculo de la cota sobre cada uno de los nuevos estados. También se puede realizar un estudio del coste de la poda explícita, si es que ésta tiene lugar.

Al igual que ocurre con la búsqueda con retroceso, la eficiencia del algoritmo depende de que la función de ramificación proporcione conjuntos de talla bien balanceada y que estos conjuntos no presenten elementos en común.

4.1.8 - El problema del viajante de comercio

Como se ha descrito anteriormente, nuestro problema de optimización de una ruta de vuelo para que un avión pase por un conjunto de puntos recorriendo la mínima distancia posible es un caso particular del problema más conocido como TSP o (*Trade Salesman Problem* por sus siglas en inglés). Para poder aplicarle una estrategia de Ramificación y Poda usaremos la siguiente notación.

Dado un digrafo $G = (V, E)$ ponderado por una función $d: E \to \mathbb{R}^+$, deseamos encontrar un ciclo *hamiltoniano* de peso mínimo. Recordemos que un ciclo hamiltoniano es un camino que visita todos los vértices una sola vez, excepto el de partida, que se visita dos veces. Sin pérdida de generalidad, supondremos que el vértice inicial es un vértice arbitrario u, pues cualquier vértice del grafo puede considerarse inicio y final del ciclo.
El conjunto de soluciones factibles es

$$X = \left\{ (v_0, v_1, \ldots, v_{|V|}) \in V^{|V|+1} \mid v_0 = v_{|V|}; \bigcup_{0 \le i < |V|} \{v_i\} = V; (v_i, v_{i+1}) \in E, 0 \le i < |V| \right\}$$

Nuestra función objetivo es

$$f((v_0, v_1, \ldots, v_{|V|})) = \sum_{0 \le i < |V|} d(v_i, v_{i+1}),$$

y deseamos calcular

$$(\hat{v}_0, \hat{v}_1, \ldots, \hat{v}_{|V|}) = \arg \min_{(v_0, v_1, \ldots, v_{|V|}) \in X} \sum_{0 \le i < n} d(v_i, v_{i+1}).$$

4.1.8.1 - Estados y ramificación

En los problemas en que se desea calcular un camino en un grafo es frecuente modelar los estados como prefijos de un camino completo. Representaremos un estado con una secuencia de vértices no repetidos. Un estado representará una solución completa cuando su talla coincida con el número de vértices y será factible si hay una arista entre el último vértice y el primero.

La ramificación de un estado $(v_0, v_1, \dots, v_k, ?)$ será el conjunto de estados de la forma $(v_0, v_1, \dots, v_k, v_{k+1}, ?)$, donde (v_k, v_{k+1}) es una arista del grafo y el vértice v_{k+1} es diferente de cualquiera de los que aparecen en el prefijo del camino.

4.1.8.2 - Catálogo de cotas implementadas

Nos resta la parte más difícil: el diseño de una función de puntuación y una cota inferior que ayude a orientar el algoritmo de Ramificación y Poda, y a reducir significativamente el espacio de búsqueda.

Vamos a presentar las diferentes cotas diseñadas e implementadas durante la realización de este proyecto. En concreto serán tres. El esquema general del algoritmo de Ramificación y Poda no variará, lo único que cambiará será la función de coste utilizada, teniendo que ser una de entre las tres descritas a continuación. Las diferentes soluciones que propondremos aparecen expuestas por orden creciente de "calidad" y "complejidad" de la cota optimista que sirve para la puntuación de estados y para su eventual poda. Todas ellas suman una cantidad al peso de la parte conocida de un estado. Con objeto de facilitar el seguimiento de la exposición, mostramos ahora una lista de las diferentes cotas implementadas, y más adelante pasamos a detallarlas:

- Cota Optimista 1: Compleción del ciclo con las aristas de mínimo coste que parten de vértices no visitados.
- Cota Optimista 2: Cota basada en la matriz de *Floyd-Warshall*.
- Cota Optimista 3: Cota basada en la matriz de costes reducida.

Para cada una de estas cotas, una vez expuesto con detalle su diseño, se pasará a indicar la implementación realizada mediante el diagrama de clases y la explicación de los métodos y atributos más importantes. Más adelante se mostrarán los resultados alcanzados sobre un conjunto de pruebas utilizando las dos primeras cotas.

4.1.8.3 - Cota 1 - Compleción del ciclo con las aristas de mínimo coste que parten de vértices no visitados.

Sea cual sea el ciclo óptimo que podamos formar ramificando un estado $(v_1, v_2, \dots, v_k, ?)$, sabemos que el resto de vértices formará un camino que empieza en v_k, pasa por todo vértice diferente de los ya visitados (una sola vez) y finaliza en v_1. Todo par de vértices contiguos, u y v, debe ser un elemento de E. Podemos relajar el problema eliminando esta restricción, es decir, escogiendo la arista de menor coste que parte de cada uno de los vértices no visitados y la de menor coste que parte del último vértice visitado.

$$optimistic\ ((v_0, v_1, \dots, v_k, ?)) = \sum_{0 \le i < k} d(v_i, v_{i+1}) + \sum_{v \in V - \{v_0, v_1, \dots, v_{k-1}\}} \min_{(v,w) \in E} d(v, w)$$

Esta es una cota inferior de f para cualquier estado, pues toda solución del problema original es una solución del problema relajado. Además, seleccionar una arista que parte de un estado hace que el resto de aristas que parten del mismo estado sean no elegibles. Podemos precalcular el valor de la arista de menor peso que parte de cada vértice:

$$m_v = \min_{(v,w) \in E} d(v, w)$$

Esta cota es monótona no decreciente en la dirección de la raíz a las hojas, su valor coincide con la función objetivo en las soluciones factibles y su cómputo es incremental y realizable en tiempo $O(1)$:

$$optimistic\ ((v_1, v_2, \dots, v_k, v_{k+1}, ?)) = optimistic((v_1, v_2, \dots, v_k, ?)) + d(v_k, v_{k+1}) - m_k$$

En la práctica, aunque esta ha sido la cota menos informada de las implementadas, es la que mejores resultados ha ofrecido en comparación con las otras dos que restan por explicar. Esto es gracias a su coste de cálculo constante. Se realiza un pequeño pretratamiento sobre los vértices del grafo teniendo que guardar simplemente una variable más que nos indique el peso de la arista E de menor peso que sale de cada uno. Como se puede apreciar, además, el coste en términos de memoria también es mínimo, ya que sólo estaremos guardando un número entero más por cada nodo que tenga el grafo.

A partir de ahora, al estado que nos implementará esta cota le llamaremos *State,* ya que es el nombre de la clase donde se implementa sobre el código el nodo vivo que representará un estado en nuestro algoritmo de Ramificación y Poda con estas condiciones.

En la siguiente figura podemos ver la representación en el diagrama de clases que muestra *State,* utilizado en el algoritmo de Ramificación y Poda.

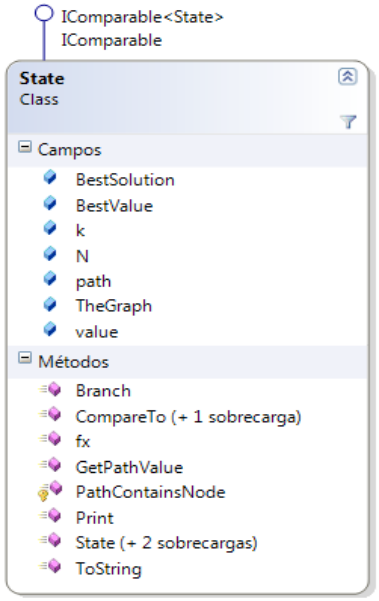

Figura 4.2: Representación en el diagrama de clases de *State*

Las siguientes tablas indican el nombre y la descripción de las variables (estáticas y dinámicas) y las funciones más importantes implementadas en la clase anterior. Con esto nos podemos hacer una idea del funcionamiento de nuestro algoritmo y de cómo se va calculando esta cota de forma incremental.

Variables estáticas (Comunes a todas las instancias)

	Tipo	Nombre	Descripción
	State	BestSolution	Estado que representa la mejor solución alcanzada hasta el momento.
	int	BestValue	El valor de la función de coste para *BestSolution*.
	Graph	TheGraph	Instancia del Grafo que está siendo usado por todos los *States*.
	int	N	Número de nodos en el grafo anterior.

Variables dinámicas

	Tipo	Nombre	Descripción
	int[]	path	Vector de enteros que indica el camino actual para este estado.
	int	k	Tamaño del camino actual.
	int	value	Valor de la función de coste (cota 1) para este *State*.

Constructores

	Nombre	Descripción
	State()	Constructor inicial de la clase *State*, empezando en el primer nodo del grafo. Representa el estado (v_1, ?) .
	State(State parent, int idNewNode, int newValue)	Constructor privado usado sólo en el método *Branch*. Copia el camino del padre en el suyo propio y añade el identificador del nuevo vértice por el que pasar, además del valor de su función de coste (cota1).

Métodos

	Tipo	Nombre y parámetros	Descripción
	List<State>	Branch()	Ramifica este nodo. Además, actualiza las variables *State.BestValue* y *State.BestSolution* en caso de ser necesario. También efectúa una poda pasiva, *i.e.*, no crea hijos peores que el *state State.BestSolution*.
	int	CompareTo(State other)	Compara este State con el especificado. Este método se usa en la *PriorityQueue* del algoritmo de Ramificación y Poda para ordenar los estados activos.
	int	fx(int linkWeight)	Función de coste para un posible nodo hijo. Tiene en cuenta el peso de la siguiente arista que escogemos al expandir el *state* actual en nuestro camino.
	bool	PathContainsNode(int idNode)	Devuelve un booleano indicando si en el *path* actual existe un nodo con el identificador especificado.

Para finalizar la explicación de esta cota implementada, se muestra a continuación la parte más importante del código desarrollado en la clase *State.cs*. Recordemos que el lenguaje de programación utilizado es c#.

Aquí se puede apreciar como, con la función fx, podemos conocer en un tiempo de $O(1)$ cuál será el valor de un posible nodo hijo a ramificar, sin tener siquiera que crearlo en caso de empeorar el mejor estado conocido hasta el momento. Además, como se ha indicado anteriormente, se sigue una estrategia de poda implícita en el método de ramificación (*branch*). No estamos eliminando estados del conjunto de estados vivos A, sino que directamente no se crean ni añaden cuando se cumple que $fx(hijo) \geq BestValue$.

El código generado para las funciones *fx* y *branch* en la clase *State.cs* es el siguiente:

```
public int fx(int linkWeight)
{
  // fx =  coste de llegar a este estado
  //     + coste del link para llegar desde este estado a otro nodo
  //     - mínimo peso de la mejor arista que salía desde el ultimo nodo de este camino
  return this.value + linkWeight -
         TheGraph.GetNode(this.path[this.k - 1]).minLinkWeight;
}


public List<State> Branch()
{
    int currentNode = path[k - 1];
    Node n = TheGraph.GetNode(currentNode);
    Link l;

    if (this.k == N) // Es solución?
    {
        l = n.GetLinkWith(this.path[0]);
        if (l != null)
        {
            this.value = fx(l.weight);
            if (this.value < State.BestValue)
            {
                // Si hemos encontrado una solución mejor, la guardamos
                BestSolution = this;
                BestValue    = this.value;
            }
        }
        return null;
    }
    else
    {
        List<State> list = null;
        int id2;
        int aux;

        foeach (Link l in n.Links)
        {
            id2 = l.node2.id;
            if (!this.PathContainsNode(id2))
            {
                aux = fx(l.weight);
                if (aux < State.BestValue)
                {
                    State s = new State(this, id2, aux); // Ramificamos
                    if (list == null)
                        list = new List<State>();
                    list.Add(s);
                }
                else
                {
                    State.NbPruned++; // Aumentamos el número de nodos podados
                }
            }
        }
        return list;
    }
}
```

Listado 4.1: Código empleado para la ramificación en la clase State.cs

4.1.8.4 - Cota 2 – Cota basada en la matriz de *Floyd-Warshall*

Esta cota se basa en la matriz de caminos mínimos que obtenemos del algoritmo de *Floyd-Warshall*, el cual compara todos los posibles caminos a través del grafo entre cada par de vértices. El algoritmo es capaz de hacer esto con sólo V^3 comparaciones (esto es notable considerando que puede haber hasta V^2 aristas en el grafo, y que cada combinación de aristas se prueba). Lo hace mejorando paulatinamente una estimación del camino más corto entre dos vértices, hasta que se sabe que la estimación es óptima. Este algoritmo será un pretratamiento que ejecutaremos una sola vez al inicio de nuestra ejecución, una vez tengamos el grafo, y por lo tanto no influirá en el coste temporal de la Ramificación y Poda. En cuanto al coste espacial, simplemente tendremos que alojar una matriz más de tamaño VxV, cosa que tampoco supone ningún problema en cuanto a gasto de memoria se refiere. [10]

El pseudocódigo de dicho algoritmo es el siguiente:

```
// Devuelve una matriz con las distancias mínimas
// de cada nodo al resto de los vértices.
// ady --> Matriz de adyacencia del grafo
// n   --> Número de vértices en el grafo
int[][] Floyd-Warshall(int[][] ady, int cn)
{
    // Copiamos todos los elementos de ady a la matriz path
    int[][] path = copy(ady);

    for(int i = 0; i < n; i++)
        path[i][i] = 0;

    for(int k = 0; k < n; k++)
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
            {
                int dt = path[i][k] + path[k][j];
                if(path[i][j] > dt)
                    path[i][j] = dt;
            }

    return path;
}
```

Como punto de partida, tendremos en cuenta que, una vez ejecutado el algoritmo de *Floyd-Warshall* sobre nuestro grafo tenemos una matriz donde conocemos la distancia que existe en el camino mínimo entre cada par de vértices (sabemos el coste, pero no el camino).

Es fácil entender que, dado un estado $(v_1, v_2, \dots, v_k, ?)$, una cota optimista puede ser la distancia del camino mínimo que hay desde el último vértice visitado v_k hasta el vértice más alejado de v_1, al que llamaremos v_z. Además, a esta distancia tendremos que sumarle también la distancia del camino mínimo desde v_z a v_1. Es decir, esta cota

optimista viene determinada por la siguiente ecuación, siendo *dfw* la distancia que obtenemos del camino mínimo entre dos vértices, extraída de la matriz de Floyd-Warshall:

$$optimistic_2\big((v_1, v_2, \dots, v_k, v_{k+1}, ?)\big) = \sum_{i=1}^{k} d(v_i, v_{i+1}) \; + \; dfw(v_{k+1}, v_z) + \; dfw(v_z, v_1)$$

Como cota optimista del estado inicial $(v_1, ?)$ tomaremos el doble de la distancia de ir desde el vértice v_1 al v_z, es decir, nuestra ruta, como mínimo, tendrá que ir y volver desde el nodo inicial al más alejado de él.

La suma de distancias del estado actual de la parte conocida del camino la podemos calcular de forma incremental. Simplemente tendremos que guardarla en cada estado, en una variable a la que llamaremos *pathValue*. La ecuación recursiva de este valor vendrá dada por la siguiente fórmula:

$$pathValue_i = pathValue_{i-1} + d(v_i, v_{i+1}) \quad \forall\, i \geq 1$$

Aprovechando esta definición, la representación final del estado de nuestra cota nos quedará de la siguiente forma:

$$optimistic_2\big((v_1, v_2, \dots, v_k, v_{k+1}, ?),\; pathValue_{k+1}\big) \\ = \; pathValue_k + \; d(v_k, v_{k+1}) \; + \; dfw(v_{k+1}, v_z) + \; dfw(v_z, v_1)$$

Esta cota es más informada que la anterior, ya que no tiene sólo en cuenta las distancias mínimas que parten de cada nodo. Ahora estamos considerando la distancia del camino mínimo que existe entre el nodo más alejado del vértice inicial de entre los vértices no visitados todavía. Por desgracia, esta cota no se puede calcular en tiempo constante. Cada vez que expandimos un nuevo nodo es necesario volver a ver qué vértice queda más alejado del vértice inicial de entre los que no estén en el camino actual. Por lo tanto, el cálculo de esta cota será de *O(V)*.

Aunque en un principio se esperaba un mejor comportamiento de esta cota, en la práctica, aunque se expanden menos nodos, el tiempo de ejecución aumenta respecto a la anterior. Esto es debido al gran número de estados que se ramifican y por tanto, al gran número de cálculos que tenemos que hacer para mantener la cota actualizada. Los resultados concretos obtenidos sobre un conjunto de pruebas se especifican más adelante.

Hemos definido una nueva clase, State2, subclase de la anterior, que servirá para implementar esta nueva cota.

En la siguiente figura podemos ver la representación en el diagrama de clases que muestra *State2,* utilizada en el algoritmo de Ramificación y Poda.

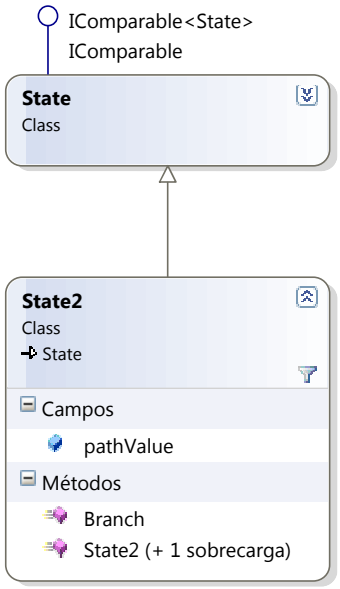

Figura 4.3: Representación en el diagrama de clases de *State2*

Puesto que esta clase hereda todas las variables y funciones que tenía *State*, simplemente se tendrá que declarar otro atributo al que llamaremos *pathValue*, para poder calcular esta segunda cota tal como se ha descrito.

La variable *pathValue* contendrá el valor de la parte conocida del camino recorrido para cada instancia de State2 donde se encuentre.

Además, se tendrá que sobrecargar los constructores y las operaciones de Branch, es decir, volverlas a definir para que el cálculo de esta cota se implemente acorde a como se ha indicado anteriormente.

4.1.8.5 - Cota 3 – Cota basada en la matriz de costes reducida.

En esta cota se incluye como información adicional una *matriz de costes reducida* para cada estado o nodo vivo que se expanda en nuestro algoritmo de Ramificación y Poda.

Antes de continuar, hay que decir que se incluye en este documento la explicación de dicha cota ya que se ha llegado a implementar en nuestro proyecto. No obstante, y debido a que su coste y sus resultados no eran nada óptimos, no se llegó a incluir en el *benchmarking* realizado sobre los algoritmos. Esta decisión fue tomada en base al elevado consumo de tiempo empleado en instancias del problema relativamente pequeñas (15-17 vértices), viéndose claramente que no llegaría a ser la cota que finalmente utilizaríamos. Seguimos pues, con la descripción de la cota.

Diremos que una fila o columna de una matriz está *reducida* si contiene al menos un elemento cero, y el resto de los elementos son no negativos. Una matriz se dice *reducida* si y sólo si todas sus filas y columnas están reducidas. Por ejemplo, dada la matriz de adyacencia:

$$\begin{pmatrix} \infty & 15 & 7 & 4 & 20 \\ 1 & \infty & 16 & 6 & 5 \\ 8 & 20 & \infty & 4 & 10 \\ 4 & 7 & 14 & \infty & 3 \\ 10 & 35 & 15 & 4 & \infty \end{pmatrix}$$

Podemos calcular su matriz reducida restando respectivamente 4, 1, 4, 3 y 4 a cada fila, y luego 4 y 3 a las columnas 2 y 3, obteniendo la matriz:

$$\begin{pmatrix} \infty & 7 & 0 & 0 & 16 \\ 0 & \infty & 12 & 5 & 4 \\ 4 & 12 & \infty & 0 & 6 \\ 1 & 0 & 8 & \infty & 0 \\ 6 & 27 & 8 & 0 & \infty \end{pmatrix}$$

En total hemos restado un valor de 23 (4 + 1 + 4 + 3 + 4 + 4 + 3), que es lo que denominaremos el *coste* de la matriz.
De esta forma, dada una matriz de adyacencia de un grafo ponderado podemos obtener su matriz reducida calculando los mínimos de cada una de las filas y restándoselos a los elementos de esas filas, haciendo después lo mismo con las columnas.

Respecto a la interpretación del *coste*, pensemos que restando una cantidad *t* a una fila o a una columna decrementamos en esa cantidad el coste de los recorridos del grafo. Por tanto, un camino mínimo lo seguirá siendo tras una operación de sustracción de filas o columnas. En cuanto a la cantidad total sustraída al reducir una matriz, ésta será una cota inferior del coste total de sus recorridos. Es decir, siendo optimistas, como mínimo, el camino que ha de pasar por todos los vértices que nos quedan por visitar, tendrá una distancia del *coste* de la matriz reducida. En consecuencia, para el ejemplo anterior hemos obtenido que 23 es una cota inferior para la solución al problema del viajante.

Esto es justo lo que vamos a utilizar como función de coste LC para podar nodos del árbol de expansión. Así, a cada estado o nodo vivo le vamos a asociar una matriz reducida y un coste acumulado. Para ver cómo trabajamos con ellos, supongamos que M es la matriz reducida asociada al nodo n, y sea n' el hijo de n que se obtiene incluyendo el arco $\{i,j\}$ en el recorrido.

- Si n' es una hoja del árbol, esto es, una posible solución, su coste va a venir dado por el coste que llevaba n acumulado más $M[i,j]+M[j,1]$, que es lo que completa el recorrido. Esta cantidad coincide además con el coste de tal recorrido.

- Por otro lado, si *n*' no es una hoja, su matriz de costes reducida M' vamos a calcularla a partir de los valores de M como sigue:

  a) En primer lugar, hay que sustituir todos los elementos de la fila *i* y de la columna *j* por ∞. Esto elimina el posterior uso de aquellos caminos que parten del vértice *i* y de los que llegan al vértice *j*.

  b) En segundo lugar, debemos asignar $M'[j,1]= \infty$, eliminando la posibilidad de acabar el recorrido en el siguiente paso (recordemos que n' no era una hoja).

  c) Reducir entonces la matriz M', y ésta es la matriz que asignamos al nodo *n*'.

Como coste para n' vamos a tomar el coste de *n* más el coste de la reducción de M' más, por supuesto, el valor de $M[i,j]$. Es importante señalar en este punto que la reducción no se realiza teniendo en cuenta los elementos con valor ∞, no obteniéndose *coste* alguno en aquellas filas o columnas cuyos elementos tomen todos tal valor.

Sea *cmr(M, {i, j})* la función que nos devuelve el coste de la matriz reducida para nuestra matriz de adyacencias *M*, eliminando la arista que va del vértice *i* al vértice *j*, la ecuación de la cota optimista que obtendremos será la siguiente:

$$optimistic_3\left((v_1, v_2, \dots, v_i, v_j, ?), M'\right) = optimistic_3\left((v_1, v_2, \dots, v_i, ?), M\right) + d(v_i, v_j) + cmr(M, \{v_i, v_j\})$$

Como hemos explicado anteriormente, el coste temporal del cálculo asociado a la función *cmr* es de $O(V^2)$. Nos sucede como en la anterior cota, ahora hemos '*informado*' mucho más nuestra función de coste, pero el precio a pagar en tiempo de cálculo es demasiado elevado y en la práctica no es viable cuando tenemos que expandir un gran número de nodos. Además, aquí sí que sufrimos un incremento importante en el gasto de memoria, ya que para cada nodo que expandamos, tendremos que asociarle su correspondiente matriz de costes reducida.

Al estado que nos implementará esta cota le llamaremos *State3.* En la siguiente figura podemos ver la representación en el diagrama de clases que muestra *State3,* utilizado en el algoritmo de Ramificación y Poda.

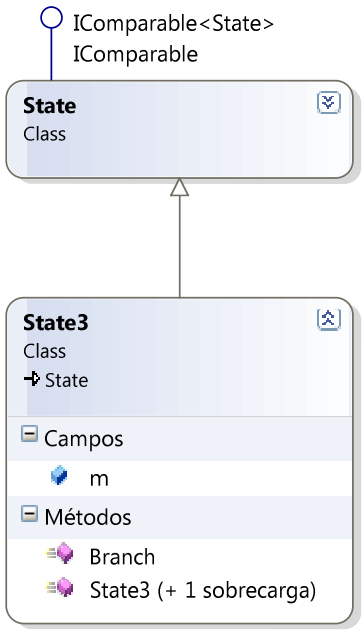

Figura 4.4: Representación en el diagrama de clases de *State3*

Esta clase también hereda todas las variables y funciones que tenía *State*. Una vez más, además de redefinir los constructores y el método de ramificación, guardaremos en la variable m la matriz de costes reducida a cada estado generado.

4.2 - ALGORITMOS GENÉTICOS

4.2.1 - Introducción

Los algoritmos genéticos (AG), fueron propuestos por primera vez en 1975 por John Holland, de la Universidad de Michigan. Los AG son, simplificando, algoritmos de optimización, es decir, tratan de encontrar la mejor solución a un problema dado entre un conjunto de soluciones posibles. Los mecanismos de los que se valen los AG para llevar a cabo esa búsqueda pueden verse como una metáfora de los procesos de evolución biológica.

Los AG son métodos adaptativos que pueden usarse para resolver problemas de búsqueda y optimización. Están basados en el proceso genético de los organismos vivos. A lo largo de las generaciones, las poblaciones evolucionan en la naturaleza acorde con los principios de la selección natural y la supervivencia de los más fuertes, postulados por Darwin (1859). Por imitación de este proceso, los AG son capaces de ir creando soluciones para problemas del mundo real. La evolución de dichas soluciones hacia valores óptimos del problema depende en buena medida de una adecuada codificación de las mismas.

En la naturaleza los individuos de una población compiten entre sí en la búsqueda de recursos tales como comida, agua y refugio. Aquellos individuos que tienen más éxito en sobrevivir y en atraer compañeros tienen mayor probabilidad de generar un gran número de descendientes. Por el contrario individuos poco dotados producirán un menor número de descendientes. Esto significa que los genes de los individuos mejor adaptados se propagaran en sucesivas generaciones hacia un número de individuos creciente. La combinación de buenas características provenientes de diferentes ancestros, puede a veces producir descendientes "superindividuos", cuya adaptación es mucho mayor que la de cualquiera de sus ancestros. De esta manera, las especies evolucionan logrando unas características cada vez mejor adaptadas al entorno en el que viven.

El poder de los AG proviene del hecho de que se trata de una técnica robusta, y pueden manejar con éxito una gran variedad de problemas provenientes de diferentes áreas, incluyendo aquellos en los que otros métodos encuentran dificultades. Si bien no se garantiza que el AG encuentre la solución óptima del problema, existe evidencia empírica de que se encuentran soluciones de un nivel aceptable, en un tiempo competitivo con el resto de algoritmos de optimización combinatoria.

En el caso de que existan técnicas especializadas para resolver un determinado problema, lo más probable es que superen al AG, tanto en rapidez como en eficiencia. El gran campo de aplicación de los AG se relaciona con aquellos problemas para los cuales no existen técnicas especializadas. Incluso en el caso en que dichas técnicas existan, y funcionen bien, pueden efectuarse mejoras de las mismas proponiendo variantes que las combinen con los AG.

Un esquema del funcionamiento general de un algoritmo genético podría ser el siguiente, donde marcaremos cada paso en lo que llamaremos fases:

1. Generar una población inicial aleatoria y suficientemente grande.
2. Iterar hasta un criterio de parada (objetivo o número máximo de generaciones alcanzado).
   2.1 Evaluar cada individuo de la población.
   2.2 Seleccionar los progenitores.
   2.3 Aplicar el operador de cruce (*crossover*) a estos progenitores y obtener sus descendientes.
   2.4 Aplicar el operador de mutación a los descendientes.
   2.5 Incluir la nueva descendencia para formar una nueva generación.

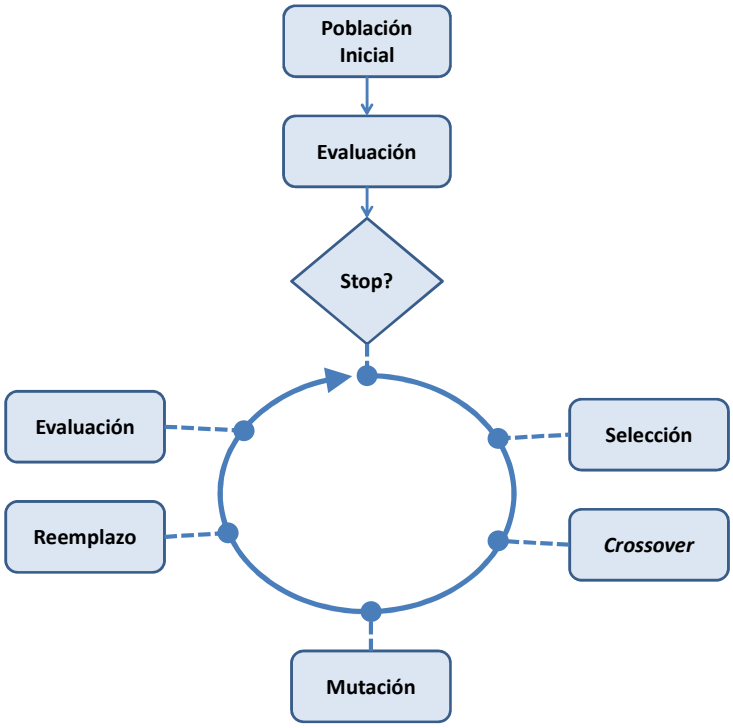

Figura 4.5: Esquema general del funcionamiento de un AG.

Dos aspectos que resultan cruciales en el comportamiento de los AG son la determinación de una adecuada **función objetivo** que nos permita evaluar

cuantitativamente la bondad de un individuo, así como la **codificación** utilizada para representar un individuo de la población.

Idealmente nos interesaría construir funciones objetivo con "ciertas regularidades", es decir funciones objetivo que verifiquen que para dos individuos que se encuentren cercanos en el espacio de búsqueda, sus respectivos valores en las funciones objetivo sean similares. Por otra parte una dificultad en el comportamiento del AG puede ser la existencia de gran cantidad de óptimos locales, así como el hecho de que el óptimo global se encuentre muy aislado.

La regla general para construir una buena función objetivo es que ésta debe reflejar bien el valor del individuo de una manera "real" y cuantitativa. Esta función objetivo es la que utilizaremos en la *fase 2.1* para **evaluar** los individuos.

El **operador de Selección** (*fase 2.2*) es el encargado de transmitir y conservar aquellas características de las soluciones que se consideran valiosas a lo largo de las generaciones. El principal medio para que la información útil se transmita es que aquellos individuos mejor adaptados (mejor valor de función de evaluación) tengan más probabilidades de reproducirse. Sin embargo, es necesario también incluir un factor aleatorio que permita reproducirse a individuos que aunque no estén muy bien adaptados, puedan contener alguna información útil para posteriores generaciones, con el objeto de mantener así también una cierta diversidad en cada población.

El **operador de cruce o *crossover*** (*fase 2.3*) permite realizar una exploración de toda la información almacenada hasta el momento en la población y combinarla para crear mejores individuos.

La **mutación** (*fase 2.4*) se considera un operador básico, que proporciona un pequeño elemento de aleatoriedad en la vecindad (entorno) de los individuos de la población. Si bien se admite que el operador de cruce es el responsable de efectuar la búsqueda a lo largo del espacio de posibles soluciones, también parece desprenderse de los experimentos efectuados por varios investigadores que el operador de mutación va ganando en importancia a medida que la población de individuos va convergiendo. El objetivo del operador de mutación es producir nuevas soluciones a partir de la modificación de un cierto número de genes de una solución existente, con la intención de fomentar la variabilidad dentro de la población. Existen muy diversas formas de realizar la mutación, desde la más sencilla (Puntual), donde cada gen muta aleatoriamente con independencia del resto de genes, hasta configuraciones más complejas donde se tienen en cuanta la estructura del problema y la relación entre los distintos genes.

Cada vez que se aplica el operador de cruce, nos encontramos con un número de nuevos individuos (la descendencia) que se han de integrar en la población para

formar la siguiente generación. La manera más común de insertarlos en dicha población es **reemplazando** (*fase 2.5*) estos nuevos individuos por otros más viejos y que parecen no presentar mejoría respecto de los nuevos.

Hasta aquí se ha presentado la idea básica de qué es un AG y su funcionamiento general. La siguiente sección presentará la solución concreta que se ha empleado para tratar nuestro problema de optimización de rutas de vuelo utilizando la técnica de los algoritmos genéticos.[11][12][13]

4.2.2 - TSP usando Algoritmos Genéticos

En primer lugar hay que mencionar que, a diferencia de la implementación de Ramificación y Poda, donde todo el código generado ha sido de principio a fin desarrollado por nosotros mismos, la parte del AG es una adaptación del código de Michael Lalena. Éste se puede encontrar en http://www.lalena.com/AI/Tsp/.[14]

Como ya sabemos, en el problema del TSP, el objetivo es encontrar la distancia más corta entre N ciudades diferentes. En esta implementación, al camino que utilicemos será llamado *tour*.

Suponiendo que utilizásemos una estrategia de *backtracking*, donde todas las combinaciones son probadas, un *tour* de N ciudades supondría un coste de N! sumas matemáticas. Un tour que pasase por 30 ciudades tiene que medir la distancia total de $2.65x10^{32}$ caminos diferentes. Asumiendo mil millones de sumas por segundo (algo desmesurado), esto tomaría 8411113007743247 años en ser calculado. Añadir una ciudad más provocaría incrementar este tiempo por un factor de 31. Obviamente, esta es una solución impracticable.

Un Algoritmo Genético puede ser usado para encontrar una solución en mucho menos tiempo. A pesar que no encontrará la mejor solución, con esta estrategia podremos encontrar una cercana a la óptima para un tour de 100 ciudades en menos de un minuto.

Existen un par de pasos básicos para resolver el problema del TSP usando un AG.

El primero, crear un grupo de muchos *tours* aleatoriamente en lo que llamaremos una Población, o *Population* por su término en inglés. Este algoritmo usa una estrategia voraz para iniciar la población, dando preferencia a enlazar cada ciudad con aquellas que tenga más cercanas.

El segundo paso será elegir 2 de los mejores (más cortos) *tours* padres en la población, y combinarlos para hacer 2 nuevos tours hijos. Con suerte, estos hijos serán mejores que ambos padres.

Un pequeño porcentaje del tiempo, los *tours* hijos serán mutados. Esto se hace para prevenir que todos los tours en la población finalmente converjan a rutas idénticas, es decir, se busca variabilidad en la población.

Los nuevos *tours* creados son insertados en la población reemplazando dos de los antiguos *tours*. Por lo tanto, el tamaño de la población seguirá siendo el mismo.

Nuevos *tours* hijos serán creados repetidamente hasta que se alcance el objetivo deseado o un número máximo de generaciones.

Como su nombre indica, los Algoritmos Genéticos imitan a la naturaleza y evolucionan usando los principios de la selección natural, también conocido como que sólo sobrevive el más fuerte.

Los dos problemas más complejos de usar un AG para resolver el TSP son la representación que hemos de darle a un tour, y el algoritmo de mezcla que es usado para combinar dos padres para crear dos tours hijos. A partir de ahora, a este algoritmo de mezcla de dos individuos para generar sus respectivos hijos, lo llamaremos *crossover*, tal y como se usa en la terminología habitual de los AG.

En un AG estándar, esta representación es una simple secuencia de números y el *crossover* se realiza eligiendo un punto aleatorio en la secuencia de los padres. Una vez hecho esto, se intercambia cada número en la secuencia que se encuentre tras ese punto de corte. En este ejemplo, el punto de crossover está entre el tercer y cuarto elemento en la lista. Para crear los hijos, cada subgrupo de las secuencias padres son intercambiados.

Padre 1	F A B | E C G D
Padre 2	D E A | C G B F
Hijo A	F A B | C G B F
Hijo B	D E A | E C G D

Tabla 4.2: Ejemplo sencillo de crossover

La dificultad con el TSP es que cada ciudad puede ser usada sólo una vez en un tour. Si las letras en el ejemplo anterior representasen ciudades, estos tours hijos creados por

esta operación de crossover no serían válidos. El hijo A parte de la ciudad F y B dos veces, y nunca parte de las ciudades D o E.

La representación no puede ser simplemente una lista de ciudades en el orden en que pasaremos por ellas. Se han creado otros métodos de representación para solucionar el problema del crossover. Aunque estos métodos no crearán tours no válidos, no toman en cuenta el hecho que el tour "A B C D E F G" es el mismo que "G F E D C B A". Para resolver el problema adecuadamente, el algoritmo de crossover será algo más complicado que el ejemplo anterior.

La solución planteada guarda los enlaces de cada vértice en ambas direcciones para cada tour. Es decir, cada vértice mantendrá dos referencias a los otros dos vértices con los que tiene conexión. En el anterior ejemplo, el Padre 1 sería guardado de la siguiente manera:

Ciudad	Enlace 1	Enlace 2
A	F	B
B	A	E
C	E	G
D	G	F
E	B	C
F	D	A
G	C	D

Tabla 4.3: Representación de un tour en el AG

En nuestro caso, la operación de *crossover* es más complicada que simplemente combinar dos *cadenas*. El *crossover* tomará cada enlace que existe en ambos padres y pondrá estos enlaces en ambos hijos. Seguidamente, para el Hijo A alternará entre tomar los enlaces que aparecen en el Padre 2 y el Padre 1. Para el Hijo B, alternará entre el Padre 2 y el Padre 1 tomando un conjunto diferente de enlaces. Para ambos hijos, existe la posibilidad en que un enlace pueda crear un *tour* no válido donde, en lugar de un único camino, existieran caminos desconectados. Estos enlaces deben ser rechazados. Para rellenar los enlaces que faltasen, se escogerán vértices de manera aleatoria. Habrá que tener en cuenta que este *crossover* no es completamente aleatorio.

En ocasiones, el AG podría hacer que todas las soluciones pareciesen idénticas. Esto no es lo ideal. Una vez que todos los *tours* en la población fuesen iguales, el AG no sería capaz de encontrar una solución mejor. Existen dos maneras de evitar que esto suceda. La primera es usar una población inicial muy grande. Esto causaría que tomase mucho tiempo el hacer que todos los individuos en ella llegasen a ser iguales. La segunda manera de evitarlo es usando un método de mutación, donde algunos *tours* hijos son aleatoriamente alterados para producir un nuevo *tour* único.

Este Algoritmo Genético genera la población inicial con un algoritmo voraz. Los enlaces entre ciudades en el tour inicial no son completamente aleatorios. El GA preferirá hacer enlaces entre ciudades que estén más próximas unas de otras. Esto no se hace así en el 100% de los casos, ya que causaría que todo *tour* inicial fuese prácticamente idéntico y lo que se busca es variedad en dicha población.

Existen 7 parámetros para controlar el funcionamiento del Algoritmo Genético implementado:

Tamaño de la Población (*Population Size*) – El tamaño de la población es el número inicial de tours que son creados cuando el algoritmo comienza. Una población grande necesita más tiempo para encontrar un resultado, pero nos asegura mucha diversidad. Una más pequeña incrementa la posibilidad de que todo *tour* en la población pudiese parecer el mismo, aumentando el riesgo de no encontrar una solución mejor en una nueva iteración, ya que la diversidad sería muy baja.

· **Tamaño del grupo (*Group Size*)** – En cada generación, este número de *tours* es escogido aleatoriamente entre la población. Los dos mejores serán los padres. Los dos peores serán reemplazados por los hijos. Un valor alto incrementará la probabilidad de que los *tours* realmente buenos sean elegidos como padres, pera también provocará que muchos otros *tours* nunca sean escogidos para serlo. Además, un valor elevado provocará que el algoritmo avance más rápido, pero difícilmente encontrará la solución óptima, o una que se le aproxime más.

% Mutación - El porcentaje en que cada hijo, después de un *crossover*, se someterá a una mutación. Cuando un *tour* es mutado, una de las ciudades es movida aleatoriamente desde un punto a otro dentro del tour.

Número de Ciudades Cercanas (*#Nearby Cities*) – Al crear la población inicial, el AG preferirá enlazar ciudades que sean cercanas unas de las otras para crear los *tours* iniciales. Cuando se crea la población inicial, este es el número de ciudades que se considerá como cercanas para cada vértice.

% Probabilidad de Ciudad Cercana (*Nearby City Odds %*) - Este es el porcentaje de probabilidad en que cualquier enlace en un *tour* aleatorio inicial preferirá usar una ciudad cercana en lugar de otra totalmente escogida al azar. Si el AG eligiese usar una ciudad cercana, entonces elegirá aleatoriamente una de las ciudades de entre las más cercanas. El número de ciudades cercanas vendrá determinado por el parámetro anterior.

Generaciones Máximas – Número de *crossovers* ejecutados antes que el algoritmo finalice.

Semilla Aleatoria (*Random Seed*) – Esta es la semilla para el generador de números aleatorios. Teniendo un valor fijo, en lugar de uno aleatorio, se pueden replicar resultados previos siempre que los otros parámetros sigan siendo los mismos. Esto es muy útil cuando se buscan errores en el algoritmo.

Los valores de los parámetros iniciales por defecto son los siguientes:

Parámetro	Valor Inicial
Tamaño de la población	10000
Group Size	5
% Mutación	3 %
#Nearby Cities	5
Nearby City Odds	90 %

Tabla 4.4: Valores por defecto para los parámetros del AG.

A continuación se muestra el diagrama de clases implementado con los atributos, propiedades y métodos más importantes para cada una.

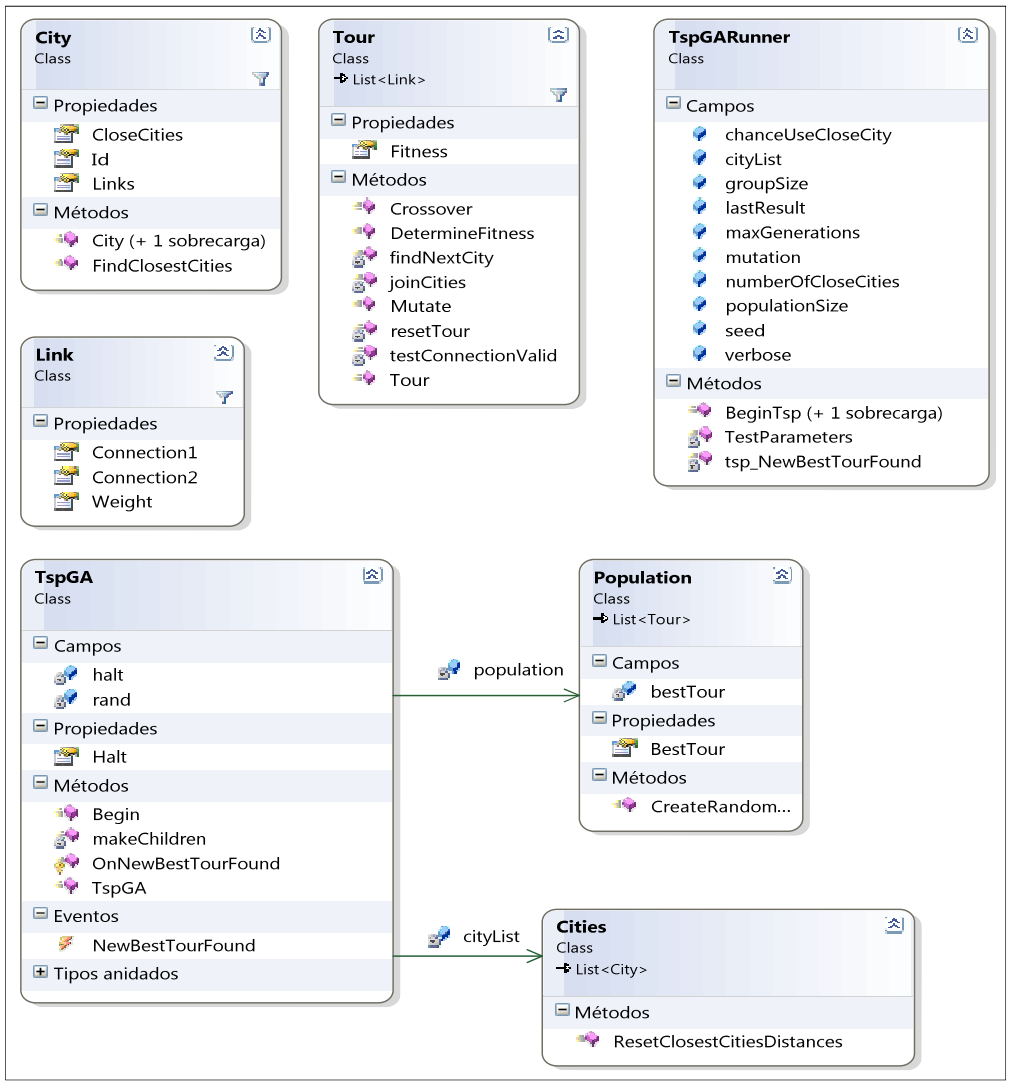

Figura 4.6: Diagrama de clases de la parte del AG de la aplicación.

4.3 – MEZCLA DE RAMIFICACIÓN Y PODA CON ALGORITMOS GENÉTICOS

Como se ha descrito anteriormente, el problema del 'Viajante de Comercio' (TSP), es la generalización que necesitamos para representar el nuestro, relativo a la optimización de rutas de vuelo.

Hemos explicado el funcionamiento de los métodos de *Ramificación y Poda* junto con las distintas cotas optimistas y maneras de ramificar implementadas y testeadas durante la realización del proyecto. Además, también se ha descrito el uso que hacemos de la técnica de los *Algoritmos Genéticos* para también tratar de encontrar una solución muy próxima a la óptima, llegando a serlo en algunos casos, para el problema del TSP.

Decir que la técnica de RyP nos asegura, cuando ha finalizado su ejecución, que ha encontrado una solución óptima, en caso que existiese. Como inconveniente, el tiempo en encontrarla puede resultar en la práctica excesivo o inviable. Este tiempo viene determinado por el espacio de búsqueda (grafo resultante de los *waypoints* por los que queremos pasar, junto con las distancias existentes entre todos ellos) y, sobretodo, su número de nodos.

Por otro lado, GA no asegura la consecución de esta solución óptima, aunque con un número adecuado de generaciones funcionando, y en un tiempo mucho más que razonable, se puede llegar a una solución aceptable para los casos que abarca este proyecto.

Llegados a este punto, y teniendo en cuenta que lo que más nos interesa es obtener la ruta que minimiza los kilómetros recorridos por el avión, pasando por todos los *waypoints* una sola vez, excepto por el primero, donde lo haremos dos veces (aeropuerto de salida y llegada), y deseando alcanzar esta solución en el menor tiempo posible, hemos llegado a la siguiente conclusión:

Mezclando las dos técnicas (Ramificación y Poda + Algoritmos Genéticos), podemos llegar a una solución óptima en un tiempo menor que si lo hiciésemos utilizando únicamente la primera.

Recordemos que cuando hablamos de solución óptima, nos referimos a la mejor solución posible, o una de ellas, de todo el conjunto de soluciones.

Para ello, en primer lugar deberíamos generar una solución inicial para el problema del *TSP* con el algoritmo genético. Una vez hecho esto, la introduciríamos como cota pesimista en el algoritmo de RyP. Es decir, podríamos afirmar que en el peor de los casos ya tenemos ese posible estado final con su respectivo valor de puntuación, y que

si encontrásemos otro estado donde no fuera posible mejorar ese valor podríamos descartarlo, es decir, podarlo.

Puesto que la solución generada por el AG estaría cerca de la mejor posible, el proceso de poda empezaría a descartar mucho más rápido soluciones peores. De este modo, en la práctica se reduce considerablemente el espacio de búsqueda, y por lo tanto, se alcanza en un tiempo más razonable una solución mejor que la del AG, si existiese.

El proceso paso a paso y en pseudo-código, sería el siguiente:

```
// Instancia del grafo que utilizaremos
Graph g = Nodos y aristas de un caso particular;

// Obtenemos una solución a través del AG
Algoritmo_Genetico GA = new Algoritmo_Genetico();
Resultado          r1 = GA.getSolution(g);

// Utilizamos la solución anterior como cota pesimista
// en el RyP y lo ejecutamos
Algoritmo_RyP     RyP = new Algoritmo_RyP();
RyP.BestSolutionFound = r1;
RyP.Start();

// Cuando finalice, r2 será la mejor solución encontrada
Resultado r2 = RyP.BestSolution;
```

Como se puede apreciar, la idea a grandes rasgos es sencilla y en la práctica ofrece los mejores resultados que hemos podido obtener, si lo que queremos es alcanzar la mejor solución de entre todas las posibles.

Si por el contrario, una solución sub-óptima nos bastase, o si debido al elevado número de nodos en el grafo, aún aplicando esta técnica, supiésemos que el tiempo de ejecución será intratable o no acorde a nuestras necesidades (si necesitamos una respuesta rápida, y la posibilidad de que tardásemos horas en obtenerla fuese elevada), lo más lógico e indicado sería utilizar solamente el AG.

Más adelante se detallan exactamente cuáles han sido los resultados en gasto de recursos temporales y espaciales al ejecutar, tanto la variante de AG+RyP, como el AG por separado, sobre un conjunto de pruebas (benchmarking). Además, se especificará una medida de lejanía donde se ha quedado de la solución óptima el algoritmo genético por separado.

4.4 - RESULTADOS OBTENIDOS CON LOS ALGORITMOS

4.4.1 - Introducción

En las siguientes secciones se detallará cómo se ha generado el conjunto de datos de pruebas.

Además se especificarán cuáles han sido los resultados obtenidos sobre el mismo, tanto para la estrategia de *'Ramificación y Poda'* como para el *'Algoritmo Genético'*.

Finalmente se extraerán unas breves conclusiones que pueden marcar el uso que hagamos de la aplicación implementada finalmente.

Las especificaciones del ordenador donde se ha realizado el *benchmarking* de los algoritmos se muestra en la siguiente tabla:

Procesador	**Intel® Core™ 2 Duo CPU E8400 @ 3.00 GHz**
RAM	**4.00 GB**
Tipo de Sistema	**32 bits**
Sistema Operativo	**Windows 7 Professional**
.Net Framework	**4.0**

Tabla 4.5: Especificaciones del ordenador usado para el *benchmarking*.

4.4.2 - Resultados obtenidos con Ramificación y Poda.

Para conseguir una medida fiable y robusta del tiempo que consume, tendremos diferentes grafos para cada número de vértices que vamos a testear.

Estos grafos han sido generados aleatoriamente gracias a una función implementada en la clase `Graph` de nuestro propio código, llamada *`CreateRandomTestSet`* y que toma los siguientes parámetros:

Tipo	Nombre	Descripción
`String`	*folder*	Directorio donde guardar los archivos que contienen los grafos.
`int`	*minNbNodes*	Mínimo número de vértices
`int`	*maxNbNodes*	Máximo número de vértices.
`int`	*repetitions*	Número de grafos a crear en cada iteración entre [min-max]

Tabla 4.6: Parámetros de la función `CreateRandomTestSet`, en la clase `Graph`

Llamaremos N al número de vértices que contiene un grafo. Así pues, si el parámetro `repetitions` toma valor 20, tendremos 20 grafos diferentes para cada N. El rango de N para nuestro conjunto de datos de prueba creado aleatoriamente es $N \in [10\text{-}25]$, es decir, `minNbNodes=10`, y `maxNbNodes=25`.

Una vez descrito cómo hemos obtenido el conjunto de datos de pruebas (también llamado *`testset`* en muchos otros campos de la informática), pasamos a explicar los resultados que sobre éstos se han conseguido con el algoritmo de Ramificación y Poda.

Tendremos en cuenta que estas pruebas se hacen sobre las primeras dos cotas implementadas. Recordemos que la Cota 1 se refería a la *'Compleción del camino con las aristas de mínimo coste que parten de vértices no visitados'*, mientras que la Cota 2 es la *'Cota basada en la matriz de Floyd-Warshall'*. La Cota 3 finalmente no se incluye en este *benchmarking* debido a su elevado coste computacional, tal y como se explicó anteriormente.

Además, recordando la sección *'Mezcla de RyP con AG'*, encontramos que podemos inicializar la cota superior del algoritmo de RyP utilizando un AG. Otra manera de hacer esta misma tarea sería utilizar una estrategia *voraz*. Esto significa que no conocemos la mejor solución en un principio, pero podemos intentar generar una buena solución utilizando una estrategia de coste polinómico. En nuestro caso, esta estrategia *'voraz'* será simplemente crear un camino escogiendo en cada iteración el nodo más cercano al que nos encontremos. Así pues, si empezamos en el v_1, escogeremos el vértice más próximo a éste formando el camino $(v_1, v_2, ?)$, y pasaremos a buscar el vértice más cercano a v_2, y así sucesivamente hasta cerrar el ciclo. Como se ha explicado, esta estrategia 'voraz' no encontrará la mejor solución posible. Pero sí nos proveerá de una cota superior con la que inmediatamente poder empezar a realizar la poda en la estrategia de RyP.

Así pues, teniendo en cuenta las dos diferentes cotas, y las dos diferentes formas de inicialización de la *Cota Superior*, tendremos las siguientes 4 combinaciones de parámetros a testear con RyP, mostradas en la tabla 4.7.

Tipo de Inicialización de la Cota Superior	Cota utilizada para ramificar estados
AG	Cota 1
AG	Cota 2
Voraz	Cota 1
Voraz	Cota 2

Tabla 4.7: Combinaciones de parámetros para el algoritmo de RyP

La figura 4.7 nos muestra una vista rápida con un gráfico de dispersión sobre todos los tiempos obtenidos en todas las ejecuciones del algoritmo en sus cuatro variantes, podemos ver que existen algunos valores que son demasiado elevados, mientras que la gran mayoría sigue una tendencia de lo que podríamos llamar una curva exponencial.

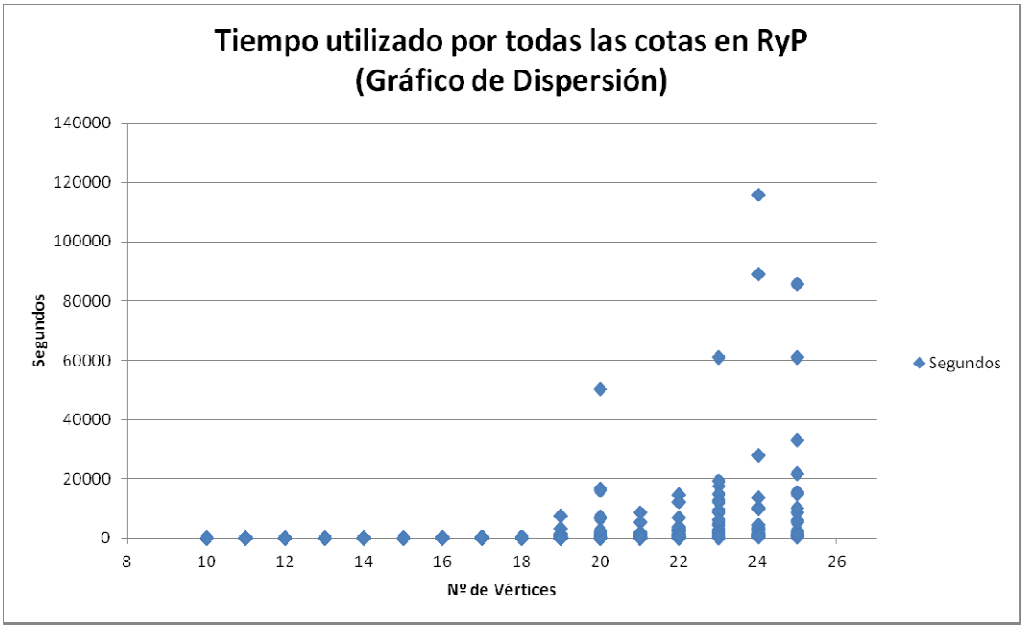

Figura 4.7: Gráfico de dispersión del tiempo utilizado por RyP en todo el `testset`.

Teniendo en cuenta que en algunos puntos concretos, el tiempo para alcanzar una solución óptima se dispara, para poder ver la tendencia real de las curvas definiremos un punto de corte sobre el que eliminar los puntos anómalos. Este punto de corte se definirá en los 40000 segundos. El número total de instancias eliminadas de nuestro

conjunto de resultados sólo es de 6, una ínfima parte de los datos teniendo en cuenta que el número total de tests realizados ha sido de 1037. Eliminar estos 6 registros nos permitirá obtener una estadística algo más fiable. De haberlos mantenido los números finales obtenidos podrían crecer excesivamente no dejándonos ver las tendencias normales que siguen los datos. Aún así hay que insistir en el hecho de que, RyP es muy dependiente de las instancias concretas del problema que esté tratando. Por lo tanto, puede darse el caso en que para un grafo concreto tengamos un tiempo desorbitado. Y también el contrario, donde la poda sea tremendamente efectiva y alcancemos una solución óptima en tiempo récord. Aún así, para la gran mayoría de grafos 'normales', los tiempos de ejecución han de seguir una tendencia determinada.

Una vez filtrados los datos, el gráfico anterior se puede ver de la siguiente manera:

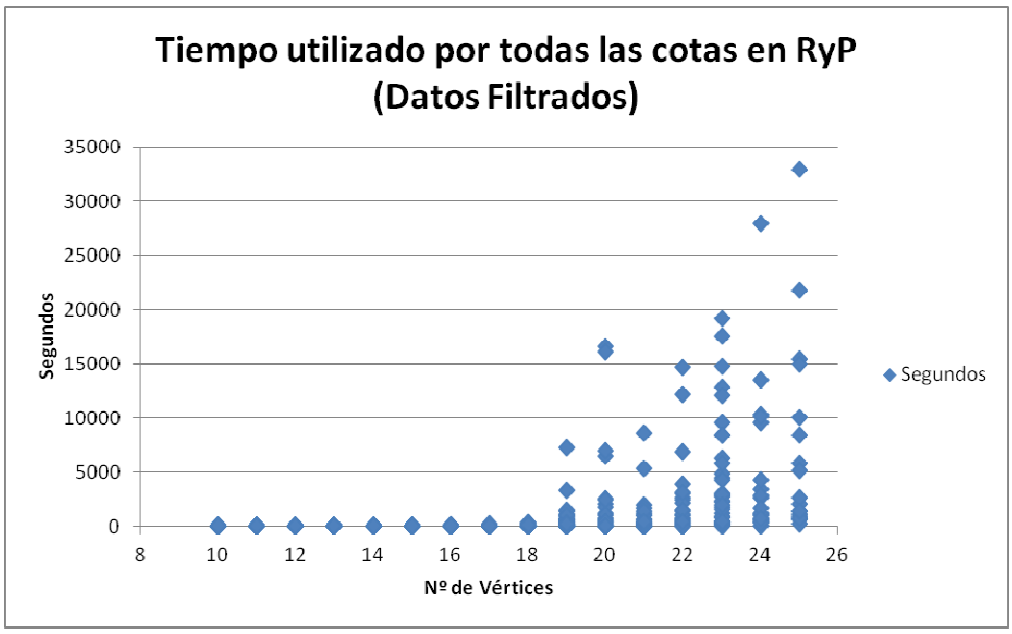

Figura 4.8: Gráfico de dispersión del tiempo utilizado por RyP en el `testset`, filtrado a partir de los 40000 segundos.

Ahora sí que se puede apreciar más la tendencia que siguen los datos sin tener valores que nos sesgarán innecesariamente la estadística.

Una vez hecho el pretratamiento de los resultados, pasamos a dividir los datos en los 4 grupos que hemos mencionado anteriormente, para distinguir entre las diferentes configuraciones con las que se han ejecutado los algoritmos de Ramificación y Poda. Hay que tener en cuenta que los siguientes gráficos mostrarán el tiempo medio de ejecución de cada configuración para cada número de nodos que haya en el grafo. Para poder apreciar mejor las diferencias, se han creado tres gráficos donde hay diferentes intervalos en el eje X.

El primer gráfico de esta serie muestra el intervalo de entre 10 y 20 vértices. Vemos como en instancias menores de 18 nodos el coste es prácticamente nulo. Es a partir de aquí donde el tiempo de ejecución utilizando la Cota 2 se empieza a disparar. Además, también se puede ver el buen comportamiento que presenta la Cota 1, sobre todo con la inicialización utilizando el AG.

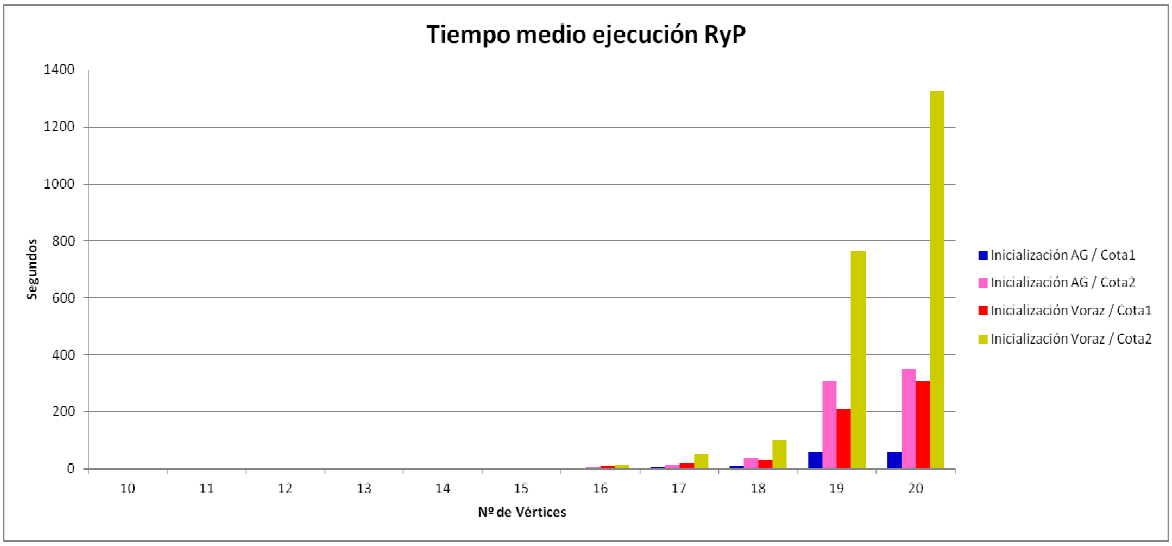

Figura 4.9: Tiempo medio de ejecución de RyP, para grafos de entre 10 y 20 vértices.

El gráfico de la figura 4.10 simplemente hace un zoom sobre el anterior en el intervalo de 10 a 16 vértices, para poder apreciar mejor los tiempos consumidos en cada configuración. Aunque se puede ver que los tiempos son prácticamente nulos, se mantiene el ranking de mejores y peores configuraciones que teníamos antes. La inicialización AG junto la cota 1 siempre es la mejor. Mientras que la peor es la inicialización voraz junto la cota 2.

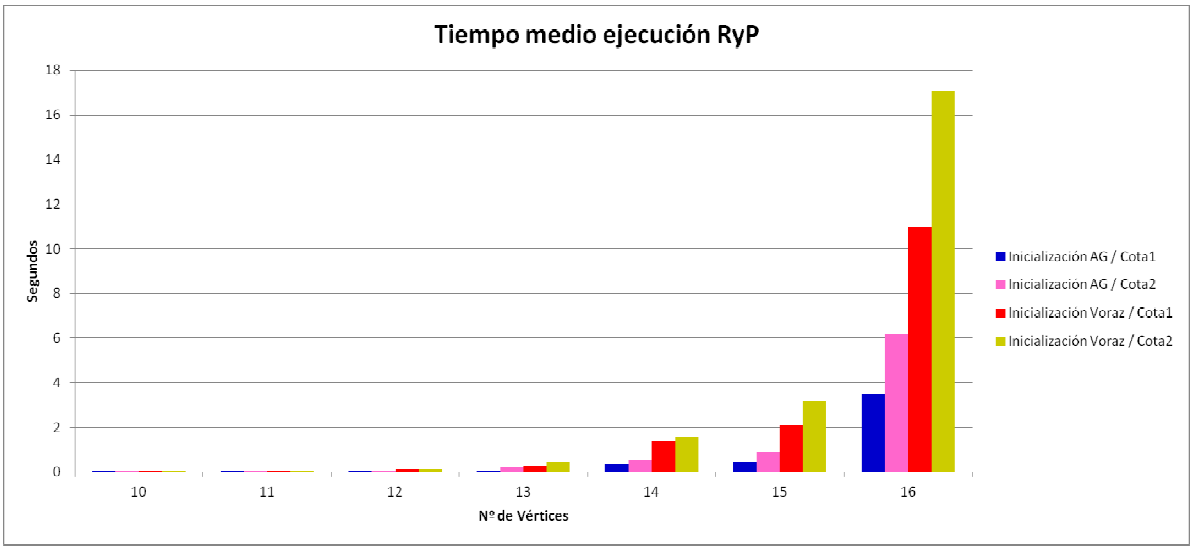

Figura 4.10: Tiempo medio de ejecución de RyP, para grafos de entre 10 y 16 vértices.

Sin embargo, también se puede apreciar como la cota 1 no es siempre mejor que la cota 2. En grafos con pocos vértices es más importante una buena inicialización que la cota que podamos estar utilizando.

Por último, pasamos a ver el intervalo de grafos de entre 21 y 25 vértices. Las configuraciones que utilizaban la cota 2 comenzaban a aumentar desmesuradamente su tiempo de ejecución. Ya que era inviable seguir calculando dichos tiempos, se decide que, a partir de este punto, dejaremos de utilizarlas en el *benchmarking*. Además, ha quedado suficientemente clara su tendencia, no siendo la mejor solución a utilizar.

Usando pues, solamente la cota 1, vemos como sigue siendo la estrategia que utiliza inicialización por AG la que mejores resultados sigue dando. Con grafos de 25 nodos nos sucedió al igual que con la cota 2 con la inicialización voraz, siendo inviable poder hacer las medidas de tiempo. Por eso en este gráfico sólo aparece la columna azul para 25 vértices en el grafo.

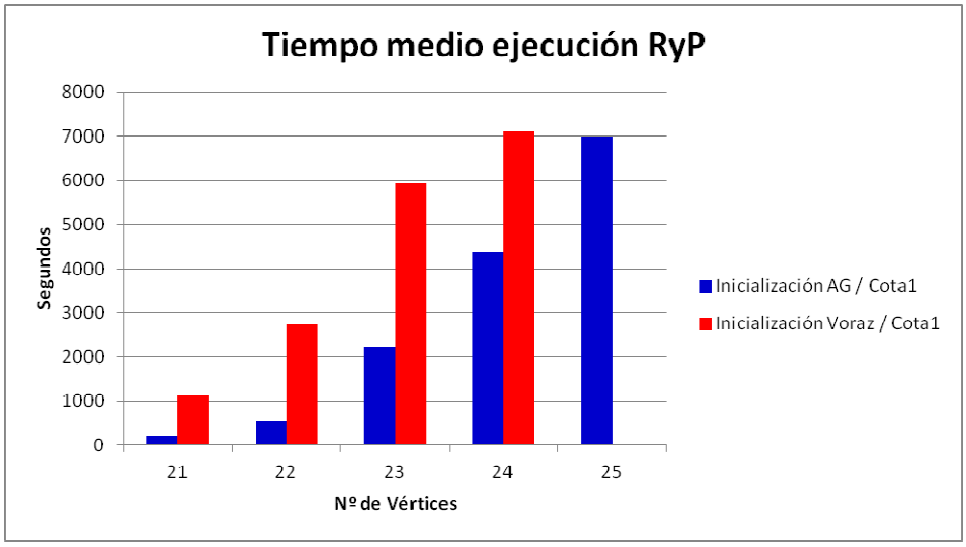

Figura 4.11: Tiempo medio de ejecución de RyP, para grafos de entre 21 y 25 vértices.

Está claro que si necesitamos procesar grafos con más de 15 nodos, el mejor resultado con la estrategia de Ramificación y Poda lo obtenemos inicializando la cota superior con el Algoritmo Genético, y ramificando y usando la Cota 1.

En la siguiente gráfica podremos ver la curva completa del tiempo medio empleado por esta configuración, siendo claramente, a partir de los 20 vértices, de $O(n^2)$. Aun siendo la mejor opción en cuanto a coste, este comportamiento no lineal de las ejecuciones limita la escalabilidad de la solución de RyP a grafos de 25-30 nodos.

Un dato curioso que podemos constatar para todas las configuraciones, es que todas empiezan teniendo un gasto de tiempo prácticamente nulo. Pero llegado a un determinado número de vértices (unas configuraciones antes, otras después), comienzan a presentar un coste exponencial, que en la práctica las hace inviables.

Se podría afirmar que, dependiendo de la cota utilizada, estamos retrasando o adelantando este punto, donde finalmente no será viable utilizar una estrategia de RyP. Por lo tanto, aún en el caso de hallar una cota 'inmejorable', que nos permitiera podar el máximo número de estados posible, al final, siempre encontraríamos un N a partir del cual nuestro tiempo de ejecución sería exponencial.

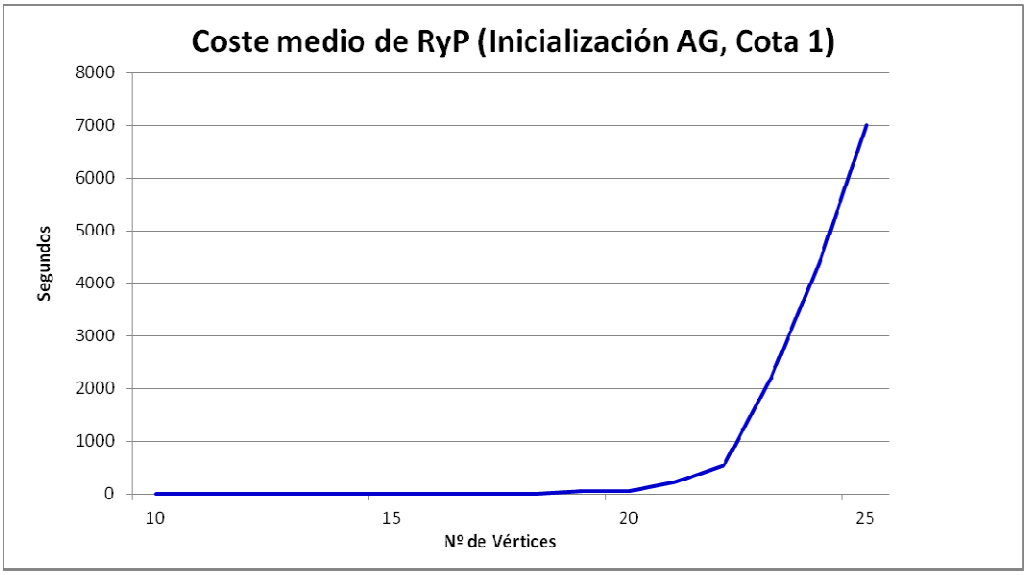

Figura 4.12: Curva de tiempo medio de ejecución de RyP (Inicialización AG, Cota 1).

A pesar de que Ramificación y Poda mejora notablemente el tiempo de ejecución sobre un algoritmo común de *backtracking*, 7000 segundos (casi 2 horas) de media sigue siendo un tiempo elevado para un problema de talla 25. Todo dependerá de la necesidad imperiosa que tengamos para que nuestra ruta sea la óptima, o de si nos es suficiente una solución aproximada.

Esta solución aproximada puede ser hallada usando el AG implementado. En la siguiente sección se pueden ver los resultados obtenidos utilizando solamente esta técnica.

4.4.3 - Resultados obtenidos con el Algoritmo Genético

Como se ha indicado anteriormente, el AG por sí sólo no nos asegura la obtención de una solución óptima, pero sí una aproximación aceptable.

Puesto que el tiempo de ejecución total se ve reducido considerablemente con respecto a 'Ramificación y Poda', en esta parte hemos podido incluir grafos de hasta 60 vértices en el conjunto de pruebas. Al igual que antes, para conseguir una medida fiable y robusta del tiempo que consume, tendremos diferentes grafos para cada número de vértices que vamos a testear. Llamaremos N al número de vértices que contiene un grafo. Así pues, por motivos de limitación del tiempo de cálculo total, tendremos 20 grafos diferentes para los $N \leq 30$, y 5 grafos diferentes para los $N > 30$. El rango de N para nuestro conjunto de datos de prueba creado aleatoriamente es $N \in [10\text{-}60]$. Por supuesto, los archivos de datos donde $N \in [10\text{-}25]$ son exactamente los mismos que en RyP para más adelante poder hacer una comparación de los resultados obtenidos.

En este conjunto de pruebas, se han mantenido los valores de los parámetros por defecto que se indicaron para el AG, variando simplemente el número de generaciones utilizadas. Además, hay que destacar que la semilla aleatoria también es diferente a cada ejecución del AG para no repetir la generación de poblaciones idénticas para un mismo grafo, una vez ejecutemos el algoritmo con otro número de generaciones. Por lo tanto, el valor del mejor *tour* encontrado puede ser diferente para un mismo grafo en las distintas ejecuciones efectuadas. En la teoría, a más generaciones, mejor individuo final hemos de tener como resultado. Pero esto no ha de ser necesariamente siempre cierto, ya que al cambiar la semilla aleatoria podemos acabar en un punto muy distinto de nuestra función de optimización.

Parámetro	Valor Inicial
Tamaño de la población	10000
Group Size	5
% Mutación	3 %
#Nearby Cities	5
Nearby City Odds	90 %
Nº Generaciones	**[inicio = 300000; fin = 1000000; incremento = 100000]**

Tabla 4.8: Valores utilizados en el *benchmarking* para los parámetros del AG.

En el gráfico de la figura 4.13 se puede apreciar el tiempo de ejecución que se ha necesitado dependiendo del número de vértices en el grafo. En el eje Y se ven

representados los segundos que ha tardado de media, para todo el conjunto de pruebas que le correspondía. Además, se ha creado una serie para cada valor del número de generaciones probado.

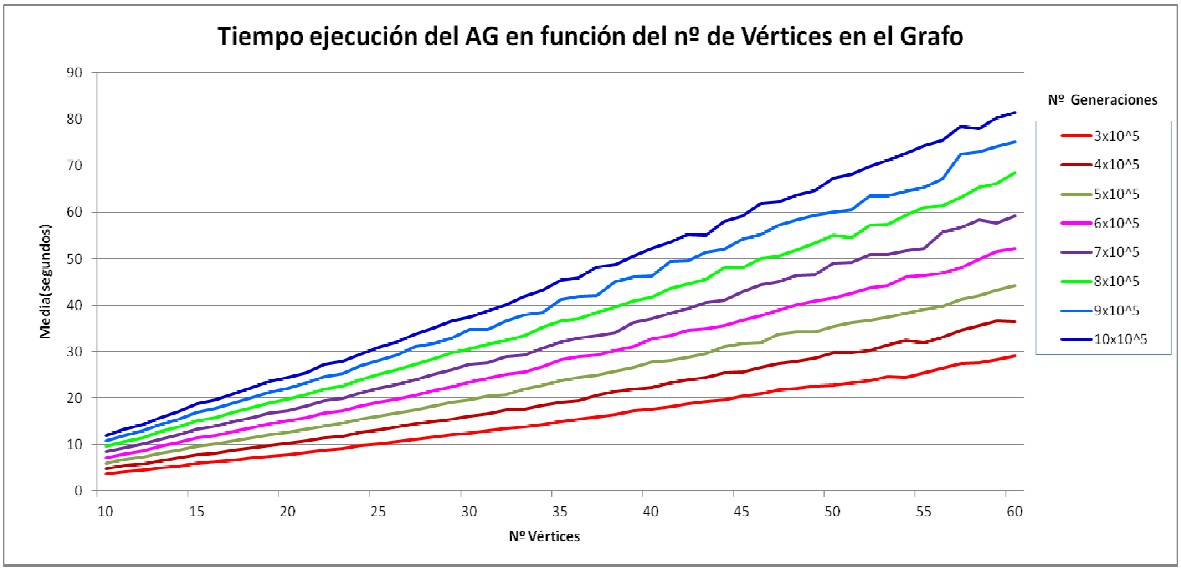

Figura 4.13: Tiempo de ejecución del AG en función del Nº de Vértices y de Generaciones

Se puede apreciar claramente como el tiempo consumido responde a una función de $O(n)$, es decir, es linear. Esto es una gran ventaja respecto al algoritmo de Ramificación y Poda, donde teníamos una función de coste exponencial $O(n^2)$. Por ejemplo, en RyP el tiempo medio para grafos con 25 vértices era de 7002 segundos, mientras que con AG es de 11 segundos.

También se puede ver como la pendiente de cada serie es función del número de generaciones. Es decir, dada la ecuación de la recta $y = mx + n$, donde m marca esta pendiente, contra más generaciones ejecute nuestro AG, mayor será el valor de m.

Un dato interesante es que con un grafo de 60 nodos, tenemos una media de tiempo de ejecución de 30 segundos con 300000 generaciones, y de 82 segundos con 1 millón. Estos tiempos eran impensables con RyP para tal cantidad de nodos. De hecho, el máximo número de vértices que se ha logrado medir con RyP ha sido 25.

Ahora podemos pasar a analizar la calidad de las soluciones generadas con el AG. Puesto que conocemos la solución óptima para las instancias de los grafos con N ϵ [10-25] gracias a las ejecuciones con el algoritmo de RyP, podemos definir la siguiente fórmula:

$$\%\, Aproximación = \frac{avg(best\_solution\_ga_{nb\_generations})}{best\_solution\_ryp} \times 100$$

Donde obtenemos el porcentaje medio de aproximación del Algoritmo Genético para un número determinado de generaciones, respecto de la solución óptima. Es decir, el valor máximo que podremos obtener con esta fórmula será 100, el cuál indicará que hemos alcanzado la solución óptima (un 100% de aproximación a la solución ideal).

Como se puede apreciar en la figura 4.14, el número de generaciones no es determinante en nuestro caso para obtener mejores o peores soluciones. Esto es debido a que el mínimo que nos hemos impuesto, 300000, es ya de por sí suficientemente elevado para producir buenas soluciones con los grafos que estamos tratando. Lo que sí se puede observar es que para grafos con menos de 15 vértices obtenemos la mayoría de las veces la solución óptima. Mientras que si vamos aumentando este número de vértices, poco a poco nos vamos alejando de dicha solución. Aún así, con 25 nodos, en el peor de los casos, estaremos un 5% alejados de la mejor solución posible, resultado que en muchos casos nos puede ser suficiente teniendo en cuenta el poco tiempo necesario para obtenerlo. En el caso que nos ocupa de los vuelos de reconocimiento, la aplicación de GA supondría un coste adicional de un 5% en tiempo de vuelo respecto al óptimo, por ejemplo un vuelo que podría haberse realizado en 2 horas, nos costaría unos 6 minutos más.

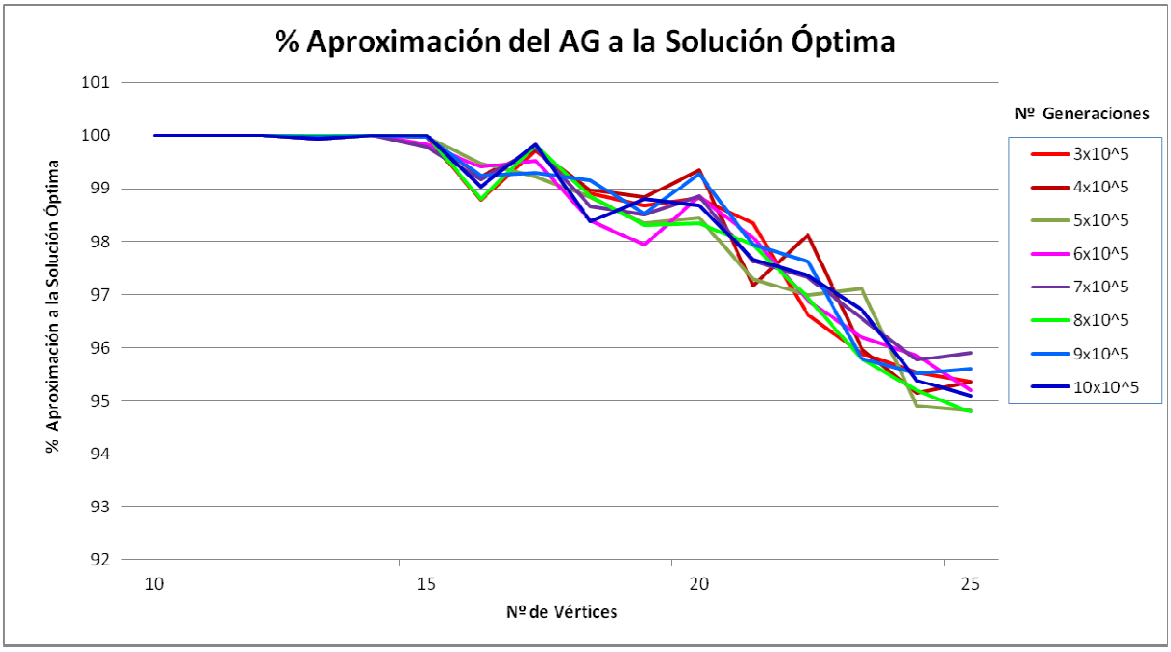

Figura 4.14: % de Aproximación medio del Algoritmo Genético a la solución óptima.

4.4.4 - Conclusiones basadas en los resultados obtenidos

A la vista de las gráficas obtenidas, tanto en la parte de RyP, con en la de AG, se podría afirmar que:

1 Para instancias de grafos con menos de 16 vértices, sería conveniente usar la estrategia de Ramificación y Poda con inicialización AG y cota 1. Esto nos permitirá hallar la solución óptima en el menor tiempo posible, siendo éste bastante bajo.

2 Para instancias de grafos con más de 20 vértices, sería conveniente utilizar la técnica del AG por separado. Aunque no se nos asegure conseguir la mejor solución posible, sí que obtendremos una bastante aceptable. Si utilizásemos RyP en estos casos, el coste de tiempo sería demasiado elevado, y pasados los 25 vértices, en muchos casos podríamos llegar a tardar días o meses en encontrar una solución.

3 Para instancias de grafos de entre 16 y 20 vértices, podremos elegir entre una de las dos soluciones anteriores, dependiendo del compromiso que tengamos con alcanzar o no una solución óptima. Utilizar la primera opción podría suponer en algunos casos usar algunas horas de cálculo. Pero bien puede valer la pena dejar una noche al ordenador trabajando, para que un avión al día siguiente ahorre el máximo combustible y tiempo posible.

CAPÍTULO 5 - INTERFAZ GRÁFICA DE USUARIO DEL PROGRAMA

Aunque la aplicación inicial en principio estaba pensada como una librería de clases para que otros programadores pudieran utilizarla, al terminar la implementación de todas las funcionalidades pedidas en la especificación de requisitos inicial, se opta por crear dos formularios de tipo *Windows Form*, para aprovechar directamente dicha implementación por parte de un usuario final y no sólo por los programadores que hicieran uso de la API.

El modo de empleo de los formularios se puede dividir en las siguientes fases:

1- Carga de los datos desde el servidor web o directamente desde un archivo ya descargado.

2- Filtrado de datos. Selección de los puntos del mapa por donde queremos que pase el avión en base a distintos criterios.

3- Creación del Plan de Vuelo (Flight Plan). Selección de los posibles algoritmos a aplicar junto a sus parámetros y fichero de salida.

Cuando iniciamos la aplicación, lo primero que podremos visualizar es un pequeño formulario en el que podremos seleccionar con qué tipo de datos queremos trabajar, ya sean los *'hotspots'* o los *'fires'*.

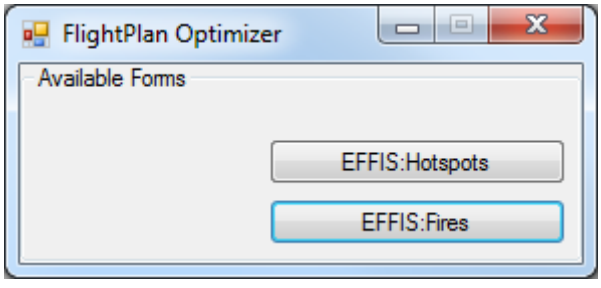

Figura 5.1: Ventana inicial de la aplicación.

A continuación se explica con más detalle la forma de uso de estos formularios en lo que podría ser el manual de usuario básico del programa.

5.1 - IGU del programa - Formulario 'EFFIS:Hotspots'

Cuando en el formulario inicial hacemos *click* sobre el botón *'EFFIS:Hotspots'* nos aparecerá maximizada la ventana de la figura 5.2.

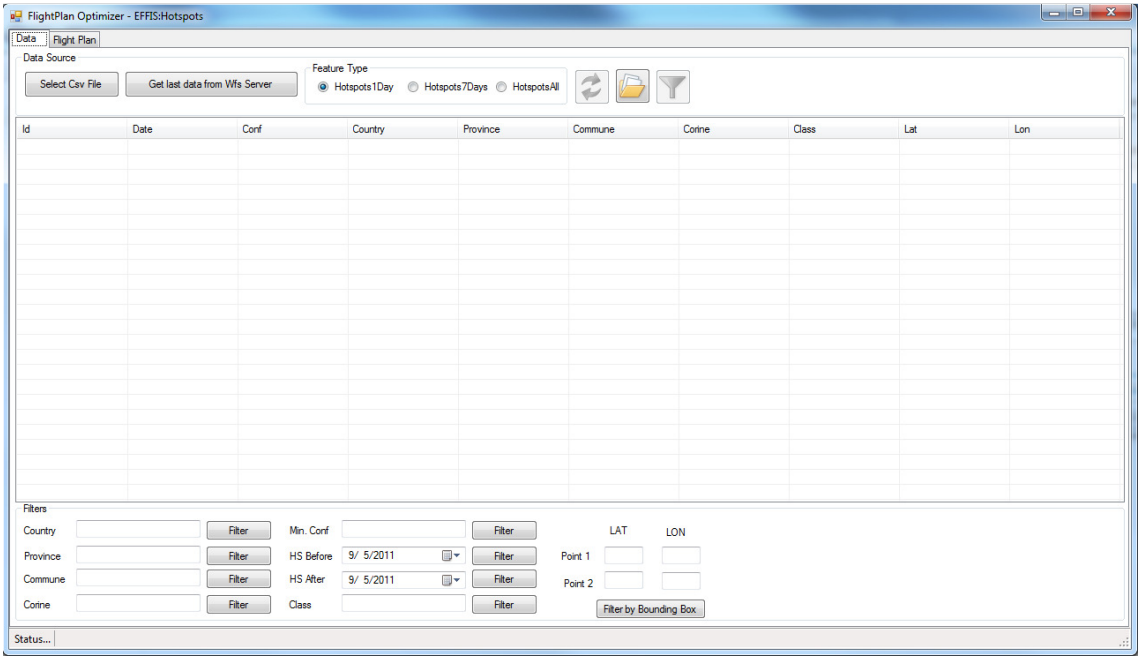

Figura 5.2: Formulario 'EFFIS:Hotspots' antes de la carga de datos.

En la parte superior se encuentran los controles para trabajar con el 'Data Source' o fuente de datos que utilizaremos.

En primer lugar seleccionaremos qué tipos de 'hotspots' queremos obtener del servidor WFS o que se encuentran en un fichero CSV en nuestro propio ordenador. Para ello, escogeremos una de las tres opciones del grupo 'Feature Type', que son las siguientes:

- **Hotspots1Day** Hotspots del MODIS en las últimas 24 horas.
- **Hotspots7Days** Hotspots del MODIS en los últimos 7 días.
- **HotspotsAll** Hotspots del MODIS desde el 01-01-2010

Una vez seleccionado el tipo de datos con el que queremos trabajar, tendremos que cargar los datos. Para ello haremos click sobre uno de los dos botones siguientes:

- **Get last data from Wfs Server**
  Descarga del servidor Wfs los datos con los que queremos trabajar en formato CSV y los deja en nuestro ordenador en un fichero. La ruta por defecto en nuestro equipo donde dejará estos archivos será *[user_documents_folder]/EFFIS_data*. Una vez descargados, lee el fichero y muestra la información que contiene en el formulario.

- **Select Csv File**

  Muestra un cuadro de diálogo para seleccionar un archivo de nuestro ordenador, que será el origen de datos que utilizaremos. Puesto que utilizando el botón anterior podemos obtener estos ficheros, esta función nos facilitará trabajar con datos que ya hubiésemos guardado anteriormente o de fechas distintas a la actual.

  Una vez seleccionado el archivo con la información correspondiente, mostrará su información al igual que antes. Nótese que el tipo de datos que contenga el fichero deberá coincidir con el seleccionado en el 'Feature Type'. De otro modo, nos aparecerá un mensaje de alerta indicándonoslo y no cargándose ningún 'hotspot'.

Una vez cargados los datos, nuestro formulario tendrá el aspecto del de la figura 5.3.

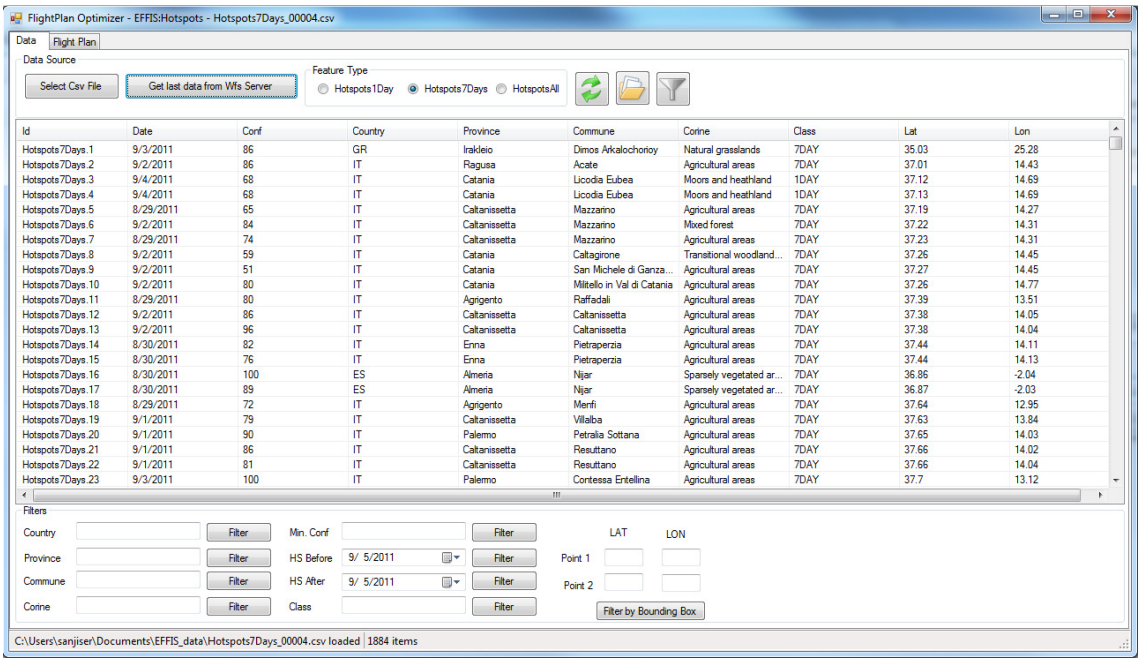

Figura 5.3: Formulario 'EFFIS:Hotspots' después de cargar los datos de los últimos 7 días (05/09/2011).

Además, los botones que aparecen a la izquierda del 'Feature Type' se activarán para poder efectuar las siguientes operaciones:

- **Refresh data from current CSV file** 

  Como se ha indicado anteriormente, tanto si descargamos los datos del servidor WFS, como si es el mismo usuario el que selecciona un fichero, nuestra fuente de datos real final es un archivo CSV. Si en algún momento nos hemos equivocado al aplicar un filtro sobre estos datos, o si queremos trabajar más adelante con otro subconjunto de ellos, pulsando este botón recargaremos la información del último fichero con el que estábamos trabajando.

- **Change downloads directory** 

  Cuando descargamos un fichero de datos del WFS, la ruta por defecto del fichero de destino es *[user_documents_folder]/EFFIS_data*. Pulsando este botón podemos cambiar el directorio de destino por el seleccionado en el explorador de directorios que nos aparecerá.

- **Filter selected 'Hotspots'** 

  En la lista donde nos aparecen los Hotspots con los que estamos trabajando, podemos realizar una selección de uno o varios de ellos utilizando las teclas CTRL y SHIFT tal y como se hace en un explorador de archivos de Windows. Pulsando este botón, filtramos los elementos seleccionados, es decir, los borramos de la lista.

Una vez cargados todos los elementos (hotspots) en el formulario, en la parte baja de éste encontramos la sección de Filtros, donde podemos eliminar de la lista algunos de estos ítems acorde al criterio, o combinación de criterios, que necesitemos.

Los filtros de datos disponibles para los *'hotspots'* y su significado son los siguientes:

- **Country:** Elimina de la lista los *waypoints* cuya abreviatura de País sea distinta a la indicada en el cuadro de texto.

- **Province:** Elimina de la lista los *waypoints* cuya Provincia sea distinta a la indicada en el cuadro de texto.

- **Commune:** Elimina de la lista los *waypoints* cuya 'Comunidad' o 'Región' sea distinta a la indicada en el cuadro de texto.

- **Corine:** Elimina de la lista los *waypoints* cuyo tipo de terreno sea distinto al indicado en el cuadro de texto.

- **Min. Conf:** (*Minimun Confidence*) Elimina de la lista los *waypoints* cuyos valores de 'confianza' sean menores al indicado en el cuadro de texto. El campo *conf* indica el porcentaje (valores entre 0 y 100) de certeza de que realmente existe un fuego en dicho punto, ya que los *Hotspots* se obtienen a partir de las imágenes del *MODIS* y existe la posibilidad donde el tratamiento utilizado para identificarlo hubiera fallado. Este error de identificación es conocido por el nombre de '*falso positivo*': indica un incendio donde realmente no lo hay.

- **HS Before:** Elimina de la lista los *waypoints* cuya fecha sea anterior a la indicada en el selector de fechas. Es decir, nos quedamos con los hotspots con fecha anterior o igual a la especificada.

- **HS After:** Elimina de la lista los *waypoints* cuya fecha sea posterior a la indicada en el selector de fechas. Es decir, nos quedamos con los hotspots con fecha posterior o igual a la especificada.

- **Class:** Elimina de la lista los *waypoints* cuya clase sea distinta a la indicada en el cuadro de texto. Los tres tipos existentes para *'hotspots'* son *1DAY*, *7DAY* y *ALL*. Hay que tener en cuenta que si descargamos los datos de tipo *Hotspots1Day*, el WFS omite esta columna en la información ofrecida, y por lo tanto aparecerá en blanco en nuestra GUI.

- **Bounding Box:** Elimina de la lista los *waypoints* cuyas coordenadas (longitud y latitud) no se encuentren dentro del rectángulo que formarían los dos puntos especificados por el usuario en un mapa.

Hay que destacar que los filtros son sensibles al uso de mayúsculas y minúsculas, es decir, no es lo mismo indicar en el filtro Country 'ES' que 'es'.
Además, para el buen funcionamiento de los filtros sobre las fechas, en la configuración regional del ordenador, el formato de fecha corta a utilizar será *dd/MM/yyyy*, y como símbolo de separación de decimales se ha de utilizar el punto *(.).*

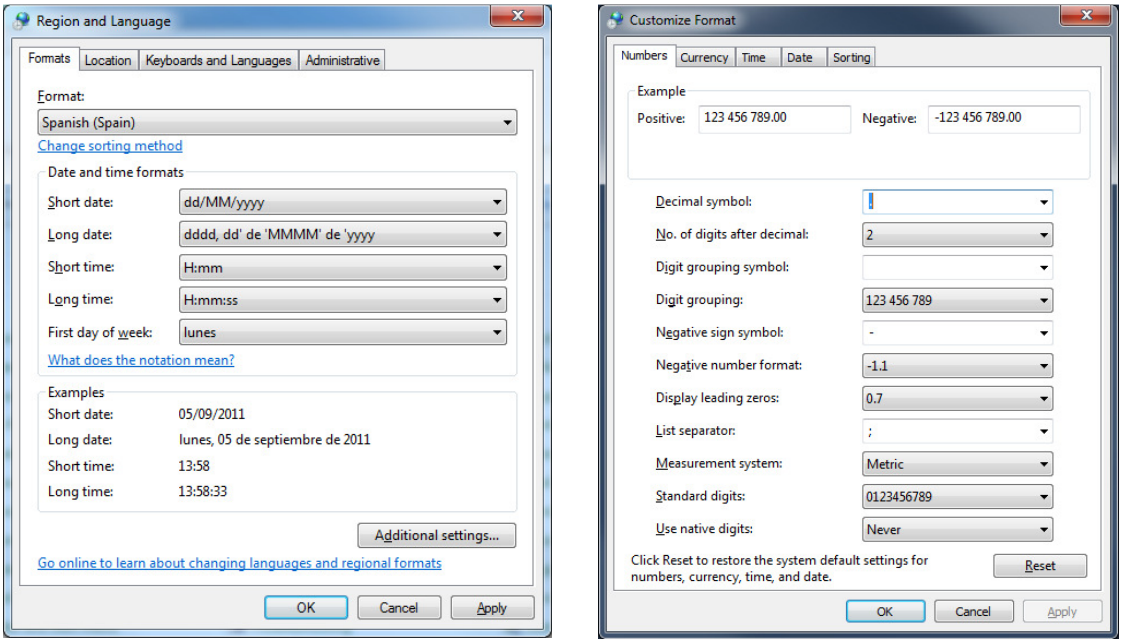

Figura 5.4: Configuración regional del ordenador que use el programa.

Suponiendo que sólo necesitásemos los *Hotspots* de los últimos 7 días vistos por el *MODIS* en España, en el campo de filtros '*Country*' introduciríamos 'ES', es decir, el código del país, y una vez pulsado el botón '*Filter*', en la lista de ítems sólo aparecerán elementos que cumplan dicho criterio. En la figura 5.5 aparece un ejemplo del uso de esta funcionalidad.

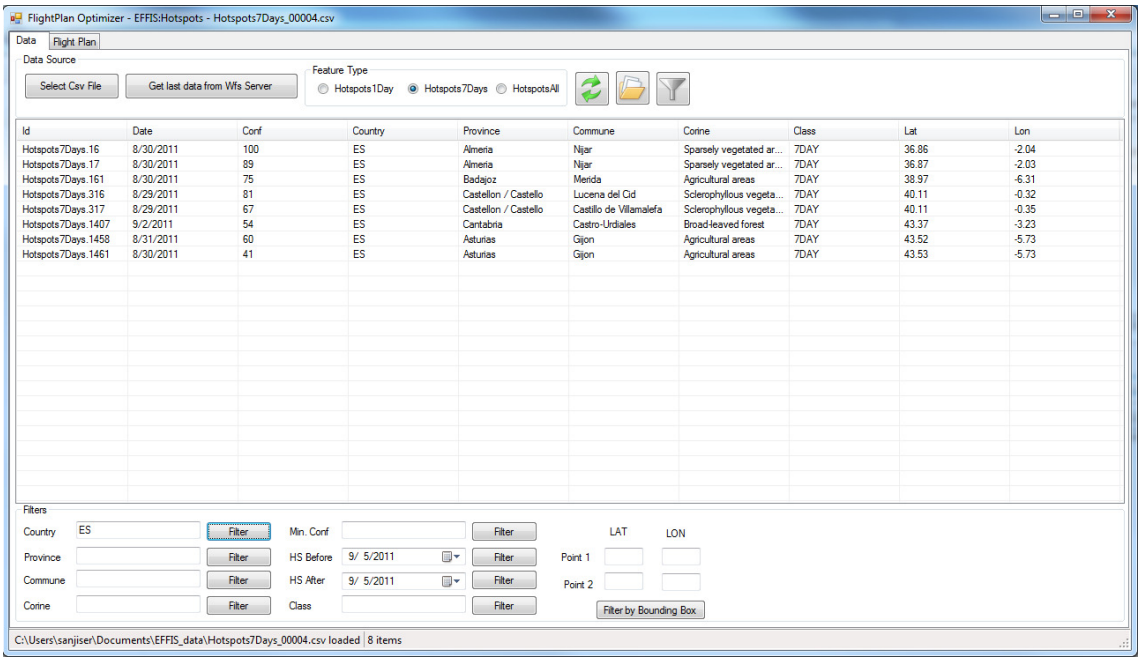

Figura 5.5: Formulario 'EFFIS:Hotspots' después de filtrar los datos para obtener los de España.

Una vez filtrados los puntos por los que queremos que pase el avión, pasaremos a la pestaña 'Flight Plan', donde aplicaremos sobre estos nodos y sus distancias el algoritmo que seleccione el usuario, generando el Plan de Vuelo en el fichero de salida indicado también por éste.

En la figura 5.6 nos aparece el aspecto que tiene esta pestaña. Además de los puntos anteriores, se ha de añadir a nuestro grafo el nodo que representa el aeropuerto de salida y llegada del avión. Para ello, el usuario podrá seleccionar uno del desplegable, o pulsar 'Set Closest' para que automáticamente, se seleccione el aeropuerto más cercano a cualquiera de los '*hotspots*' por donde ha de pasar el avión. Esto se ha conseguido introduciendo en el código las coordenadas geográficas de cada aeropuerto seleccionable en dicho desplegable.

Cuando hayamos seleccionado el aeropuerto, será necesario indicar qué algoritmo queremos aplicar para la resolución del camino óptimo que pasa por todos los nodos. En el desplegable tendremos dos opciones:

- Branch And Bound (Ramificación y Poda)
- Genetic Algorithm (Algoritmo Genético)

Anteriormente se han descrito cuáles son las ventajas e inconvenientes de utilizar cada uno, que en este implementación concreta se podría resumir en lo siguiente: En caso de tener 20 o menos *hotspots* por los que pasar, utilizar el *Branch and Bound*, ya que obtendremos una solución óptima. Para más *hotspots*, utilizar el *algoritmo genético*, que nos dará una buena solución en un tiempo razonable, sin asegurarnos que ésta sea la óptima.

Una vez hayamos seleccionado en el desplegable qué tipo de algoritmo vamos a utilizar, en el editor de propiedades nos aparecerán los parámetros que podemos ajustar acorde a nuestras necesidades, junto con una descripción de estos.

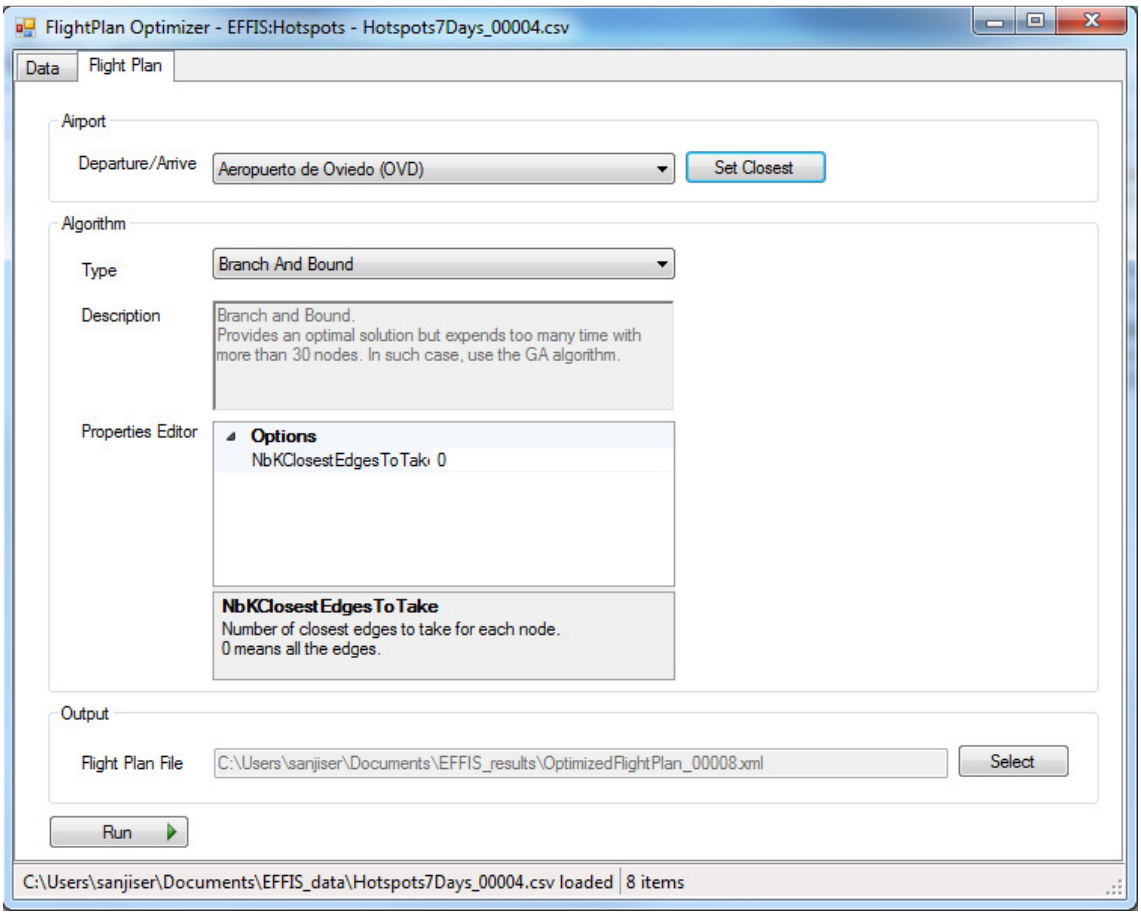

Figura 5.6: Formulario 'EFFIS:Hotspots', pestaña Flight Plan.

Los parámetros modificables por el usuario sobre el algoritmo de ramificación y poda son los siguientes:

- **NbKClosestEdgesToTake**: Número de aristas más cercanas a cada waypoint a tener en cuenta. Este parámetro nos limita el grafo resultante que obtenemos al utilizar los waypoints filtrados en la pestaña *Data*. Si su valor se deja en 0, significa que no se altera dicho grafo, es decir, se tienen en cuenta todas las distancias existentes desde un nodo (*waypoint*) a todos los demás. En caso de

ser distinto de 0, se hace un pretratamiento del grafo, donde se eliminan las aristas más lejanas. Por ejemplo, dado un grafo de 20 nodos y poniendo el valor de este parámetro a 10, cada nodo sólo tendrá en cuenta las 10 aristas más cercanas a él, y no las 19 que en principio tendría. En la práctica, este pre-proceso reduce considerablemente el tiempo de ejecución, aunque en contrapartida puede producir resultados extraños para determinadas instancias del problema. Por ejemplo, en las que hay un elevado número de waypoints concentrados en una superficie determinada. Además, no asegura la optimalidad (encontrar la mejor solución posible) del algoritmo. Un buen uso de este parámetro sería dejar siempre un valor igual o superior al 50% del número de nodos del grafo. Más adelante se hace un análisis sobre cómo varían los resultados dependiendo del número de nodos y número de aristas empleadas al resolver el problema.

Los parámetros modificables por el usuario sobre el algoritmo genético son los siguientes:

- **MaxGenerations**: Número de ciclos principales (*crossover*, mutación y selección) o generaciones que realizará el algoritmo. Para tallas pequeñas del problema (menos de 15 waypoints), bastará con usar 300000 generaciones para obtener un buen resultado. Para tallas medianas (entre 15 y 40 waypoints) y asegurarnos obtener una buena solución es recomendable usar 1000000 de generaciones, y para tallas grandes (> 40 waypoints) se puede aumentar al doble del anterior, aunque dependiendo de la instancia concreta del problema el resultado puede variar en cuanto a calidad. Más adelante se hace un análisis exhaustivo sobre cómo varían los resultados dependiendo del número de nodos y generaciones empleadas en resolver el problema.

En las opciones del grupo *output* se indica el archivo XML donde se guardará el Plan de Vuelo resultante. Por defecto, una vez se inicie la aplicación, ésta escogerá un fichero que tenga por nombre el siguiente formato y que actualmente no exista en nuestro sistema de archivos:

[user_documents_folder]/EFFIS_results/OptimizedFlightPlan_[xxxxx].xml

Donde *[xxxxx]* sera un número entero y *[user_documents_folder]* el directorio de los documentos del usuario en el sistema, por defecto. Este path podrá ser modificado por el usuario pulsando en el botón ‘Select’ que aparece justo al lado. Si hacemos esto, nos aparecerá un cuadro de diálogo para seleccionar el lugar y el nombre donde guardar la salida resultante.

Una vez efectuados todos los pasos anteriores, ahora sólo falta ejecutar el algoritmo y esperar. La figura 5.7 muestra el aspecto de la aplicación durante esta ejecución, donde los controles aparecen en estado desactivado y podemos ver una barra de progreso. No obstante, esta barra de momento no contiene funcionalidad para indicar cuanto falta para que termine la ejecución, pero sí nos da una idea de que el programa está trabajando.

Cabe destacar también que en la barra de estado del formulario podremos ver los distintos mensajes del estado en qué se encuentra el programa, además del número de elementos que hay en la lista de *hotspots*, parámetro que nos indicará la complejidad que tendrá la instancia de nuestro problema.

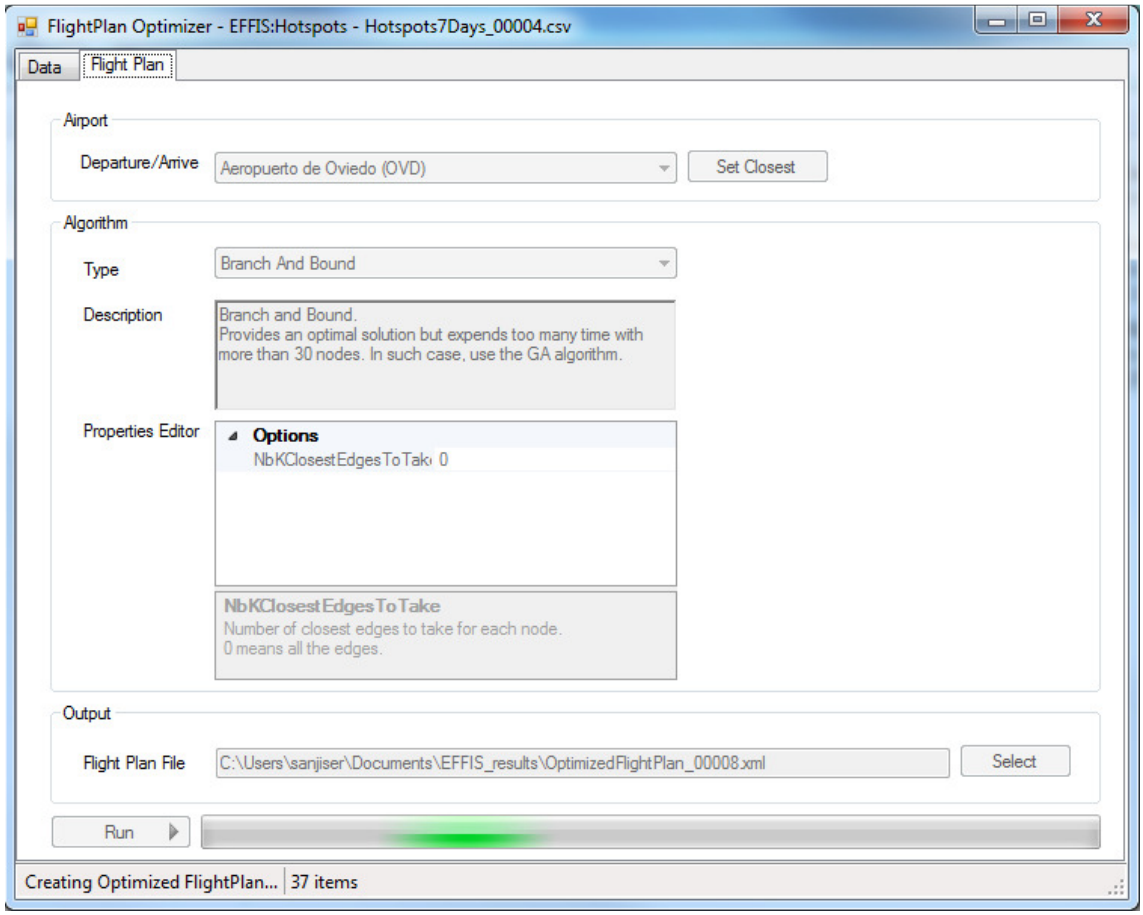

Figura 5.7: Formulario 'EFFIS:Hotspots', pestaña Fligh Plan durante la ejecución.

Una vez se haya generado el archivo de salida, para visualizar su contenido tendremos que usar el *'Flight Plan Monitor'* de la plataforma ISIS. Hay que tener en cuenta que esta es sólo una de las posibles aplicaciones que tiene dicho archivo XML, ya que, como se ha descrito anteriormente, dentro de dicha plataforma se podría utilizar el archivo para simular un vuelo e incluso hacer volar un aparato real. Sin embargo, de momento el uso principal que se le dará será su visualización en el mapa para que un

piloto, en este caso, *humano,* es decir, no automático, sepa por donde ha de hacer pasar el avión.

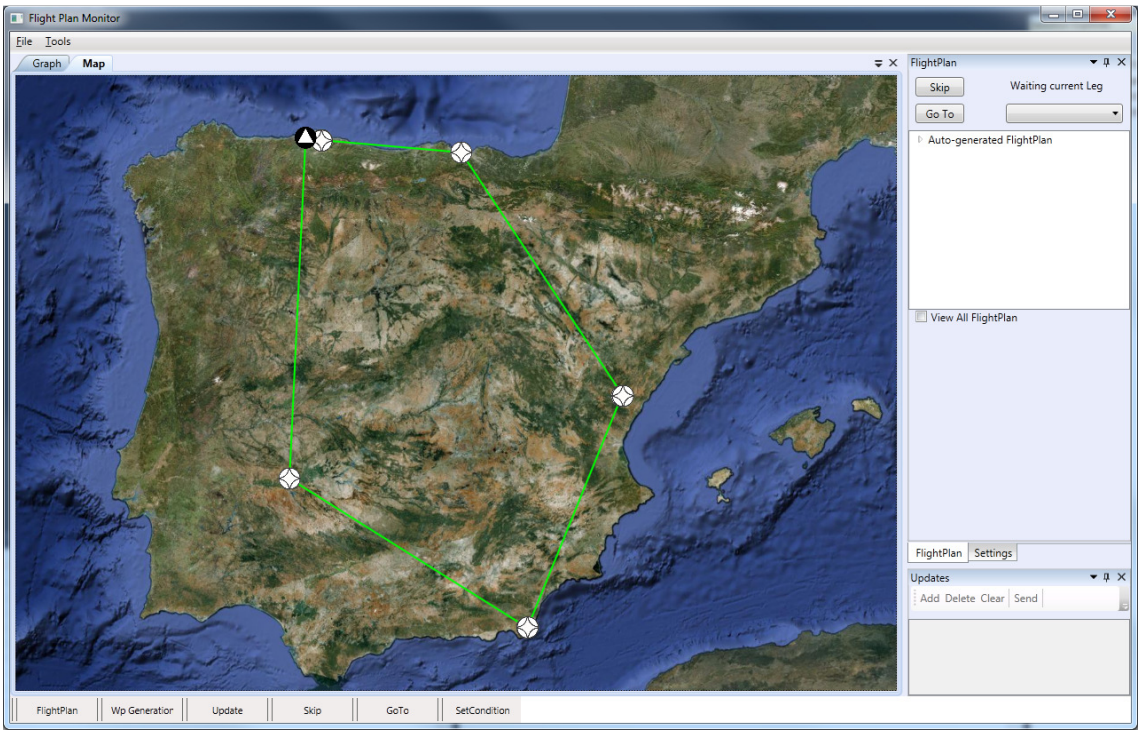

Figura 5.8: Visualización del FlighPlan resultante sobre los datos anteriores de 'Hotspots' en el Flight Plan Monitor de la plataforma ISIS.

La figura 5.8 muestra la ruta mínima generada con los 'hotspots' filtrados en las figuras anteriores. Hay que tener en cuenta que, en este mapa a primera vista sólo vemos cinco nodos por los que pasaría el avión, pero contiene 8, donde algunos se superponen en el mapa debido a su cercanía y simplemente tendríamos que hacer un zoom para poder distinguirlos.

Otra cosa a tener en cuenta es que este plan de vuelo no es muy realista, ya que nuestro avión en principio recorrerá sólo una o dos provincias de una comunidad autónoma, y no España de punta a punta. Pero este ejemplo nos sirve como figura ilustrativa de lo que podremos hacer cuando encontremos más puntos por los que pasar en una o dos regiones cercanas.

5.2 - IGU del programa - Formulario 'EFFIS:Fires'

Cuando en el formulario inicial hacemos *click* sobre el botón *'EFFIS:Fires'* nos aparecerá maximizada la ventana de la figura 5.9 sin datos cargados todavía.

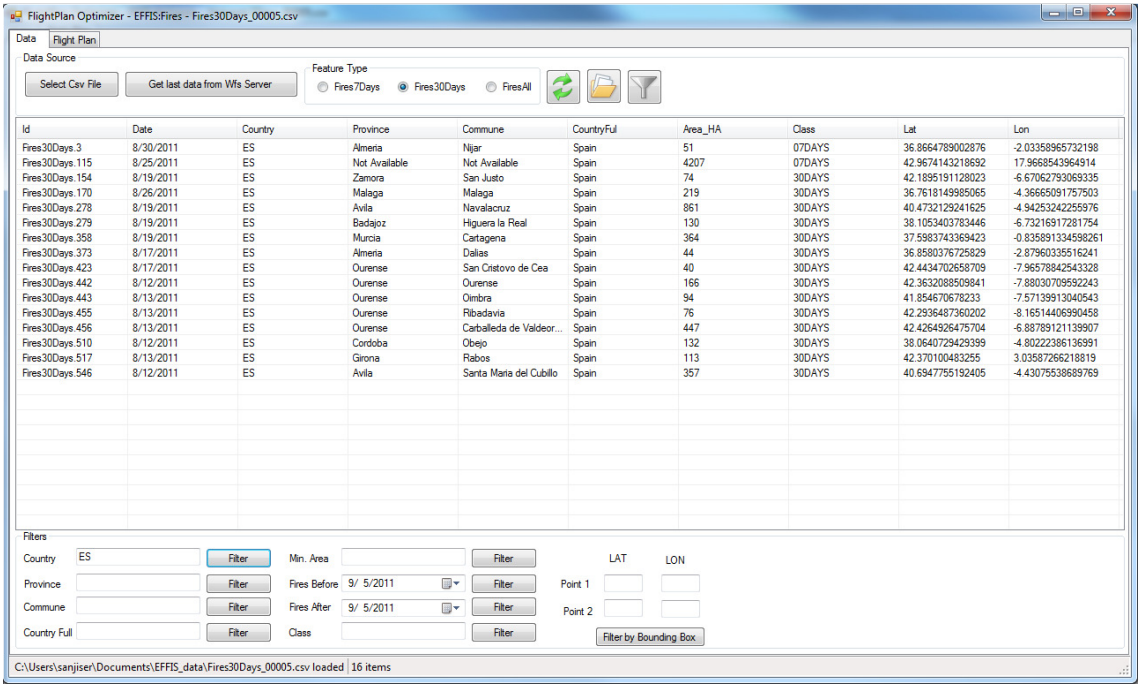

Figura 5.9: Formulario 'EFFIS:Fires' después de cargar los datos de los últimos 30 días (05/09/2011) y haber filtrado sólo los datos del territorio español.

El funcionamiento de este formulario es idéntico al visto anteriormente con Hotspots, salvo por los tipos de datos con los que trabaja, y por lo tanto, algunos filtros también variarán ligeramente.

Los tipos de *'fire'* (*Feature Type*) que podremos descargar o utilizar a través de un fichero en nuestro ordenador serán los siguientes:

- **Fires30Days** Fuegos en los últimos 30 días.
- **Fires7Days** Fuegos en los últimos 7 días.
- **FiresAll** Fuegos desde el 01-01-2010.

Los filtros de datos disponibles para los *'fires'* y su significado son los siguientes:

- **Country:** Elimina de la lista los *waypoints* cuya abreviatura de País sea distinta a la indicada en el cuadro de texto.

- **Province:** Elimina de la lista los *waypoints* cuya Provincia sea distinta a la indicada en el cuadro de texto.

- **Commune:** Elimina de la lista los *waypoints* cuya 'Comunidad' o 'Región' sea distinta a la indicada en el cuadro de texto.

- **Country Full:** Elimina de la lista los *waypoints* cuya nombre completo de País sea distinto al indicado en el cuadro de texto.

- **Min. Area:** Elimina de la lista los *waypoints* cuyos valores de área incendiada en hectáreas (Area_HA) sean menores al indicado en el cuadro de texto.

- **Fires Before:** Elimina de la lista los *waypoints* cuya fecha sea anterior a la indicada en el selector de fechas. Es decir, nos quedamos con los *fires* con fecha anterior o igual a la especificada.

- **Fires After:** Elimina de la lista los *waypoints* cuya fecha sea posterior a la indicada en el selector de fechas. Es decir, nos quedamos con los *fires* con fecha posterior o igual a la especificada.

- **Class:** Elimina de la lista los *waypoints* cuya clase sea distinta a la indicada en el cuadro de texto. Los tres tipos existentes para *'fires'* son *07DAYS*, *30DAYS* y *ALL*. Hay que tener en cuenta que si descargamos los datos de tipo Fires7Days, el WFS omite esta columna en la información ofrecida, y por lo tanto aparecerá en blanco en nuestra GUI.

- **Bounding Box:** Elimina de la lista los *waypoints* cuyas coordenadas (longitud y latitud) no se encuentren dentro del rectángulo que formarían los dos puntos especificados por el usuario en un mapa.

Hay que tener en cuenta que los filtros son sensibles al uso de mayúsculas y minúsculas, tal y como ocurría en el formulario de *'Hotspots'*. El formato de fecha corta a utilizar sigue siendo *dd/MM/yyyy*, y el símbolo de separación de decimales el punto *(.).*

Para mostrar el comportamiento que presenta el algoritmo genético con los datos filtrados tal y como aparecen en la figura 5.9, donde encontramos 16 nodos, se lanza una ejecución con 1000000 de generaciones tal y como se muestra en la figura 5.10.

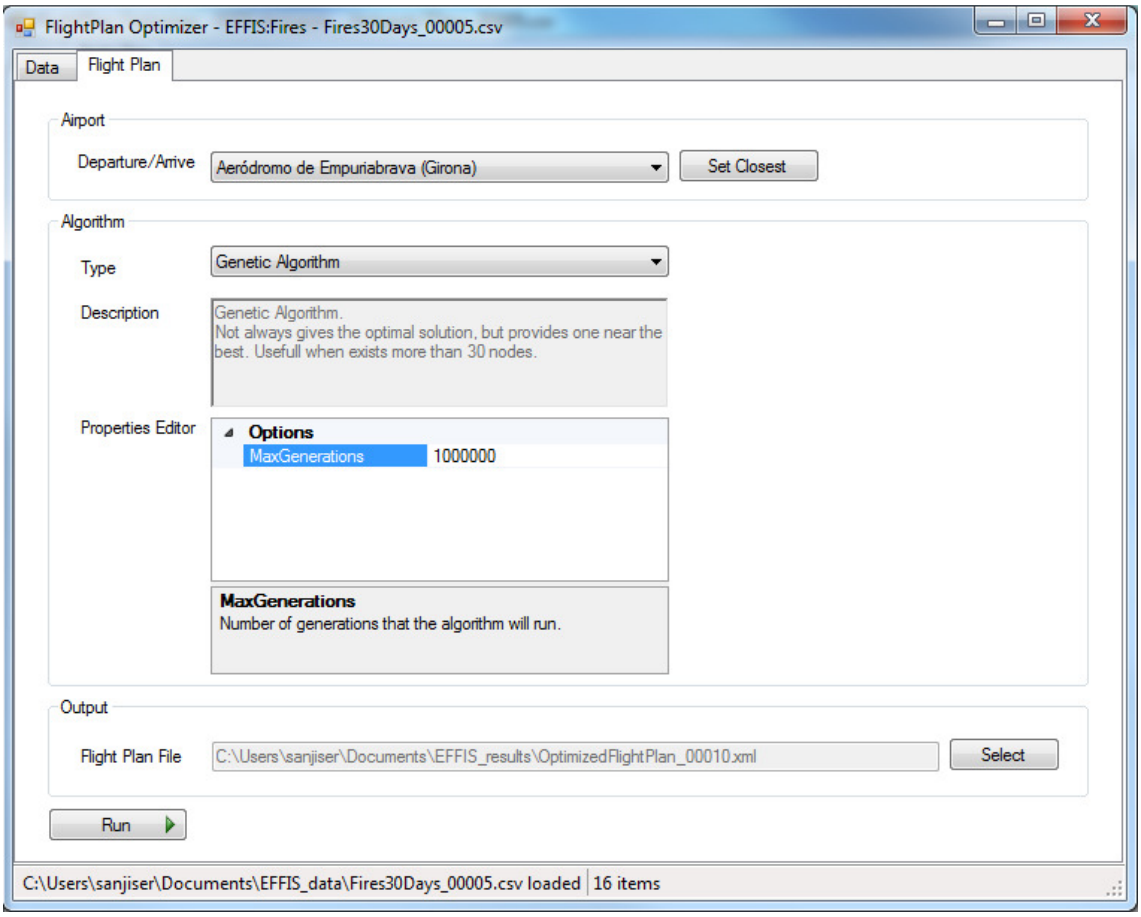

Figura 5.10: Formulario 'EFFIS:Fires', tab Flight Plan, utilizando el Algoritmo Genético.

El Plan de Vuelo resultante lo podemos ver en la figura 5.11, donde se puede apreciar que ha encontrado una ruta más que razonable para el conjunto de 'fires' anterior.

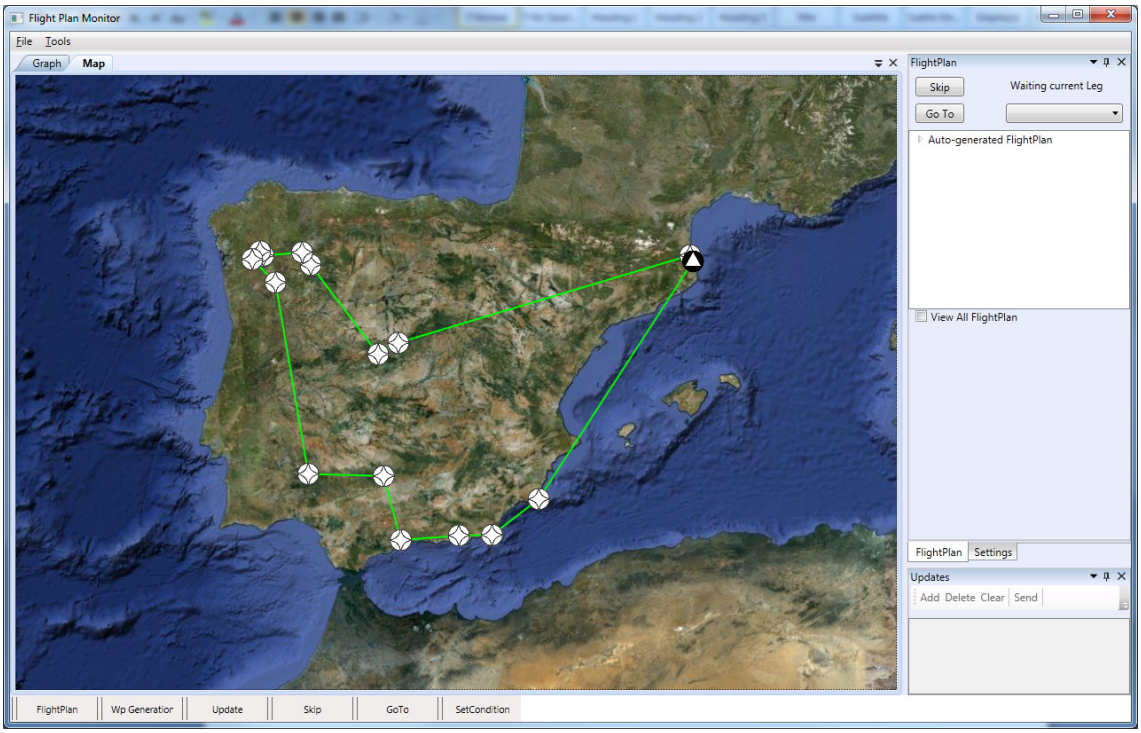

Figura 5.11: Visualización del FlighPlan resultante sobre los datos anteriores de '*Fires*' en el Flight Plan Monitor de la plataforma ISIS

CAPÍTULO 6 – PLANIFICACIÓN Y ARQUITECTURA DEL SOFTWARE DESARROLLADO

El presente capítulo muestra cuál ha sido la planificación seguida durante el desarrollo del software y la arquitectura resultante. En todo caso, el orden y las explicaciones mostradas en los capítulos anteriores muestran también el orden cronológico y de implementación seguido. En este capítulo se pretende resumirlo para tener una visión general y esquemática de las tareas realizadas.

6.1 Especificación de requisitos

Creación de una API, y de una interfaz de usuario que nos permita realizar las siguientes tareas:

1. Obtener la información en tiempo real que nos ofrece el EFFIS a través de su página web, relativa a *hotspots* (posibles fuegos detectados por un satélite y su sistema de tratamiento de imágenes MODIS), y a *fires* (fuegos ya confirmados).

2. Filtrado y selección de las coordenadas por los que queremos que pase un avión de extinción o prevención de incendios, y que se corresponderán con los *hotspots* o *fires* anteriores.

3. Generación de un plan de vuelo **óptimo** por el que habrá de pasar dicho avión. El estándar a utilizar para definir el plan de vuelo ha de ser el '*Flight Plan Specification Language'*. Este plan de vuelo ha de ser posible visualizarlo en el módulo '*Flight Plan Monitor'* de la plataforma ISIS.

6.2 Etapas en la creación de la aplicación

A la hora de implementar la aplicación se han dividido las tareas de la siguiente manera:

1 Creación de la capa de software que genere automáticamente un archivo XML siguiendo el estándar *Flight Plan Specification Language,* a partir del modelo de objetos. Recordar que ya existía un modelo de objetos implementado en la plataforma ISIS para representar dicho estándar. Lo que no existía era la parte

donde esos objetos se podían transformar en representaciones XML. Nuestra tarea ha sido crear todos los métodos que nos conviertan, de la forma más clara y sencilla para el programador, una instancia de tipo *FlightPlan* en un archivo XML. No sólo se ha añadido nuevas funciones a las clases ya existentes, sino que se han creado otras nuevas que también eran necesarias. Es decir, se aumenta la funcionalidad de la API existente.

2 Creación y test de los algoritmos que resuelven el problema del TSP. Puesto que el TSP es una abstracción de nuestro problema de optimización de rutas de vuelo, se implementan diferentes soluciones para resolver este problema. Las técnicas implementadas para esta tarea han sido 'Ramificación y Poda', con sus respectivas cotas, y una adaptación de los 'Algoritmos Genéticos'. Como se ha mostrado en el Capítulo 4, también se ha realizado un *benchmarking* y se han extraído unas conclusiones del rendimiento de cada una de las técnicas.

3 Implementación de la API para la obtención de los datos del servidor WFS. Esta etapa ha incluido la creación del modelo de objetos que representan estos datos, su descarga de la red y lectura desde el archivo descargado.

4 Implementar las capas de software que conectan las tres etapas anteriores entre ellas. Puesto que cada una de las tres etapas anteriores presenta un modelo de objetos distinto, es necesario proveer una API que las conecte. La figura 6.1 ilustra cuál es el funcionamiento de dicha API. Esta etapa vendría representada por los círculos verdes de la figura.

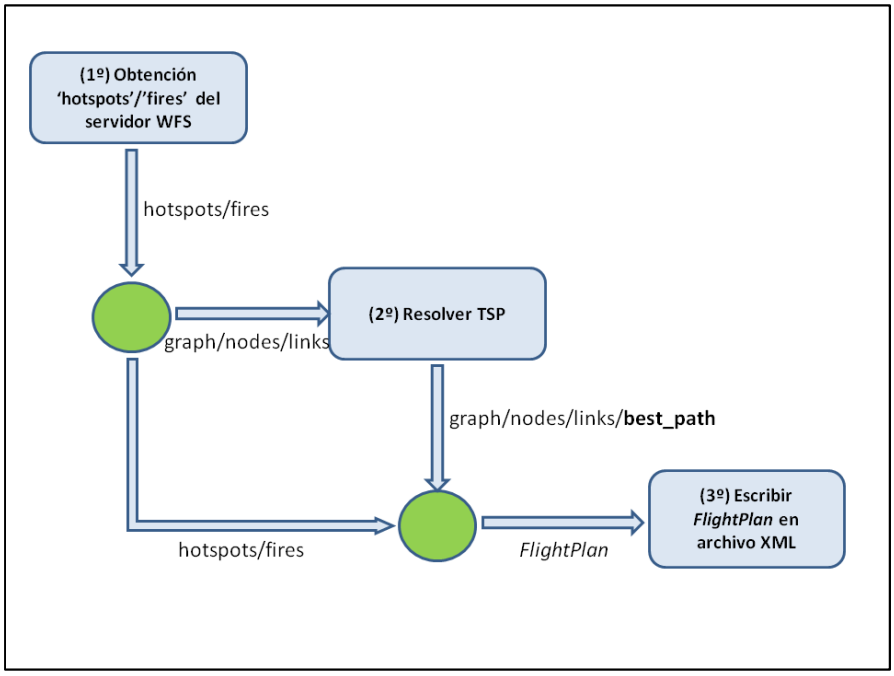

Figura 6.1: Conexión entre las distintas APIs implementadas.

5 Implementación de la Interfaz Gráfica del Usuario. Como se ha mostrado en el Capítulo 5, aprovechando toda la API generada en las 4 etapas anteriores, se crea una IGU para que un usuario final, y no sólo un programador, pueda hacer uso de todas las funcionalidades creadas. Este usuario podrá descargar del servidor WFS un conjunto de puntos de riesgo, ya sea de tipo '*Hotspot*' o '*Fire*', por los que haríamos sobrevolar el avión de prevención o extinción de incendios. El usuario podrá filtrar estos puntos acorde a sus criterios, seleccionar un aeropuerto de despegue y aterrizaje y, posteriormente, generar el archivo XML que represente el plan de vuelo óptimo a seguir.

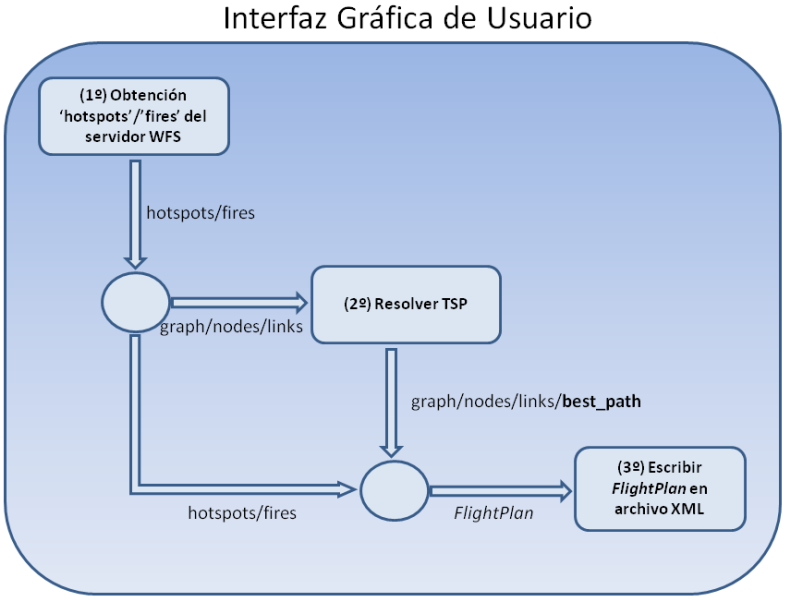

Figura 6.2: Esquema de encapsulamiento en la interfaz gráfica de las etapas [1-4].

6.3 Arquitectura de la aplicación

La estructura del programa está separada en tres capas, esto recibe el nombre de programación por capas. Este estilo de programación tiene como objetivo principal la separación de la capa de presentación, capa de negocio o dominio y capa de datos. En la figura 6.3 se puede observar las diferentes capas y su forma de interactuar entre ellas.

La ventaja principal de este tipo de arquitectura es que el desarrollo del programa se puede llevar a cabo en varios niveles, y en el caso que se quiera hacer un cambio solo habrá que modificar el nivel requerido. Además, este estilo de programación también nos permite avanzar en la programación del proyecto de una forma ordenada. Esto nos beneficia en una reducción en el tiempo necesario de implementación del

software, debido a que se podrá adelantar de forma más segura durante su desarrollo. Todo esto gracias a que la aplicación general está dividida en distintos módulos que pueden ser tratados de manera independiente e incluso paralela.

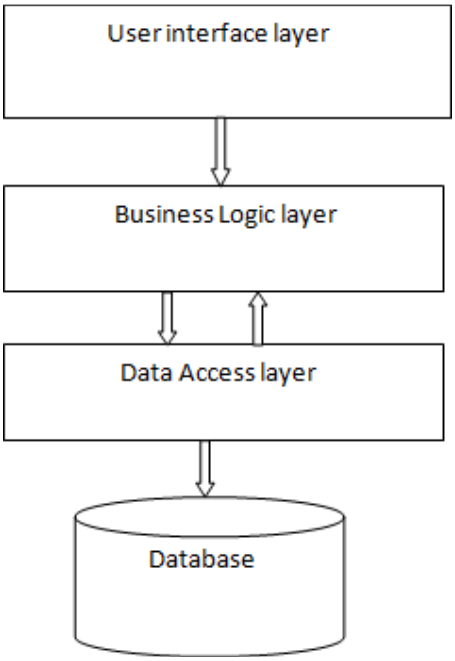

Figura 6.3: División de la arquitectura del programa en 3 capas.

• Capa de presentación: Es la capa que presenta el sistema al usuario, le comunica la información y captura la interacción entre ambos. La interfaz debe ser amigable y fácil de utilizar, ya que el usuario final es el que se va encargar de utilizar el sistema. Esta capa se comunica únicamente con la capa de negocio.

• Capa de negocio: Es la capa donde se encuentra toda la lógica del programa, así como las estructuras de datos y objetos encargados para la manipulación de los datos existentes, y el procesamiento de la información ingresada o solicitada por el usuario en la capa de presentación. Esta capa es el núcleo de la aplicación ya que se comunica con todas las demás capas para llevar a cabo la ejecución. Se comunica con la capa de presentación para recibir las solicitudes del usuario y presentar los resultados, y con la capa de datos para solicitar a la base de datos que almacene o recupere datos.

• Capa de datos: Es la capa donde residen los datos y es la encargada da acceder a los mismos. Está formada por uno o más gestores de bases de datos que realizan todo el almacenamiento de datos, reciben solicitudes de almacenamiento o recuperación de información desde la capa de negocio. Esta capa se comunica únicamente con la capa de negocio.

En nuestro caso, podríamos decir que la base de datos es el servidor web *WFS* o los ficheros *CSV* ya descargados en nuestro ordenador. La capa de acceso a datos serían las clases que interactúan con los anteriores y cargan en una lista toda esta información, siendo la capa de negocio la que hace el trabajo de filtrado y creación de la ruta óptima con dichos *waypoints* para generar el *FlightPlan* resultante y su representación en XML. La capa de interfaz con el usuario no será más que los formularios creados, cuya única función es realizar las llamadas oportunas sobre los objetos que implementan la capa de negocio.

En la figura 6.4 se pueden apreciar todas las clases generadas durante la realización del proyecto.

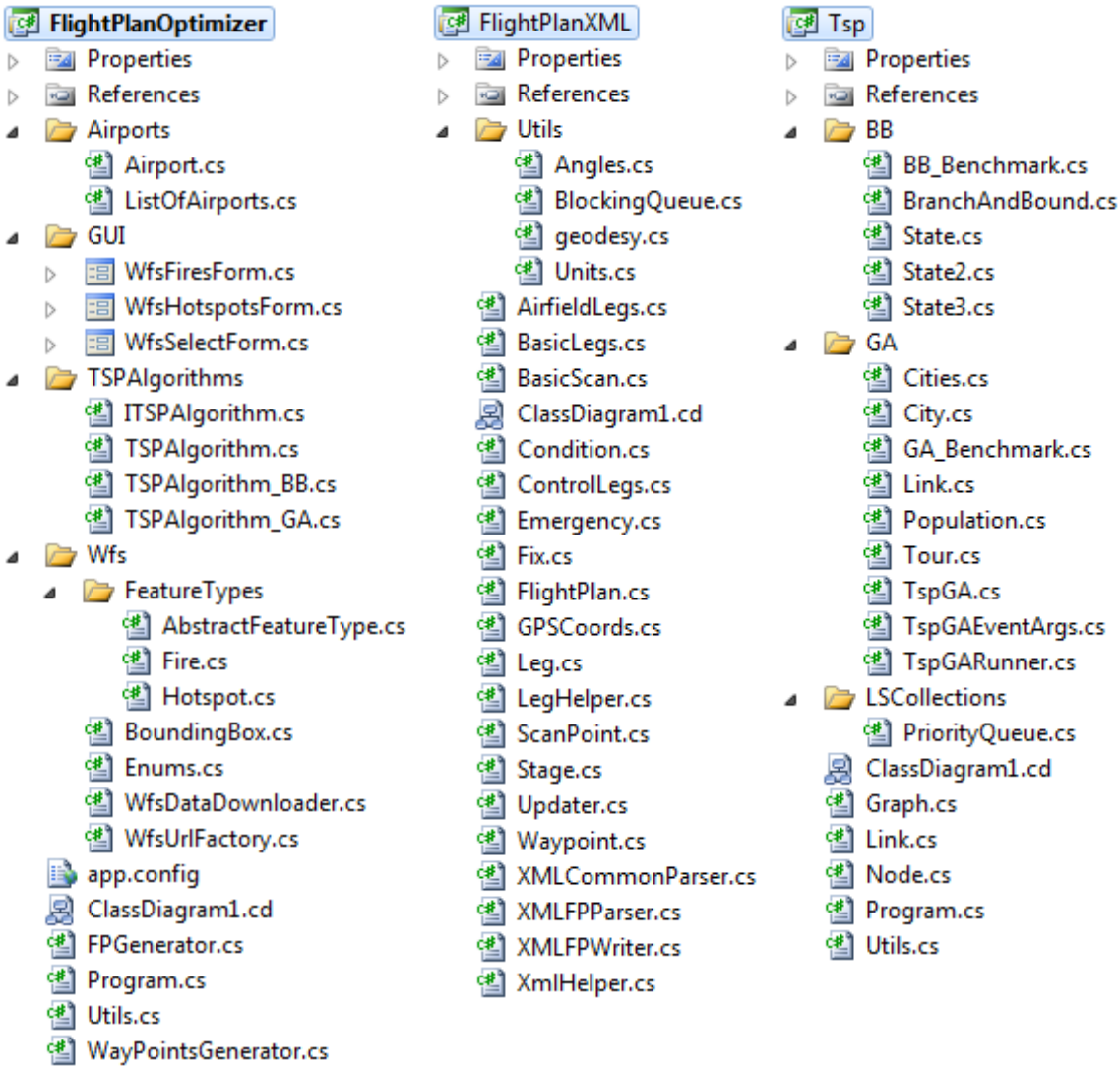

Figura 6.4: Vista del Explorador de Soluciones de Visual Studio, de todas las clases implementadas, divididas por sus respectivos proyectos.

Realmente hemos creado dos proyectos nuevos, y añadido funcionalidad al proyecto *FlightPlanXML*, en Visual Studio.
El proyecto *Tsp* es el encargado de resolver la generación de un camino óptimo dado un grafo, y correspondería a la capa de negocio.

El proyecto *FlightPlanXML* es el encargado de guardar el plan de vuelo acorde a las especificaciones del 'Flight Plan Specification Language'. Correspondería a la capa de datos.

Y finalmente, el proyecto *FlightPlanOptimizer* es el encargado de conectar los otros dos proyectos, y ofrecer, tanto la API para desarrolladores, como la GUI para usuarios finales. Además, contiene las funciones para conectar con el WFS y representar la información descargada en el modelo de objetos. Por lo tanto, este último proyecto contiene clases que pertenecen a cada una de las 3 capas de software especificadas anteriormente (presentación, negocio y datos).

En los anexos de este documento, se incluye una carpeta llamada VS2010Projects con los tres proyectos mostrados en la figura 6.4, incluyendo todo el código y los diagramas de clases generados.

CAPÍTULO 7 – CONCLUSIONES DEL PROYECTO

El objetivo inicial de este trabajo era crear una librería (API) que calculara el plan de vuelo más corto que pasara por un conjunto de puntos de interés en una ruta de un hidroavión en sus tareas de prevención o extinción de incendios, pudiendo visualizarse ésta en la plataforma ISIS. Dicho objetivo se ha cumplido, llegándose incluso a desarrollar una GUI para que un usuario final, y no sólo desarrolladores, pueda hacer uso de todas estas funciones.

Se comprueba, además, que el problema que estamos tratando tiene un coste exponencial con respecto a su talla, estando entre los problemas denominados NP-Completos, teniendo que usar, para tamaños grandes de ciertas instancias Algoritmos Genéticos, técnica que no provee una solución óptima, pero sí se le aproxima. En los demás casos, y gracias al uso de Ramificación y Poda, podemos llegar a una solución óptima sin un consumo excesivo de tiempo en este cálculo, teniendo en cuenta siempre, que todo esto dependerá no sólo del tamaño del problema, sino también de la instancia concreta que estemos tratando.

Otra conclusión que se puede derivar de este trabajo es que, el método de ramificación en RyP y el cálculo de cotas para este problema han de utilizar una función de coste constante si queremos obtener un buen rendimiento, ya que de otro modo, debido al elevado número de nodos que podemos llegar a expandir, el tiempo de ejecución se eleva considerablemente. Es decir, vale la pena para este tipo de problemas tener una función de coste menos informada pero más rápida, que no al contrario, si para conseguir informarla hemos de añadirle cierto coste computacional. En caso contrario, es evidente que una función de coste muy informada y computable en tiempo constante sería la situación ideal para podar los nodos que expandamos antes y en más cantidad. Pero en nuestro caso, cuanto más informadas estaban estas funciones, más recursos se tenían que consumir para calcularlas, siendo la heurística menos informada (suma de distancias mínimas entre los nodos no recorridos) la que mejores resultados nos ha ofrecido en la práctica.

Además, queda demostrada la potencia y eficacia del Algoritmo Genético para tallas grandes del problema, donde en un tiempo más que razonable se obtienen rutas adecuadas y que satisfacen las necesidades de este proyecto.

Otra conclusión que podemos obtener es que la división de todo el proyecto en pequeñas tareas a realizar, y el cumplimiento de una serie de hitos a lo largo de la duración del proyecto, han sido muy efectivos a la hora de crear la aplicación sin llegarse a estancar en exceso en cualquiera de los tres apartados importantes (creación del *flightplan*, creación y testeo de los algoritmos, obtención de datos reales de *hotspots* e incendios), llegando incluso a dejar un pequeño margen de tiempo para crear la GUI. La arquitectura de la aplicación en 3 capas (datos, negocio, interfaz) también ha sido muy efectiva y ha permitido la creación de un código limpio y entendible para añadir o mantener futuras características.

Otra cuestión a destacar son los esfuerzos que se hacen desde la Comisión Europea y otros organismos, para ofrecer datos de calidad y en tiempo real en la red, ya sea en nuestro caso, referentes a incendios, u otros, como los referentes a riesgos de inundaciones. Estos datos son un estímulo y un punto de apoyo básico en los desarrollos actuales y futuros de aplicaciones que ayuden a la prevención de este tipo de situaciones. Además, queda demostrada la necesidad de crear y utilizar estándares comunes para que todos esos datos sean comprensibles por los desarrolladores y por quienes finalmente interpretaran dicha información. Estos estándares son OGL, y los servidores Web Maps Server (WMS) y Web Features Services (WFS), basados todos ellos en XML y DTD.

Aún así, en mi opinión, todo estos servicios se podrían mejorar todavía más usando Web Services, donde quedaría todavía más claro y limpio a qué información estamos accediendo y qué parámetros necesitamos pasarle a las funciones que corresponden para obtenerla. Gracias a los lenguajes (c#, java,…) y plataformas de desarrollo actuales (eclipse, Visual Studio, …), el acceso a ellos es realmente sencillo y no hacen casi necesario conocer los estándares anteriores a los desarrolladores, facilitándoles un poco más la creación de aplicaciones que consuman estos datos.

A nivel personal este proyecto me ha ayudado a ampliar un poco más la visión de cómo se trabaja en un departamento de otra universidad que no es la mía. Con un equipo multidisciplinar donde el espíritu de trabajo en colaboración y la visión de un objetivo común final para desarrollar algo que puede mejorar un poco la vida de todos, para mí ha sido muy gratificante. Además, he aprendido bastantes cosas sobre especialidades distintas a la informática, como la prevención de incendios y las etapas de vuelo de un avión, y ampliado mis conocimientos sobre ciertos algoritmos y técnicas que, sin duda, pueden ser muy útiles aplicadas a cualquier otro campo.

7.1 Posibles ampliaciones y mejoras

- Tener en cuenta el FWI (Fire Weather Index) para la generación de los nodos en las rutas.[15]

El modulo 'fire danger forecast' (pronóstico del peligro de fuego) del EFFIS genera diariamente mapas de previsión del nivel de peligro de incendios para los 6 días siguientes en la Unión Europea usando datos de pronóstico del tiempo. Este módulo está activo desde el 1 de marzo hasta el 31 de octubre de cada año y se alimenta de datos meteorológicos recibidos diariamente desde los servicios franceses y alemanes de meteorología (Meteo-France y DWD).

Después de una fase de pruebas de 5 años, durante los cuales, diferentes métodos de peligro de incendios han sido implementados en paralelo, en 2007 la red EFFIS finalmente adoptó el Fire Weather Index (FWI) desarrollado en Canadá, como el método para evaluar el nivel de peligro de incendios de forma armonizada en toda Europa.

En la capa actual que EFFIS ofrece a través de su WMS, este riesgo de incendios se clasifica en 5 tipos distintos (muy bajo, bajo, medio, alto y muy alto) con una resolución espacial de unos 45 km (datos de Meteo-France) y de 36 km (datos de DWD). Estos tipos de peligro de incendios son los mismos para todos los países y mapas, mostrando una imagen armonizada de la distribución espacial del nivel de riesgo de incendios a lo largo de la UE.

Estos mapas de pronóstico de alerta de incendios pueden ser consultados a través de la interfaz web de mapas de EFFIS, y además están disponibles a través del servidor WMS. Además del pronóstico actual de riesgo de incendio, se pueden obtener también los de días anteriores, o medias para un periodo de tiempo dado, especificando el intervalo de tiempo de interés.

En un principio, nosotros deseábamos utilizar estos datos, pero se presentó el inconveniente de la resolución espacial, la cual era demasiado alta (36 o 45 km), y que además, EFFIS no ofrecía estos datos en un formato utilizado por el WFS, sino por el servidor de mapas WMS, el cual está más bien pensado para devolver una imagen. Finalmente, y tras algunas gestiones, conseguimos que EFFIS ofreciera el valor real del FWI a través de una función especial del WMS, pero para entonces ya habíamos tomado la decisión e implementado, el tomar como puntos de interés para hacer pasar el avión, los *hotspots* y los *fire* anteriormente descritos.

Sin embargo, sería una ampliación interesante el que, una vez EFFIS haya mejorado un poco esta interfaz, obtener los datos de FWI para la zona que ha de ser vigilada, y tras

un pretratamiento, tomar como puntos de interés las zonas donde este índice sea más elevado. Cabe destacar que los valores reales de FWI no comprenden sólo los cinco tipos que se pintan en el mapa, sino que es un número real comprendido entre 0 y 100, aunque pasado el umbral de 50 ya se considera alto riesgo.

La figura 7.1 muestra la capa de datos anteriormente mencionada en formato *layer*, es decir, una imagen transparente que se puede pintar sobre un mapa. La ampliación consistiría en poder obtener estos datos con una resolución mayor y utilizarlos como se ha explicado anteriormente. En la *Universitat Politècnica de Catalunya* se está negociando con el *EFFIS* cómo obtener esta información. Una vez disponible, añadir esta nueva funcionalidad a nuestra aplicación puede llegar a ser una tarea bastante trivial pero muy importante.

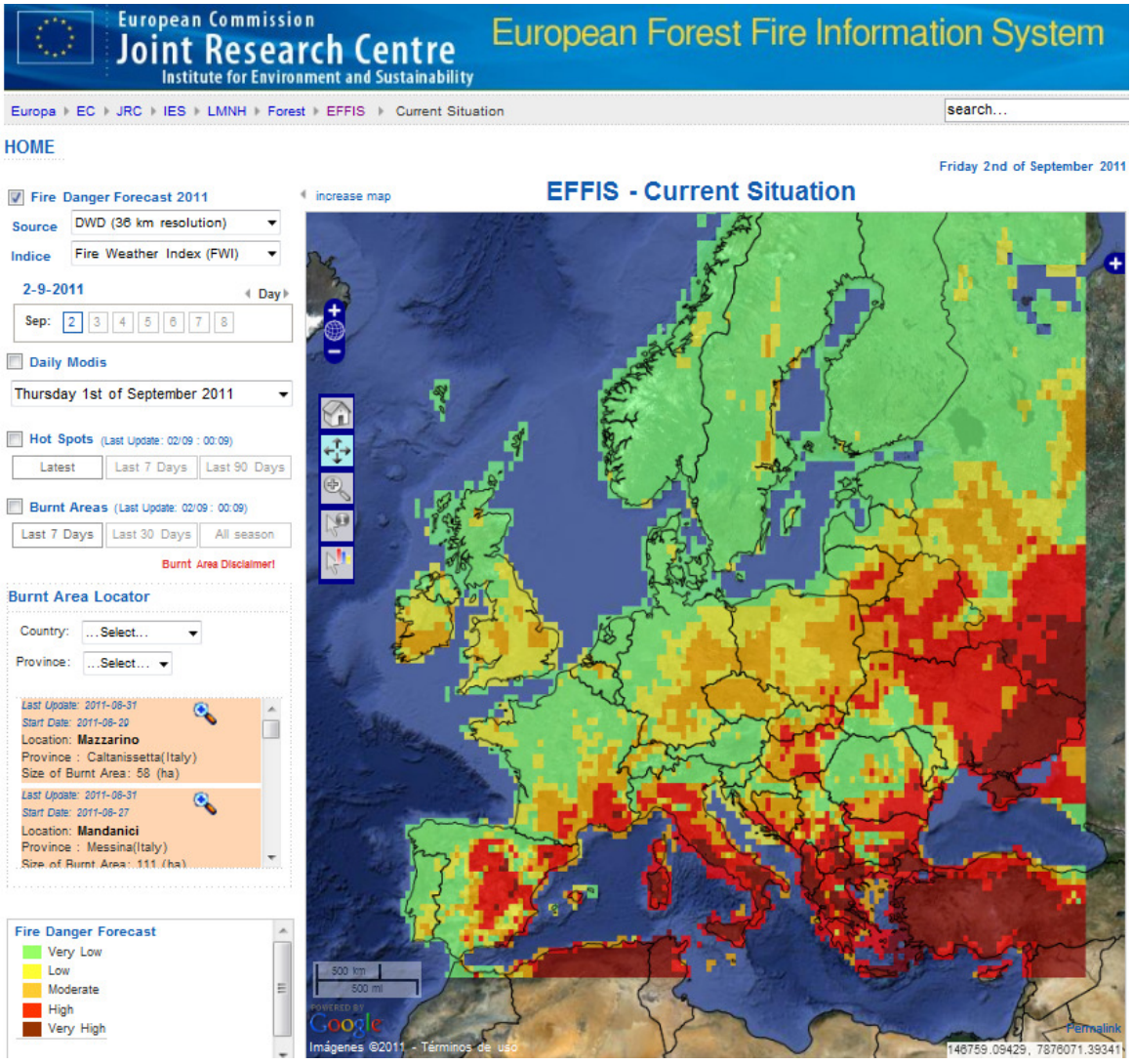

Figura 7.1: Interfaz web del EFFIS *(http://effis.jrc.ec.europa.eu/current-situation).*

Otra posible ampliación sería la siguiente:

- Sistema de alertas para nuevos hotspots detectados por el MODIS en una zona determinada.

Se trataría de tener un '*daemon*' que tuviera en cuenta los hotspots ya detectados y controlados en un área de interés, y estuviera haciendo consultas con una relativa frecuencia al servidor WFS para saber si se ha detectado un nuevo hotspot que anteriormente no existía.

Cuando esto sucediese, hay una gran variedad de cosas que la aplicación podría hacer, una de ellas sería enviar un e-mail a quien correspondiera, o incluso un SMS de alerta a uno o varios teléfonos móviles, además de automáticamente generar el *FlightPlan* que nos llevase desde el aeropuerto más cercano a ese conjunto de nuevos hotspots detectados, adjuntándolo en el e-mail.

BIBLIOGRAFÍA

[1] Service Oriented Fast Prototyping Environment for UAS Missions
Pablo Royo, Juan López, Joshua Tristancho, Juan Manuel Lema, Borja López, and Enric Pastor
Computer Architecture Dept., UPC, Spain

[2] A Middleware Architecture for Unmanned Aircraft Avionics
Juan López, Pablo Royo, Enric Pastor, Cristina Barrado, Eduard Santamaria
Technical University of Catalonia

[3] Formal Mission Specification and Execution Mechanisms for Unmanned Aircraft Systems
Eduard Santamaría Barnadas
Departament d'Arquitectura de Computadors Universitat Politécnica de Catalunya (UPC)

[4] OpenGIS Web Feature Service 2.0 Interface Standard
http://www.opengeospatial.org/standards/wfs

[5] Especificaciones del aparato MODIS a borde de los satélites TERRA y AQUA
http://modis.gsfc.nasa.gov/about/

[6] Especificación técnica sobre la política detección de fuegos activa realizada por el EFFIS.
http://effis.jrc.ec.europa.eu/about/technical-background/active-fire-detection

[7] Página Web principal del 'Open Geospatioal Consortium'
http://www.opengeospatial.org/

[8] Apuntes de Algorítmica
Andrés Marzal, María José Castro, Pablo Aibar
2006/2007—Universitat Jaume I de Castelló

[9] Foundations of algorithms using C++ pseudocode
Richard E. Neapolitan,Kumarss Naimipour

[10] Descripción del algoritmo de Floyd-Warshall
http://es.wikipedia.org/wiki/Algoritmo_de_Floyd-Warshall

[11] Descripción de los algoritmos genéticos
http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/temageneticos.pdf

[12] Introducción a los algoritmos genéticos.
Marcos Gestal Pose
Depto. Tecnologías de la Información y las Comunicaciones. Universidade da Coruña
http://sabia.tic.udc.es/mgestal/cv/AAGGtutorial/TutorialAlgoritmosGeneticos.pdf

[13] Algoritmos Genéticos
Natyhelem Gil Londoño
Universidad Nacional de Colombia. Escuela de Estadística
http://www.monografias.com/trabajos-pdf/algoritmos-geneticos/algoritmos-geneticos.pdf

[14] Traveling Salesman Problem Using Genetic Algorithms
http://www.lalena.com/AI/Tsp/

[15] Descripción del "Canadian Forest Fire Weather Index (FWI) System"
http://cwfis.cfs.nrcan.gc.ca/en_CA/background/summary/fwi

[16] XML 1.0, *Extensible Markup Language (XML) 1.0*, World Wide Web Consortium Recommendation, Bray, T., Paoli, J., Sperberg-McQueen, C.M., and Maler, E., eds., available at *http://www.w3.org/TR/*

[17] XML Schema, *XML Schema Part 1: Structures*, World Wide Web Consortium Recommendation, Thompson, H.S., Beech, D., Maloney, M., and Mendelsohn, N., eds., available at *http://www.w3.org/TR/*