DESARROLLO DE UN SISTEMA PARA TOMA DE DECISIONES EN SITUACIONES DE INTRUSIÓN EN INSTALACIONES E INFRAESTRUCTURAS

Proyecto Fin de Máster en Sistemas Inteligentes

Máster en Investigación en Informática, Facultad de Informática, Universidad Complutense de Madrid

Autor: Jesús M. Conesa Muñoz

Director: Gonzalo Pajares Martinsanz

Colaborador de dirección: Ángela Ribeiro Seijas

Curso académico: 2008/09

Resumen

El presente trabajo describe la arquitectura de un sistema inteligente desarrollado dentro de un proyecto de seguridad para la protección de grandes infraestructuras. El principal objetivo es conseguir un sistema experto capaz de tomar decisiones en tiempo real que neutralicen ataques de uno o más intrusos a objetivos clave de una instalación. Para ello se abordarán diferentes problemas como la representación espacial del entorno, planificación de caminos sobre el espacio representado, análisis y caracterización del escenario de intrusión, toma de decisiones y aprendizaje en función de la evolución del sistema, así como las comunicaciones con diferentes módulos mediante integraciones a través de un *middleware*. En cada sección se incidirá en el planteamiento escogido para resolver cada problema concreto y las razones que han motivado su elección. Algunos de estos enfoques son: triangulaciones de *Delaunay* (para la representación espacial), razonamiento basado en casos (para la toma de decisiones) y *Ice* (como *middleware* de integración entre módulos). Finalmente, se mostrarán y discutirán los resultados globales del sistema y las diferentes alternativas implementadas en algunas fases. Entre las técnicas de análisis de resultados se emplearán evaluaciones de tipo cruzado, tales como *leave-one-out* o k-validaciones.

Palabras clave: Sistema Experto, Triangulaciones de *Delaunay*, Planificación de Caminos, Razonamiento Basado en Casos, Función de Similitud, Árboles de Decisión, Algoritmos Genéticos, *Middleware*, *Ice*, Evaluaciones Cruzadas.

Abstract

The present paper describes the architecture of an expert system involved in a crisis management project for infrastructural building security. The main goal is to achieve an expert system, capable of making decisions in real-time to quickly neutralize one or more intruders that threaten strategic installations. To accomplish that many different problems will be deal with like the spatial representation of the environment, the path planning through the represented space, the analysis and characterization of the intrusion scenario, the making decision and learning related to the evolution of the system as well as the communications with different modules using integrations based in *middlewares*. The chosen approach to solve each particular matter will be discussed in each section as well as the selection reasons. Some of these approaches are: *Delaunay* triangulations (for the spatial representation), case-based reasoning (for the making decision) and *Ice* (as integration middleware among the modules). Finally, results on the retrieving phase performance are shown and analyzed according to well-known cross-validations, such as leave-one-out or k-validations.

Keywords: Expert System, *Delaunay* Triangulation, Path Planning, Case-Based Reasoning, Similarity Function, Decision Tree, Genetic Algorithms, Middleware, *Ice*, Cross-Validation.

1 Introducción

1.1 Marco de la aplicación

Debido a la sensible situación internacional causada por los últimos ataques terroristas, existe una necesidad común de proteger y garantizar la seguridad de infraestructuras de grandes dimensiones como edificios gubernamentales, aeropuertos o estaciones de energía. Esta demanda es especialmente fuerte en los territorios de la Unión Europea y en Estados Unidos donde las situaciones de crisis desencadenan desestabilizaciones en cadena que pueden llegar a afectar en gran medida a multitud de sectores de la sociedad.

Una crisis es un evento generalmente súbito y con escaso tiempo de reacción en donde las decisiones pueden estar afectadas por una gran cantidad de variables que conllevan muchos elementos de incertidumbre. Por ello, la utilización de las técnicas contenidas en las llamadas tecnologías de la información puede facilitar enormemente la gestión de los datos para la toma de decisiones adecuadas que conduzcan a paliar la crisis con los recursos disponibles, en el menor tiempo posible y con el menor número de pérdidas.

Existe un gran vacío en la aplicación de tecnologías en los procesos de gestión de crisis. Así, disponer de un volumen organizado de información y usar sistemas de gestión de la misma será crucial en todas las etapas, permitiendo optimizar la planificación y la toma de decisiones. Cuando, como en una situación de crisis, la información distribuida espacialmente es decisiva en la decisión, el sistema de gestión de la información deberá incluir algún mecanismo para gestionar adecuadamente la información espacial y las relaciones espaciales entre los elementos que integran la crisis y la gestión de la misma. Por ejemplo en la distribución de equipos humanos en una situación de crisis, se pueden tener en cuenta criterios de necesidad y criterios espaciales como la cercanía al foco del desastre o a otros equipos. Además de los espaciales también existen otros factores como la detección de movimientos y sonidos que pueden ser cruciales en el proceso de toma de decisiones. Una vez determinados todos los criterios relevantes para la evaluación de la escena será necesario ponderar cada uno de estos factores en su justa medida y determinar en función de sus diferentes pesos cual es la actuación óptima que se debe generar.

A las dificultades inherentes para tratar la información asociada a los sistemas de representación espacial, de audio y vídeo cognitivo, hay que sumar los problemas habituales de las grandes aplicaciones como son la interfaz de usuario, la arquitectura, las redes de comunicación, la seguridad, etc. Por ello, para afrontar todos estos problemas es necesario un desarrollo conjunto en varias áreas de investigación capaz de integrar todos los requisitos necesarios para el buen funcionamiento del sistema final.

1.2 Hesperia y programa CENIT

El proyecto *Hesperia* [Hesperia] tiene por objeto el desarrollo de tecnologías que permitan la creación de sistemas punteros de seguridad, vídeo vigilancia y control de operaciones de infraestructuras y espacios públicos. El proyecto surge para dar respuesta a una demanda sostenida

a medio y largo plazo, en particular, en países de la Unión Europea y en Estados Unidos. La gestión integrada de seguridad y control de operaciones permitirá la implantación de sistemas rentables que, en este momento, no existen en el mercado.

Las tecnologías del proyecto resolverán la seguridad en dos tipos de escenarios. Por un lado, permitirán gestionar la seguridad y las operaciones de infraestructuras públicas especialmente sensibles, como subestaciones eléctricas, en gas, depósitos de agua o estaciones de telecomunicaciones. Por otro, incrementarán de forma sustancial los niveles de seguridad de grandes espacios públicos, como aeropuertos, estaciones de ferrocarril, puertos, centros de ciudades especialmente en zonas peatonales, centros comerciales, etc.

El *CDTI*, organismo adscrito al Ministerio de Industria Turismo y Comercio, creó en 2005 un programa de Consorcios Estratégicos Nacionales en Investigación Técnica (*CENIT*), cuyo principal objetivo es fomentar la cooperación público privada en I+D+i mediante la financiación de proyectos conjuntos de investigación industrial. El organismo ha dotado a CENIT con doscientos millones de euros durante cuatro años, el 46,5 % de la inversión prevista para todo el programa. Los otros 230 millones de euros los aportará el sector privado.

El consorcio está integrado por Indra Software Labs, Unión Fenosa, Tecnobit, SAC Control, Technosafe, Visual Tools y Brainstorm Multimedia. Asimismo, participan las universidades de Castilla La Mancha, de Granada, de Extremadura, la Politécnica de Madrid, la de las Palmas, la Politécnica de Valencia y la Politécnica de Cataluña. La lista se completa con la colaboración del Consejo Superior de Investigaciones Científicas (*CSIC*) y el Centro Tecnológico del País Vasco (Ikerlan).

1.3 Gestor de crisis

El proyecto Hesperia está dividido en varios módulos con sus respectivas funcionalidades y uno de ellos es el gestor de crisis. Este es el componente que se describe en el presente proyecto y su funcionalidad consiste en ser capaz de generar actuaciones que minimicen un ataque de intrusión en una infraestructura conocida. Para ello debe ser capaz de analizar las características relevantes de la propia infraestructura y de su entorno, y a partir de ellas inferir cuál es la estrategia óptima para neutralizar el ataque.

El propio gestor de crisis se divide en tres grandes partes: el planificador de caminos, el *CBR* y las clases de conexión con el *middleware*. Gracias a la integración de estos tres módulos se consigue generar un sistema experto capaz de tomar decisiones que salvaguarden la seguridad de grandes infraestructuras. De este modo, aplicando técnicas ya conocidas, se diseña un sistema avanzado capaz de aportar una solución para un problema complejo y real.

El planificador agrupa tanto las estructuras de datos que soportan la representación espacial del recinto sobre el que se ejecuta el sistema como los métodos de consulta sobre ella. En particular destaca la operación de búsqueda entre dos puntos por resolver el problema de encontrar el camino más corto en un plano con obstáculos.

El CBR, razonamiento basado en casos, implementa el sistema de toma de decisiones. A partir de unos parámetros determinados previamente como relevantes, deduce la actuación más apropiada, es decir, aquella que contribuye más a minimizar el impacto del ataque. El CBR se apoya en el planificador para calcular las distancias entre elementos relevantes de la escena, de este modo, consigue más información a tener en cuenta en el proceso de toma de decisiones.

Por último, la integración con el resto de módulos que conforman el proyecto se realiza mediante un *middleware* llamado *Ice*. De ahí la existencia de un tercer componente del gestor de crisis que agrupa las clases de conexión y cuya función es gestionar la comunicación con el resto de aplicaciones del sistema. El diseño del gestor de crisis con sus tres módulos y métodos que lo forman constituye el objetivo de la investigación propuesta.

En los siguientes apartados se explicará el funcionamiento de estos tres componentes. La sección dos describe el funcionamiento del planificador así como la estructura implementada para soportar la representación espacial sobre la que se aplica la búsqueda de caminos. El apartado tres contiene las explicaciones relacionadas con el razonamiento basado en casos o CBR. En él se detallan las diferentes etapas del ciclo de vida del CBR y los diferentes esquemas de funcionamiento abordados. El apartado cuarto se centra en el *middleware* y el quinto en la integración del proyecto, tanto a nivel interno (relaciones entre los diferentes componentes del gestor) como a nivel externo (comunicación con aplicaciones externas). Por último, las secciones seis y siete abarcan respectivamente los resultados evaluados y las conclusiones obtenidas.

2 Planificador

El objetivo principal del gestor de crisis es generar actuaciones capaces de neutralizar ataques sobre la infraestructura que se está protegiendo. Para poder cumplir este cometido, uno de los parámetros a tener en cuenta son las características de la infraestructura, por ello, es necesario encontrar una estructura de datos adecuada para la representación espacial de un recinto. En esta sección se detalla la estructura propuesta y sus características. Debe ser fácil de construir y de modificar, así como soportar búsquedas de caminos mínimos ya que en etapas posteriores, el CBR (el módulo que se encargará de la toma de decisiones) se apoyará en esta información para ofrecer la actuación más apropiada. Es por ello que se distinguen dos partes bien diferenciadas: la construcción y modificación de la estructura que represente espacialmente al recinto y la búsqueda de rutas mínimas entre dos elementos dentro de este espacio. En la primera parte de la sección se hablará de la estructura auxiliar *QuadEdge* necesaria para construir la estructura de tipo *Triangulation* que representará espacialmente un plano con obstáculos. La segunda parte describirá las operaciones de construcción y modificación de la triangulación y la tercera parte se centrará en el algoritmo de búsqueda de caminos óptimos sobre la triangulación.

El planteamiento por el que se ha optado consiste en una triangulación restringida de *Delaunay* [Anglada, 1997] [Preparata et al., 1985] [Floriani et al., 1992] o CDT (*constraint Delaunay triangulation*) de carácter dinámico [Kallmann et al., 2003], es decir, con posibilidad de añadir o eliminar restricciones en tiempo de ejecución. Una vez construida la triangulación se aplicará sobre ella una búsqueda *TA** para encontrar el camino óptimo (más corto) entres dos puntos dados. Este tipo de búsqueda es una adaptación del *A** sobre una triangulación en vez de sobre un grafo con nodos adimensionales (sin área, sólo con posición). El algoritmo *TA** se apoyará en el algoritmo de *funnel* [Chazelle, 1982] [Lee et al., 1984] para encontrar el camino más corto dentro de un polígono.

La principal ventaja para confiar en una representación triangulada es que el tamaño del grafo de adyacencia, aquél sobre el que se realizarán las búsquedas de caminos, es $O(n)$ donde n es el número de vértices y es habitualmente mucho más pequeño que los que se usan en métodos basados en grids [Koenig, 2004] o en grafos de visibilidad [Kreveld et al., 2000] ya que los primeros tienden a generar grafos grandes y los segundos tienen complejidad $\Omega(n^2)$ en el peor de los casos. De este modo se reduce el tiempo de determinar el camino más corto al tiempo necesario para la búsqueda sobre el grafo, la cual puede ser implementada en tiempo $O(n \log n)$ con cualquier método de búsqueda de tipo *Dijkstra* [Cormen et al., 1993]. Además, como ya se ha mencionado, la CDT se puede actualizar eficientemente con la inserción de nuevos obstáculos o con la eliminación de los ya incluidos.

2.1 Representación del espacio

2.1.1 Triangulaciones de Delaunay

La técnica propuesta para segmentar el espacio en celdas sobre las que poder realizar una búsqueda se basa en un desarrollo incremental de una triangulación restringida de *Delaunay*. A

partir de un conjunto de obstáculos poligonales se construye una malla con estructura de grafo sobre la que se realizarán las búsquedas. Las aristas del grafo serán de dos tipos: restringidas (representan el contorno de algún obstáculo) o sin restringir (delimitan alguna región del espacio). Cada región tendrá forma de triángulo y estará, por tanto, delimitada por 3 aristas, de ahí el nombre de triangulación. Se podrá pasar de una región a otra adyacente siempre que la arista fronteriza (aquella que tienen en común) no sea restringida. Dentro de cada región el espacio es libre, es decir, no hay ningún obstáculo y por tanto, para cualesquiera dos puntos internos la ruta mínima entre ellos es la recta que los une.

La construcción de la malla se realizará incrementalmente a partir de procedimientos de inserción de puntos (vértices) y de inserción de segmentos (aristas). No sólo será posible insertar obstáculos sino que también se debe permitir su eliminación, por ello los procedimientos de inserción incluirán los pasos necesarios para las posibles posteriores eliminaciones.

En la figura 1 se muestra un ejemplo de triangulación restringida de *Delaunay*. La triangulación se encuentra delimitada por el rectángulo rojo más externo. Todo el espacio interno de este rectángulo está segmentado en triángulos. Las aristas de color rojo representan contornos de un obstáculo y no pueden ser atravesadas, es decir, no se puede pasar de un triángulo a otro adyacente a través de su arista en común si ésta está marcada como restringida (de color rojo). Por el contrario, las aristas no restringidas (de color gris) sí pueden ser atravesadas y separan diferentes zonas del espacio; son necesarias para formar la triangulación.

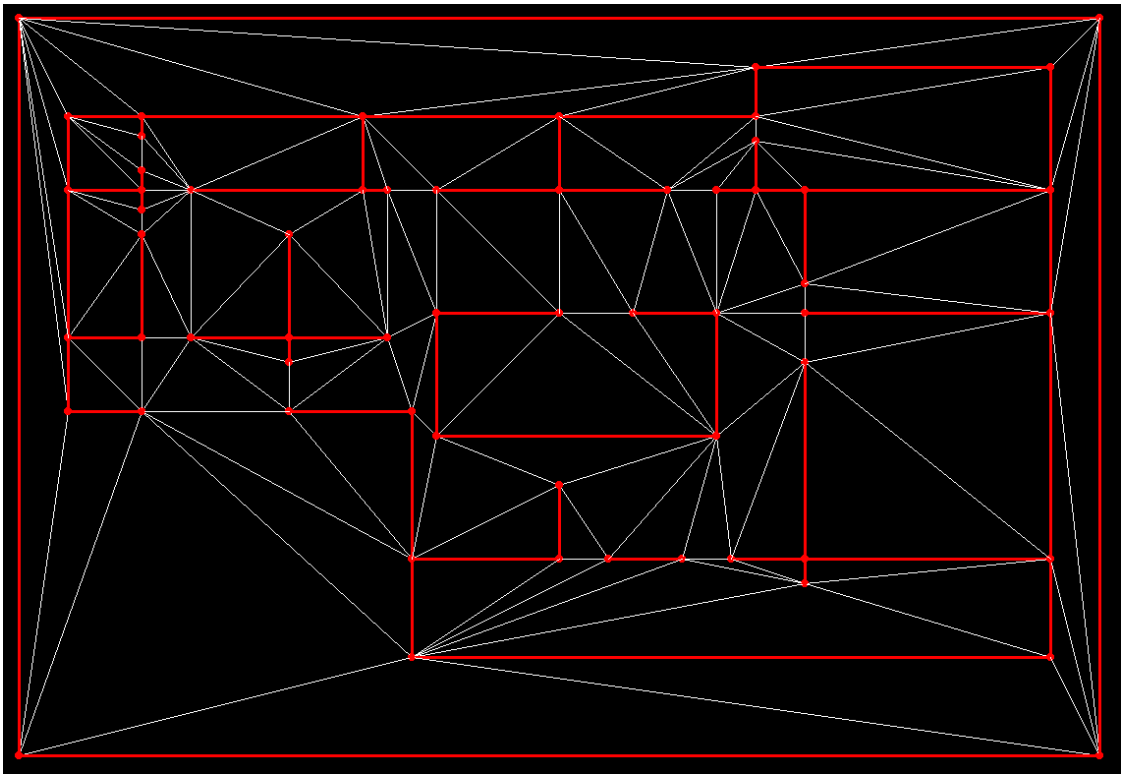

Figura 1. Ejemplo de espacio segmentado según una triangulación restringida de *Delaunay*. Las líneas rojas son los obstáculos (aristas restringidas del grafo). Las líneas grises completan las triangulaciones (aristas sin restringir)

2.1.2 Álgebra QuadEdge

Para garantizar un buen rendimiento en las búsquedas, la estructura de datos que soporte la triangulación debe ser eficiente. Esto es, debe ser capaz de devolver las relaciones de adyacencia en tiempo constante. Para ello, las aristas del grafo no sólo llevarán una referencia a los vértices que la definen, sino que estarán compuestas por una estructura con 4 aristas 'tradicionales'. Se basarán en una estructura conocida como *QuadEdge* [Guibas & Stolfi, 1985]. Además, puesto que la malla estará sujeta a cambios (inserción y eliminación de obstáculos) debe soportar también modificaciones eficientes. La estructura QuadEdge puede adaptarse para cumplir este último requisito. A continuación se detallará la implementación de las estructuras QuadEdge y *Edge* (que sirve de soporte a la primera) y el álgebra que forman para poder construir una triangulación restringida de *Delaunay* eficiente.

De ahora en adelante se usarán indistintamente los términos malla, triangulación o CDT. Todos ellos hacen referencia a la misma estructura: triangulación restringida de *Delaunay*.

2.1.2.1 Edge

Se corresponde con el concepto básico de arista. Lleva una referencia a su vértice origen y otra a la siguiente arista. Por siguiente arista *e.next* de una arista *e* se entiende la primera arista en sentido antihorario de todas las que comparten el mismo origen que *e*.

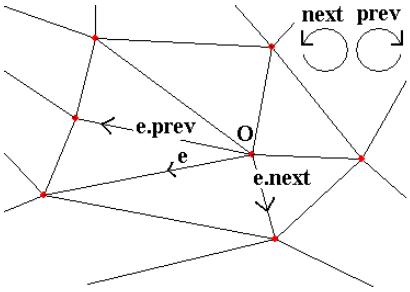

Figura 2. Arista (edge) siguiente y previa de *e* respecto a su origen *O*

La figura 2 muestra los sentidos 'next' y 'prev' que deben entenderse cuando se habla respectivamente de la arista siguiente o previa a una arista dada *e* respecto a su origen *O*.

Sobre cada arista de tipo Edge se definen las siguientes funciones [Guibas & Stolfi, 1985]:

Onext: Siguiente arista con el mismo punto de origen.
Oprev: Anterior arista con el mismo punto de origen.
Dnext: Siguiente arista con el mismo punto de destino.
Dprev: Anterior arista con el mismo punto de destino.
Lnext: Siguiente arista alrededor de la cara izquierda.
Lprev: Anterior arista alrededor de la cara izquierda.

Rnex: Siguiente arista alrededor de la cara derecha.
Rprev: Anterior arista alrededor de la cara derecha.

Aplicando estas operaciones se accede a las diferentes aristas incidentes en los vértices origen y destino de la arista evaluada.

Existen otras 3 funciones que también pueden resultar de utilidad:

Sym: Devuelve la arista simétrica a la actual.
Rot: Arista que une la cara derecha con la izquierda con origen en la cara derecha.
InvRot: Arista que una la cara izquierda a la derecha con origen en la cara izquierda.

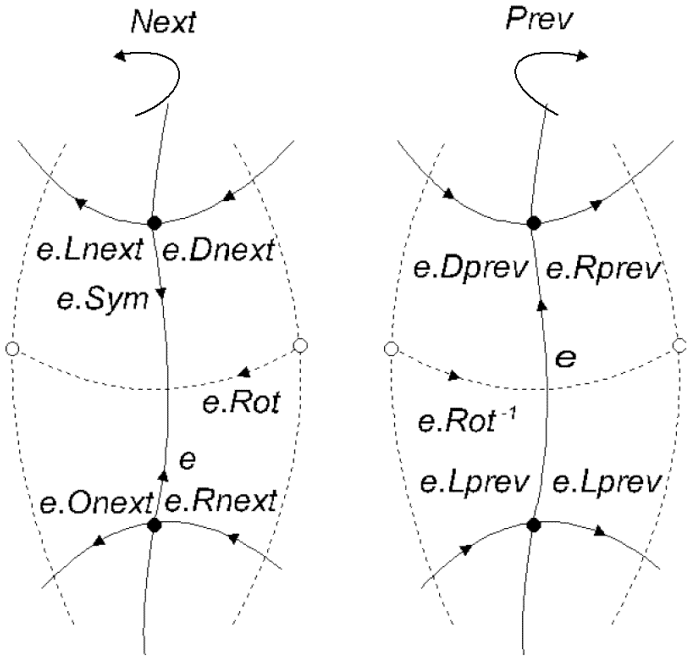

Figura 3. Funciones de adyacencia sobre un eje

En la figura 3 se muestran todas las operaciones que pueden aplicarse sobre una arista e y la arista resultante.

Las operaciones *Rot* e *InvRot* (o Rot^{-1}) sólo pueden aplicarse cuando además de la estructura del grafo principal lleva asociada la de su dual. En teoría de grafos, un grafo dual *G'* de un grafo planar *G* es un grafo que tiene un vértice por cada región de *G*, y una arista por cada arista en *G* uniendo a dos regiones vecinas. En la figura 4 se muestra un grafo planar G y su dual G'. En G se diferencian 4 regiones diferentes que corresponden con los 4 vértices de G'. Por cada arista de G que separa dos regiones, G' presenta una arista que las une.

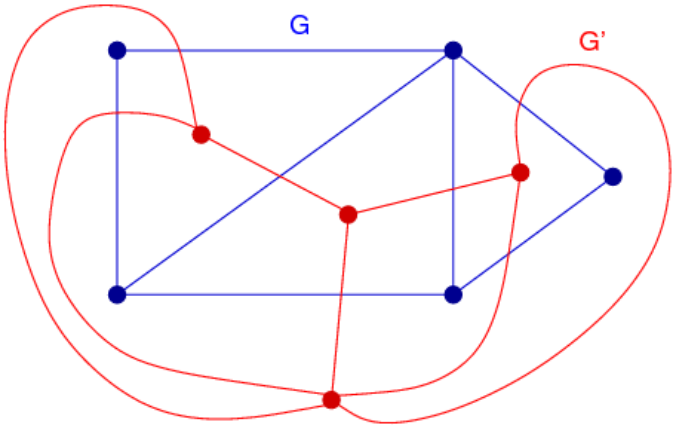

Figura 4. *G* es un grafo planar y *G'* su dual

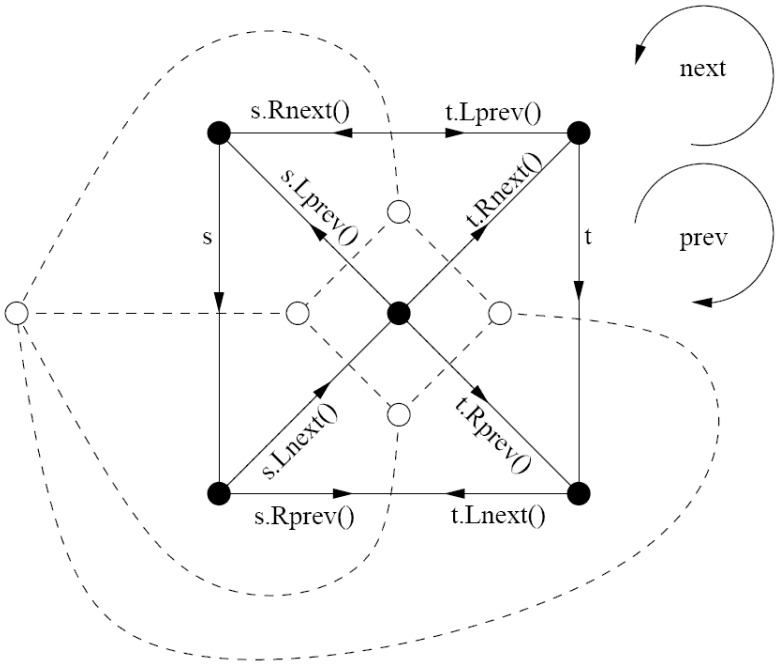

Figura 5. Grafo y su dual (en discontinua) con funciones de adyacencia

En la figura 5 se muestra un grafo, su dual y dos aristas *s* y *t* del grafo normal. Aplicando diferentes operaciones sobre estas dos aristas se puede alcanzar cualquier otra arista del grafo. Para moverse por el grafo dual habría que utilizar las operaciones *Rot* e *InvRot*.

Para implementar este tipo de estructuras capaces de soportar un grafo y su dual se utiliza la estructura QuadEdge [Guibas & Stolfi, 1985] que se presenta en el siguiente punto.

2.1.2.2 QuadEdge

Consiste en un vector de 4 aristas de tipo edge. Además de la arista habitual contiene la simétrica, la rotada y la inversa. La normal y su simétrica se mueven por el grafo principal y la rotada y la

inversa por el dual, es decir conectan las caras o regiones del espacio. De este modo es posible moverse por los contornos de las regiones representadas o por las propias regiones saltando de una celda a sus adyacentes en tiempo constante.

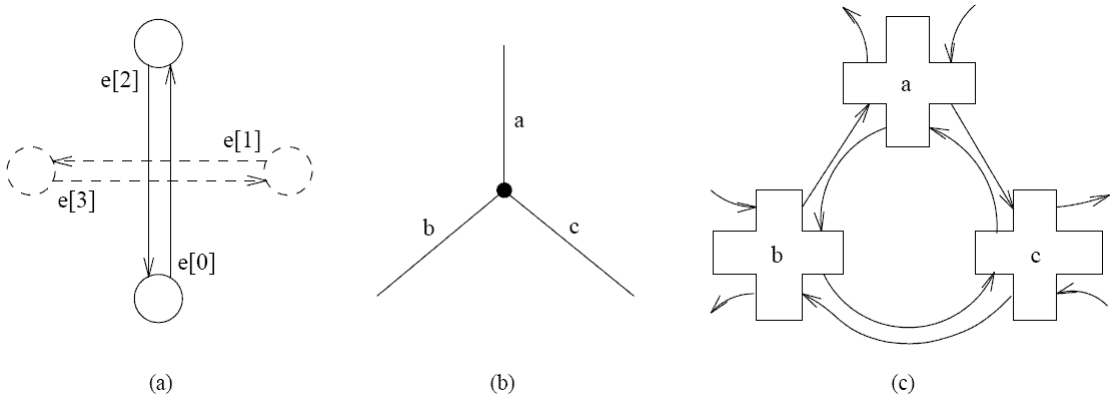

Figura 6. Estructura de un QuadEdge [Lischinski, 1994]

En la figura 6a se muestra la estructura de una arista QuadEdge: un array de 4 aristas Edge donde *e[0]* y e*[2]* son simétricas así como *e[1]* y *e[3].* Además se cumple *que e[i].Rot = e[i+1 mod 4]* y *e[i].InvRot = e[i – 1 mod 4]* . Es importante destacar que, aunque sobre un plano, cada arista QuadEdge representa solo un 'segmento', realmente lleva toda la información necesaria para obtener la arista (edge) que representa ese segmento, su simétrica, y aquellas dos que unen la región izquierda con la derecha y viceversa.

En la figura 6b aparecen 3 aristas conectadas por un mismo vértice. Si se opta por la estructura QuadEdge de la figura 6a para implementar esta conexión el resultado a nivel de estructura de datos sería el de 6c.

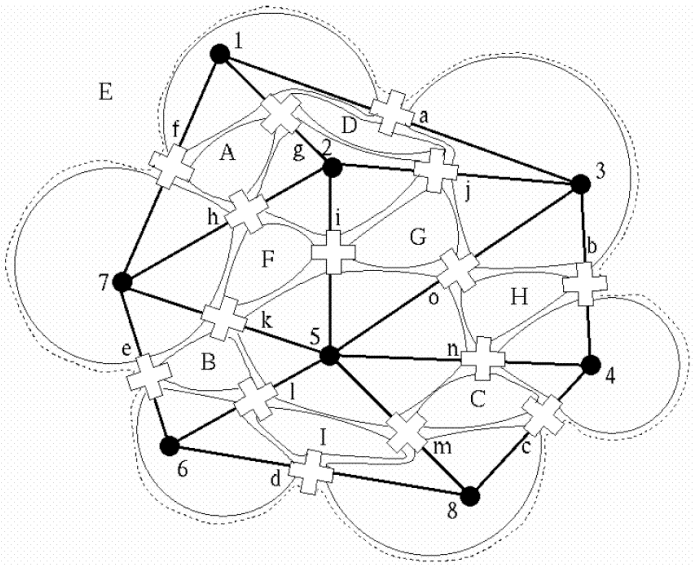

Figura 7. Malla representada con QuadEdges

La figura 7 muestra una malla con cada estructura QuadEdge superpuesta sobre la arista que representa. Cada arista tiene una referencia hacia sus aristas siguientes y previas tanto respecto a su origen como a su destino, por eso se pueden obtener en tiempo constante las aristas adyacentes a una dada. Lo mismo sucede para las caras sólo que entonces es por el grafo dual por el que hay que moverse.

Puesto que una arista de tipo QuadEdge puede ser restringida (pertenece a un obstáculo) o no (forma parte de la triangulación sin delimitar ningún obstáculo) la estructura QuadEdge debe contener la información necesaria para indicar ante qué tipo de arista se está. Además, una arista restringida puede pertenecer no sólo a un único obstáculo sino a varios (por estar éstos solapándose), por tanto es preciso llevar una lista con los identificadores de todas los obstáculos a las que pertenezca. Observar que en el caso de que la lista sea vacía, la arista es libre (sin restringir) y de este modo se puede discernir el tipo de arista ante el que se está.

2.1.2.3 Operadores entre aristas (QuadEdges):

La principal ventaja del tipo de datos QuadEdge es que la construcción (así como la modificación) de un grafo planar se puede hacer usando simplemente dos operadores: uno de creación de aristas (*makeQuadEdge*) y otro de interconexión (*splice*).

2.1.2.3.1 MakeQuadEdge:

Crea un nuevo QuadEdge cuyas cuatro aristas están sin conectar con el resto de la malla. Devuelve una referencia a la primera arista del array. Se puede especificar si el QuadEdge es restringido o no. Es decir, si las dos caras separadas comparten una arista restringida.

2.1.2.3.2 Splice(a, b):

Interconecta dos aristas de tipo Edge que se van a situar en el mismo origen. Para ello se debe intercambiar los valores de adyacencia asociados a las aristas implicadas y a sus caras:

- Se intercambian las referencias a las aristas siguientes a las implicadas respecto a su origen de coordenadas, de este modo la siguiente a la arista ***a*** será la siguiente a la arista ***b*** y viceversa:
    - <*a.next, b.next*> = <*b.next, a.next*>
- Se intercambian las referencias a las caras siguientes a las aristas implicadas. De este modo de la cara izquierda de ***a*** se saltará a la cara izquierda de ***b*** y viceversa:
    - *alpha = a.next.rot*
    - *beta = b.next.rot*

- <*alpha.next, beta.next*> := <*beta.next, alpha.next*>

En la figura 8 se conectan dos aristas ***a*** y ***b*** (renombradas con los números 1 y 5 respectivamente): partiendo de 8a se alcanza 8b. Se deben actualizar los anillos que contienen las aristas y las caras, es decir, la arista siguiente a 5 debe dejar de ser la arista 2 y pasa a ser la arista 6 que es la siguiente a 1. A su vez la arista siguiente a 1 pasa a ser la arista 2 que es la siguiente a 5. En definitiva, se intercambian los siguientes de las aristas que se están conectando. Análogamente se hace lo mismo con los anillos de las caras.

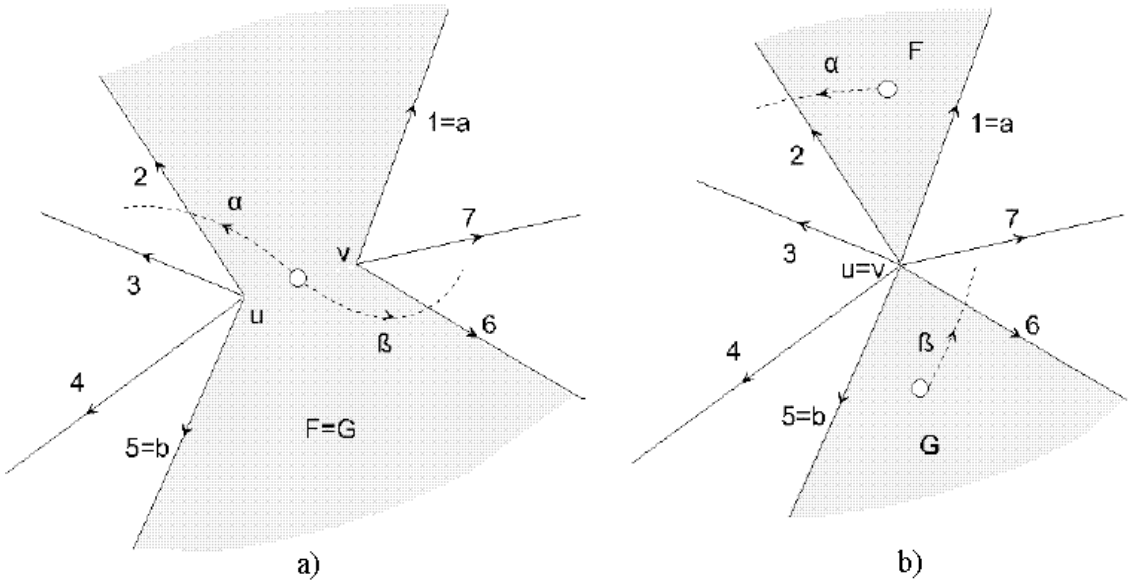

Figura 8. Estados de una malla antes y después de aplicar el operador s*plice[a, b].* Partiendo de cualquiera de los dos estados, y aplicando *splice*, se alcanza el contrario.

La interconexión de dos aristas ***a*** y ***b*** puede entenderse como la fusión entre los anillos de sucesión de las dos aristas (ver anillos de las figuras 6c y 7) en torno al origen común (uno representa la sucesión de las aristas y otra la sucesión de las caras) para formar uno solo a partir de ellos.

La operación *splice* es su propia inversa, es decir, también se puede utilizar para desconectar dos aristas ***a*** y ***b*** conectadas previamente. En la figura 8 podemos partir de cualquiera de los dos estados y alcanzar el contrario aplicando la operación *splice(a,b).*

2.1.3 Construcción de la malla o CDT

Dados un conjunto de obstáculos poligonales (restricciones), se debe construir una malla triangulada. Estos obstáculos poligonales no tienen por qué ser polígonos convexos, basta con que sean una sucesión de puntos, siendo válido un sólo punto o incluso que las líneas que lo componen intersecten entre sí. La única condición es partir de una triangulación inicial que llamaremos caja (ver figura 9): un rectángulo cuyos lados son aristas restringidas más una cualquiera de sus diagonales,

una arista sin restringir, y que todas las inserciones estén dentro de sus límites (todos los puntos deben ser internos a la caja).

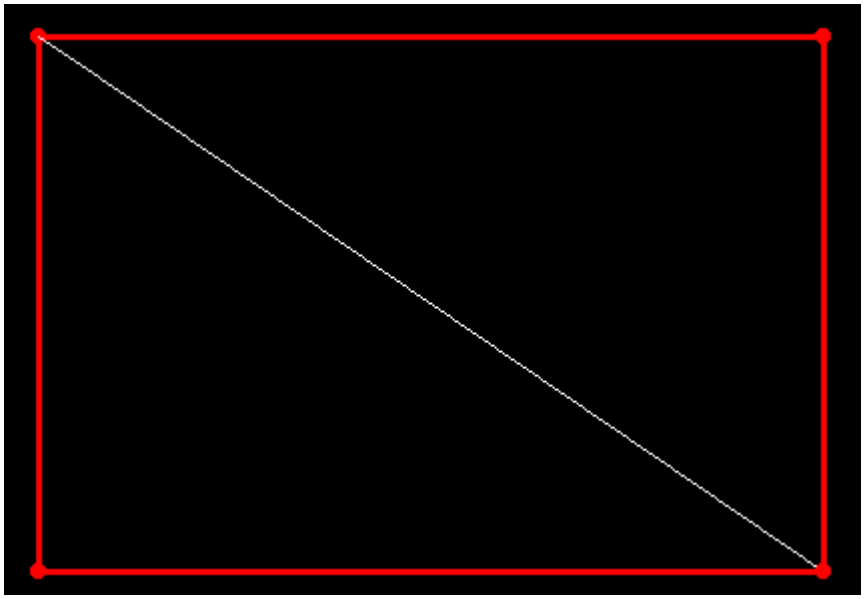

Figura 9. Caja o CDT inicial

La malla se construye incrementalmente a partir de los métodos de inserción de puntos y de inserción de segmentos. Este último se apoya a su vez sobre el primero ya que un segmento está delimitado por dos puntos. La mecánica es siempre la misma: localizar el triángulo en el que se emplazará el nuevo punto, insertar el punto y modificar el triángulo convenientemente para conservar las propiedades de un CDT (y si es necesario, propagar las modificaciones hacia fuera comenzando por los triángulos adyacentes).

2.1.3.1 Localización de un punto

Gran parte de las operaciones de manipulación sobre una malla necesitan hallar el triángulo o cara en el que se encuentra incluido un punto. Para ello se buscará una de las 3 aristas (de tipo Edge) cuya cara izquierda coincida con el triángulo requerido. Este criterio de referenciar un triángulo o cara mediante alguna de las aristas que lo encierran por la izquierda es el que se aplica a lo largo de todo el documento. El procedimiento implementado para realizar la búsqueda del triángulo en el que se encuentra un punto se basa en el enfoque propuesto en [Guibas & Stolfi, 1985] de complejidad O(n) para una triangulación de *n* vértices.

Se parte de una arista cualquiera de la triangulación, si su origen o su destino es el punto buscado se devuelve la propia arista (o la simétrica para respetar el criterio de representar a un triángulo por una de las aristas que lo contenga en su cara izquierda), en caso contrario, se trata de avanzar saltando a alguna arista del triángulo izquierdo que deje a su izquierda el punto buscado (puede ser la propia arista en sentido contrario). Finalmente, si la arista actual ***e*** ya tiene el punto buscado *p* a

su izquierda y las otras dos aristas que forman el triángulo izquierdo, *e.Onext* y *e.Dprev*, tienen el punto *p* a su derecha, podemos afirmar que el punto *p* está contenido en el triángulo izquierdo de la arista *e*. En ese caso se devuelve la arista *e*.

El procedimiento que lo implementa es Locate, descrito en el apéndice C En dicho procedimiento conviene observar que si *p* recae sobre una arista ***e***, se devuelve la propia arista *e*. Este procedimiento siempre va a devolver la arista que contenga el punto buscado o la arista cuya cara o triángulo izquierdo lo contenga. En caso de que el punto *p* se encuentre en el destino de la arista encontrada se devolverá la simétrica, de este modo siempre que se busque un punto ya insertado en la malla, el origen de la arista será el propio punto *p*.

2.1.3.2 Operación *Swap*:

Durante la construcción de la malla, en algunos momentos se deberá comprobar si los triángulos formados cumplen la propiedad de *circuncírculo* vacío y por tanto la malla es una triangulación *Delaunay*. Esta propiedad consiste en que dado un triángulo, la circunferencia circunscrita no contiene ningún otro punto de la triangulación.

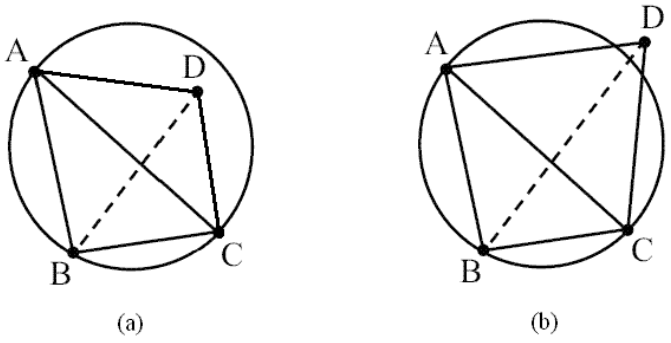

Figura 10. El triángulo ABC de (a) no cumple la propiedad de *circuncírculo* vacío ya que el punto D se encuentra inscrito dentro de la circunferencia definida por ABC. En (d) el triángulo ABC sí cumple la propiedad.

Cuando se modifica la malla, pueden alcanzarse triangulaciones que contengan triángulos que no cumplen la propiedad de circuncírculo vacío, es decir que no se alcancen triangulaciones que no sean de *Delaunay*. Para arreglarlo se aplica la operación *swap*. Esta operación consiste en cambiar la arista más cercana al punto que queda incluido en la circunferencia por la arista que resulta de unir este punto con su opuesto respecto a la arista inicial. Posteriormente se comprueba que se sigue cumpliendo la propiedad para los nuevos triángulos formados y de no ser así se seguiría aplicando sucesivamente la operación swap. En la figura 11 se puede ver el resultado de aplicar la operación swap sobre 2 triángulos que comparten una arista ***e***.

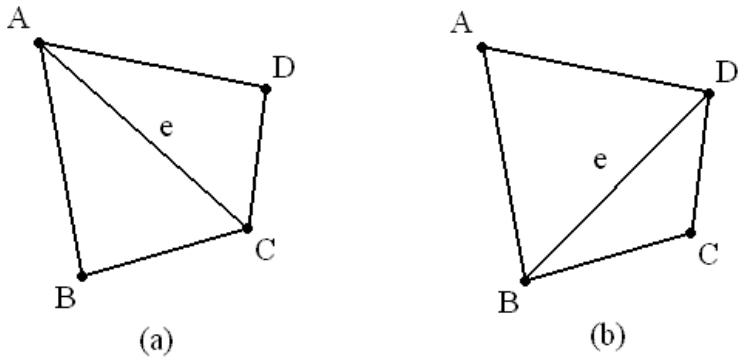

Figura 11. Resultado de aplicar la operación swap sobre dos triángulos que comparten una arista *e*. Partiendo de (a) se alcanza (b) o viceversa.

2.1.3.3 Inserción de un punto

Se trata de un método necesario para insertar un punto *p* que se encuentre dentro de los límites de la caja. Partiendo de esta premisa, se localiza la arista que contenga el punto *p* o que lo deje en su triángulo izquierdo, el procedimiento que lo implementa aparece en el apéndice C. Pueden darse 3 casos:

1) El punto está en la misma posición que algún otro punto ya insertado (extremo de una arista): en ese caso no es necesario modificar nada puesto que el punto ya está insertado.

2) El punto cae en el interior de una cara o triángulo:

   - Se inserta el punto.
   - Se une por medio de aristas con todos los vértices de la cara donde se ha insertado.
   - Si la arista eliminada era restringida, las dos nuevas aristas que la reemplazan (conectadas por medio del vértice insertado) se configuran como restricciones.
   - Se recorren los triángulos formados comprobando si el punto opuesto respecto del insertado se encuentra dentro de la circunferencia determinada por *p*, *e.Org* y *e.Dest*, siendo **e** la arista que comparten el triangulo opuesto y el actual. Si es así se aplica el operador swap a la arista que separa *p* y su opuesto (ver paso de la figura 11a a 11b)
   - En caso de aplicar el operador swap, el paso anterior debe volverse a aplicar en los nuevos triángulos formados.

3) El punto *p* recae en el interior de una arista:

   - Se elimina la arista.

- Se aplican los mismos pasos que en el caso anterior con la excepción de que si la arista original es restringida se deben mantener las dos resultantes (no se puede aplicar la operación swap sobre ellas) y sus listas de identificadores de restricción deben ser iguales a la de la original.

En la figura 12 se muestra la inserción de un punto en una cara (12a). En el paso siguiente (12b) se crean las aristas que unen el nuevo punto con los vértices de la cara triangular en la que está contenido, surgen por tanto 3 nuevos triángulos que hay que evaluar, para ver si cumplen la condición de circuncírculo vacío. El primero de ellos si la cumple (12c) y el segundo no (12d). Se aplica la operación swap y se vuelve a comprobar la propiedad para los nuevos triángulos formados

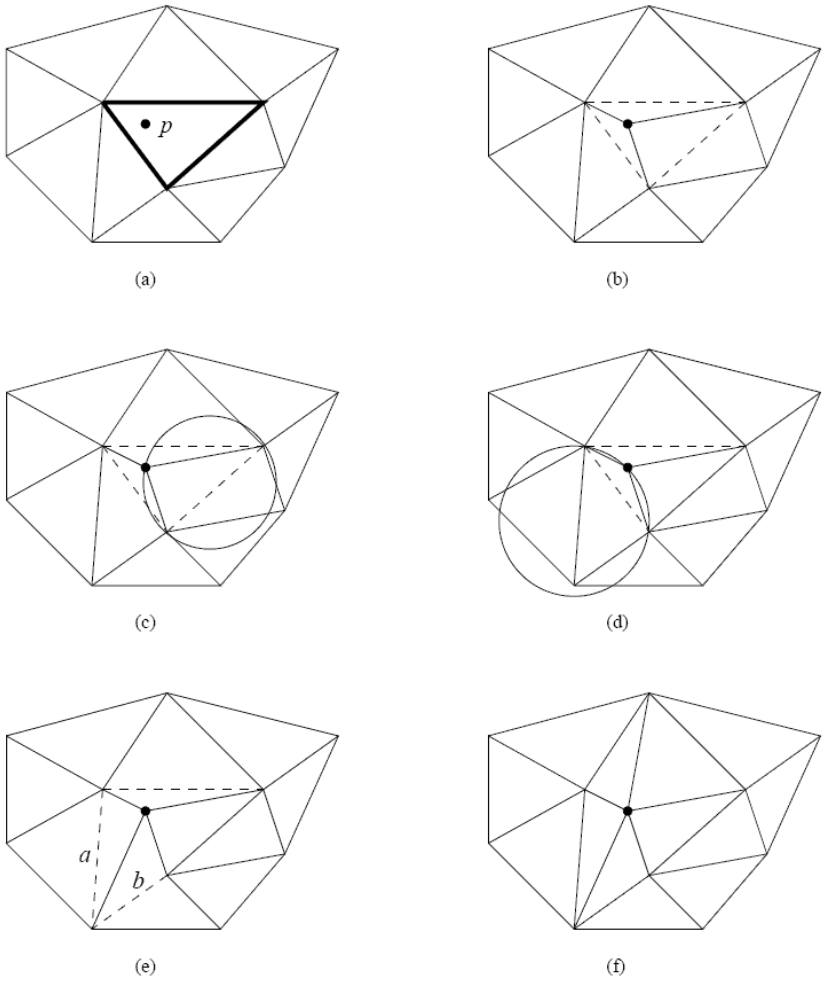

Figura 12. Inserción de un punto en una malla. Las aristas discontinuas son las que deben ser evaluadas para comprobar si la nueva triangulación es de *Delaunay*

2.1.3.4 Inserción de una arista

Para la inserción de una arista ***ab*** se seguirá el método propuesto por [Kallmann et al., 2003] que consta de los siguientes pasos:

1) Se insertan los puntos *a* y *b.*
2) Se calculan todos los puntos de corte de la arista ***ab*** con las aristas restringidas ya presentes y se insertan en la malla.
3) Se eliminan todas las aristas $e_1, \ldots, e_k$ cortadas por ***ab*** de tal modo que la región por la que pasa ***ab*** quede vacía.
4) Se añade la arista ***ab***.
5) Se triangulan las caras a ambos lados de la arista ***ab***.

En la figura 13 se muestra una evolución de los pasos descritos. Partiendo de 13a se determinan los puntos de corte de las aristas restringidas (en negrita) con la arista *ab* (en discontinua) que se desea insertar. En 13b aparecen ya insertados los dos puntos de corte hallados *pc1* y *pc2*. Posteriormente se eliminan todas las aristas cortadas por la nueva arista ***ab***. No se tienen en cuenta las aristas solapadas ni las intersectadas a través de puntos de corte ya insertados, por tanto, debido a los pasos anteriores, todas las aristas intersectadas son no restringidas y por ello se pueden eliminar sin mayor problema. En 13c se muestra la figura con las aristas ya eliminadas. Por último, se crea la nueva arista ***ab*** como arista restringida conectando los puntos ***a*** y ***b*** y los diferentes puntos de corte generados en el paso de 13a a 13b y se triangulan las nuevas caras formadas. En el apéndice C puede encontrarse el método que lo implementa.

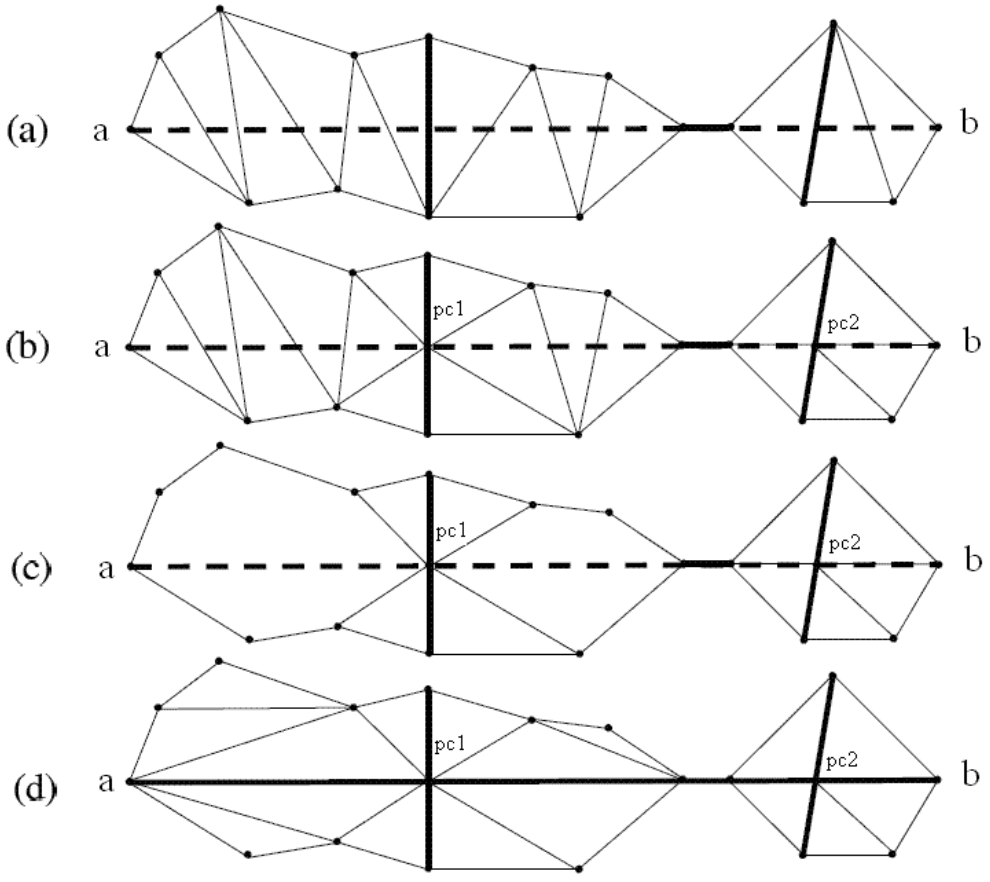

Figura 13. Pasos en la inserción de una restricción o segmento *ab*

El proceso de triangulación de una cara se explica con todo detalle más adelante, en el apartado 3.1.3.6

2.1.3.5 Inserción de una restricción

Puesto que una restricción no es más que una secuencia de puntos $p_1, \ldots p_n$ donde cada par p_i, p_{i+1} representa una arista, el proceso de insertar una restricción se traduce en insertar todos los segmentos que la componen usando el método *InsertSegment* descrito en el apartado anterior. El procedimiento que lo implementa aparece descrito en el apéndice C..

2.1.3.6 Triangulación de una cara

En el proceso de inserción de una restricción o segmento, uno de los pasos consiste en triangular las caras adyacentes a la arista insertada (ver paso de la figura 13c a 13d). Estas caras tienen forma de polígonos y para triangularlas se sigue el enfoque detallado en [Anglada, 1997] que se apoya en la preservación de la propiedad de *circuncírculo* vacío. El procedimiento consiste en recorrer los puntos de la cara diferentes de *ab* (siendo *ab* la arista insertada) y quedarse con aquél cuya circunferencia (la que forma junto con *a* y *b*) no incluya a ningún otro. De este modo se asegura por construcción que la propiedad de *circuncírculo* vacío se sigue cumpliendo. Una vez encontrado este punto, se conecta mediante dos nuevas aristas con los puntos *a* y *b* y se aplica recursivamente el mismo procedimiento para las dos posibles nuevas caras sin triangular construidas. En el apéndice C se muestra el procedimiento que lo implementa.

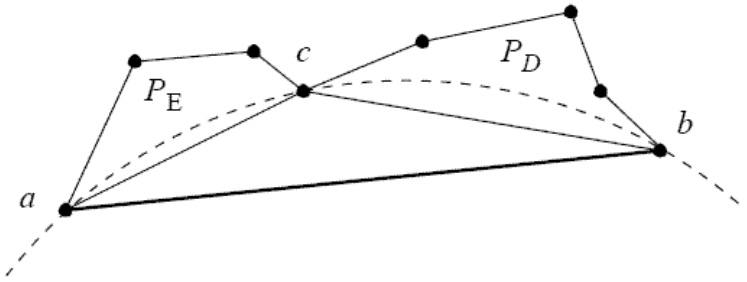

Figura 14. Procedimiento recursivo de triangulación de una cara o polígono *P*. Tras encontrar *c* en *P*, se crea un nuevo triángulo y se vuelven a triangular P_E y P_D.

2.1.3.7 Eliminación de una restricción de la CDT

El proceso de eliminación de una restricción *i* se basa en el descrito en [Kallmann et al., 2003] que consta de dos pasos. El primero busca todas las aristas que formen parte de la restricción *i* y elimina de cada una de ellas la referencia a la restricción *i*. Esto simplemente consiste en eliminar el identificador id_i de la restricción *i* de la lista de identificadores de todas las aristas que lo tengan. En

este punto las aristas afectadas pueden volverse libres (sin restringir, es decir, sin formar parte de ningún obstáculo). El segundo paso consiste en eliminar aquellos vértices que pertenecían al obstáculo eliminado y que posiblemente no estén siendo usados por ningún otro (ver figura 14).

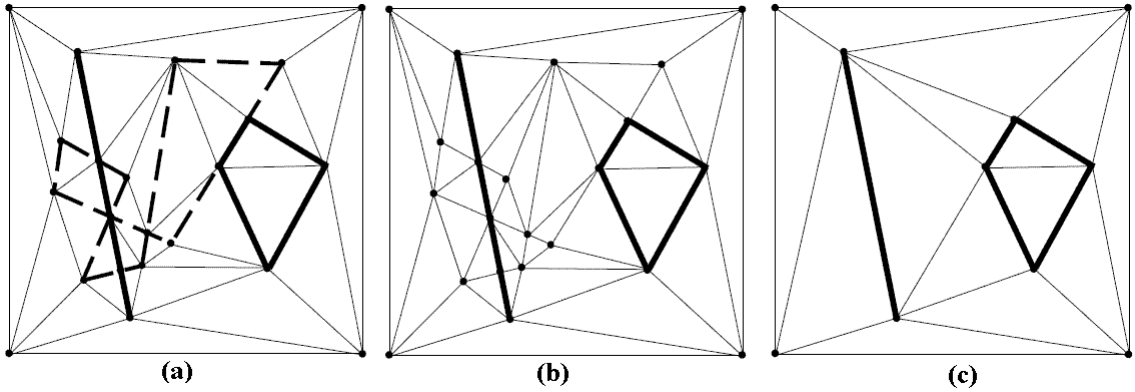

Figura 15. La restricción C que se va a eliminar aparece en trazos discontinuos y en negrita las otras dos con las que intersecta (15a). Una vez encontradas todas las aristas de C se elimina su indicador de aristas de C (15b). Finalmente, los puntos que no son necesarios se eliminan (15c).

Se necesita un vértice *v* de cada restricción *i* para comenzar la búsqueda de todas las aristas que componen la restricción *i*, por ello, cada vez que se inserta un nuevo obstáculo debe guardarse uno de sus puntos en una lista que lo asocie con su respectiva restricción. Esta lista debe mantenerse a lo largo de las diferentes operaciones que se apliquen sobre la malla. En el apéndice C aparece descrito el método que lo implementa.

La eliminación de una restricción se apoya en el método auxiliar *RemoveVertex,* este método elimina un vértice *v* de la CDT así como todos las aristas (restringidas o no) que inciden en *v*. Después de la eliminación, la cara resultante no está triangulada y por tanto necesita ser triangulada de nuevo. Para ello se sigue el enfoque de [Anglada, 1997] explicado en el apartado anterior.
Cuando se elimina un vértice para simplificar dos aristas restringidas colineales, se realiza una llamada al método *InsertSegment* justo después de la eliminación del vértice para crear una nueva arista restringida donde estaban las dos colineales.
La descripción del método RemoveVertex puede encontrarse en el apéndice C.

2.2 Búsqueda sobre la CDT

Una vez construida y actualizada la malla, el proceso de encontrar el camino mínimo que une dos puntos dados se realiza en dos pasos:

1) Se encuentra una secuencia de triángulos adyacentes que comparten aristas sin restringir (canal) que una los dos puntos dados. El algoritmo utilizado es el *TA*,* está basado en un *A** pero con las modificaciones necesarias para que funcione sobre nodos que ahora son triángulos (y por tanto, por tener dimensión, tienen diferentes puntos de entrada y de salida) y no vértices (cuyo punto de entrada y de salida es el mismo).

2) Se calcula el camino mínimo sobre este canal o secuencia de triángulos utilizando el algoritmo de *funnel* (similar a encontrar el camino mínimo que una dos vértices por dentro de un polígono [Chazelle, 1982] [Lee et al., 1984])

Este proceso se repite hasta que ninguno de los posibles canales que queden sin evaluar pueda devolver un camino más corto que el encontrado hasta el momento.

2.2.1 Busqueda *TA*.* Busqueda de un canal

El algoritmo Triangulación *A** o, abreviadamente, *TA** implementa la búsqueda del camino mínimo sobre una malla. Se basa en el algoritmo *A** y por ello, ambas estrategias, mantienen muchas características en común, como por ejemplo, una cola de prioridad o el mantenimiento de costes acumulados así como el de las estimaciones del coste por gastar. En [Demyen, 2006] se puede encontrar amplia información sobre el algoritmo *TA*.*

2.2.1.1 Algoritmo A*

El algoritmo *A** es un algoritmo de búsqueda sobre grafos capaz de encontrar el camino óptimo entre dos nodos. Para ello, se apoya en el coste real consumido hasta un determinado nodo y en el coste estimado desde ese nodo hasta el objetivo. Al coste real para alcanzar un determinado nodo se le llama *g*, y al estimado *h*. La suma de *g* y *h* representa una estimación del coste global (coste del camino desde el inicio hasta el objetivo) y a su valor asociado se le llama *f*. Todo nodo tiene un estado de búsqueda asociado en el que se almacenan estos valores. La idea es seleccionar, en cada iteración de la búsqueda, el nodo por analizar que tenga menor valor *f*, es decir, aquel que estime el menor coste global. De este modo se da prioridad en la exploración a los nodos más prometedores. Cuanto mejor sea el valor *h*, es decir, cuanto más ajustada sea esta estimación y por tanto más aproximada la estimación del coste global, mejor será el rendimiento del algoritmo, puesto que será más acertada la selección del nodo más prometedor. Es decir, la heurística juega un papel clave en el rendimiento.

En cada paso, se generan los sucesores o hijos del nodo seleccionado, se calculan los costes reales de alcanzar cada uno de estos hijos (coste de alcanzar el nodo padre, más el coste de ir desde el padre a su sucesor), y, si no han sido todavía alcanzados por algún otro camino, se almacenan en el conjunto de nodos por explorar. Si, en caso contrario, ya han sido explorados, se comprueba si su coste real asociado es inferior al almacenado y si es así se actualiza el camino con el nuevo mejor encontrado. De este modo, se consigue llevar siempre el camino y el coste mínimo (dentro del espacio explorado) para cualquiera de los nodos ya analizados.

La heurística debe estimar valores siempre inferiores al coste real para que de este modo, la primera vez que se alcance el nodo objetivo se pueda detener la búsqueda puesto que ya se ha encontrado el camino óptimo. Si hubiera otro camino mejor, pasaría por algún nodo mejor, es decir por algún nodo que tuviera un coste global estimado *f* inferior y por tanto durante el proceso de elección del nodo más prometedor hubiera sido seleccionado. Como esto no ha sucedido porque todos los nodos

de la cola presentan valores *f* superiores, y *f* es una cota inferior del coste real, se deduce que todos los caminos que quedan por explorar no pueden mejorar el ya encontrado.

2.2.1.2 Algoritmo *TA**

En esta adaptación del *A*,* cada triángulo puede entenderse como la representación geométrica de un nodo. Sin embargo no basta con hacer esta abstracción sino que es necesario tener en cuenta sus implicaciones en la implementación. A un triángulo se accede por alguna de las tres aristas que lo determinan, y por tanto el punto de entrada final puede ser cualquiera de los infinitos que componen estas aristas. En concreto aquel que minimice el camino global, sin embargo este punto no puede ser determinado en el momento en que se alcanza el triángulo actual sino más adelante. Esta consideración se explicará con más detalle a continuación, dentro de los pasos que componen el algoritmo.

En primer lugar se encuentra el triángulo contenedor del punto inicial. Para ello se utiliza el procedimiento *Locate* descrito con detalle previamente. Este primer triángulo se corresponde con el nodo o estado inicial de la búsqueda y por tanto sus costes acumulado y estimado son respectivamente 0 y la distancia euclídea entre el punto inicial y el final. La heurística adoptada para determinar el valor *h* de un nodo cualquiera es la distancia euclídea entre el punto objetivo y su punto más cercano a la arista de entrada del nodo (triángulo). Esta heurística es conocida por ser consistente y admisible y por tanto es válida para el planteamiento del problema. En cada paso se elige el estado con *f* mínima, se generan sus sucesores así como sus valores *g* y *h*.

En este paso es muy importante destacar la gran diferencia respecto al algoritmo *A**. En el *A**, los costes imputados hasta un nodo pueden ser llevados de forma exacta, sin embargo en una CDT, es imposible conocer el coste en cada nodo puesto que no se puede saber el punto de entrada de un estado o triángulo hasta saber por dónde se alcanza el punto final. Es decir, se puede calcular exactamente el coste hasta cualquiera de los puntos de la arista de entrada de un triángulo, lo que no se puede saber es cual, de todos los puntos de esa arista, será finalmente el punto de entrada, puesto que según la ubicación del punto objetivo final, puede resultar mejor entrar por un punto u otro de la arista.

En la figura 16 se pueden observar diferentes caminos desde el mismo punto inicial por los mismos triángulos (mismo canal) pero con diferentes puntos de entrada según dónde se encuentre finalmente el punto objetivo.

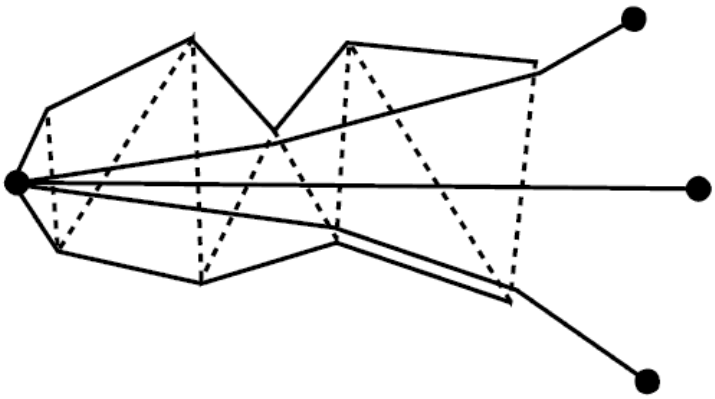

Figura 16. El camino hasta un triángulo depende de por donde continúe el camino

Debido a este inconveniente, el valor *g* no puede ser calculado con exactitud y por tanto lo que se tiene realmente es una estimación del coste real. La estimación *g*, al igual que el valor *h*, debe ser una cota inferior del coste real para preservar las propiedades de admisibilidad y consistencia y poder posteriormente realizar una poda sobre el espacio de búsqueda.
Tampoco tiene sentido mantener un conjunto con los nodos explorados, sus caminos y costes mínimos, puesto que como se ha dicho, no se puede saber el punto de entrada y por tanto no se puede garantizar que los datos almacenados sean los óptimos.

Por tanto el algoritmo *TA** se reduce a una búsqueda voraz guiada únicamente por la heurística. Por ello se deben ajustar al máximo las estimaciones porque causan un gran impacto sobre el rendimiento del algoritmo. Cuanto más ajustados sean *g* y *h* más amplia será la poda y más se podrá acotar el espacio de estados. Para el valor *h* se calculará la distancia euclídea entre el objetivo y su punto más cercano de la arista de entrada. Para el valor *g* se calcularán varias cotas inferiores y se seleccionará la máxima de ellas por ser la que más se ajuste al valor real. Las cotas inferiores de *g* evaluadas son:

- Valor *g* del nodo padre
- Distancia euclídea entre el punto inicial y su más cercano de la arista de entrada
- Distancia real al punto más cercano: se calcula mediante el algoritmo de *funnel* que se detallará más adelante, la distancia exacta desde el punto inicial hasta el punto de la arista de entrada que minimice el coste.

En el apéndice C se detalla el procedimiento que lo implementa.

En la figura 17 se muestra un canal (sucesión de triángulos adyacentes) en azul que contiene un camino en verde. En este caso el camino es óptimo. El algoritmo TA*, partiendo de un nodo (triángulo) inicial ha ido expandiendo sus sucesores (triángulos adyacentes) hasta encontrar un sucesión de nodos que contiene un camino. Para ello se apoya en un procedimiento conocido como algoritmo de *funnel* capaz de calcular las distancias dentro de un polígono triangulado.

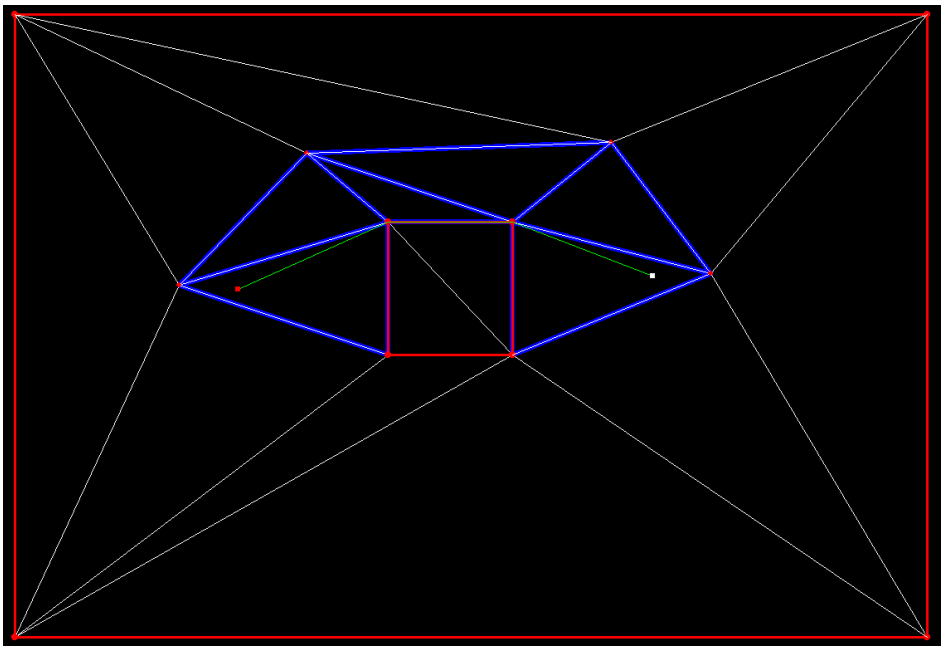

Figura 17. Secuencia de triángulos que contiene un camino óptimo en una CDT. El canal se muestra en azul y en rojo las restricciones.

2.2.2 Algoritmo de *funnel*

El algoritmo TA* va produciendo sucesiones de triángulos adyacentes sobre las que se puede trazar un camino desde el punto inicial, contenido en el primer triángulo, hasta el punto final, contenido en el último triángulo. Sobre estas sucesiones o canales se debe calcular la ruta mínima exacta. El problema es similar al de encontrar el camino mínimo interior entre dos puntos de un polígono triangulado sin restricciones internas (no puede haber obstáculos puesto que, por construcción, las regiones internas de los triángulos son libres y el TA* sólo salta de un triángulo a otro si la arista que comparten no es restringida). Un algoritmo capaz de resolver este problema es el algoritmo de *funnel*. En [Demyen, 2006] se puede encontrar amplia información sobre este procedimiento.

El algoritmo de *funnel* (embudo en inglés) se basa en la idea de construir una estructura capaz de delimitar el rango o espacio por el que puede extenderse el camino mínimo. Se parte del punto inicial y se van recorriendo los triángulos del canal. Cada nuevo triángulo puede 'abrir' o 'cerrar' el embudo. Por abrir se entiende que el rango de espacio por el que puede oscilar el camino se amplia, y por cerrar, que el rango se estrecha y por tanto el embudo se hace más angosto.

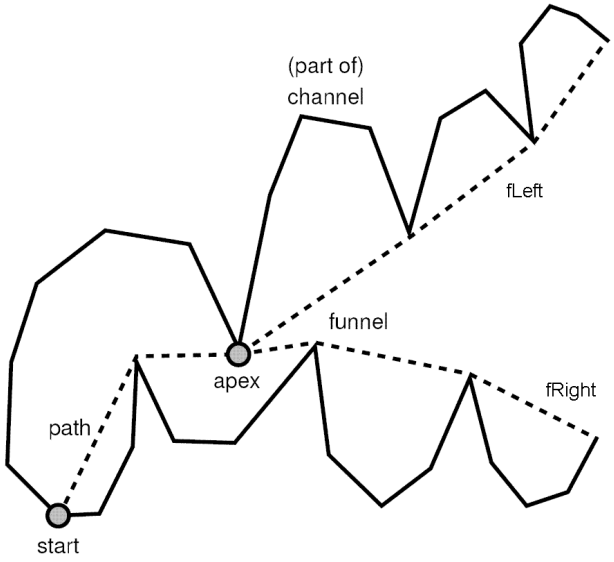

Figura 18. Estructura del algoritmo de *funnel*. Las dos líneas discontinuas *fLeft* y *fRight* que parten del elemento apex delimitan el espacio por el que puede oscilar el camino mínimo hasta ese instante.

Los elementos necesarios para soportar esta estructura son:

- *Path*: la lista de puntos que guarda la parte del camino ya definida.
- *Funnel*: dos listas de puntos que delimitan el embudo (parte izquierda y parte derecha). Ambas listas comparten el mismo punto inicial del que divergen. Les llamaremos *fLeft* y *fRight* respectivamente o *funnel* izquierdo y *funnel* derecho.
- *Apex*: punto de unión entre el *path* y el *funnel*. Es el primer elemento tanto de la parte izquierda como de la derecha del embudo.

La figura 18 muestra las tres listas empleadas en el algoritmo de funnel, path, fLett y fRight con 3, 4 y 4 puntos respectivamente. Las tres listas tienen un mismo punto en común: el apex.

El procedimiento es el siguiente: el apex se inicializa con el punto de partida, el *path* comienza vacío y se añade el apex tanto al *funnel* izquierdo como al derecho. Además, se añaden también los puntos extremos de la arista que separa el triángulo inicial del siguiente (el extremo izquierdo respecto del canal en el *funnel* izquierdo y el extremo derecho en el *funnel* derecho). Posteriormente se van recorriendo las aristas restantes que unen cada triángulo con el siguiente. Cada una de estas aristas mantiene en común uno de sus extremos con uno de los últimos elementos insertados en el *funnel* izquierdo y el derecho. Es decir, si se define el origen de la arista como el extremo que incide en la parte izquierda del canal y el destino como el que toca la parte derecha, en cada paso, o el origen coincide con el último punto insertado en *fLeft* o el destino coincide con el último punto insertado en el *fRight*. Se escoge el nuevo punto a analizar, aquel que no coincide, para esta explicación supongamos que el nuevo punto es el origen de la arista y por tanto está más próximo al lado izquierdo del canal. El siguiente paso es determinar si el nuevo punto 'cierra' el lado izquierdo del embudo. Es decir, se evalúa si está más hacia el interior del embudo que el anterior, si no es así, se añade el punto al lado izquierdo *fLeft*. En caso contrario, se van eliminando los puntos de *fLeft*

hasta que el nuevo punto quede más hacia el exterior del embudo y finalmente se añade. Las comprobaciones para saber si un punto queda más hacia el exterior o hacia el interior del embudo consisten en determinar si quedan a la izquierda o a la derecha del eje formado por los dos últimos puntos del lado correspondiente del embudo.

Cuando el nuevo punto de *fLeft* ha 'cerrado' el embudo, puede suceder que quede tan en el interior que el nuevo lado izquierdo *fLeft* intersecte con el derecho *fRight*, por ello, en este caso, se dice que el embudo se ha colapsado, y se debe 'avanzar' el apex hasta el punto en el que ambos lados del embudo vuelven a divergir hasta el final. Para ello se va recorriendo el lado derecho *fRight* desde el apex hasta que el nuevo punto insertado en *fLeft* quede a la izquierda del eje formado por el *apex* y su siguiente. Todos los puntos de *fRight* que se van descartando en este último paso, se van añadiendo a la lista path. El nuevo apex alcanzado se actualiza también en *fLeft*.

El proceso es análogo para el caso en el que el nuevo punto se deba insertar en *fRight*.

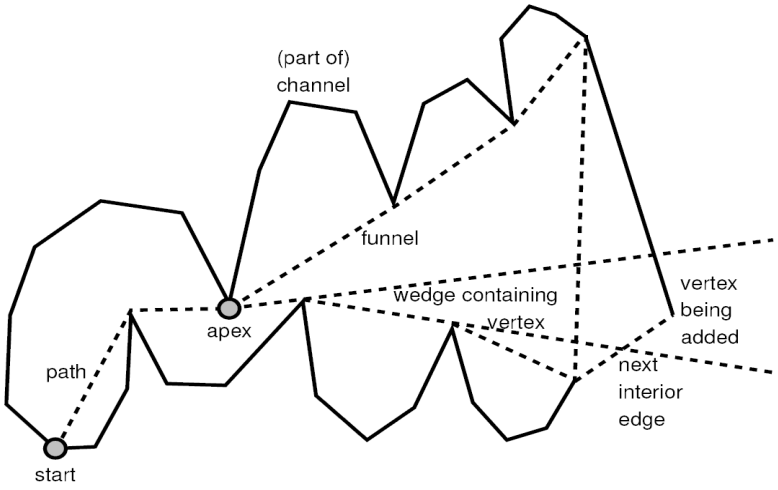

Figura 19. Añadiendo un punto al lado izquierdo del embudo

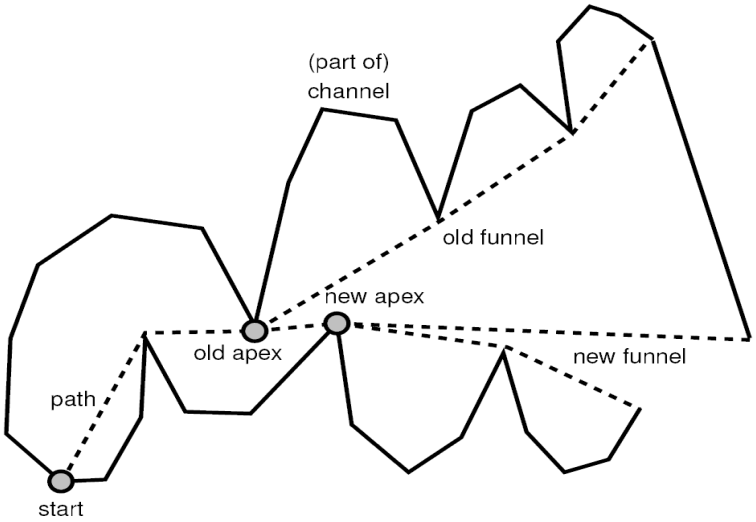

Figura 20. Nuevo embudo y nuevo apex tras añadir un punto en *fLeft*

En las figuras 19 y 20 se observa la inserción de un punto en el lado izquierdo del embudo. En este caso el embudo se colapsa y es necesario avanzar el apex y construir un nuevo embudo que parta de él.

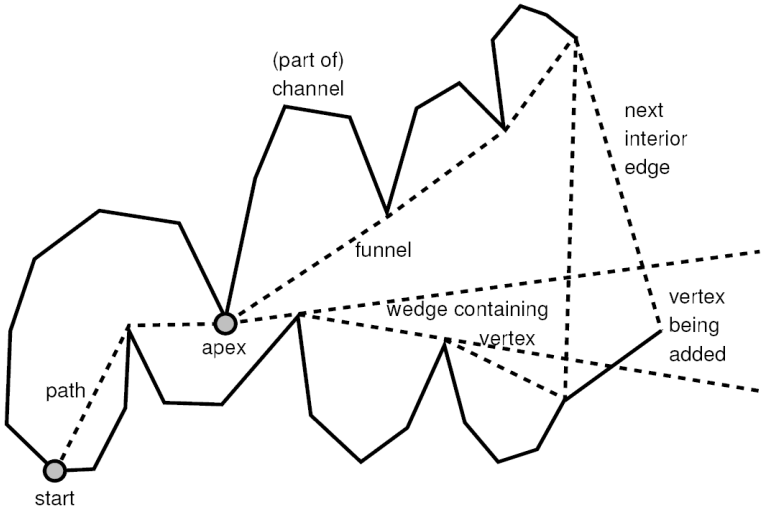

Figura 21. Añadiendo un punto en el lado derecho del embudo

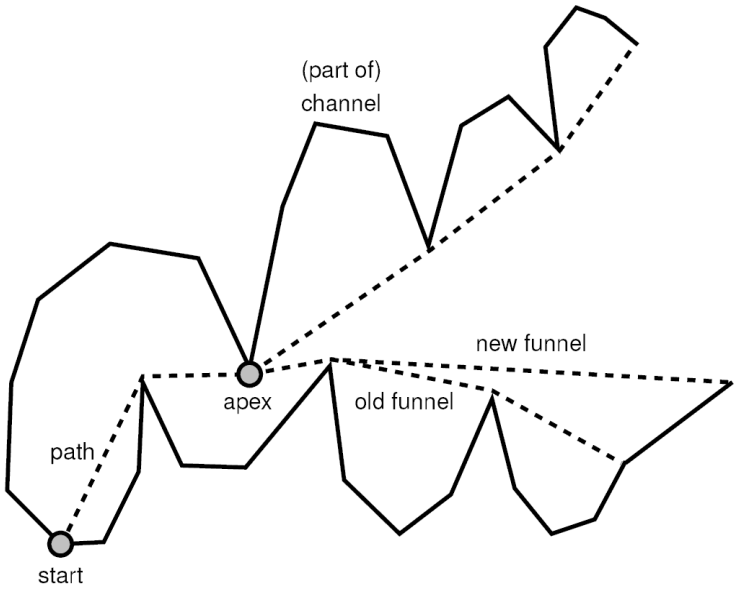

Figura 22. Nuevo embudo tras añadir un punto en *fRight*

En las figuras 21 a 22 se inserta un nuevo punto en la parte derecha que 'cierra' este lado del embudo pero sin llegar a colapsarlo. Por ello sólo es necesario modificar la lista correspondiente a la parte derecha del embudo, fRight, mientras que el apex y la parte izquierda, fLeft, permanecen invariables. En el apéndice C aparece la implementación de este algoritmo.

3 Razonamiento basado en casos

En el apartado anterior se ha descrito el planteamiento adoptado para representar el espacio y los algoritmos implementados para realizar búsquedas de caminos sobre él. En este apartado se describe el sistema experto capaz de razonar sobre la información representada y de tomar la decisión o actuación adecuada en cada instante con el objetivo de neutralizar la intrusión. En concreto, el sistema experto implementado consiste en un razonamiento basado en casos o CBR (Case-Based Reasoning). A continuación se hace una pequeña introducción sobre los razonamientos basados en casos y en los siguientes subapartados se explica la representación de casos empleada y los diferentes métodos utilizados en la fase de recuperación, a saber, árboles de decisión y funciones de similitud con coeficientes sin ajustar y ajustados mediante algoritmos genéticos.

El CBR es una técnica de resolución de problemas que se basa en la manera de razonar del hombre. Muchos estudios en Psicología afirman que la mente del ser humano trata de resolver determinadas situaciones utilizando información específica de experiencias anteriores [Ross, 1989]. De esta manera, en un CBR la resolución de un nuevo problema se basa en encontrar casos pasados similares y reutilizarlos, adaptando así una solución anterior al nuevo problema. Entre las ventajas destaca la incorporación a la base de conocimiento del sistema, de los casos nuevos incrementando así la experiencia acumulada y poniéndola a disposición de los problemas futuros, de este modo, a medida que se resuelven nuevos problemas se mejorarán los resultados obtenidos en problemas pasados.

Un CBR está compuesto generalmente por cuatro fases: recuperación, reutilización, revisión y recuerdo [Aamodt et al., 1994].

En la fase de recuperación se determina qué características del caso nuevo son las que van a permitir encontrar los casos relevantes de la base de casos del sistema. Después se accede a la memoria para recuperar los casos más similares, ordenándolos a partir del grado de similitud y escogiendo aquéllos cuyo grado de similitud sea superior a un valor umbral o simplemente se selecciona aquél caso que presente mayor semejanza. El concepto de semejanza se define posteriormente.

En la fase de reutilización se utiliza el conocimiento incluido en el caso recuperado para resolver/clasificar el problema/situación actual. Aquí existen dos posibles alternativas, se puede ofrecer simplemente la solución recuperada sin modificar o se puede adaptar la misma a la situación actual. Esto último requiere encontrar las diferencias entre el caso recuperado y el actual y aplicar algún mecanismo que sugiera cambios en función de esas diferencias encontradas.

En la fase de revisión se pone a prueba la solución propuesta, ya sea siendo evaluada por un experto, o aplicándola directamente a un sistema real. En caso de no ser una solución adecuada se incluye la información al respecto, reparando la solución o incorporando algún mecanismo en el sistema que se encargue de realizar alguna estrategia de corrección.

En la fase de recuerdo se integra, en la base de casos del sistema, el nuevo caso, con sus características más relevantes y su solución. Es esta fase la que hace que el sistema CBR mejore su funcionamiento a medida que va adquiriendo nuevas experiencias.

De estas cuatro fases, la que tiene más importancia en este proyecto es la de recuperación ya que será crucial encontrar el caso que más se parezca a la situación de intrusión analizada. Esta etapa comienza con una buena descripción o representación del problema, y finaliza cuando se encuentra el caso que más se ajusta.

En la recuperación se trata de identificar las variables más relevantes del problema y devolver un conjunto de casos suficientemente similares al nuevo en función del criterio de similitud escogido. Los procesos involucrados en la recuperación de un caso de una base de casos son muy dependientes del modelo de memoria y de los procedimientos de indexación usados. A continuación se describen someramente los principales criterios de similitud y algunos de los algoritmos más utilizados como son k-vecinos más cercanos, árboles de decisión y sus derivados [Pal & Shiu, 2004]. Estas técnicas necesitan de una medida de similitud para determinar la proximidad entre casos; por ejemplo, la distancia euclídea, de hamming o de levenshtein.

En el algoritmo de k-vecinos más cercanos el caso recuperado es aquél cuyas características se ajusten más a las del caso comparado. Por características debe entenderse al conjunto de variables o propiedades en las que se ha dividido el caso para su representación. Las características tienen un peso asociado y se combinan mediante una función para obtener un valor global. Si los pesos de las características son iguales, un caso que ajusta con n características será recuperado antes que otro caso que empareja con *k* características, siendo $k < n$. Se puede asignar un peso mayor a las características consideradas más importantes.

Una función de evaluación empleada con frecuencia para encontrar el vecino que mejor se empareja o ajusta es la siguiente:

$$similarity\ \ (Case_I, Case_R) = \frac{\sum_{i=1}^{n} w_i \times sim(f_i^I, f_i^R)}{\sum_{i=1}^{n} w_i} \qquad (1)$$

Donde w_i es el peso o importancia de una característica, *sim* es la función de similitud de las características y f_i^I y f_i^R son los valores del caso de entrada y del recuperado respectivamente para una característica *i*.

La figura 23 muestra un esquema simple de definición de la similitud basado en el vecino más cercano. En el ejemplo se seleccionaría el caso 3 como el vecino más cercano al nuevo caso (NC) porque:

similarity(NC, case3) > similarity(NC, case1) and similarity(NC, case3) > similarity(NC, case2).

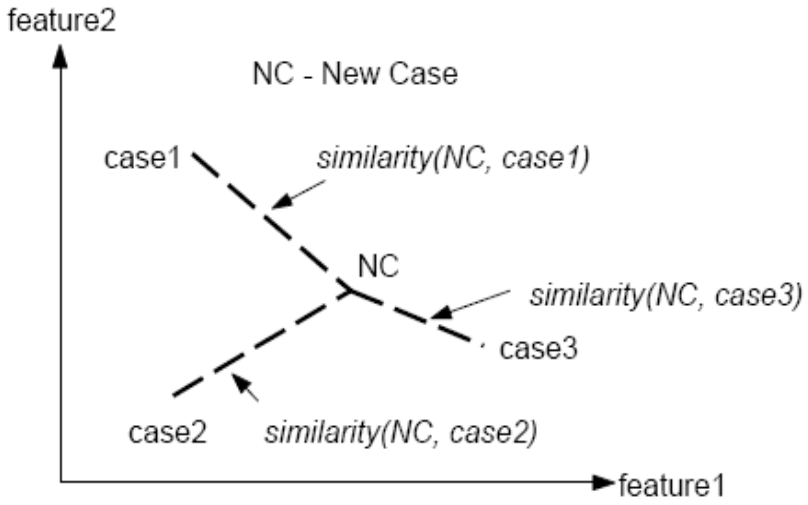

Figura 23. Estructura vecino más cercano

Además del método k-vecinos, otra de las estrategias más usadas son los árboles de decisión. Con los procedimientos de inducción (aprendizaje automático) es posible extraer reglas o construir árboles de decisión a partir de datos pasados. Cuando se utilizan árboles de decisión en la fase de recuperación es necesario determinar qué características son las que mejor diferencian los casos entre sí para generar (inducir) un árbol de decisión que clasifique (indexe) los casos. El algoritmo inductivo más utilizado es ID3 [Quinlan, 1986].

En la figura 24 se muestra el árbol de decisión generado para los datos de la tabla 1. La tarea es cual sería la actuación correcta a partir de características de un escenario de intrusión(relación entre el número de guardias y el de intrusos, estado de la alarma, dirección de los intrusos, distancias entre elementos clave de la escena…).

	Descripción					**Solución**
Caso	**Relación guardias-intrusos**	**Alarma**	**Dirección de los intrusos**	**Distancia intrusos-objetivo**	**Resto de propie-dades**	**Actuación**
1	Igual	Apagada	Objetivo	Lejana	…	Encender alarma
2	Menos	Apagada	Salida	Cercana	…	Encender alarma
3	Igual	Encendida	Objetivo	Cercana	…	Capturar intrusos
4	Igual	Encendida	Objetivo	Cercana	…	Capturar intrusos
5	Menos	No disponible	Objetivo	Cercana	…	Ir al objetivo
6	Menos	No disponible	Objetivo	Media	…	Ir al objetivo
7	Más	No disponible	Salida	Media	…	Ir a la salida
8	Igual	No disponible	Salida	Lejana	…	Ir a la salida
9	Menos	Encendida	Objetivo	Cercana	…	Ir a los intrusos
10	Menos	No disponible	Objetivo	Cercana	…	Ir a los intrusos

Tabla 1. Casos de intrusión

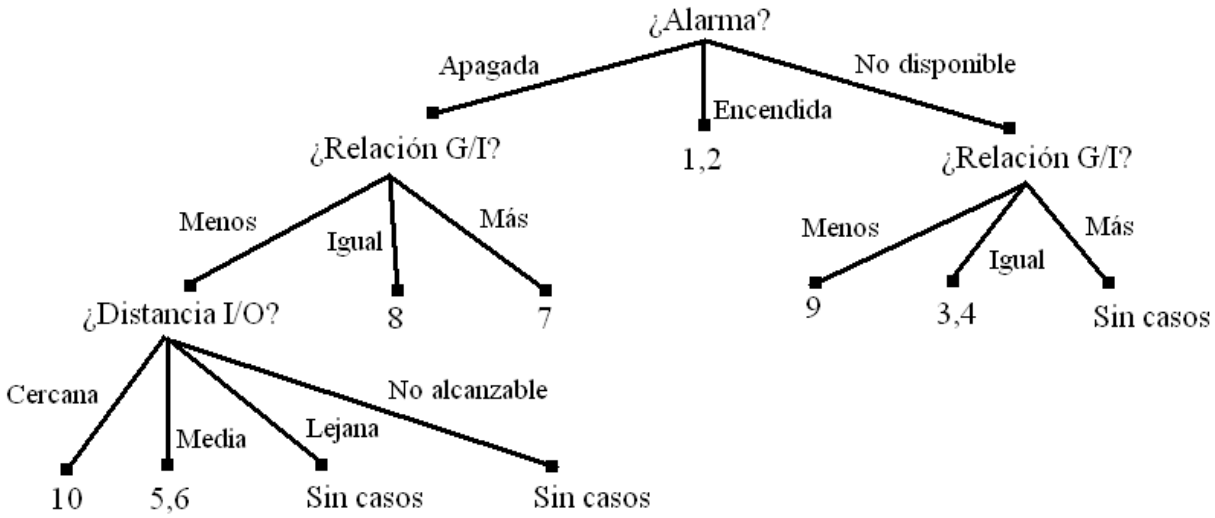

Figura 24. Árbol de decisión para casos de intrusión

Una vez construido el árbol de decisión (figura 24), la clasificación de un nuevo caso se realiza seleccionando la rama que se ajuste al valor del atributo asociado a cada nodo, hasta alcanzar un nodo hoja que representa la clase. Los nodos hoja contienen una lista de casos que cumplen las condiciones que figuran en los nodos intermedios desde la raíz. Además, al pertenecer todos los casos de la misma hoja a la misma clase, todos ellos comparten el mismo valor para el atributo-objetivo.

El vecino más cercano es una técnica muy simple que proporciona una medida de similitud entre un caso objetivo y un caso almacenado en la base. Su principal desventaja es la velocidad de recuperación ya que para encontrar el caso más similar, el caso objetivo debe compararse con cada caso de la base de casos. Esto quiere decir que una comparación de similitud (tal como una distancia) debe calcularse para cada característica indexada, lo que hace que el algoritmo sea ineficiente a medida que aumenta el número de casos almacenados. Por ejemplo, para una base de 100 casos con 10 características indexadas habría que realizar 1.000 cálculos de similitud. Si el tamaño de la base aumenta a 10.000 casos sería necesario realizar 100.000 cálculos de similitud.

La principal ventaja de los CBR proviene del hecho de que habitualmente es más fácil abordar la descripción de las características de un problema que formalizar las reglas que lo resuelven. Cuando el sistema a estudiar es muy complejo, resulta más sencillo recoger todas las experiencias anteriores junto con sus soluciones y tratar de inferir un conocimiento general a partir de lo particular (inducción) que tratar de encontrar una serie de reglas generales que expliquen y resuelvan el sistema (deducción).

Entre las principales desventajas de los sistemas CBR está su dependencia de la caracterización de problemas y de la medida de similitud que se emplee en la recuperación de casos. Un mal diseño en este punto puede hacer que los casos recuperados no sean los más similares al caso nuevo en lo que a la resolución del problema se refiere y, por lo tanto, que los resultados no sean en ningún caso satisfactorios. Otro punto débil es su gran dependencia de la 'calidad' de los datos de entrenamiento.

Si no existe una cierta diversidad en la gama de ejemplos de entrenamiento, el CBR no tiene base para aprender y su rendimiento se resiente. Este sesgo inductivo es bastante importante y debe ser tenido en cuenta a la hora de entrenar el sistema de razonamiento. Estas circunstancias son las que aparecen en la aplicación objeto de este trabajo.

3.1 Representación de un caso

Para poder evaluar correctamente un caso, el primer paso consiste en definir cuáles son los parámetros que lo representan. Es decir, las características sobre las que se apoyaría un experto para caracterizar la situación y posteriormente decidir entre todas las posibles soluciones. Decidir qué almacenar en la representación de un caso es un proceso muy importante puesto que la estructura de éste condicionará fuertemente todo el diseño del sistema [Aamodt et al., 1994].

En nuestro caso, la situación inicial de la que se parte es el plano de un edificio con las posiciones de los intrusos, guardias, posibles objetivos (o zonas sensibles) y salida. En la figura 25, se puede observar una representación espacial de un escenario. Las líneas rojas representan obstáculos, están delimitadas por segmentaciones del espacio y no son relevantes. Las posiciones de los guardias, intrusos, salida y objetivo están unidas por líneas de colores que representan el camino más corto entre ellas. También se dispone de información sobre el estado de la alarma (si está activada, en silencio o sonando), la intención de los intrusos (si van en dirección al objetivo o a la salida) y las distancias entre los elementos de la escena. En la figura 26 se puede observar la configuración de algunos de estos parámetros.

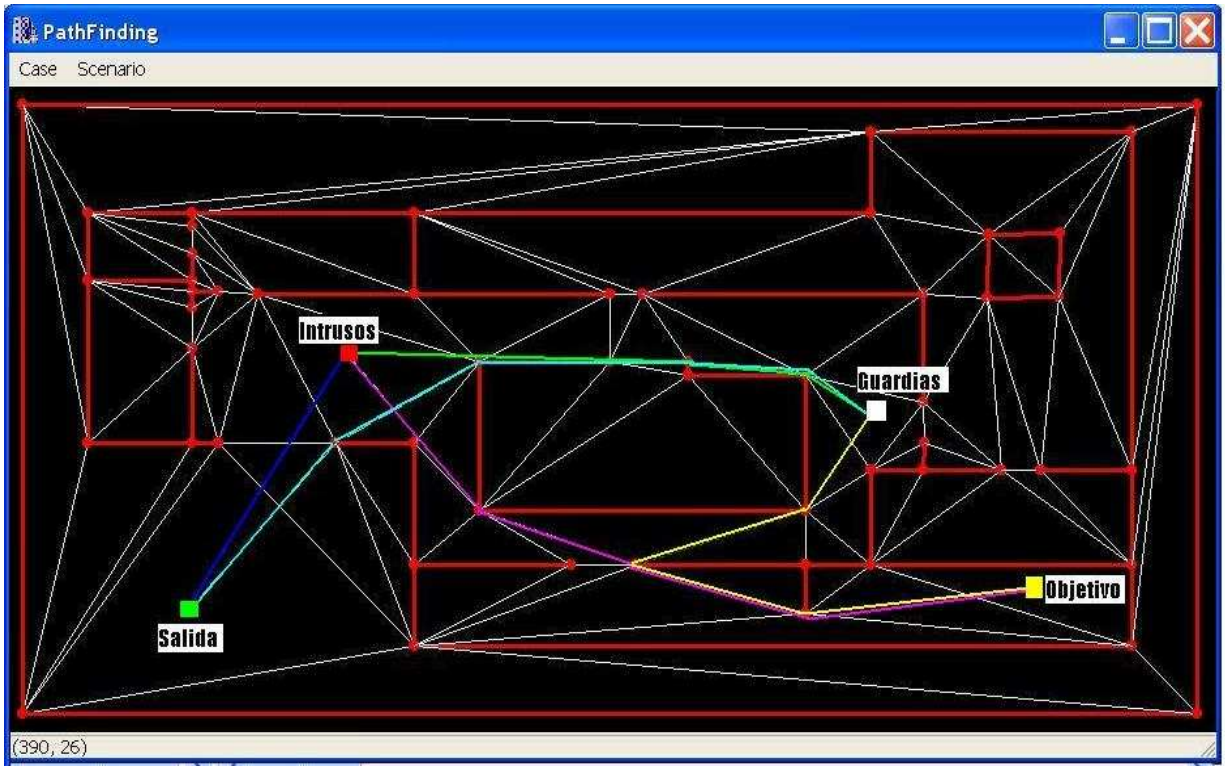

Figura 25. Representación espacial de una escena

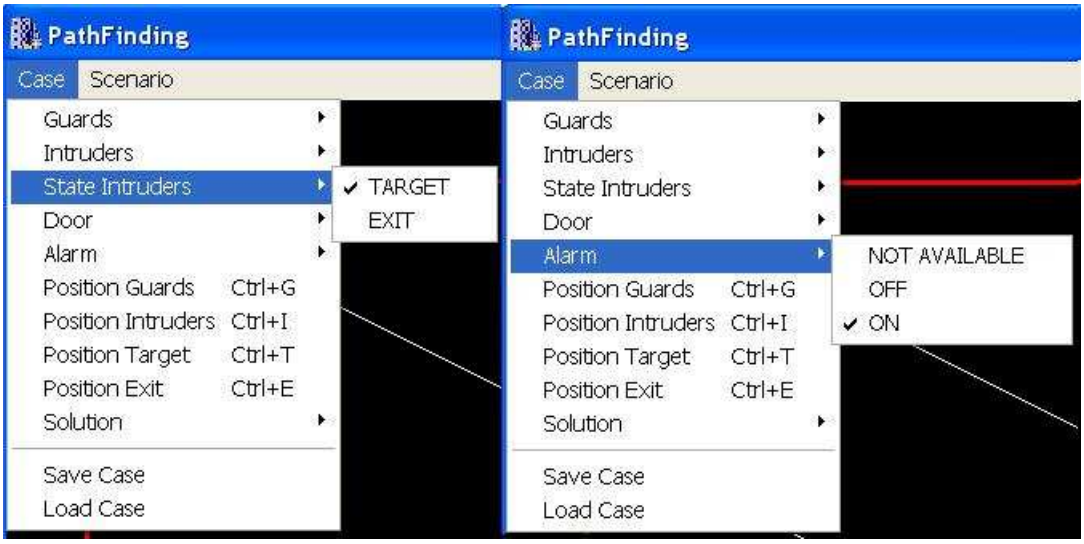

Figura 26. Configuración del estado y la alarma

A partir de toda esta información, la representación del caso que resulta más natural es la siguiente:

<nG, nI, state, door, alarm, dIT, dIE, dGI, dGT, dGE>

donde:

- nG: número de guardias
- nI: número de intrusos
- state: intención de los intrusos
- door: estado de la puerta
- alarm: estado de la alarma
- dIT: distancia entre intrusos y objetivo (Target)
- dIE: distancia entre intrusos y salida (Exit)
- dGI: distancia entre guardias e intrusos
- dGT: distancia entre guardias y objetivo
- dGE: distancia entre guardias y salida

Sin embargo, si se analiza un poco más exhaustivamente el problema, se puede reducir la información de las variables *nG* (número de guardias) y *nI* (número de intrusos) a una única, denominada ratio, que almacena la proporción entre guardias e intrusos, que es lo que realmente valoraría un experto.

Por tanto, si incluimos este cambio y además añadimos el atributo-solución asociado a cada caso, la representación final del caso quedaría como sigue:

<ratio, state, door, alarm, dIT, dIE, dGI, dGT, dGE, opSol>

donde:

- ratio: nG/nI
- opSol: tipo o clase de la solución a aplicar (actuación más adecuada)

3.2 Discretización de valores

En el ejemplo que nos ocupa, los valores considerados para cada atributo se muestran en la tabla 2:

Atributo	**Valores**	**Descripción**
ratio	LESS	Menos guardias que intrusos
	MORE	Igual número de guardias que de intrusos
	EQUAL	Más guardias que intrusos
state	TARGET	Los intrusos se dirigen hacia el objetivo
	EXIT	Los intrusos se dirigen hacia la salida
door	NOT_AVAILABLE	No existe ninguna puerta entre intrusos y su objetivo
	CLOSED	La puerta entre los intrusos y su objetivo está cerrada
	OPEN	La puerta entre los intrusos y su objetivo está abierta
alarm	NOT_AVAILABLE	La alarma no está disponible
	OFF	La alarma no está sonando
	ON	La alarma está sonando
distances	CLOSE	Distancia cercana entre dos puntos
	MEDIUM	Distancia media entre dos puntos
	FAR	Distancia lejana entre dos puntos
	NOT_REACHABLE	No existe ningún camino entre el origen y el destino
opSol	GO_EXIT	Los guardias deben ir a la salida
	GO_TARGET	Los guardias deben ir al objetivo
	TURN_ON_ALARM	La alarma debe comenzar a sonar
	CAPTURE_INTRUDERS	Los guardias deben atacar y capturar por los guardias
	GO_INTRUDERS	Los guardias deben ir a la posición de los intrusos
	CLOSE_DOOR	Se debe cerrar la puerta disponible

Tabla 2. Lista de valores que toman las variables

Las distancias calculadas sobre el plano del recinto se discretizan en función del tamaño del propio recinto. Dadas dos posiciones sobre el plano, si no existiera ningún camino entre ellas, se asigna a la distancia entre ambas el valor *NOT_REACHABLE*. Si existiera algún camino, se calcula la distancia sobre el mínimo, y si ésta es menor que 1/3 de la diagonal del plano, entonces se asigna

el valor *CLOSE*, si está entre 1/3 y 2/3, el valor que corresponde es *MEDIUM*, finalmente, si es mayor de 2/3 el valor es *FAR*.

Para los datos mostrados en las figuras 25 y 26 la representación o instancia concreta del caso tendría la siguiente forma:

< EQUAL, TARGET, NOT_AVAILABLE_D, ON, MEDIUM, CLOSE, MEDIUM, MEDIUM, MEDIUM, UNDEFINED* >

*El valor del atributo-solución es igual a *UNDEFINED* porque no se le asignó ningún valor por defecto.

3.3 Función de similitud

Dada una base de conocimiento o base de casos *BC* de *n* elementos o casos definida como:

$$BC = \{C_1, ..., C_n\}$$

se debe encontrar una función de similitud capaz de medir el grado de relación entre dos casos. Esta relación se mide en términos de semejanza.

3.3.1 Coeficientes equiponderados

Respecto a la función de similitud global, su forma general es la siguiente [Althoff et al., 1995] :

$$SIM(A, B) = F(sim_1(a_1, b_1), sim_2(a_2, b_2), ..., sim_p(a_p, b_p)) \quad (2)$$

donde *A* y *B* son dos casos, *p* es el número de atributos del caso que se tienen en cuenta para medir la similitud y $F:[0,1]^p \rightarrow [0,1]$. El valor de máxima similitud es 1 y el de mínima 0. La función *sim* es la función de similitud local, es decir atributo por atributo, que se aplica para comparar los valores de los atributos internos de un caso.

Puesto que se ha realizado una discretización y se está trabajando con atributos simbólicos ordenados y univaluados, la función de similitud local entre dos atributos a y b se puede definir mediante la siguiente expresión [Althoff et al., 1995]:

$$sim(a, b) = 1 - \frac{|ord(a) - ord(b)|}{card(O)} \quad (3)$$

donde *a* y *b* son los valores de los atributos comparados, *O* el conjunto de todos los posibles valores que puede tomar el atributo, *ord(a*) y *ord(b)* sus respectivos ordinales dentro de su tipo de datos y *card(O)* la cardinalidad del tipo de datos.

Teniendo en cuenta la expresión (3) la similitud global SIM entre dos casos *A* y *B* con la misma representación (los mismos atributos) se puede expresar de la siguiente forma:

$$SIM\ (A, B) = \frac{1}{p} \sum_{i=1}^{p} sim_{i}(a_i, b_i) \qquad (4)$$

siendo:

$$\sum_{i=1}^{p} \frac{1}{p} = 1$$

Como *p* es el número de atributos que se comparan, en esta primera aproximación, se ha optado por asignar el mismo peso a todos los atributos (*1/p*). En la figura 27 se muestra un listado de todos los valores de similitud calculados para un mismo caso sobre otros 10. Se puede observar el caso comparado, el más similar y todos los valores de similitud respecto del primero.

Es interesante destacar que la función de similitud global no está ajustada en función de la base de conocimiento y por tanto, la similitud global es la suma equiponderada de las similitudes parciales o locales de todas las propiedades que componen el caso. Es decir, a todas las similitudes parciales se les aplica el mismo valor o peso (1/p), lo que se traduce en que todos los atributos tienen la misma importancia a la hora de valorar la similitud entre casos, cuando en la realidad no debería ser así, ya que existen atributos con mayor relevancia que otros. Por consiguiente, para cada atributo o propiedad del caso, no se está aplicando el valor o peso verdadero que se merece en función de su importancia para la similitud entre casos.

Para solventar este problema, se decide abordar la similitud entre casos con otra técnica: ajuste de coeficientes mediante algoritmos genéticos. En nuestro caso, los coeficientes serán los valores o pesos asociados de la función de similitud global para cada atributo del caso. En el siguiente apartado se detalla esta última implementación.

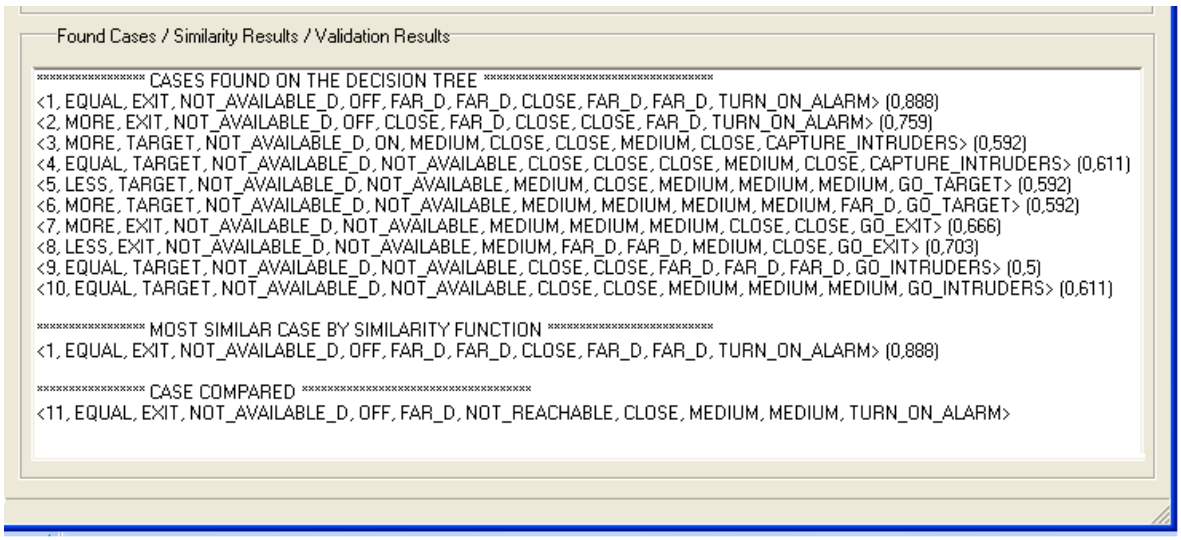

Figura 27. Valores de similitud aplicados a un caso sobre otros 10

3.3.1.1 Ajuste de coeficientes mediante algoritmos genéticos

Como ya se ha comentado en el apartado anterior, el problema de encontrar una función de similitud entre casos de una base de conocimiento se ha reducido a un problema de optimización. En concreto hay que buscar los valores de los coeficientes (pesos que se aplican en la función de similitud global para ajustar la relevancia de cada atributo del caso en la propia función) que maximicen un determinado criterio que se formula como una combinación lineal de criterios parciales. Generalizando la fórmula (4) se obtiene la siguiente expresión:

$$SIM(A,B) = \sum_{i=1}^{p} w_i \cdot sim_i(a_i, b_i) \quad \text{con} \quad \sum_{i=1}^{p} w_i = 1 \tag{5}$$

Cada coeficiente w_i representará el peso de los atributos a_i y b_i en el grado de similitud global entre casos.

Por otro lado, se define como *Clase de pertenencia al valor*$_i$ o simplemente *Clase*$_i$ al conjunto de casos que comparten el mismo *valor*$_i$ en el atributo-solución:

$$Clase_i = \{C_1, ..., C_k\} \tag{6}$$

donde *valor_solución*(C_1) = *valor_solución*(C_2) = … = *valor_solución*(C_k) = *valor*$_i$

Dada una base de casos *BC* y un caso $C_i \in BC$ se define como el caso más similar, *most_similar*(C_i), al caso $C_s \in BC$ tal que cumple:

$$\max_{j=1}^{j=0} {}_i \quad (SIM\ (C_i, C_j)) \tag{7}$$

Así, dados dos casos C_i y C_r perteneciente a una base de casos *BC*, donde C_i es el caso recuperado en *BC* para C_i y por tanto cumple que C_r = *most_similar*(C_i), diremos que la recuperación es correcta si C_i y C_r pertenecen a la misma Clase de pertenencia, es decir si los casos comparten el mismo *valor*$_i$ en el atributo solución.

Una vez definida la expresión de la función de similitud global (5) y el concepto de *recuperación correcta*, el problema se reduce a calcular el vector de pesos $W = (w_1, \ldots, w_p)$ que maximice el número de recuperaciones correctas. Es decir, hay que buscar los valores para estos coeficientes que maximicen un determinado criterio. En nuestro caso, este criterio es recuperar el máximo número de casos correctos, teniendo además en cuenta la disimilitud respecto a las demás clases de pertenencia para distinguir entre aquellas situaciones en las que los aciertos en recuperación sean similares. Es decir, a igualdad de número de aciertos, se dará preferencia a las soluciones que discriminen mejor las clases de pertenencia.

Como el vector de pesos $W = <w_1, \ldots, w_p>$ debe ser un vector de p reales no negativos, el espacio de búsqueda sobre estos coeficientes es infinito y por tanto es imposible evaluar computacionalmente de modo sistemático todo el espacio de soluciones. Por tanto, es necesario utilizar alguna técnica que permita una exploración apropiada a un espacio de soluciones infinito, algo muy en la línea de las técnicas de optimización basadas en los algoritmos genéticos. Este tipo de algoritmos se inspiran en aquellos procedimientos biológicos que guían los procesos de evolución y selección natural en los seres vivos. La idea consiste en evaluar fracciones del espacio de posibles soluciones, aquéllas que se vayan *adaptando* mejor en cada instante, es decir la que maximice un criterio de evaluación conocido como función de *adaptación* o función *fitness* a la vez que se evalúan otras zonas elegidas de modo aleatorio evitando de este modo que el algoritmo de búsqueda caiga en máximos locales.

Inicialmente se parte de una pequeña población de posibles soluciones, generada aleatoriamente, cuyos individuos (en nuestro problema, vectores de pesos) se clasifican en función de su *fitness* (el grado de optimización de la función de similitud que producen el conjunto de pesos). Por tanto, los mejores individuos, aquellos que poseen las *fitness* más altas, son los que ocupan los primeros puestos en la clasificación. Posteriormente se realiza un proceso de selección probabilística según la cual los mejores individuos tienen más posibilidades de ser seleccionados. En nuestro caso concreto, la función probabilística empleada es la estocásticamente uniforme, también conocida como *stochastic universal sampling.* A continuación, sobre los individuos seleccionados se aplica un operador genético de cruce (se genera, de acuerdo con una probabilidad de cruce, dos nuevos individuos a partir de los seleccionados) y un operador de mutación (se genera, con arreglo a una probabilidad de mutación, un nuevo individuo modificando el individuo resultante del cruce anterior). En el punto 3.3.1.1.2 se ofrece más información sobre las funciones empleadas tanto en el proceso de selección, como en los operadores de cruce y mutación.

Finalmente, una vez formada la nueva población, se reemplaza por la inicial y se vuelven a ejecutar los mismos pasos hasta que se cumpla algún criterio de terminación, tal como un estancamiento en la mejora de la *fitness*, un número determinado de iteraciones o alcanzar un individuo, en nuestro caso, se elige como criterio el máximo valor para la *fitness*.

La forma de búsqueda de la mejor solución con un Algoritmo Genético se basa en la idea de que las 'buenas soluciones' tienen tendencia a generar nuevas 'buenas soluciones', así, mediante este procedimiento, se consigue obtener una solución suficientemente buena con una exploración del espacio de soluciones sin necesidad de recorrer el espacio completo o tener algún conocimiento heurístico para la construcción de la solución. En [Michalewicz, 1994] se puede encontrar más información sobre algoritmos genéticos.

3.3.1.1.1 Función de fitness o evaluación

Según se ha explicado previamente la función de *fitness* mide la calidad de una solución y una solución, en nuestro caso, es de mayor calidad que otra si produce más recuperaciones correctas, además en el supuesto de no realizar una recuperación correcta se considerará mejor a aquella solución que devuelva un valor de similitud más bajo. De este modo se trata de favorecer la distancia entre clases de pertenencia, es decir que si la función de similitud no es capaz de devolver un caso incluido en la clase de pertenencia correcta que por lo menos devuelva un valor bajo de similitud.

Resumiendo, se trata de encontrar una función de *fitness f* de modo que a partir de un conjunto de clases de pertenencia *CS* y un vector de pesos $\boldsymbol{w} \in W$ devuelva un valor entre 0 y 1 (donde 1 es lo mejor y 0 lo peor) y que aumente con el número de recuperaciones correctas y con la distancia entre clases. W es el conjunto de todos los posibles vectores y por tanto coincide con el espacio de búsqueda de soluciones.

$$f : W \times CS \rightarrow [0,1]$$

La función *fitness* propuesta es la siguiente:

$$\mathrm{f(W,CS)} = \left(\frac{\sum_i f\_local(w, C_i)}{\#CS} \right) \tag{8}$$

donde *#CS* es la suma de todos los casos que hay en todas las clases de pertenencia de *CS* y *f_local* mide la *fitness* local entre un caso C_i y todos los demás de *CS* para un vector de pesos $\boldsymbol{w} \in W$. La *fitness* local se define como:

$$\mathrm{f\_local(w,C)} = \begin{cases} 1 & \text{Si la recuperación} \\ & \text{es correcta} \\ 1 - SIM(C, C') & \text{E.O.C.} \end{cases} \tag{9}$$

donde *C' = most_similar(C)*

Cuando la recuperación es correcta, es decir, volviendo a los términos del CBR, cuando los atributos-solución del caso consultado y del caso recuperado coinciden, la *fitness* local devuelve el máximo valor posible y cuando no es así, devuelve un valor que se toma como disimilitud. Posteriormente, en la función de *fitness* global, se realiza la media de todas las *fitness* locales y de este modo se obtiene una medida de la calidad de recuperación que tendría la función de similitud, y por ende el CBR, si se cargara con los pesos de $\boldsymbol{w} \in W$.

Para evaluar la función *fitness* se utilizó un conjunto de casos ya clasificados (con el atributo-solución ya determinado) de 50 casos. A partir de este conjunto que podemos llamar de entrenamiento se determinaron unos pesos, que posteriormente fueron utilizados y evaluados frente a diferentes bases de conocimiento. Los resultados obtenidos y su comparación con los diferentes métodos utilizados se incluyen en el apartado siete.

3.3.1.1.2 Operadores y funciones aplicadas

La implementación se ha realizado usando la *toolbox* de genéticos de matlab. Cada individuo fue codificado como un vector 9-dimensional de números (donde cada componente del vector representa un atributo de un caso). El atributo-solución no se tiene en cuenta para este proceso

puesto que no debe influir en la similitud entre casos, simplemente debe almacenar la solución usada en el pasado para resolverlo.
Cada componente del vector permite cualquier valor positivo o negativo pero para la evaluación de la *fitness* se interpretan siempre como positivos, es decir se toma el valor absoluto del vector generado. El tipo de selección es estocásticamente uniforme, también conocida como *stochastic universal sampling*, esto es, se traza un segmento dividido en tramos de forma que a cada individuo le corresponde un tramo de longitud proporcional a su *fitness*; posteriormente se divide el tamaño total del segmento entre el número de individuos de la población y a esta distancia se le llama paso. Se escoge un número de comienzo *start* entre 0 y el tamaño de un paso y partiendo de este punto se va recorriendo el segmento sumando cada vez un paso al punto anterior. En cada paso se selecciona el individuo asociado al tramo en el que se está incidiendo. La operación se repite *n* veces hasta alcanzar una selección de *n* individuos. Este método presenta una ventaja respecto al de la ruleta [Blickle & Thiele, 1995], y es que muestrea todo el 'ancho' del espacio de progenitores y respeta además el principio de que cuanto mayor sea la *fitness* mayor probabilidad tiene un individuo de ser seleccionado. En la ruleta, cada sección o porción también es proporcional a la fitness de cada individuo, posteriormente se van escogiendo aleatoriamente puntos pertenecientes a la ruleta y extrayendo los individuos asociados a los secciones donde recaen los puntos. Por tanto, mediante la ruleta no se puede garantizar que no se seleccione un conjunto de progenitores poco diverso, ya que, por ejemplo, podría ocurrir que todos los puntos recayeran en una misma sección.

La función de cruce es aleatoria, se crea un vector binario aleatorio que indicará el origen de los genes del nuevo individuo, las posiciones donde haya un valor numérico de uno toman sus genes del primer individuo padre y donde haya un cero del segundo individuo padre. La probabilidad de cruce es de 0.8, que es utilizado por defecto en Matlab. El tipo de mutación es gaussiana ya que es la más comúnmente usada y la probabilidad de mutar es de 0.2 (= 1 – 0.8 de la probabilidad de cruce). Existe elitismo, en concreto los dos mejores individuos de la población original saltan a la siguiente generación. Finalmente, para calcular los coeficientes, y tras conjuntos de experimentaciones de prueba y error, se determinó partir de una población inicial aleatoria de 20 individuos y tras 100 generaciones la función *fitness* ya alcanza el valor de 0,9492. Tras mil generaciones más, el valor había mejorado levemente hasta alcanzar 0,9498 y mantenerse estable en torno al 0,95. En la figura 38 se pueden observar estos últimos valores.

3.3.2 Árbol de decisión

Otra alternativa válida para solventar el problema de ajuste de pesos en la función de similitud es la utilización de un árbol de decisión en la clasificación de un nuevo caso y por tanto en la etapa de recuperación. Los árboles de decisión son una buena técnica de recuperación y organización de la información. El árbol se construye a partir de un conjunto de entrenamiento, consistente en nuestra aplicación en casos clasificados. Con el conjunto de atributos que definen estos casos (ver tabla 2), se genera un clasificador con una estructura en forma de árbol capaz de clasificar un nuevo caso a partir del valor para los atributos que definen el caso [Buckinx et al., 2004].

El algoritmo implementado para construir el árbol de decisión es el ID3. Este método ya se utilizó previamente para realizar una búsqueda en el espacio de estados, si bien ahora se utiliza para el cómputo de pesos, por tanto aún tratándose del mismo método su aplicación es bien diferente. Este método selecciona en cada paso un atributo de caso todavía sin desplegar y crea una nueva rama

por cada uno de sus posibles valores. Posteriormente se divide la lista de casos actuales en tantas sublistas como valores tenga el atributo de selección. En cada sublista se almacenan los casos que encajen con el valor correspondiente para esa rama del atributo seleccionado. Si alguna sublista presenta el mismo valor del atributo-objetivo en todos sus elementos, entonces se etiqueta ese nuevo nodo como solución (nodo hoja) con dicho valor y sus casos asociados son todos los presentes en la sublista. Todos esos casos presentan características comunes que permiten definir una solución asociada.

El criterio para elegir en cada paso el mejor atributo es seleccionar aquel que produzca una mayor reducción de la entropía. Para medir la entropía de un nodo se utiliza la siguiente fórmula:

$$entropia\ \ (X) = -\sum_{i=1}^{n} p(x_i) \log\ p(x_i) \tag{10}$$

donde $p(x_i)$ es la proporción de ejemplos de la clase x_i. Para nuestro CBR el valor de n es 6, que es la cardinalidad del atributo-solución y por tanto el número de clases distintas (tabla 2).

Una vez definido el concepto de entropía, se puede definir el de ganancia como la diferencia entre la entropía de un nodo y la suma ponderada de sus sucesores. Su ecuación es la siguiente:

$$ganancia(S,A) = entropia(S) - \sum_{v \in vectores(A)} \frac{|S_v|}{|S|} entropia\ (S_v) \tag{11}$$

siendo S el conjunto de ejemplos del nodo padre y S_v el subconjunto de elementos de S para los que el atributo A toma el valor v.

Por tanto, como cada paso del ID3 implica una selección del mejor atributo posible, en cada iteración se debe evaluar la ganancia para todos los atributos todavía disponibles y se escoge aquél que genere la partición con el valor más alto de ganancia.

En la figura 28 se muestra el árbol de decisión generado para los mismos 10 casos que se evaluaron en la figura 27 (valores de similitud entre casos).

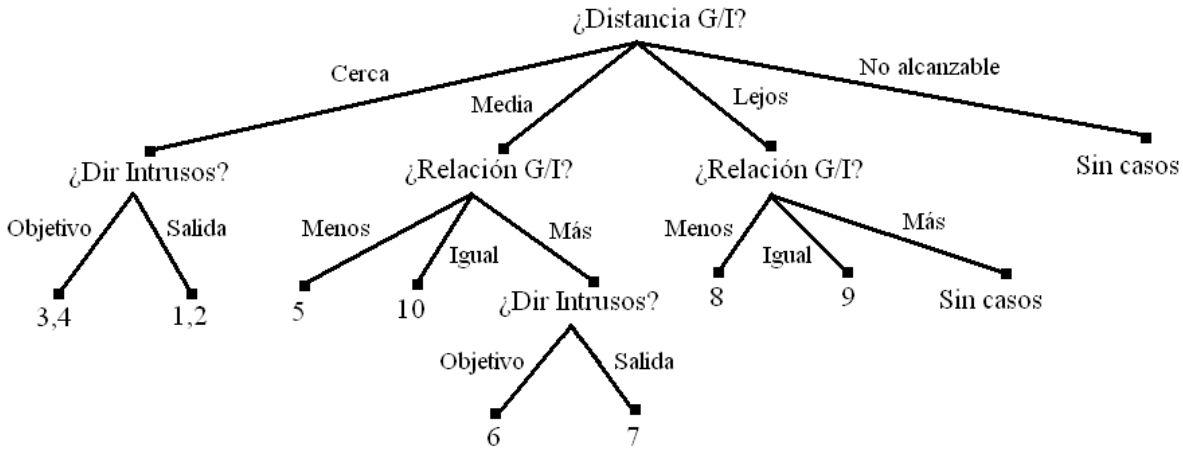

Figura 28. Árbol de decisión para 10 casos

Es interesante observar que, cuando no existe ningún caso relacionado con algún nodo final, la solución que se devuelve es la más frecuente en el árbol padre. El fundamento de esta elección radica en que el nodo final que no dispone de casos asociados, comparte con sus hermanos una serie de valores en común: todos aquellos que se han recorrido desde la raíz global hasta el padre. Por ello, es lógico pensar que su solución estará relacionada con la de sus hermanos, y por estadística se asigna la más probable, que es la más frecuente, éste es un modo de tratar con los valores perdidos.

4 Aplicación sobre plataformas distribuidas: Middleware

Desde mediados de los noventa, la industria de la computación ha estado usando plataformas *middleware* orientadas a objetos, tales como DCOM [Grimes, R. 1998] y Corba [Henning, M., and S. Vinoski. 1999]. Los *middleware* orientados a objetos han supuesto desde su aparición un importante paso hacia delante haciendo disponible la programación distribuida a los desarrolladores de aplicaciones. Por primera vez han posibilitado la construcción de aplicaciones distribuidas sin necesidad de tener que ser un experto en redes: ya que el *middleware* pasa a encargarse de los principales puntos de comunicación, tales como *marshaling* and *unmarshaling* (codificación y decodificación de los datos para la transmisión), mapeo lógico de las direcciones de objetos a los puntos físicos finales del transporte, cambio de la representación de datos de acuerdo a la arquitectura nativa de la máquina donde estén el servidor y el cliente, y el arranque automático de servidores bajo demanda.

En este proyecto, se utilizará el *middleware* Ice como el software que hace posible la conexión entre las diferentes aplicaciones distribuidas del proyecto Hesperia. Funcionará como una capa de abstracción que se sitúa entre las capas de aplicaciones y las capas inferiores (sistema operativo y red).

4.1 Ice

Ice es una plataforma *middleware* orientada a objetos. Fundamentalmente esto significa que Ice provee herramientas, APIs y soporte de librerías para construir aplicaciones cliente servidor orientadas a objetos. Las aplicaciones Ice son ideales para usar en entornos heterogéneos: cliente y servidor pueden ser escritos en diferentes lenguajes de programación, pueden ejecutarse en diferentes sistemas operativos y en máquinas con arquitecturas diferentes y pueden comunicarse usando gran variedad de tecnologías de redes. El código fuente de estas aplicaciones es portable sin tener en cuenta el entorno de instalación.

Como cualquier tecnología, Ice tiene su propio vocabulario. En Ice, los términos cliente y servidor más que designar partes concretas de una aplicación, denotan los papeles que son interpretados por partes de una aplicación durante una petición. Los clientes son entidades activas que emiten peticiones de servicio a los servidores. Los servidores son entidades pasivas que proveen servicios en respuesta a las peticiones.

Frecuentemente, los servidores no actúan como servidores 'puros' en el sentido de que nunca emiten una petición y que sólo responden a las peticiones. En vez de eso, los servidores a menudo actúan como servidor para algún cliente, pero para completar el servicio tienen que acudir a otro servidor como cliente para poder terminar el servicio y satisfacer al cliente inicial. Análogamente, los clientes a menudo no actúan como clientes 'puros' en el sentido de que sólo realizan peticiones. A menudo suelen funcionar como entidades híbridas cliente-servidor. Por ejemplo, un cliente puede comenzar un largo proceso en un servidor y como parte del arranque de la operación, pasarle un objeto *callback* al servidor que será usado por el servidor para notificar al cliente que la operación se ha completado. En este caso, el cliente actúa como cliente cuando arranca el proceso y como servidor cuando es notificado del fin de la operación.

Un objeto Ice es una entidad conceptual o abstracción que se caracteriza por:

- Ser una entidad local o en un espacio remoto que puede responder a las peticiones de los clientes.
- Puede ser instanciado en uno o varios servidores, sin dejar de ser el mismo objeto Ice.
- Cada objeto Ice tiene una o más interfaces. Una interfaz es una colección de operaciones soportadas por un objeto. Las peticiones que emiten los clientes son invocaciones a estas operaciones.
- Una operación tiene cero o más parámetros así como un valor de retorno.
- Cada objeto tiene una única identidad que la distingue de los demás objetos. Por tanto no pueden existir dos objetos con la misma identidad dentro de un dominio de comunicación Ice.

Para permitir a un cliente contactar con un objeto Ice, el cliente debe tener un *proxy* al objeto. Un *proxy* es un 'ente' local del espacio de direcciones del cliente que representa al objeto Ice (posiblemente remoto) en el cliente. Es decir, actúa como un embajador local del objeto Ice remoto.

El *proxy* encapsula toda la información necesaria para conectar con el objeto remoto:

- Direcciona información que permite al cliente contactar con el servidor correcto.
- Un identificador de objeto que identifica qué objeto en concreto es el objetivo de la petición.
- Un identificador de interfaz que indica qué interfaz en concreto es la referenciada.

En la figura 29 se puede observar cual es la arquitectura en capas de Ice. En los niveles más altos están las aplicaciones cliente y servidor e inmediatamente por debajo y respectivamente, los proxies y el adaptador de objetos. Por último, en el nivel más interno se encuentran los núcleos cliente y servidor que se conectan mediante la red.

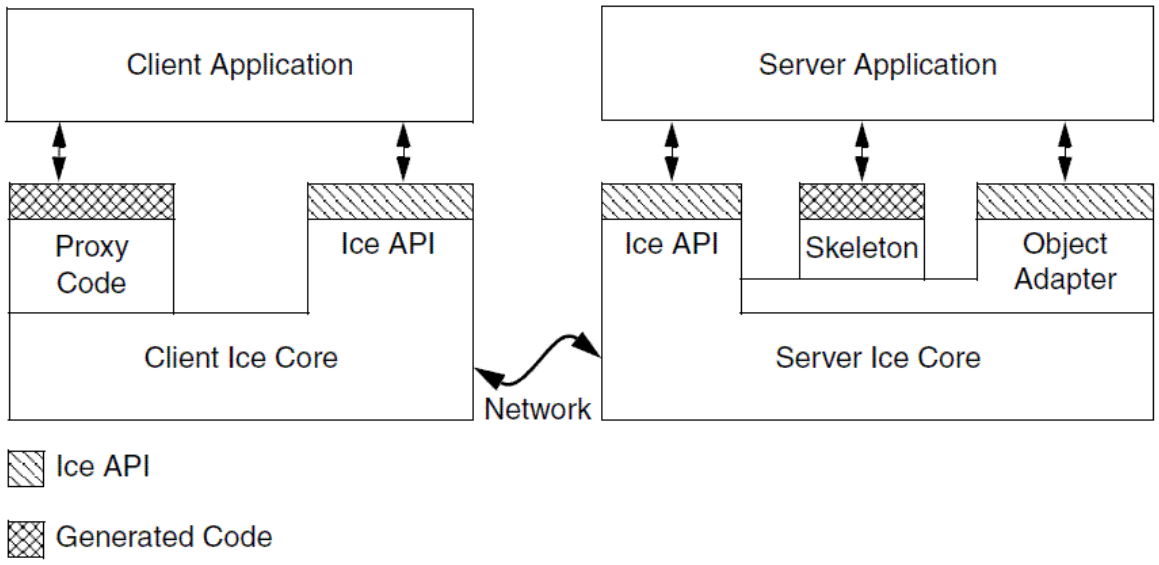

Figura 29. Arquitectura Ice

Como se ha mencionado anteriormente, cada objeto Ice tiene una interfaz con un número de operaciones. Las interfaces, operaciones y los tipos de datos que se intercambian entre cliente y servidor se definen usando el lenguaje Slice. Slice es un lenguaje que permite definir el contrcto cliente-servidor de manera que sea independiente de un lenguaje de programación específico, tal como C++, Java o C#. Las definiciones Slíce son traducidas por un compilador a una API en un determinado lenguaje de programación. Es decir, basta con que cliente y servidor tengan los mismos archivos *slice*, que son aquellos que definen las interfaces, y posteriormente según la implementación de uno u otro se usa el compilador que traduzca los *slices* al lenguaje de implementación en el que se esté trabajando.

4.2 Gestor de crisis con Ice

El gestor de crisis tiene una función clara que es la de generar actuaciones que neutralicen una intrusión. Para ello, de acuerdo con la terminología Ice explicada en el apartado anterior, a veces actuará como cliente y otras como servidor. Por ejemplo, en la interfaz *Vigilante*, el gestor actúa como cliente cada vez que ordena la ejecución de una actuación mediante el método *EjecutarAccion* ya que invoca un objeto Ice en el elemento que deba ejecutar la acción. Este método se utiliza para ordenar una actuación a un vigilante. En el objeto de tipo *AccionGC* que se le pasa va toda la información necesaria para llevar a cabo la acción: identificadores de elementos sobre los que se debe operar, tipo de acción, camino hasta el lugar de la acción, etc.
Asimismo, en el resto de operaciones que contiene también hace de cliente, ya que solicita datos de un elemento remoto invocando las operaciones *getCoord*, *ConsultarEstado* o *getId*. *GetCoord*, *ConsultarEstado* y *getId* son métodos que devuelven respectivamente la posición, el estado y el identificador del elemento sobre el que se invocan, Todas estas operaciones son métodos que se invocan en objetos Ice remotos a través de un *proxy.*, Por ser el gestor quién los invoca y realiza la petición, y otros elementos quienes facilitan la respuesta, se dice que el gestor actúa como cliente.

```
interface Vigilante{
        Coord getCoord();
        void EjecutarAccion(AccionGC a);
        EstadoGC ConsultarEstado(string id);
        string getId();
};
```

En las interfaces *GestorIntrusion* y *Malla* sin embargo, el gestor actúa como servidor. Estas dos interfaces ofrecen una colección de operaciones a las que se puede acceder desde procesos externos. Las operaciones de *GestorIntrusion* permiten consultar información acerca del estado de la escena global así como modificar la posición de algunos de sus elementos o consultar la última acción generada por el gestor. Mediante *ConsultarPosicionElemento* se puede obtener las coordenadas en el plano de un objeto presente en la escena. Los métodos *ConsultarCaminoEntrePuntos* y *ConsultarCaminoEntreElementos* devuelven respectivamente las rutas óptimas entre dos posiciones o entre dos elementos de la escena determinados por sus idenetificadores. *ConsultarAccionesSobreElemento* ofrece todas las actuaciones o acciones que se han realizado sobre un determinado elemento. *MoverGuadias* y *moverIntrusos* desplazan respectivamente a vigilantes e intrusos por la escena y *consultarAccion* devuelve la última actuación realizada.

```
interface GestorIntrusion{
        Coord ConsultarPosicionElemento(int id);
        RutaGC ConsultarCaminoEntrePuntos(Coord a, Coord b);
        RutaGC ConsultarCaminoEntreElementos(int e1, int e2);
        HistorialGC ConsultarAccionesSobreElemento(int e);
        void moverGuardias(Coord p);
        void moverIntrusos(Coord p);
        AccionGC consultarAccion();
};
```

En la interfaz *Malla* los métodos están pensados para poder modificar la estructura del plano de la escena. De este modo se pueden insertar nuevos obstáculos o eliminar los ya presentes utilizando respectivamente insertarObstaculo y quitarObstaculo.

```
interface Malla{
        bool insertarObstaculo(RutaGC obs, int id);
        bool quitarObstaculo(int id);
};
```

Tanto en la interfaz *GestorIntrusion* como en *Malla,* el gestor actúa como servidor porque se encarga de alojar a los objetos Ice que responderán a las peticiones que se hagan remotamente.

5 Integración

En este apartado se describe la estructura interna del proyecto así como la integración con el resto de aplicaciones que componen el proyecto Hesperia, aquél del que forma parte el gestor de crisis expuesto en este trabajo. En el primer punto se explica un diagrama de las principales clases que componen el proyecto y su funcionalidad, mientras que en el segundo se listan las diferentes aplicaciones o entidades externas al gestor pero que interactúan con él formando parte del sistema global.

5.1 Estructura interna del proyecto

El diagrama de clases simplificado del gestor de crisis es, a grandes rasgos, el mostrado en la figura 30. La clase *Manager* contiene las clases necesarias para representar el espacio, *Triangulation*, y para implementar el razonamiento basado en casos, *CBR*. *CBR* se apoya a su vez en la clase *TreeNode* que le permite implementar el árbol de decisión necesario para la toma de decisiones. *Manager* contiene además en la propia clase todas las propiedades necesarias para llevar el estado de la aplicación, como son las listas de elementos presentes en la escena o los historiales de acciones. Por último, se encuentran los servicios Ice que acceden a la clase *Manager* para poder implementar los procesos necesarios para responder a las peticiones externas.

Desde el punto de vista de la investigación realizada en el presente proyecto, este diagrama aporta una estructura compacta para entender cómo se integran las diferentes técnicas utilizadas para así abarcar todos los aspectos y necesidades del complejo sistema final.

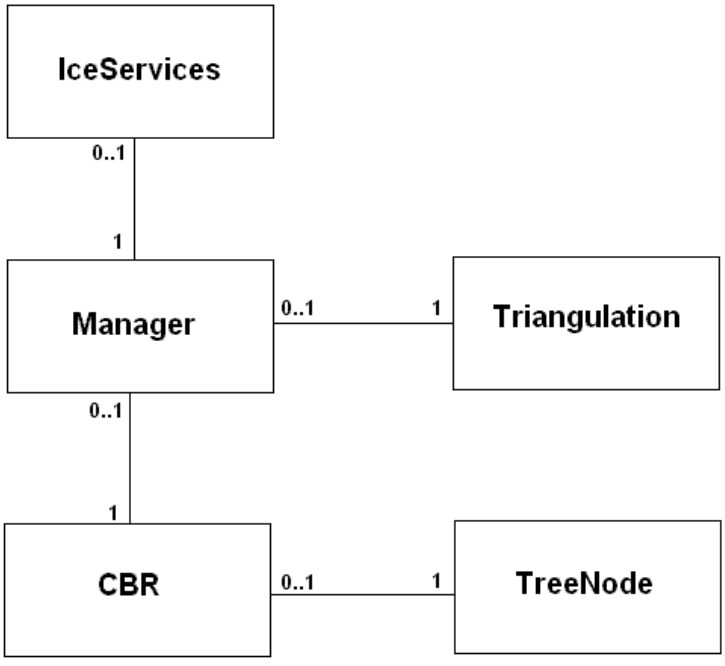

Figura 30. Diagrama de clases simplificado

5.2 *Integración con aplicaciones externas*

Como ya se ha comentado previamente, el proyecto Hesperia, como se puede observar en la figura 31, está compuesto por varios módulos:

- Interfaz: Visualiza el estado del sistema permitiendo monitorizar las decisiones que se van tomando y la evolución de la escena.
- Sensores: Captan las señales del medio y las envían al gestor de conocimiento.
- Gestor de conocimiento: A partir de las señales captadas del entorno, realiza un análisis y determina si se está en una situación de intrusión o no. En caso afirmativo lo notifica al gestor de crisis.
- Gestor de crisis: A partir del estado de la escena genera una serie de actuaciones para neutralizar la intrusión. Puede conectarse a los sensores para adquirir información de la intrusión, y envía a los vigilantes (equipados con una PDA) las decisiones que va tomando. Todo su proceso de actuación es monitorizado por la interfaz, lo que permite realizar un seguimiento del gestor.
- PDA's: Van asociadas a los vigilantes y se utilizan para recibir las órdenes del gestor de crisis y para permitir a los vigilantes acceder a información mediante consultas al gestor.
- Base de datos: Almacena la información estática del gestor.

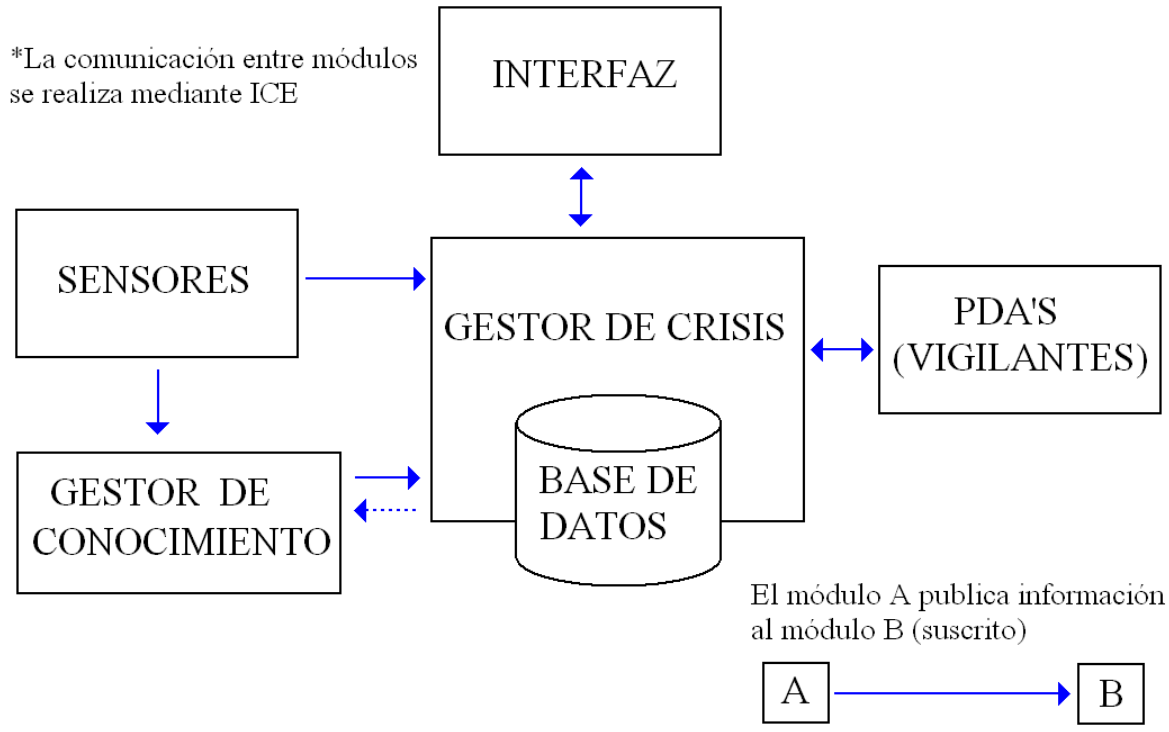

Figura 31. Estructura proyecto Hesperia

6 Resultados

En este apartado se describen las técnicas empleadas para evaluar el rendimiento del sistema. Primero se describe el concepto de evaluación cruzada y sus diferentes tipos y posteriormente se comentan y comparan los resultados obtenidos utilizando estas técnicas en los diferentes enfoques utilizados en el diseño de la implementación (recuperación con función de similitud equiponderada, función de similitud con algoritmos genéticos y árbol de decisión).

6.1 Evaluaciones cruzadas

Una evaluación cruzada es un método estadístico que consiste en segmentar el espacio de muestras en varios subconjuntos y utilizar uno de ellos para inicializar un sistema y extraer un primer análisis, mientras que los restantes se utilizarán para realizar posteriores evaluaciones y contrastarlas con los resultados del primer análisis. En el caso del CBR, la base de conocimiento se divide en varios conjuntos [Duda, R. O. et al, 2001] de casos, donde unos se utilizan como conjuntos de entrenamiento y otros como conjuntos de verificación o contraste. Para este trabajo se han utilizado bases de conocimiento de 10, 30, 50, 70 y 100 casos. Como se ve más adelante, se empleará un parámetro *k* para determinar los conjuntos de entrenamiento y los de evaluación.

6.1.1 K-fold

En *k-fold cross-validation*, el muestreo original se divide en *k* subconjuntos. De estos *k* subconjuntos, uno se retiene como conjunto de evaluación del modelo esperado y los restantes *k*-1 subconjuntos como ejemplos de entrenamiento. Se repite entonces la evaluación cruzada *k* veces usando cada vez uno de los *k* subconjuntos como conjunto de contraste. El resultado final o estimación del rendimiento del sistema se calcula realizando la media de los *k* resultados obtenidos. Los valores de k empleados en las evaluaciones presentes en este apartado se muestran más adelante.

6.1.2 Leave-one-out

Leave-one-out es un caso extremo de una evaluación cruzada *k-fold*. Para este método, se toma cada muestra como conjunto de validación y todas las restantes como datos de entrenamiento. Para este caso, *k* es igual al número de elementos. La desventaja respecto al método *k-fold* se encuentra en que es mucho más costoso por incrementar considerablemente el número de conjuntos de entrenamiento del muestreo y no necesariamente mejora los resultados de una evaluación *k-fold* [Kohavi, R., 1995], [Bishop, C., 2007]. Sin embargo, es el más adecuado cuando se dispone de pocos ejemplos de entrenamiento.

6.2 Comparativas entre la función de similitud equiponderada y el árbol de decisión

A continuación se muestran los resultados obtenidos para las tres implementaciones realizadas, basadas en los tres modos de recuperación explicados anteriormente, a saber: la función de similitud

con pesos equiponderados y con pesos determinados mediante genéticos y el árbol de decisión. La medida que determina el rendimiento es el número de casos recuperados correctamente entre el número de casos consultados, expresando el resultado en porcentaje. La técnica empleada ha consistido en aplicar diferentes evaluaciones cruzadas modificando en cada caso el número de particiones del espacio de ejemplos. El parámetro *k* representa el número de particiones realizadas sobre el espacio de ejemplos. Es interesante observar que para *k* = NC (número de casos total), la evaluación cruzada puede considerarse una evaluación de tipo *leave-one-out*. En concreto, para hallar los resultados mostrados en este apartado se han aplicado tres tipos evaluaciones resultantes de modificar el parámetro *k*. Los valores empleados han sido 5, 10 y el número total de casos del conjunto evaluado (*leave-one-out*). En cada evaluación, uno de los conjuntos de ejemplos se ha utilizado como conjunto de contraste y los restantes como entrenamiento.

Los *k* grupos formados para cada evaluación están estratificados de modo que exista un porcentaje similar de cada posible valor del atributo-solución. De este modo, se mejora la evaluación, al no permitir conjuntos de entrenamiento donde no haya una 'diversidad' razonable.

En la fila "FP" se muestra el porcentaje de casos bien recuperados para el CBR con función de similitud de pesos equiponderados y en la fila "Árbol" los casos bien recuperados cuando se utiliza un árbol de decisión en la fase de recuperación.

Para los casos con ruido se dejaron 3 ejemplos mal clasificados en cada base de conocimiento aplicada. Por caso 'con ruido' se debe entender un caso mal clasificado, es decir, cuyo parámetro solución no contiene un valor correcto de acuerdo al conocimiento que aplicaría un experto.
Todos los resultados obtenidos son peores, especialmente para bases de conocimiento de tamaño pequeño (número de casos = 10), donde lógicamente el porcentaje de ejemplos mal clasificados es significativamente más alto.

k = 5										
Num Casos	10	%	30	%	50	%	70	%	100	%
FP	4	**40**	14	**47**	27	**54**	52	**74**	74	**74**
Árbol	2	**20**	18	**60**	31	**62**	50	**71**	82	**82**

k = 10										
Num Casos	10	%	30	%	50	%	70	%	100	%
FP	2	**20**	13	**43**	25	**50**	51	**73**	72	**72**
Árbol	1	**10**	16	**53**	32	**64**	55	**79**	80	**80**

k = NC										
Num Casos	10	%	30	%	50	%	70	%	100	%
FP	2	**20**	13	43	25	**50**	50	**71**	72	**72**
Árbol	1	**10**	13	43	31	**62**	48	**69**	83	**83**

Tabla 3. Datos con ruido

k = 5										
Num Casos	10	%	30	%	50	%	70	%	100	%
FP	6	**60**	15	**50**	31	**62**	54	**77**	80	**80**
Árbol	3	**30**	18	**60**	33	**66**	55	**79**	86	**86%**

k = 10										
Num Casos	10	%	30	%	50	%	70	%	100	%
FP	5	**50**	14	**47**	29	**58**	53	**76**	78	**78**
Árbol	3	**30**	17	**57**	35	**70**	60	**86**	85	**85**

k = NC										
Num Casos	10	%	30	%	50	%	70	%	100	%
FP	5	**50**	14	**47**	29	**58**	52	**74**	78	**78**
Árbol	3	**30**	13	**43**	35	**70**	53	**76**	88	**88**

Tabla 4. Datos sin ruido

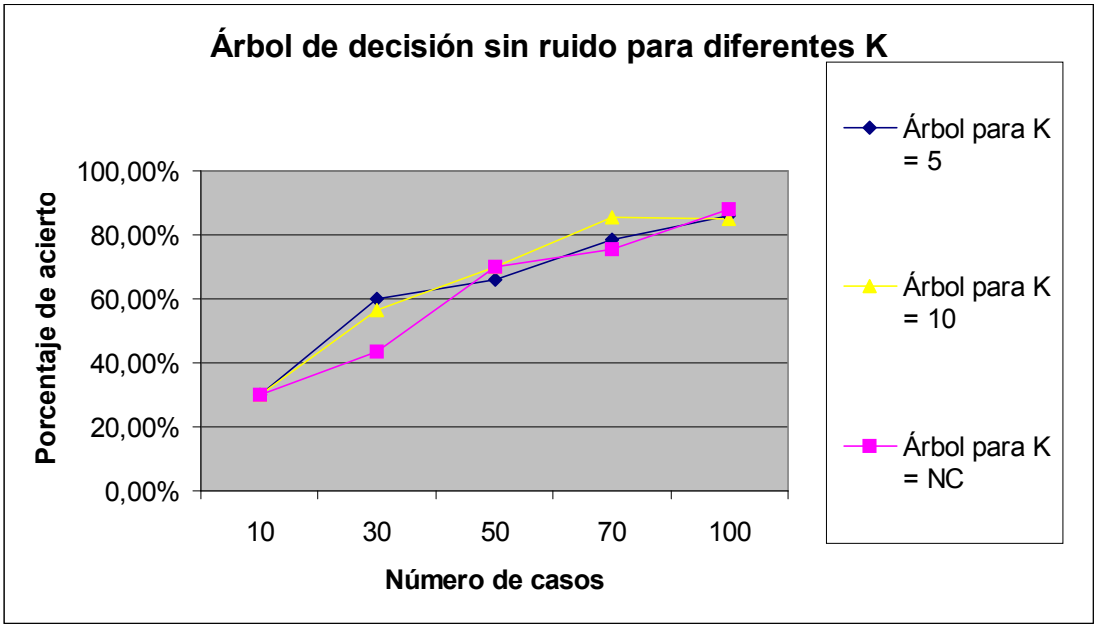

Figura 32. Árbol de decisión sin ruido par diferentes *k*

En las siguientes figuras (32 a 37) se muestra una mejor visualización de los datos mostrados en las tablas 3 y 4. En concreto, en las figura 32, correspondiente a las filas etiquetadas como 'Árbol' en la tabla 4, se observa un mejor rendimiento del árbol de decisión cuando hay una cantidad suficiente de ejemplos. A partir de 50 casos, ya hay suficiente conocimiento almacenado para que la clasificación del nuevo caso con el árbol de decisión sea correcta. Para conjuntos de ejemplos más pequeños, la precisión de la recuperación con la función de similitud iguala a la precisión en la clasificación con el árbol de decisión y en algunos casos incluso la supera.

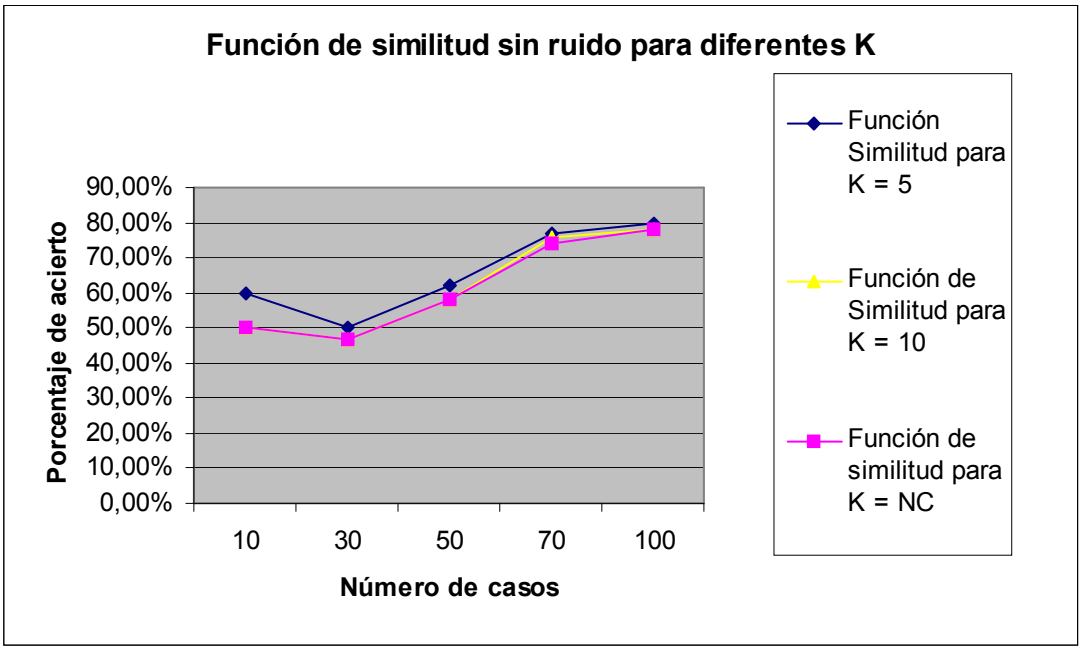

Figura 33. Función de similitud sin ruido para diferentes k

Para la función de similitud, como se ve en la figura 33 (construida con los datos de las filas etiquetadas como 'FP' de la tabla 4), en un primer momento podría pensarse, que el porcentaje de recuperaciones correctas debería mantenerse estable, sin embargo los números demuestran, que al igual que sucede con el árbol de decisión, el rendimiento mejora con el número de ejemplos. Esto se debe a que en este problema en concreto, todas la variables juegan un papel más o menos similar e importante (coincide con el peso equivalente aplicado en al función) y ninguna acaba de predominar sobre las otras ni hay redundancia entre ellas. Además cuanto mayor es el número de casos, hay más diversidad y por tanto es más difícil encontrar casos que no devuelvan ningún valor próximo a una alta similitud para la función utilizada.

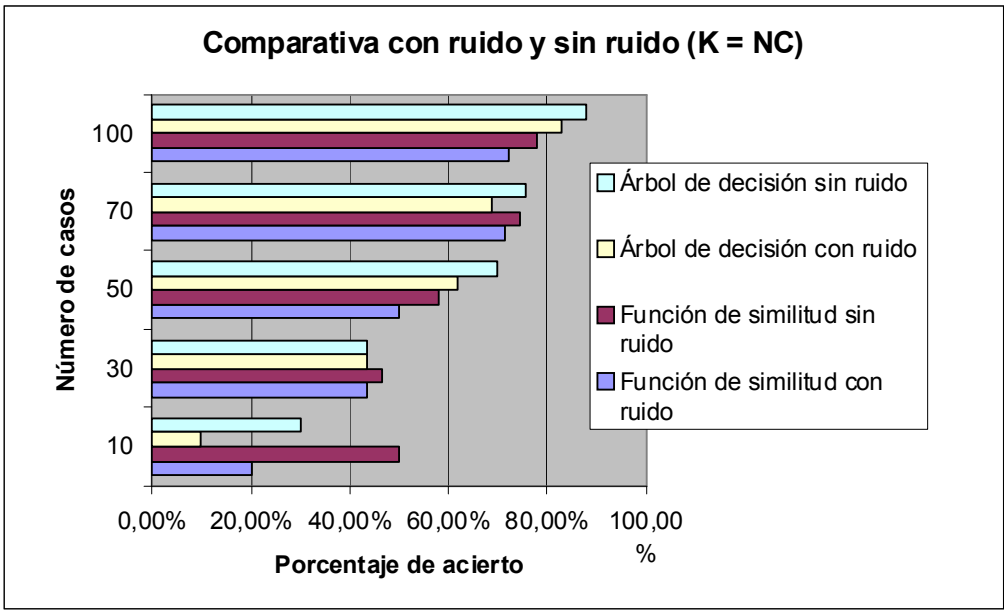

Figura 34. Comparativa con ruido y sin ruido para k = número de casos

Asimismo, en la figura 34 (datos obtenidos de las filas 'árbol' y 'FP' de las tablas 3 y 4 para k = NC), se observa que el ruido afecta al rendimiento del sistema. Para los datos del funcionamiento del sistema con ruido se dejaron 3 ejemplos mal clasificados en cada base de conocimiento aplicada. Todos los resultados obtenidos son peores, especialmente para bases de conocimiento de tamaño pequeño (número de casos = 10), donde el porcentaje de ejemplos mal clasificados es más alto. Aún así, en bases de conocimiento grandes (número de casos = 100), se constata una disminución en la precisión de un 3%.

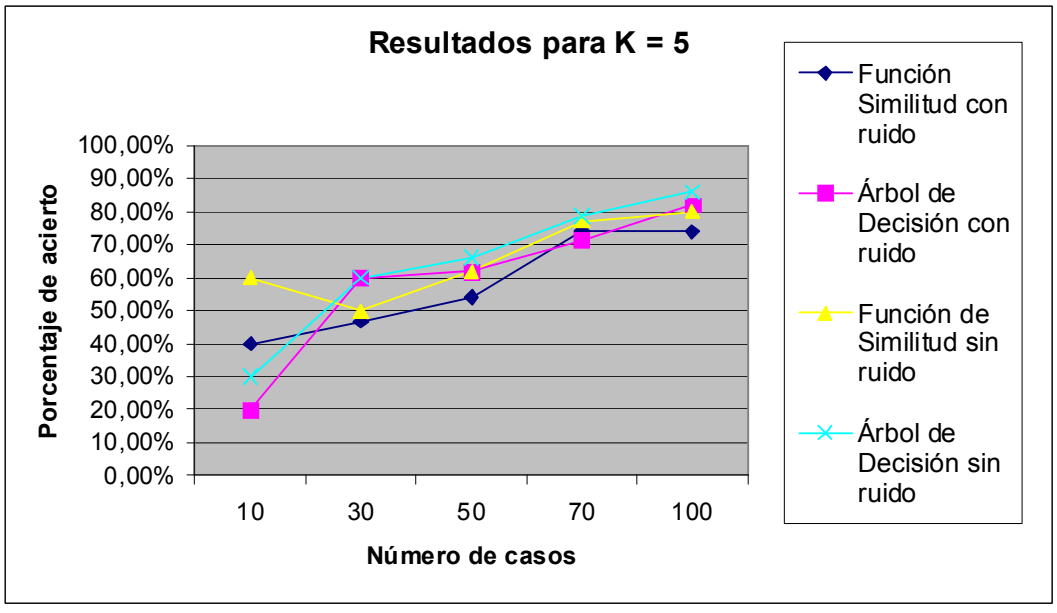

Figura 35. Árbol de decisión y función de similitud equiponderada para *k* = 5

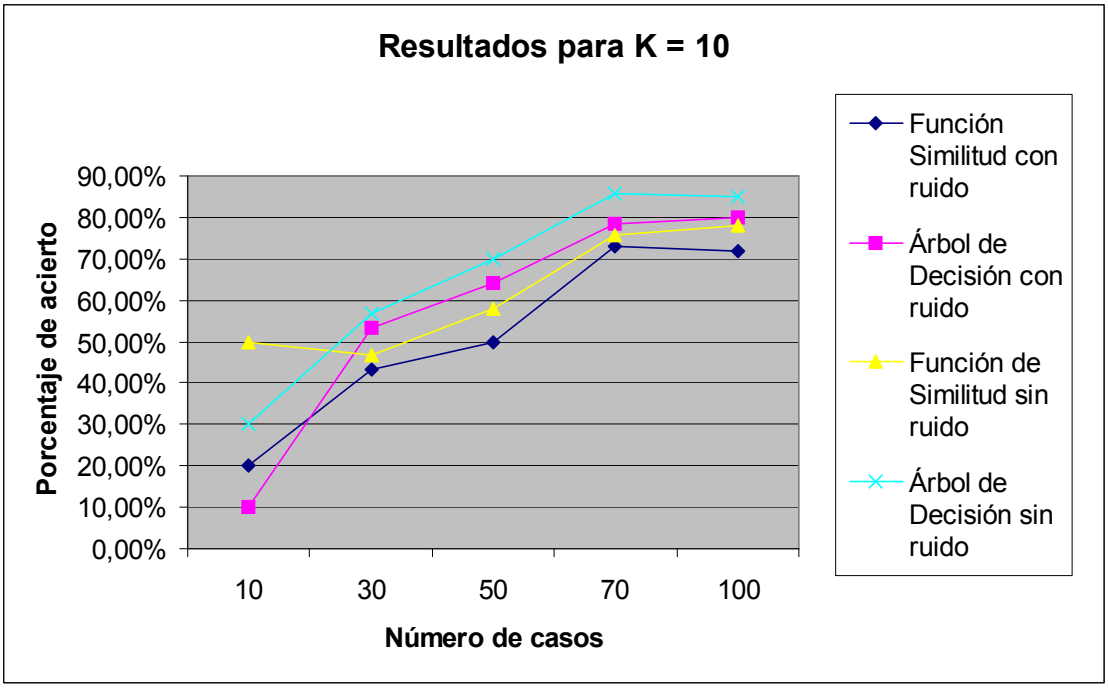

Figura 36. Árbol de decisión y función de similitud equiponderada para *k* = 10

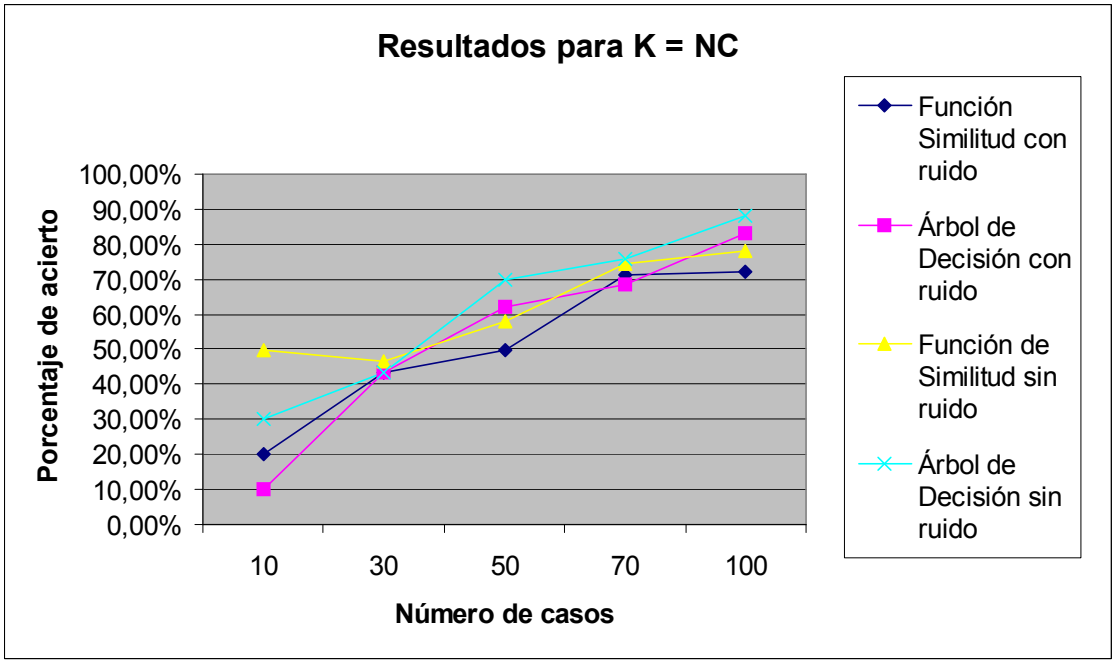

Figura 37. Árbol de decisión y función de similitud equiponderada para *k* = NC

Por último, sobre las figuras 35, 36 y 37 (formadas a partir de los datos de la tabla 3 y 4, en concreto con las filas 'FP' y 'Árbol' respectivamente para k = 5, 10 y NC), se esperaría encontrar unos valores

claramente mejores para las evaluaciones de tipo *leave-one-out* (k = número de casos), ya que los conjuntos de entrenamiento resultan más grandes. Sin embargo, no se aprecia ninguna mejora significativa en los resultados, a pesar de que el mejor porcentaje se obtiene para la situación en la que k = 100. Esto se debe a que el mayor conjunto de datos de entrenamiento, ramifica más el árbol de decisión, pero solamente lo suficiente para cubrir casos singulares de la base de conocimiento. No se llegan a inferir reglas más generales y por tanto esto no repercute en el rendimiento global del sistema. Si el sistema fuera mucho más complejo (con un mayor número de variables por caso), el rendimiento sí se vería mejorado, ya que el incremento de casos de entrenamiento contribuiría a añadir no tanto conocimiento particular y sí más general.

6.3 Comparativa entre las funciones de similitud ponderada mediante algoritmos genéticos y el árbol de decisión

Para el cálculo real de los pesos en el vector ***w***, definido en la sección cuatro, se ha utilizado una base de conocimiento de 50 casos. En total, el número de valores diferentes del atributo-solución, definido con el nombre de *opSol* (tabla 2 presente en la sección 4) presentes entre esos 50 casos es 5 (aunque el atributo-solución puede tomar 6 valores el valor *CLOSE_DOOR* no se ha usado en la base de casos por ser una actuación que no se empleará en la práctica al no estar incluida dentro de las especificaciones del proyecto Hesperia), por tanto el número de clases de pertenencia es también 5. Además cada Clase de pertenencia está formada exactamente por 10 casos, por lo que el espacio de entrenamiento (casos de entrenamiento) está equitativamente distribuido entre las 5 clases.

Como se puede observar, en la figura 38 se muestra el mejor individuo y el mejor valor fitness, tras 100 y 1000 generaciones del algoritmo genético para 50 casos, respectivamente los valores son 0.9492 y 0.9498. En la figura 38 aparecen los valores 0.0508 y 0.0502 en vez de los mencionados 0.9492 (= 1 – 0.0508) y 0.9498 (= 1 - 0.0502) esto se debe a que fueron obtenidos mediante la herramienta Matlab que por defecto trata de acercar la función *fitness* lo más posible al valor 0 en vez de al 1 como se ha definido en el punto 3.3.1.1.1. Para adaptarlo a lo explicado en dicho punto basta con restar a 1 el valor devuelto por la función.

En la figura 38 también se muestra el mejor individuo encontrado durante todo el proceso. Observar que los pesos están sin normalizar.

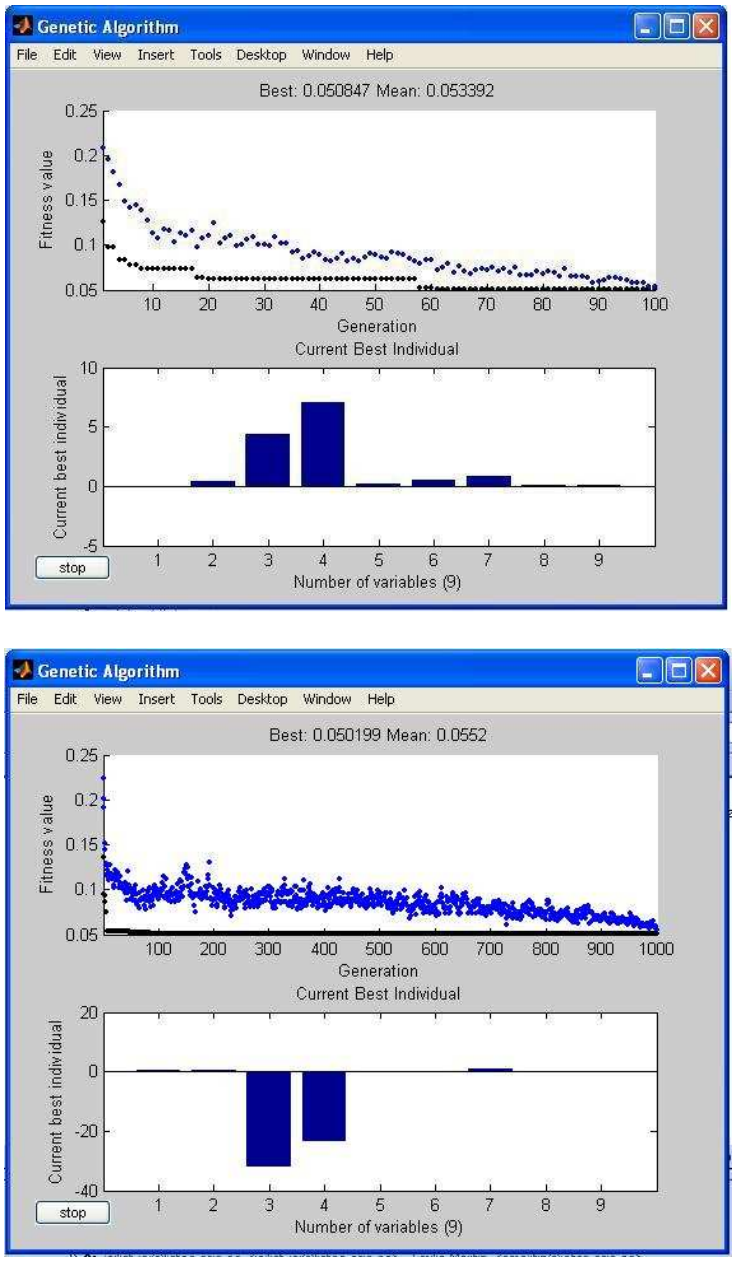

Figura 38. Mejor individuo y evolución del *fitness* a lo largo de 100 generaciones para 50 casos (arriba) y para 1000 iteraciones (abajo)

A partir del vector de pesos ***w*** obtenido en 100 iteraciones sobre 50 casos, se han realizado una serie de evaluaciones cruzadas sobre conjuntos de casos de 50, 70 y 100 elementos. Los resultados son los mostrados en la tabla 5.

K = 10						
N Casos	50	%	70	%	100	%
FP	29	**58**	53	**76**	78	**78**
FW	44	**88**	63	**90**	88	**88**
Árbol	35	**70**	60	**86**	85	**85**

K = NC						
N Casos	50	%	70	%	100	%
FP	29	**58**	52	**74**	78	**78**
FW	43	**86**	63	**90**	89	**89**
Árbol	35	**70**	53	**76**	88	**88**

Tabla 5. Aciertos en recuperación CBR para diferentes métodos y números de casos. FP función de similitud con pesos iguales para cada atributo. FW función de similitud ajustada por AG

El parámetro *k* es el número de conjuntos en los que se ha dividido el espacio de conocimiento para realizar las evaluaciones cruzadas: *k - 1* conjuntos para entrenar el sistema y 1 para evaluarlo. NC representa el número total de casos. En cada fila de la tabla se muestra cada criterio de recuperación (FP función de similitud con pesos iguales, FW función de similitud ajustada por AG y Árbol referencia al árbol de decisión) y sus recuperaciones en términos absolutos y en porcentaje para bases de conocimiento de diferentes tamaños (50, 70 y 100)

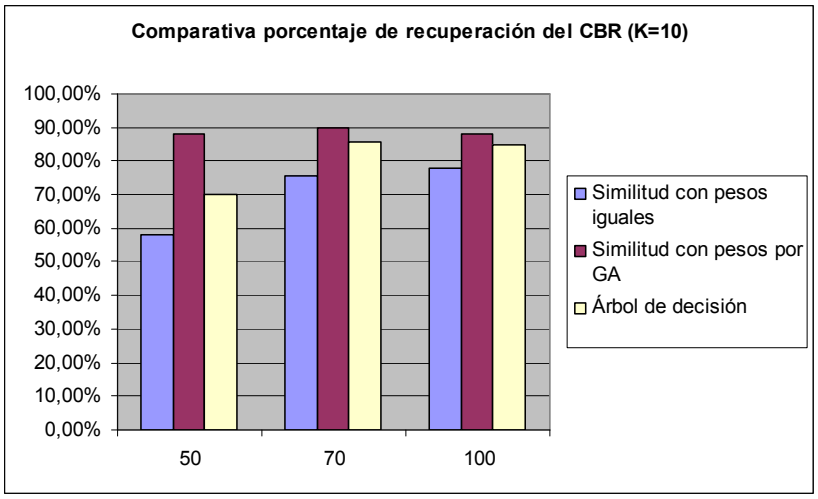

Figura 39. Comparativa porcentaje de recuperación para *k* = 10

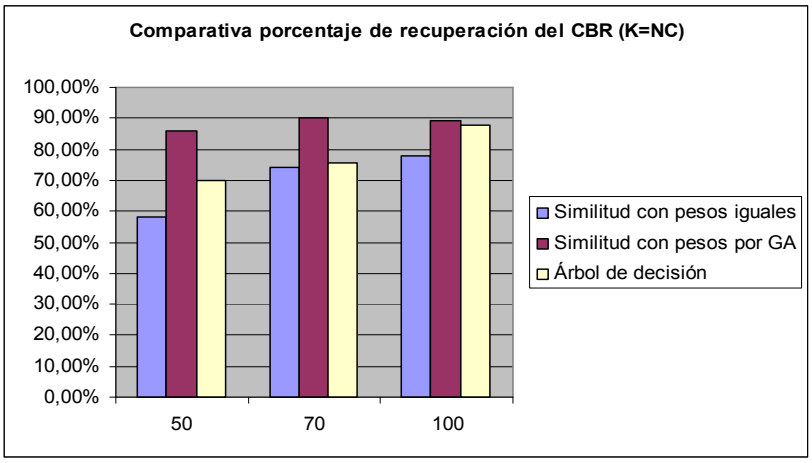

Figura 40. Comparativa porcentaje de recuperación para *k* = número de casos

Tanto en la figura 39 como en la figura 40 se puede comprobar que los resultados obtenidos para la función de similitud ajustada con algoritmos genéticos no sólo mejoran siempre, con respecto a los de similitud con pesos iguales, sino que además, también superan siempre a los del árbol de decisión para cualquiera de los conjuntos de casos evaluados.
Ninguno de los porcentajes de acierto en recuperación obtenidos para la similitud ajustada con algoritmos genéticos es inferior al 86%, lo cual demuestra que la función de similitud consigue un alto rendimiento en recuperación siempre que se ajuste bien, como es el caso que nos ocupa.

7 Conclusiones

Gracias a la integración de estos tres módulos se consigue generar un sistema experto capaz de tomar decisiones que salvaguarden la seguridad de grandes infraestructuras. De este modo, aplicando técnicas ya conocidas, se diseña un sistema avanzado capaz de aportar una solución para un problema complejo y real.

A continuación se detallan las ventajas e inconvenientes correspondientes al comportamiento del sistema global. Por tratarse de un sistema complejo compuesto por tres módulos bien definidos, se han detallado por separado tanto las ventajas como los inconvenientes. Además también se ha considerado el comportamiento del gestor de crisis, es decir del sistema global, y también aparecen analizadas sus ventajas e inconvenientes….

7.1 Ventajas

Sobre el planificador:

- Se puede construir incrementalmente
- No sólo permite una fácil inserción de obstáculos sino que también soporta su eliminación.
- Permite abstraer planos de diferentes obstáculos

Sobre el razonamiento basado en casos:

- Para la toma de decisiones, el CBR es capaz de abstraer reglas complejas. En particular un árbol de decisión puede expresarse como un sistema basado en reglas IF-THEN. Por cada nodo hoja existe una regla que se extrae de la conjunción de los atributos y valores presentes en el camino que une la raíz del árbol con la hoja. Finalmente, para cada regla, se dispararía la actuación (atributo-solución) asociada al nodo hoja que se esté analizando.
- Con un número relativamente bajo de ejemplos de entrenamiento, pero bien seleccionado, alcanza un buen rendimiento
- El árbol de decisión obtiene relaciones entre las variables de los casos más afinadamente que la función de similitud de pesos equiponderados. Es decir no se limita a dar un valor numérico (peso) que indique su relevancia en la similitud sino que es capaz de abstraer reglas de tipo IF-THEN.
- La función de similitud con pesos mediante algoritmos genéticos puede llegar a competir en rendimiento con el árbol de decisión.
- La aplicación de la ganancia como criterio de selección del mejor atributo, permite que el árbol no profundice en exceso, lo que ayudaría a entender más fácilmente la interpretación del conocimiento subyacente en la clasificación , si finalmente el árbol se traduce en reglas..

Sobre *Ice*:

- Permite un desarrollo independiente de todos los módulos del proyecto
- Soporta una gran cantidad de lenguajes de implementación

Sobre el gestor de crisis en general:

- Aborda un problema complejo que abarca varios campos y da una primera solución con resultados razonables, según se deduce de los mostrados en las tablas 4 y 5 y figuras asociadas.
- Se integra fácilmente con aplicaciones externas. Por ejemplo, podría montarse fácilmente un cliente de consulta del estado de un escenario que se conectara al gestor de crisis utilizando las interfaces definidas mediante Ice. Este cliente accedería mediante estas interfaces al estado manejado por el gestor de crisis y de este modo podría crearse una interfaz remota para visualizar la evolución de la escena.
- Al estar desarrollado en módulos, permite cambiar fácilmente cualquiera de sus partes para adoptar nuevas estrategias ya sean de planificación, razonamiento o comunicación. Es decir se trata de una estructura de diseño abierta.
- Se da una solución razonable a un problema real realizando a partir de la integración de varias técnicas de diferentes disciplinas: planificación de caminos, toma de decisiones y aplicaciones distribuidas.

7.2 Inconvenientes

Sobre el planificador:

- La búsqueda de caminos puede demorarse demasiado si el escenario es grande y la heurística no se ajusta lo suficiente. Para evitar esto, es necesario tratar de ajustar al máximo la función heurística al coste real, de este modo, se evita explorar nodos innecesarios, es decir se acota el espacio de búsqueda.

Sobre el CBR:

- Si el conjunto de ejemplos de entrenamiento es pequeño y no contiene un grado de diversidad elevado, no alcanza un buen rendimiento. Por tanto, lo ideal es conseguir ejemplos con el número de ejemplos lo más elevado posible
- Difícil de mantener, Un cambio en el diseño del caso por ejemplo añadir o eliminar una nueva variable en la representación del caso alteraría las cuatro fases del ciclo de vida del CBR: recuperación, reutilización, revisión y recuerdo.
- El ruido, aún en pequeñas magnitudes, afecta excesiva y negativamenteal rendimiento
- Si se determinan los coeficientes mediante genéticos y se modifica la base de conocimiento, por ejemplo añadiendo más casos, es necesario recalcularlos, o en caso contrario, los coeficientes no estarían ajustados a todo el conocimiento almacenado. No sucede así con el árbol de decisión, puesto que se reconstruye cada vez que se modifica la base de conocimiento.

Sobre Ice:

- Complejo de manejar de cara a la implementación

Sobre el gestor de crisis en general:

- El razonamiento basado en casos no parece el mejor enfoque para la toma de decisiones en situaciones de intrusión ya que es difícil encontrar la similitud entre las disposiciones espaciales de los escenarios. Para solventar este problema una opción puede ser entender los escenarios como grafos cuyos nodos son los elementos clave de la escena y calcular el nivel de riesgo asociado a un estado del escenario y navegar entre todos los posibles actuadores aplicables buscando aquél que disminuya más el nivel. De este modo el problema se simplificaría a una búsqueda sobre un espacio de operadores orientada a alcanzar un estado donde el riesgo sea mínimo.

7.3 Trabajo futuro

- El árbol de decisión no tiene por qué encontrar siempre un caso similar al consultado y por tanto puede suceder que no se encuentre ninguna experiencia anterior en la que basarse y no disponer de ninguna solución. En el planteamiento actual del CBR, para solventar esta situación, se devuelve la solución más frecuente en el entorno más similar del caso consultado. Sin embargo, ésta es una solución basada en probabilidad que puede desviarse mucho del verdadero valor. Una posible mejora futura sería implementar alguna técnica capaz de tratar con este tipo de situaciones.
- Otra alternativa es buscar un nuevo enfoque para el sistema experto encargado de la toma de decisiones. Una opción es tratar de calcular el nivel de riesgo asociado a un estado del escenario y navegar entre todos los posibles actuadores aplicables buscando aquél que disminuya más el nivel. De este modo el problema se simplificaría a una búsqueda sobre un espacio de operadores orientada a alcanzar un estado donde el riesgo sea mínimo.

8 Bibliografía

[Aamodt et al., 1994] Aamodt, A. and Plaza, E. (1994). Case-Based Reasoning: Foundational Issues, Methodological Variations and System Approaches AI Communications, Vol. 7 Nr. 1, pp 39-59

[Althoff et al., 1995] Althoff K.-D., Auriol E., Barletta R. and Manago M. (1995). A Review of Industrial Case-Based Reasoning Tools. AI Intelligence, Oxford UK.

[Anglada, 1997] Anglada, M. V. (1997). An Improved Incremental Algorithm for Constructing Restricted Delaunay Triangulations, Computer & Graphics, 21(2):215-223.

[Bishop, C.M., 2007]. Bishop, C.M. (2007) Pattern Recognition and Machine Learning. Springer, New York

[Blickle & Thiele, 1995] Blickle, T. and Thiele, L. (1995). A comparison of selection schemes used in genetic algorithms. Technical Report 11, Computer Engineering and Comunnication Network Lab (TIK), Gloriastrasse 35, 8092 Zurich, Switzerland

[Buckinx et al., 2004] Buckinx, M., Moons, E., Poel, D. V. D. and Wets, G. (2004). Customer-adapted coupon targeting using feature selection. Expert Systems with Applications, 26, 509-518.

[Chazelle, 1982] Chazelle, B. (1982). A Theorem on Polygon Cutting with Applications. In Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science, 339-349.

[Conesa & Ribeiro, 2009] Conesa, J and Ribeiro, A. (2009). Performing the retrieve step in a case-based reasoning system for decision making in intrusion scenarios. In Proceedings of the 11th International Conference on Enterprise Information Systems.

[Cormen et al., 1993] T. Cormen, C. Leiserson, and Rivest, R. (1993). Introduction to Algorithms, MIT Press, Cambridge, MA.

[Demyen, 2006] Demyen, D. J. (2006). Efficient Triangulation-Based Pathfinding. Thesis, University of Alberta.

[Duda, R. O. et al, 2007] Duda, R.O., Hart, P.E. and Stork, D.G. (2001). Pattern Classification, Wiley and Sons, New York.

[Floriani et al., 1992] Floriani, L., and Puppo, A. (1992). An On-Line Algorithm for Constrained Delaunay Triangulation, Computer Vision, Graphics and Image Processing, 54:290-300.

[Grimes, R. 1998]. Grimes, R. (1998). Professional DCOM Programming. Chicago, IL: WroxPress.

[Guibas & Stolfi, 1985] Guibas, L. and Stolfi, J. (1985). Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. ACM Transactions on Graphics, 4(2), 75-123.

[Hesperia, 2009] www.proyecto-hesperia.org

[Henning, M., and S. Vinoski. 1999]. Henning, M. and S. Vinoski. (1999). Advanced CORBA Programming with C++. Reading, MA: Addison-Wesley.

[Kallmann et al., 2003] Kallmann M., Bieri, H. and Thalmann, D. (2006). Fully Dynamic Constrained Delaunay Triangulations, In Geometric Modelling for Scientific Visualization, G. Brunnett, B. Hamann, H. Mueller, L. Linsen (Eds.), ISBN 3-540-40116-4, Springer-Verlag, Heidelberg, Germany, pp. 241-257.

[Koenig, 2004] Koenig, S. (2004). A Comparison of Fast Search Methods for Real-Time Situated Agents, AAMAS'04, July 19-23, New York.

[Kohavi, R., 1995] Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence 2 (12): 1137–1143.(Morgan Kaufmann, San Mateo)

[Kreveld et al., 2000] Kreveld, M. V., Overmars, M., Schwarzkopf, O. and de Berg, M. (2000). Computational Geometry: Algorithms and Applications, ISBN 3-540-65620-0 Springer-Verlag.

[Lee et al., 1984] Lee, D. T., and Preparata, F. P. (1984). Euclidean Shortest Paths in the Presence of rectilinear barriers. Networks. 14(3):393-410.

[Lischinski, 1994] Lischinski, D. (1994). Incremental Delaunay triangulation, Graphics gems IV, Academic Press Professional, Inc., San Diego, CA, 1994

[Michalewicz, 1994] Michalewicz, Z. (1994). Genetic Algorithms + Data Structures + Evolution Programs (2nd edition ed.), Springer, New York.

[Pal & Shiu, 2004] Pal, S. K. and Shiu, S. C. K. (2004). Foundations of soft case-based reasoning. Editorial John Wiley & sons, inc.

[Preparata et al., 1985] Preparata, F. P., and Shamos, M. I. (1985) Computational Geometry: An Introduction. Springer-Verlag, ISBN 3540961313.

[Quinlan, 1986] Quinlan, J. R. (1986). Induction of Decision Trees. Mach. Learn. 1, 1, 81-10

[Ross, 1989] Ross, B. H. (1989). Some psychological results on case-based reasoning Case-Based Reasoning, Workshop, DARPA 1989. Pensacola Beach. Morgan Kaufmann. pp. 144-147.

9 Apéndice A: Aplicaciones desarrolladas

En esta sección se describe un breve tutorial sobre las aplicaciones implementadas para el desarrollo del trabajo.

9.1 PathFinding

En primer lugar, se describe la aplicación PathFinding. Su funcionalidad es la de representar el plano de un recinto y ser capaz de calcular sobre él los caminos mínimos entre distintas posiciones. Mediante este programa podemos representar visualmente escenas o casos y guardarlos en un formato adecuado consistente con la entrada del CBR. De este modo se pueden crear casos rápidamente y de forma visual, lo que resulta mucho más fácil e intuitivo que editar un fichero de texto.

En la figura 41, se observan tres posiciones destacadas: intrusos, objetivo y guardias, así como los caminos mínimos que las conectan. Las líneas rojas determinan los obstáculos y las grises segmentan el espacio (triangulación), estas últimas no son relevantes para caracterizar el caso o escena. Si se pulsa en la opción de menú "Case", se puede configurar las variables del caso (figura 42)

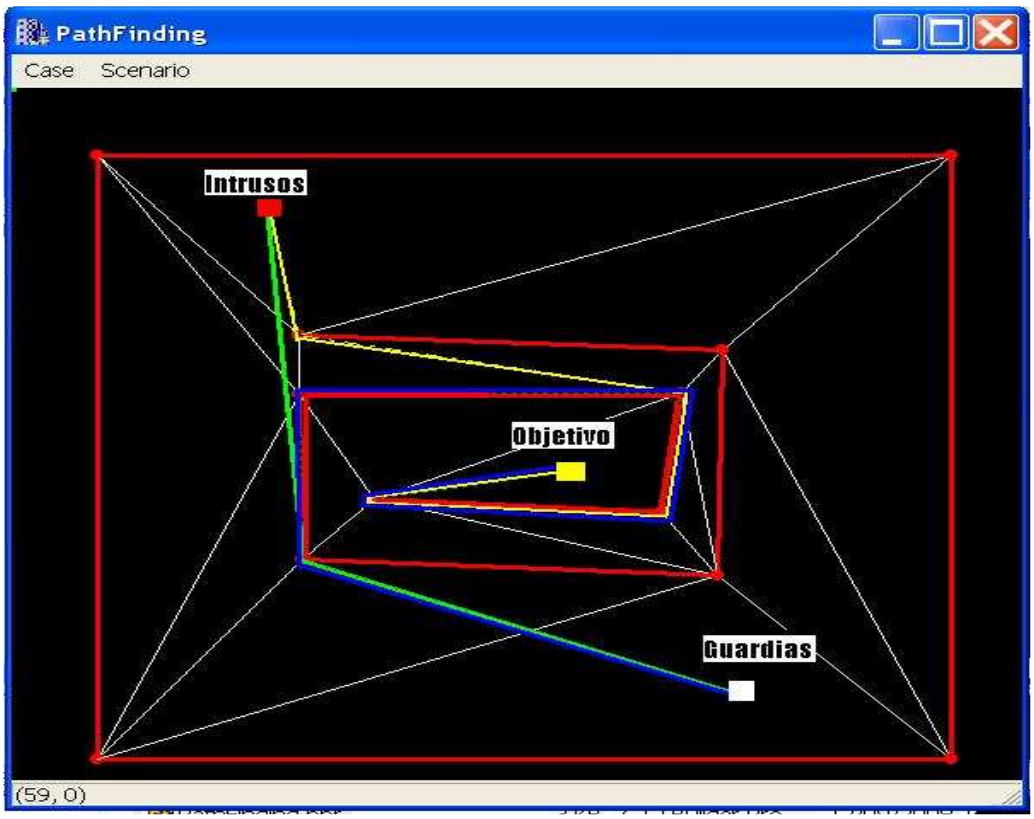

Figura 41. Aplicación PathFinding

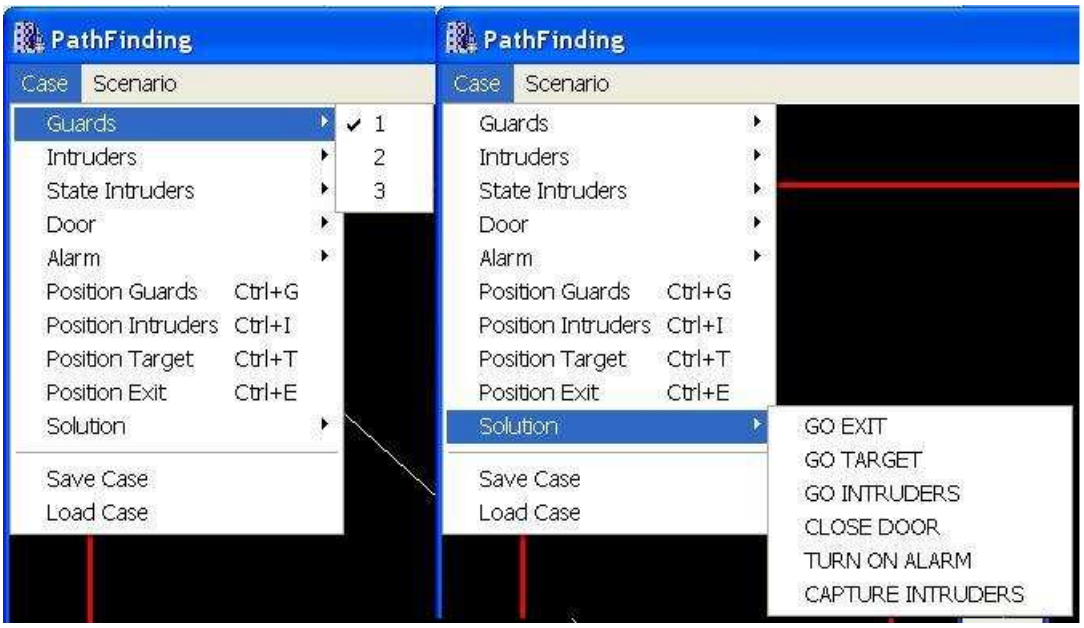

Figura 42. PathFinding: Configuración de un caso

La opción "Save Case" almacena la representación del caso en pantalla en un fichero de texto. El formato usado es el siguiente:

<idCase , vRatio, nG, nI, vState, vDoor, vAlarm, dIT, dIE, dGI, dGT, dGE, pxG pyG pxI pyI pxT pyT pxE pyE vSol >

- idCase = identificador del caso
- vRatio = valor de la variable ratio
- nG = número de guardias
- nI = número de intrusos
- vState = valor de la variable state
- vDoor = valor de la variable door
- vAlarm = valor de la variable alarm
- dIT = distancia entre intrusos y objetivo
- dIE = distancia entre intrusos y salida
- dGI = distancia entre guardias e intrusos
- dGT = distancia entre guardias y objetivo
- dGE = distancia entre guardias y salida
- pxG = coordenada x de los guardias dentro del mapa
- pyG = coordenada y de los guardias dentro del mapa
- pxI = coordenada x de los intrusos dentro del mapa
- pyI = coordenada y de los intrusos dentro del mapa
- pxT = coordenada x del objetivo dentro del mapa
- pyT = coordenada y del objetivo dentro del mapa
- pxE = coordenada x de la salida dentro del mapa

- pyE = coordenada y de la salida dentro del mapa
- vSol = valor de la variable solución

Ejemplo:

<1, EQUAL, 1, 1, TARGET, NOT_AVAILABLE_D, OFF, FAR_D, MEDIUM, FAR_D, FAR_D, MEDIUM, 1042 729 155 111 1035 149 172 717 TURN_ON_ALARM >

Se guarda información no relacionada directamente con el CBR (como las coordenadas de las posiciones de algunos elementos o el número exacto de guardias o intrusos) por ser necesarios para representar visualmente la información en PathFinding si se selecciona la opción “Load Case”.

9.2 Intruders CBR

La aplicación Intruders CBR implementa el sistema de razonamiento basado en casos analizado a lo largo del trabajo. Dispone de un panel de visualización del árbol de decisión generado, un panel de listados de resultados y un panel de creación de casos para llevar a cabo operaciones sobre el caso creado, tales como inserción en el árbol o encontrar los casos más similares mediante una función de similitud o mediante una clasificación utilizando el árbol de decisión. En la figura 43 se pueden apreciar los diferentes paneles y menús que componen la aplicación.

9.2.1 Panel del árbol de decisión generado

Muestra una representación del árbol de decisión generado. En cada nodo y en mayúsculas aparece anotada la variable seleccionada en ese nivel (aquélla que producía mayor ganancia). Desde cada variable (nodo) se despliega una rama por cada uno de lo posibles valores (anotados en minúsculas) hasta el siguiente nivel. En el siguiente nivel se vuelve a repetir el mismo proceso hasta alcanzar un nodo final (aquél que alberga el atributo-objetivo o clase). Los nodos finales están por tanto marcados con la variable solución y con una lista de casos relacionados. En la figura 43, si por ejemplo se sigue la rama ALARM = on, DISTANCE_GI = close, DISTANCE_IE = close, se alcanza un nodo final cuya lista de casos asociados es [42, 44, 45]. Todos ellos comparten la solución GO_INTRUDERS, que es la que devolvería el sistema en caso de ser consultado por un caso que encajase en los valores de este camino.
En el supuesto de que el nodo final no tuviera ningún caso asociado, se devolvería como solución la más frecuente en los nodos del mismo nivel. En la figura 43 se aprecia esta situación en la rama ALARM = on, DISTANCE_GI = medium, DISTANCE_GT = medium, DISTANCE_IT = far, donde la solución devuelta es CAPTURE_INTRUDERS por ser la más frecuente en el grupo de nodos del mismo nivel. .

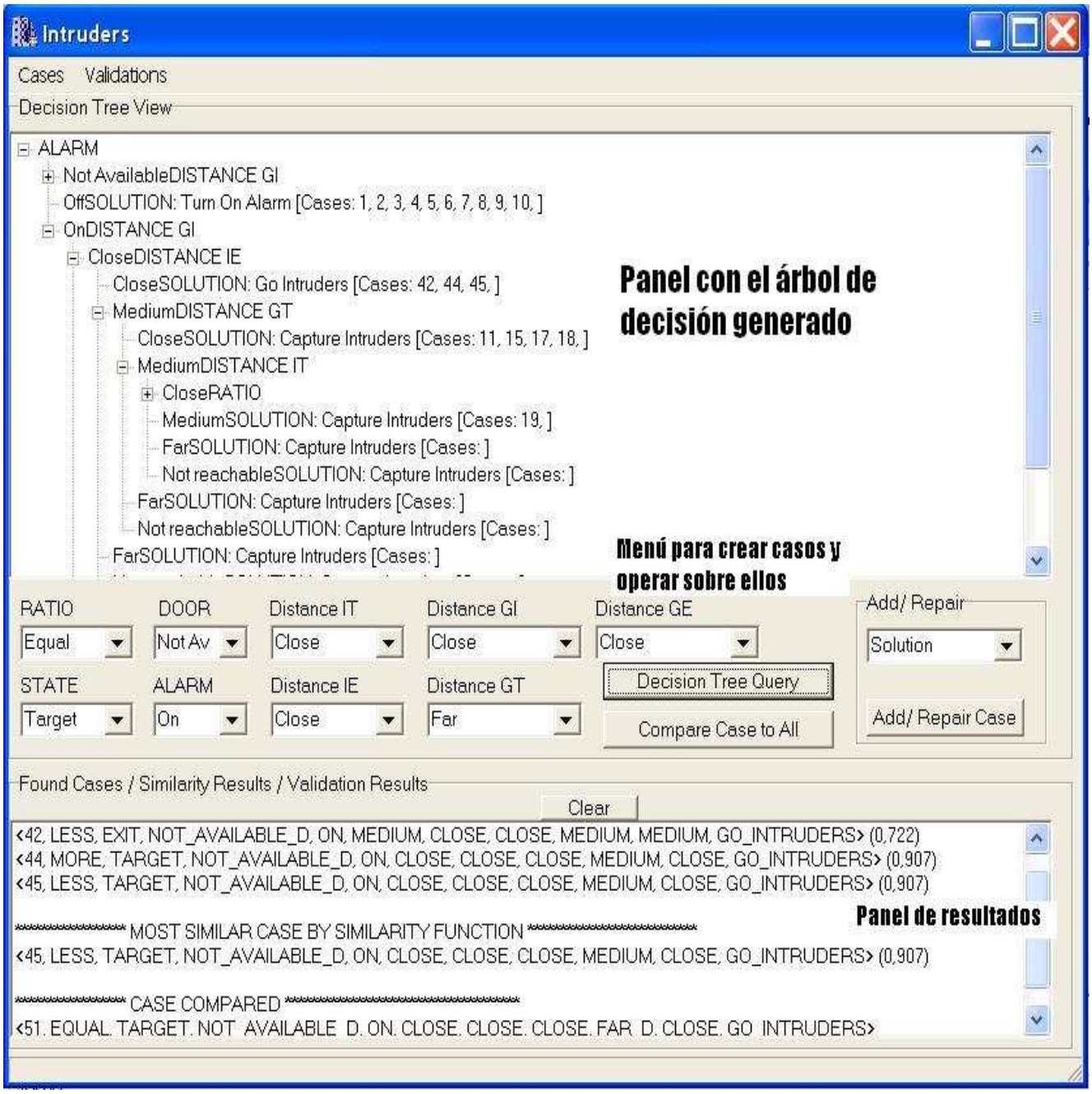

Figura 43. Aplicación Intruders CBR

9.2.2 Menú de creación de casos

La finalidad del menú de creación de casos, es permitir evaluar un caso frente a todos los presentes en el sistema. Las operaciones admitidas son:

- Consultar en el árbol: busca los casos relacionados siguiendo las ramas correspondientes del árbol de decisión (aquéllas que contiene los valores del caso evaluado)
- Comparar con todos: compara el caso creado, aplicando la función de similitud, con todos los demás cargados en el sistema. Devuelve un listado con los valores de similitud entre cada caso presente y el evaluado. Además, muestra destacado el caso más similar de todos.
- Añadir/ Reparar: Tanto la operación de consultar como la de reparar, tras encontrar los casos más similares, asignan el valor del atributo solución de los casos encontrados al caso comparado. Con la opción de añadir/ reparar, se pretende simular el proceso de reparación propio de la fase de revisión. Una vez realizada la consulta y obtenido el caso más similar y su solución, el experto tendría la posibilidad de añadir el caso al sistema (si considera satisfactoria la solución ofrecida), y ampliar así la base de conocimiento.

9.2.3 Panel de resultados

En este panel se muestra la información asociada a los resultados de las operaciones descritas en el apartado anterior. Además, visualiza los resultados de las validaciones cruzadas que se pueden aplicar al CBR. Para ejecutar una validación cruzada sobre el sistema, basta con pinchar sobre una de las opciones disponibles en la opción "Validations" del menú superior. El parámetro *k*, determina el número de grupos en el que se segmentará el conjunto de casos. Es interesante recordar que para *k* = Número de Casos (NC), la validación realizada corresponde con una de tipo *leave-one-out.*

Los resultados de las validaciones también se pueden guardar en un fichero de texto.

10 Apéndice B: Base de casos

10.1 Base de conocimiento y árbol de decisión generado

Este es un conjunto de casos de 50 elementos junto con el árbol de decisión generado a partir de él. Los casos están agrupados según su actuación (atributo-solución) asociada:

<Identificador de caso, Ratio, State, Door, Alarm, Distance IT, Distance IE, Distance GI, Distance GT, Distance GE >
Solution**: TURN_ON_ALARM**
<1, EQUAL, TARGET, NOT_AVAILABLE_D, OFF, FAR_D, MEDIUM, FAR_D, FAR_D, MEDIUM >
<2, EQUAL, TARGET, NOT_AVAILABLE_D, OFF, FAR_D, MEDIUM, MEDIUM, MEDIUM, MEDIUM >
<3, LESS, EXIT, NOT_AVAILABLE_D, OFF, FAR_D, MEDIUM, MEDIUM, MEDIUM, MEDIUM >
<4, MORE, EXIT, NOT_AVAILABLE_D, OFF, FAR_D, MEDIUM, MEDIUM, MEDIUM, CLOSE >
<5, EQUAL, EXIT, NOT_AVAILABLE_D, OFF, MEDIUM, CLOSE, CLOSE, MEDIUM, CLOSE >
<6, LESS, EXIT, NOT_AVAILABLE_D, OFF, MEDIUM, CLOSE, CLOSE, MEDIUM, CLOSE >
<7, LESS, TARGET, NOT_AVAILABLE_D, OFF, CLOSE, CLOSE, CLOSE, CLOSE, CLOSE >
<8, MORE, EXIT, NOT_AVAILABLE_D, OFF, CLOSE, FAR_D, CLOSE, CLOSE, FAR_D >
<9, LESS, TARGET, NOT_AVAILABLE_D, OFF, CLOSE, FAR_D, MEDIUM, MEDIUM, FAR_D >
<10, LESS, EXIT, NOT_AVAILABLE_D, OFF, CLOSE, FAR_D, MEDIUM, MEDIUM, FAR_D >
Solution**: CAPTURE_INTRUDERS**
<11, MORE, TARGET, NOT_AVAILABLE_D, ON, CLOSE, MEDIUM, CLOSE, CLOSE, MEDIUM >
<12, MORE, TARGET, NOT_AVAILABLE_D, NOT_AVAILABLE, FAR_D, CLOSE, CLOSE, MEDIUM, CLOSE >
<13, EQUAL, TARGET, NOT_AVAILABLE_D, NOT_AVAILABLE, MEDIUM, CLOSE, CLOSE, MEDIUM, CLOSE >
<14, MORE, EXIT, NOT_AVAILABLE_D, NOT_AVAILABLE, MEDIUM, MEDIUM, CLOSE, FAR_D, FAR_D >
<15, EQUAL, EXIT, NOT_AVAILABLE_D, ON, CLOSE, MEDIUM, CLOSE, CLOSE, FAR_D >
<16, EQUAL, EXIT, NOT_AVAILABLE_D, NOT_AVAILABLE, CLOSE, MEDIUM, CLOSE, CLOSE, MEDIUM >
<17, EQUAL, EXIT, NOT_AVAILABLE_D, ON, CLOSE, MEDIUM, CLOSE, CLOSE, CLOSE >
<18, EQUAL, TARGET, NOT_AVAILABLE_D, ON, CLOSE, MEDIUM, CLOSE, CLOSE, CLOSE >
<19, EQUAL, TARGET, NOT_AVAILABLE_D, ON, MEDIUM, MEDIUM, CLOSE, MEDIUM, MEDIUM >
<20, EQUAL, TARGET, NOT_AVAILABLE_D, ON, CLOSE, MEDIUM, CLOSE, MEDIUM, MEDIUM >

Solution: **GO_TO_TARGET**
<21, LESS, TARGET, NOT_AVAILABLE_D, NOT_AVAILABLE, FAR_D, CLOSE, MEDIUM, MEDIUM, CLOSE >
<22, EQUAL, TARGET, NOT_AVAILABLE_D, NOT_AVAILABLE, MEDIUM, CLOSE, MEDIUM, CLOSE, CLOSE >
<23, MORE, TARGET, NOT_AVAILABLE_D, ON, MEDIUM, FAR_D, MEDIUM, CLOSE, FAR_D >
<24, LESS, TARGET, NOT_AVAILABLE_D, ON, MEDIUM, CLOSE, MEDIUM, CLOSE, MEDIUM>
<25, LESS, TARGET, NOT_AVAILABLE_D, NOT_AVAILABLE, MEDIUM, CLOSE, MEDIUM, CLOSE, MEDIUM >
<26, MORE, TARGET, NOT_AVAILABLE_D, NOT_AVAILABLE, MEDIUM, MEDIUM, MEDIUM, MEDIUM, CLOSE >
<27, MORE, TARGET, NOT_AVAILABLE_D, NOT_AVAILABLE, FAR_D, FAR_D, FAR_D, MEDIUM, MEDIUM >
<28, EQUAL, TARGET, NOT_AVAILABLE_D, ON, FAR_D, FAR_D, FAR_D, CLOSE, CLOSE, GO_TARGET >
<29, EQUAL, TARGET, NOT_AVAILABLE_D, ON, MEDIUM, MEDIUM, MEDIUM, CLOSE, CLOSE >
<30, LESS, TARGET, NOT_AVAILABLE_D, NOT_AVAILABLE, MEDIUM, FAR_D, FAR_D, MEDIUM, CLOSE >
Solution: **GO_TO_EXIT**
<31, LESS, EXIT, NOT_AVAILABLE_D, NOT_AVAILABLE, MEDIUM, FAR_D, FAR_D, MEDIUM, CLOSE >
<32, EQUAL, EXIT, NOT_AVAILABLE_D, NOT_AVAILABLE, MEDIUM, FAR_D, MEDIUM, CLOSE, CLOSE >
<33, EQUAL, EXIT, NOT_AVAILABLE_D, ON, CLOSE, MEDIUM, FAR_D, FAR_D, MEDIUM >
<34, LESS, EXIT, NOT_AVAILABLE_D, ON, CLOSE, FAR_D, FAR_D, FAR_D, FAR_D >
<35, MORE, EXIT, NOT_AVAILABLE_D, ON, FAR_D, FAR_D, FAR_D, FAR_D, FAR_D >
<36, MORE, EXIT, NOT_AVAILABLE_D, ON, FAR_D, FAR_D, FAR_D, CLOSE, MEDIUM >
<37, MORE, EXIT, NOT_AVAILABLE_D, NOT_AVAILABLE, MEDIUM, FAR_D, MEDIUM, CLOSE, MEDIUM >
<38, LESS, EXIT, NOT_AVAILABLE_D, NOT_AVAILABLE, FAR_D, FAR_D, MEDIUM, MEDIUM, MEDIUM >
<39, EQUAL, EXIT, NOT_AVAILABLE_D, NOT_AVAILABLE, FAR_D, MEDIUM, MEDIUM, MEDIUM, CLOSE >
<40, MORE, EXIT, NOT_AVAILABLE_D, NOT_AVAILABLE, MEDIUM, MEDIUM, MEDIUM, CLOSE, CLOSE >
Solution: **GO_TO_INTRUDERS**
<41, LESS, EXIT, NOT_AVAILABLE_D, NOT_AVAILABLE, CLOSE, CLOSE, CLOSE, CLOSE, MEDIUM >
<42, LESS, EXIT, NOT_AVAILABLE_D, ON, MEDIUM, CLOSE, CLOSE, MEDIUM, MEDIUM >

<43, EQUAL, TARGET, NOT_AVAILABLE_D, ON, CLOSE, MEDIUM, CLOSE, MEDIUM, MEDIUM >
<44, MORE, TARGET, NOT_AVAILABLE_D, ON, CLOSE, CLOSE, CLOSE, MEDIUM, CLOSE >
<45, LESS, TARGET, NOT_AVAILABLE_D, ON, CLOSE, CLOSE, CLOSE, MEDIUM, CLOSE >
<46, LESS, TARGET, NOT_AVAILABLE_D, NOT_AVAILABLE, CLOSE, MEDIUM, CLOSE, MEDIUM, CLOSE >
<47, EQUAL, TARGET, NOT_AVAILABLE_D, NOT_AVAILABLE, CLOSE, FAR_D, MEDIUM, MEDIUM, FAR_D >
<48, MORE, TARGET, NOT_AVAILABLE_D, NOT_AVAILABLE, MEDIUM, FAR_D, MEDIUM, FAR_D, FAR_D >
<49, MORE, EXIT, NOT_AVAILABLE_D, NOT_AVAILABLE, FAR_D, MEDIUM, MEDIUM, FAR_D, FAR_D >
<50, MORE, EXIT, NOT_AVAILABLE_D, NOT_AVAILABLE, FAR_D, CLOSE, FAR_D, FAR_D, FAR_D >

Tabla 6. Base de conocimiento de 50 casos

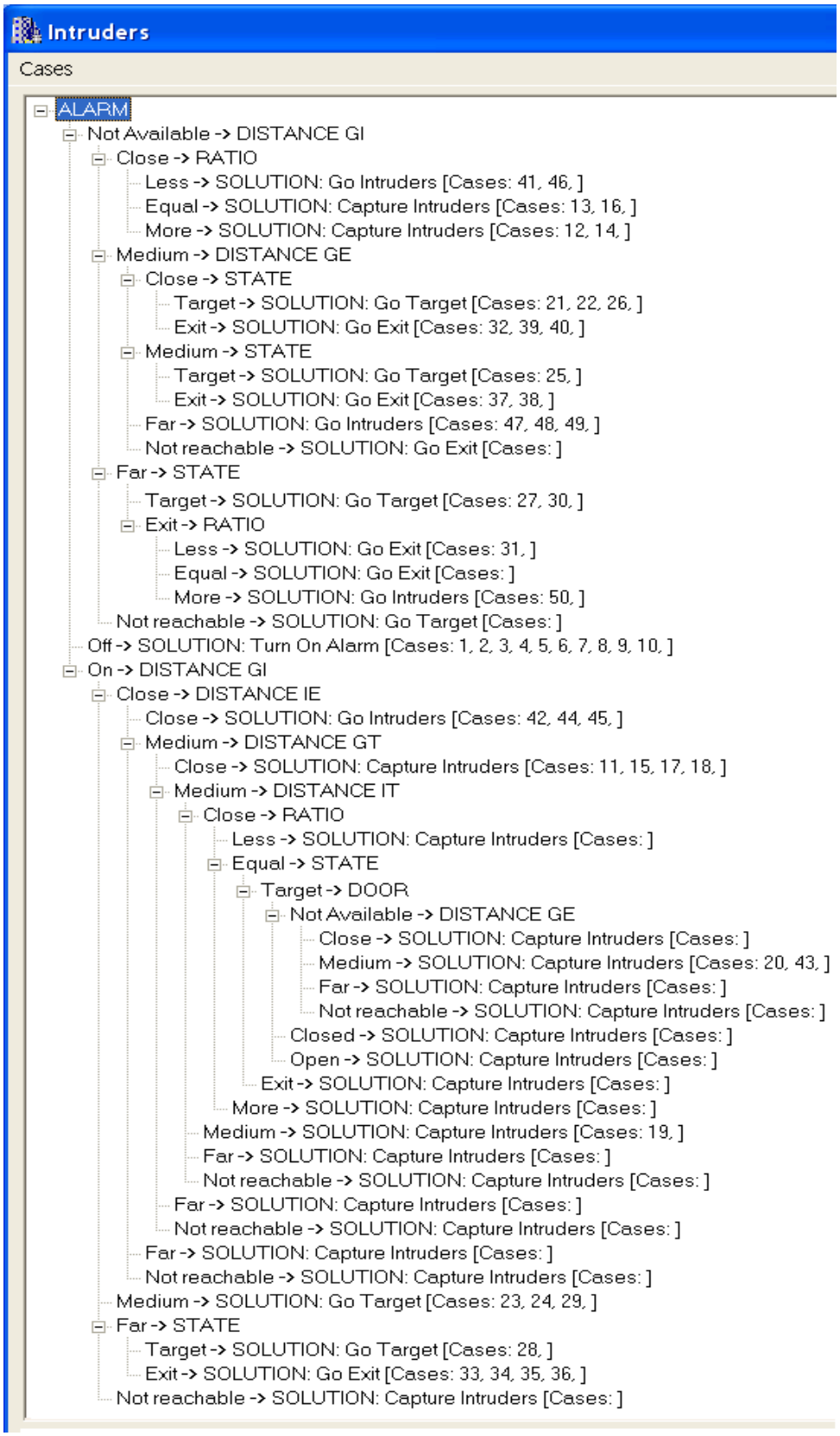

Figura 44. Árbol de decisión para los 50 casos de la tabla 6

11 Apendice C: Procedimientos y listado de clases

11.1 Planificador

11.1.1 Locate

```
PROCEDURE Locate(p:Point) RETURNS [e]:
        Edge e ← some edge;
        DO
                IF (p = e.Org) THEN
                        RETURN e;
                ELSIF (p = e.Dest) THEN
                        RETURN e.Sym;
                ELSIF (RightOf(p, e)) THEN
                        e ← e.Sym;
                ELSIF (NOT RightOf(p, e.Onext)) THEN
                        e ← e.Onext;
                ELSIF (NOT RightOf(p, e.Dprev)) THEN
                        e ← e.Dprev;
                ELSE
                        RETURN e;
                FI
        OD
END Locate.
```

11.1.2 InsertPoint

```
PROCEDURE  InsertPoint(p:Point)
        Edge e ← Locate(p);
        IF (p = e.Org OR p = e.Dest) THEN
                { Point is already in}
                RETURN;
        ELSIF (OnEdge(p,e)) THEN
                Point a ← e.Org;
                Point b ← e.Dest;
                DeleteEdge(e);
                InsertPointInLeftFace(e.Lnext, p);
                {Locates the two 'subedges' created from the original edge e}
                Edge e1 ← LocateEdge(p, a);
                Edge e2 ← LocateEdge(p, b);
                IF (e IS constrained) THEN
                e1.ListOfConstraints ← list of constraint identifiers of e;
                        e2.ListOfConstraints ← list of constraint identifiers of e;
                FI
        ELSE
```

```
            InsertPointInLeftFace(e, p);
        FI;
END InsertPoint.
```

11.1.3 InsertPointInLeftFace

```
PROCEDURE InsertPointInLeftFace (e:Edge, p:Point)
        {Connect X to vertices around it}
        Edge base ← MakeQuadEdge();
        Point first ← e.Org;
        base.Org ← first;
        base.Dest ← p;
        Splice(base, e);
        REPEAT
            base ← Connect(e, base.Sym);
            e ← base.Oprev;
        UNTIL (e.Dest = first);
        e ← base.Oprev;
        {The suspect edges (from top to bottom) are e (. Onext.Lprev)^k for k = 0, 1, . . ..) (The bottom
        edge has .Org = first.}
        DO
            Edge t ← e.Oprev;
            IF (RightOf (t.Dest, e) AND InCircle(e.Org, t.Dest, e.Dest, p)) THEN
                Swap(e);
                e ← t;
            ELSIF (e.Org = first) THEN
                {No more suspect edges}
                RETURN;
            ELSE {Pop a suspect edge}
                e ← e.Onext.Lprev;
            FI;
        OD;
END InsertPointInLeftFace.
```

11.1.4 InsertSegment

```
PROCEDURE InsertSegment (a:Point, b:Point, i:Identifier)
        InsertPoint(a);
        InsertPoint(b);
        edge_list ← all constrained edges crossed by segment {a, b};
        FOR all edges e IN edge_list
            pc ← intersection point between e and segment {a, b};
            InsertPointInEdge(pc, e);
        END;
```

```
        edge list ← all edges crossed (not overlapped) by segment {a, b};
        FOR all edges e IN edge_list
                DELETE edge e from the CDT;
        END;
        vertex_list ← all vertices crossed by segment {a, b};
        FOR all vertices v IN vertex_list
                IF (v IS NOT the last element IN vertex list) THEN
                        vs ← successor vertex of v in vertex list;
                        IF (v and vs are connected by an edge) THEN
                                e ← edge connecting v and vs;
                                add index i to list of constraints of e;
                        ELSE
                                e ← MakeQuadEdge(v, vs, true);
                                e ← add new edge in the CDT connecting v and vs;
                                add index i to list of constraints of e;
                                retriangulate the two faces adjacent to e;
                        FI;
                FI;
        END;
END InsertSegment;
```

11.1.5 InsertConstraint

```
PROCEDURE InsertConstraint(c:Constraint, i: Identifier)
        FOR all vertIces v IN c.vertex_list
                IF (v is not the last element in c.vertex list) THEN
                        vs ← successor vertex of v in c.vertex list;
                        InsertSegment(v, vs, i);
                FI;
        END;
END InsertConstraint;
```

11.1.6 TriangulatePseudopolygonDelaunay

```
PROCEDURE TriangulatePseudopolygonDelaunay(pol:VertexList, ab:Edge, t:CDT)
        IF (pol has more than one element) THEN
                c ← First vertex of pol;
                FOR  EACH vertex v IN pol DO
                        IF (v is into Circumcircle(a,b,c)) THEN
                                c ← v;
                        FI;
                OD;
                Divide pol into polE and polD, giving pol = polE + {c} + polD;
                TriangulatePseudopolygonDelaunay(polE,ac,t);
```

```
            TriangulatePseudopolygonDelaunay(polD,cb,t);
        FI;
        IF (pol is NOT empty) THEN
            Add triangle with vertices a, b, c into t;
        FI;
END TriangulatePseudopolygonDelaunay.
```

11.1.7 RemoveConstraint

```
PROCEDURE RemoveConstraint (i:Identifier)
        { step 1}
        v ← one vertex of the constraint i;
        put on stack all incident edges to v representing the constraint i;
        mark all edges in stack;
        WHILE ( stack IS NOT empty )
            Edge e ← stack.pop();
            add e to edge list;
            push to stack all incident edges to e representing constraint i and which are not
            marked; ensure all edges in stack are marked;
            FOR all edges e IN edge list DO
                remove index i from list of constraints identifier of e;
            OD;
        ENDWHILE;
        { step 2 }
        vertex list ← all vertices which are endpoints of edges in edge list;
        FOR all vertices v IN vertex list DO
            ref ← number of different indices referenced by the remaining constrained edges
        adjacent to v;
            n ← number of remaining constrained edges adjacent to v;
            IF ( n = 0 ) THEN
                Edge eFace ← RemoveVertex ( v );
                TriangulateFace(eFace);
            ELSIF ( n = 2 ) THEN
                let e1 and e2 be the remaining constrained edges adjacent to v;
                IF ( list of constraints of e1 = list of constraints of e2 AND e1 is collinear to
                e2 ) THEN
                    let v1 and v2 be the two vertIces incident to e1 and e2, and which
                    are different than v;
                    crep ← lisf of constraints identifiers of e1;
                    Edge eFace ← RemoveVertex ( v );
                    TriangulateFace(eFace);
                    Edge e = InsertSegment ( v1, v2, -1);
                    list of constraints of e ← crep;
                FI;
            FI;
        OD;
```

END RemoveConstraint.

11.1.8 RemoveVertex

```
PROCEDURE RemoveVertex(p:Point)
        Edge e ← Locate(p);
        Edge eAux ← e;
        WHILE (e.Onext IS NOT e) DO
                e ← eAux.Onext;
                DELETE eAux from the CDT;
                eAux ← e;
        ENDWHILE;
        DELETE eAux from the CDT;
END RemoveVertex.
```

11.1.9 TA*

```
PROCEDURE TA*(t:CDT, a:Point, b:Point) RETURN [c:Triangles_List]
        Triangle ta ← Locate(a);
        ta.g ← 0;
        ta.h ← Euclidean Distance between a and b;
        Priority_Queue pq ← [ta];
        WHILE (NOT (goal node found OR pq is empty))
                bestNode ← node with minimal f from pq;
                IF (bestNode CONTAINS b point) THEN
                        goalNode ← bestNode;
                ELSE
                        Generate a successor of bestNode for each not constraint side and not
                        explored yet;
                        Add each successor on pq;
                FI;
        ENDWHILE;
        RETURN goalNode.channel;
END TA*.
```

11.1.10 GenerateSuccessor

```
PROCEDURE GenerateSuccessor(parent:Node, e:Edge, a:Point, b:Point) RETURN [suc:Node]
        suc ← CreateNode();
        p ← Point on e closest to b;
        suc.h ← Euclidean Distance between p and b;
        p ← Point on e closest to a;
        distEA ← Euclidean Distance between p and a;
```

```
        distFunnel ← funnel(a, Point that minimices cost);
        suc.g ← MAX(parent.g, distEA, distFunnel);
        RETURN suc;
END GenerateSuccessor.
```

11.1.11 Funnel

```
PROCEDURE Funnel(ch:Channel, start:Point, goal:Point) RETURN [path:PointList]
        path.Clear();
        IF (NumEdges(Ch) < 1) THEN
                path.Add(start);
                path.Add(goal);
                RETURN path;
        FI;
        apex ← start;
        path.Add(start);
        funnelLeft.Add(start);
        funnelRight.Add(start);
        crossedEdge ← First edge from the channel ch;
        funnelLeft.Add(crossedEdge.Org());
        funnelRight.Add(crossedEdge.Dest());
        FOR each inner edge ON the channel ch DO
                crossedEdge ← inner edge i from the channel ch;
                pLeft ← funnelLeft.Front();
                pRight ← funnelRight.Front();
                pL ← crossedEdge.Org();
                pR ← crossedEdge.Dest();
                IF (pRight == pR) THEN {The next point is on the left side of the funnel}
                        IF (pL opens the Funnel) THEN
                                funnelLeft.Add(pL);
                        ELSE
                                CloseFunnelLeft(funnelLeft, funnelRight, pL, path, apex);
                        FI;
                ELSE {The next point is on the right side of the funnel}
                        IF (pR opens the Funnel) THEN
                                funnelRight.Add(pR);
                        ELSE
                                CloseFunnelRight(funnelRight, funnelLeft pR, path, apex);
                        FI;
                FI;
        OD;
        CloseFunnelLeft (funnelLeft, funnelRight, goal, path, apex);
        CloseFunnelRight (funnelRight, funnelLeft, goal, path, apex);
        RETURN path;
END Funnel.
```

11.1.12 CloseFunnelLeft

```
PROCEDURECloseFunnelLeft(funnelLeft:PointList, funnelRight:PointList, p:Point, path:PointList, apex: Point)
        lastEdge ← funnelLeft.LastEdge();
        WHILE (p is on the right of lastEdge) DO
                funnelLeft.PopEnd();
                lastEdge = funnelLeft.LastEdge();
        OD;
        funnelLeft.Add(p);
        firstEdge ← funnelRight.FirstEdge();
        WHILE (p is on the right of FirstEdge) DO
                path.Add(funnelRight.Front());
                funnelRight.PopFront();
                firstEdge ← funnelRight.FirstEdge();
        OD;
        apex ← funnelRight.Front();
        funnelLeft.PopFront();
        funnelLeft.Add(apex);
        path.Add(apex);
END CloseFunnelLeft.
```

11.1.13 CloseFunnelRight

El procedimiento *CloseFunnelRight* es análogo a *CloseFunnelLeft*.

11.2 Detalles del código fuente

11.2.1 Planificador

11.2.1.1 Listado de clases

Clase	Descripción
Point2d	Según se hable de plano o malla, representa respectivamente un punto espacial o un vértice de la malla o triangulación.
Vector2d	Representa una dirección. Es útil para algunas operaciones entre puntos
Edge	Estructura de datos básica de una arista: vértice de origen y puntero hacia la siguiente arista (en sentido antihorario) con el mismo origen.
QuadEdge	Estructura de datos que consiste en un array de 4 Edges. El primer Edge representa una arista normal y el tercero su simétrica. El segundo Edge lleva la arista que une las dos zonas (izquierda y derecha) delimitadas por el primer y el tercer Edge. El cuarto Edge es el simétrico del segundo.
Constraint	Lista de puntos que representa un obstáculo en el plano. Se puede entender

	por una lista de segmentos, estando delimitados cada uno de ellos por el punto i y su siguiente i+1 en la lista.
Triangulation	Estructura de datos que soporta la malla o CDT. Se construye incrementalmente mediante inserciones de puntos y segmentos. Sobre ella se realiza la búsqueda T*A**. Ofrece métodos de búsqueda sobre la malla que implementan el algoritmo T*A** y el algoritmo de *funnel*.
AStarNode	Estructura que almacena los datos necesarios para un estado de búsqueda del algoritmo T*A**.

11.2.2 Procedimientos y atributos por clase

Point2d	**Tipo/Tipo devuelto**	**Descripción**
Atributos		
x, y	double	Coordenadas del punto
Procedimientos		
Point2d()	void	Constructor. Se crea un Point2d vacío (0,0).
Point2d(double a, double b)	void	Constructor. Se crea un Point2d a partir de una abcisa x y una ordenada y.
Point2d(const Point2d& p)	void	Constructor. Se crea un Point2d a partir de otro punto p.
operator+(const Vector2d v)	Point2d	Suma de dos puntos
operator+(const double& a)	Point2d	Suma de un punto y un escalar a
operator-(const double & b)	Point2d	Resta de un punto y un escalar b
operator==(const Point2d&)	int	Igualdad entre puntos
operator!=(const Point2d&)	int	Desigualdad entre puntos
operator<(const Point2d p);	bool	Menor de dos puntos
operator>>(istream& is, Point2d& p)	istream&	Carga de un punto desde el flujo is
operator<<(ostream& os, const Point2d& p)	ostream&	Impresión de un punto en el flujo os
toAnsiString();	AnsiString	Pasa de Point2d a AnsiString
draw(TCanvas* cv, int r, TColor c = clRed)	void	Pinta el punto en el canvas cv con color c y de radio r
drawR(TCanvas* cv, int r, TColor c = clRed);	void	Pinta un rectángulo en el canvas cv con color c y de longitud r
drawText(TCanvas* cv, int r, TColor c, AnsiString s)	void	Pinta el punto en el canvas cv con color c y de radio r y encima el texto s

drawTextR(TCanvas* cv, int r, TColor c, AnsiString s)	void	Pinta un rectángulo en el canvas cv con color c y de longitud r y encima el texto s
isZero()	bool	Devuelve cierto si un punto es igual a (0,0)
moveDistanceBetweenPoints(Point2d p2, int d);	Point2d	Avanza el punto una distancia d sobre la recta que le une con p2
add(int ix, int iy)	void	Incrementa las coordenadas del punto en ix e iy
isInRectangle(int xLeft, int yTop, int xRight, int yBot)	bool	Indica si el punto está en el triángulo delimitado por xLeft, yTop, xRight e yBot
resize(double coefX, double coefY)	void	Multiplica las coordendas del punto por los factores coefX y coefY

Vector2d	**Tipo/Tipo devuelto**	**Descripción**
Atributos		
x, y	double	Direccion del vector
Procedimientos		
Vector2d()		Constructor
Vector2d(Real a, Real b)		Constructor
norm()	double	Modulo del vector
void normalize()	void	Normaliza el vector (lo hace unimodular)
operator+(const Vector2d&)	Vector2d	Suma de vectores
operator-(const Vector2d&)	Vector2d	Resta de vectores
operator*(Real, const Vector2d&)	Vector2d	Producto de un escalar por un vector
dot(const Vector2d&, const Vector2d&)	double	Producto escalar de vectores
operator>>(istream& is, Vector2d& v)	istream&	Vuelca el flujo is en el vector v
operator<<(ostream& os, const Vector2d& v)	ostream&	Vuelca el flujo os en el vector v

Edge	**Tipo/Tipo devuelto**	**Descripción**
Atributos		
num	int	Posición de la arista en el array de 4 aristas de la estructura QuadEdge
data	Point2d*	Origen de la arista

next	Edge*	Referencia a la siguiente arista con el mismo origen (data)
Procedimientos		
Edge()		Constructor. Crea una arista vacía
rot()	Edge*	Devuelve la arista que va de la cara derecha a la izquierda de la actual
invRot()	Edge*	Simétrica de la devuelta por rot()
sym()	Edge*	Simétrica de la actual
oNext()	Edge*	Siguiente arista con mismo origen
oPrev()	Edge*	Arista anterior con mismo origen
dNext()	Edge*	Siguiente arista con el mismo destino
dPrev()	Edge*	Anterior arista con el mismo destino
lNext()	Edge*	Siguiente arista de la cara izquierda
lPrev()	Edge*	Anterior arista de la cara izquierda
rNext()	Edge*	Siguiente arista de la cara derecha
rPrev()	Edge*	Anterior arista de la cara izquierda
org()	Point2d	Origen de la arista
dest()	Point2d	Destino de la arista
rightOf(const Point2d& x)	bool	Indica si el punto x está a la derecha de la arista
leftOf(const Point2d& x)	bool	Indica si el punto x está a la derecha de la arista
onEdge(const Point2d& x)	bool	Indica si el punto x está en la propia arista
collinear(Edge* e2)	bool	Indica si la arista y e2 son colineales
endPoints(Point2d o, Point2d d)	void	Cambia el origen y destino de la arista por los puntos o y d respectivamente
qEdge()	QuadEdge*	Devuelve la estructura QuadEdge de la que forma parte la arista
drawRecursive(unsigned int i, TCanvas* cv)	void	Dibuja todas las aristas conectadas en cualquier número de pasos a la actual en el canvas cv. El parámetro i indica si ya se ha pintado o no esa arista.
draw(TCanvas* cv, TColor c, int r, int g);	void	Dibuja la arista en el canvas cv con color c, radio r y grosor g
closestPointOnEdge(const Point2d& p)	Point2d	Devuelve el punto de la arista más cercano a p

QuadEdge	**Tipo/Tipo devuelto**	**Descripción**
Atributos		

e[Aamodt et al., 1994]	Edge	Dados dos puntos p1 y p2 entre los que existe una arista: e[0] iría de p1 a p2 e[1] de la cara derecha de e[0] a la izquierda e[2] de p2 a p1 (simétrica de e[0]) e[Ross, 1989] de la cara izquierda a la derecha (simétrica de e[1])
timeDraw	unsigned int	En procesos recursivos en los que se recorre toda la malla saltando de cada arista a todas sus adyacentes, el atributo timestamp de la malla se incrementa al inicio y cada vez que se pasa por una arista se incrementa su ts particular (igualándolo al de la malla) para indicar que esa arista ya se ha recorrido y por tanto no se debe analizar.
crep	list<int>	Lista con los identificadores de los obstáculos de los que forma parte la arista. Si está vacía es que no es una arista restringida.
Procedimientos		
remove(int i)	void	Elimina el identificador i de la lista crep
add(int i)	void	Añade el identificador i a la lista crep
setCrep(list<int> cr)	void	Asigna la lista cr a la lista crep
crepList()	list<int>	Devuelve el atributo crep
hasConstraint(int i)	bool	Comprueba si el identificador i está incluido en la lista crep
edge()	Edge*	Devuelve la arista principal de tipo edge
timeStamp(unsigned int time)	int	Devuelve cierto si el atributo ts es distinto del parámetro time y además asigna su valor a ts.
isConstrained()	bool	Indica si es restringido o no
draw(TCanvas* canvas, TColor c, int r, int g)	void	Pinta una arista en el canvas cv de color c, radio r y grosor g.
drawRecursive(int td, TCanvas* cv, int r, int g, bool drawSeg, TColor cCEdge, TColor cUEdge);	void	Pinta recursivamente en el canvas cv todas las aristas conectadas a la actual en cualquier número de pasos.

Triangulation	Tipo/Tipo devuelto	Descripción
Atributos		

startingEdge	Edge*	Arista a partir de la cual se accede a la malla
timestamp	unsigned int	Atributo que se incrementa cada vez que se realiza el proceso de pintado de la malla
links	list<link>	Lista de enlaces (pares punto, identificador de restricción) para encontrar fácilmente un punto que forme parte de una restricción
quadEdges	list<QuadEdge*>	Lista con todas las aristas QuadEdge de la CDT
Procedimientos		
Triangulation()		Constructor
~Triangulation()		Destructor
Triangulation(Point2d, Point2d, Point2d)		Constructor. Crea una malla con forma de triángulo: 3 aristas unidas entre sí
Triangulation(Point2d, Point2d, Point2d, Point2d)		Constructor. Crea un rectángulo triangulado.
locate(Point2d)	Edge*	Localiza la arista en la que se encuentra un punto o cuya cara izquierda lo contiene.
locateTriangle(Point2d a, Point2d b)	Edge*	Encuentra el triángulo que es cortado por el segmento ab
splice(Edge* a, Edge* b)	void	Conecta dos aristas ya existentes
swap(Edge* a)	void	Aplica el operador swap sobre una arista
connect(Edge* a, Edge* b);	Edge*	Conecta dos aristas creando una desde a.Dest hasta b.Org
listOfConstrainedEdgesCrossed(Point2d a, Point2d b, Edge* e = NULL)	list<Edge*>	Lista con las aristas restringidas cruzadas entre (incluidas las solapadas) entre a y b.
listOfCrossedEdges(bool constraints, Point2d a, Point2d b, Edge* e = NULL)	list<Edge*>	Lista con las aristas cruzadas entre (incluidas las solapadas) entre a y b.
intersectionPoint(Point2d aO, Point2d aD, Point2d bO, Point2d bD)	Point2d	Calcula el punto de interseccion entre las rectas aOaD y bObD
insertPointInLeftFace(Edge* e, Point2d p)	void	Inserta el punto p en la cara izquierda y sin triangular de e
remove(Edge* e)	void	Elimina de la CDT la arista e
remove(QuadEdge* qe)	void	Elimina de la CDT la arista qe
makeQuadEdge(Point2d o, Point2d d, int idConstr)	Edge*	Crea una nueva arista con origen o y destino d y perteneciente a la restricción idConstr
makeQuadEdge(Point2d o, Point2d	Edge*	Crea una nueva arista sin restringir con

d)		origen en o y destino en d
triangulateLeftFace(Edge* e);	void	Triangula la cara izquierda de e
edgesOfConstraint(Point2d p, int i)	list<Edge*>	Devuelve todas las aristas del obstaculo i conectadas en cualquier numero de pasos al punto p
difference_set(list<Edge*> le1, list<Edge*> le2)	list<Edge*>	Realiza la diferencia de conjuntos entre las listas le1 y le2
union_set(list<Edge*> le1, list<Edge*> le2)	list<Edge*>	Realiza la union de conjuntos entre le1 y le2
edgesOfConstraint(Point2d p)	list<Edge*>	Obtiene todas las aristas restringidas con origen en p
removePoint(Point2d p)	Edge*	Elimina el punto p y todas las aristas con origen en él.
calculateSuccessor(const AStarNode& parentNode, Edge* eInner, Point2d a, Point2d b)	AStarNode	Calcula el sucesor del nodo parentNode cuya entrada es eInner en el camino del punto a al destino b.
*funnel*Algorithm(AStarNode a, Point2d sP, Point2d gP)	list<Point2d>	Algoritmo de *funnel* entre los puntos sP y gP siguiendo el camino incluido dentro del nodo a
euclideanDistance(list<Point2d> points)	double	Distancia euclidea entre una secuencia de puntos
containedOnLeftTriangle(Edge* e, Point2d p)	bool	Determina si un punto p está incluido en la cara izquierda de e
draw(TCanvas* c, int radio, int grossor, bool drawSeg, TColor cCEdges, TColor cUEdges)	void	Pinta la malla en el canvas c usando el radio, el grosor y los colores indicados. Las segmentaciones se pintan en funcion de drawSeg.
drawRecursive(TCanvas* canvas, int radio, int grossor, bool drawSeg, TColor cCEdges, TColor cUEdges)	void	Pinta recursivamente la malla en el canvas c usando el radio, el grosor y los colores indicados. Las segmentaciones se pintan en funcion de drawSeg.
insertPoint(Point2d p)	void	Inserta el punto p
insertConstraint(Constraint c, int i)	void	Inserta la restriccion c con identificador i
insertSegment(Point2d p, Point2d psuc, int i, bool insertPoints = true)	void	Inserta el segmento entre p y psuc con identificador i. Los puntos extremos se insertan en función de insertPoints.
removeConstraint(int i)	void	Elimina la restriccion i
countQuadEdges()	int	Cuenta el número de aristas de la malla
countQuadEdgesRecursive()	int	Cuenta el número de aristas de la malla
searchTAStar(Point2d a, Point2d b)	list<Point2d>	Encuentra el camino mínimo entre a y b aplicando el algoritmo *A**
resize(double coefX, double coefY)	void	Redimensiona los puntos de la malla
loadScenario(list<Constraint> scenario)	void	Carga un escenario

Constraint	Tipo/Tipo devuelto	Descripción
Atributos		
points	list<Point2d>	Lista de puntos que definen los segmentos que forman el obstáculo.
id	int	Identificador de la restricción
Procedimientos		
Constraint(int i = 0)		Constructor. Se le puede pasar el identificador de la restriccion
~Constraint()		Destructor
getPoints()	list<Point2d>&	Devuelve la lista de puntos
add(Point2d p)	void	Añade un punto a la lista (y por tanto un segmento al obstáculo salvo que sea el primer elemento de la lista)
id()	int	Devuelve el identificador de la restricción
moveTo(int x, int y)	void	Desplaza cada punto de la restricción en x e y unidades
isInRectangle(int xLeft, int yTop, int xRight, int yBot)	bool	Comprueba si todos los puntos de la restricción están incluidos en el rectángulo que se le pasa
resize(double coefX, double coefY)	void	Redimensiona la restricción aplicando los factores coefX y coefY a todos los puntos de la restricción
load(AnsiString path, TRect rect, int count)	list<Constraint>	Carga una lista de restricciones del archivo con ruta path en la malla delimitada por rect
save(AnsiString path,list<Constraint> constraints)	bool	Salva en un fichero con ruta path la lista de restricciones

AStarNode	Tipo/Tipo devuelto	Descripción
Atributos		
eLeftFace	Edge*	Arista cuya cara izquierda es el triangulo representado por el nodo
pathEdges	list<Edge*>	Secuencia de aristas seguida hasta alcanzar el nodo (El nodo raiz tendría la lista vacia)
path	list<Point2d>	Si el nodo es un nodo final contiene el camino óptimo a seguir desde la raiz

g	double	Coste estimado gastado hasta alcanzar este nodo
h	double	Coste estimado que queda por gastar hasta alcanzar el objetivo
goal	bool	Indica si el nodo es objetivo
Procedimientos		
AStarNode()		Constructor
AStarNode(Edge* eL, double gg, double hh, bool gn)		Constructor
AStarNode(list<Edge*> pEs, Edge* eL, double gg, double hh, bool gn)		Constructor
~AStarNode()		Destructor
operator<(const AStarNode&) const	bool	Comparador entre nodos. Un nodo es menor que otro si su f-value es menor
operator>(const AStarNode&) const	bool	Comparador entre nodos. Un nodo es mayor que otro si su f-value es mayor
innerEdge(){	Edge*	Arista de entrada al nodo
gValue()	double	Devuelve el coste gastado
hValue()	double	Devuelve el coste estimado hasta el final
setGValue(double gg)	void	Cambia el coste gastado
setHValue(double hh)	void	Cambia el coste estimado hasta el final
getEdge()	Edge*	Devuelve la arista que representa el nodo (bordea su triangulo)
calculateOperators()	list<Edge*>	Calcula los operadores que se pueden aplicar sobre un nodo
isGoal()	bool	Indica si el nodo es final o no
setGoal(bool b)	void	Asigna el valor b al atributo goal
getPath()	list<Point2d>	Devuelve el atributo path
getPathEdges()	list<Edge*>	Devuelve el atributo pathEdges
setPath(list<Point2d> pth)	void	Asigna el atributo path

Funciones relacionadas con Edge y QuadEdge	**Tipo/Tipo devuelto**	**Descripción**
TriArea(const Point2d& a, const Point2d& b, const Point2d& c)	double	Devuelve el área por el triángulo definido abc
InCircle(const Point2d& a, const Point2d& b,const Point2d& c, const Point2d& d)	int	Determina si el punto d está dentro de la circunferencia definida por a, b

		y c
ccw(const Point2d& a, const Point2d& b, const Point2d& c)	int	Determina si los puntos a, b y c están en sentido horario

11.3 Razonador basado en casos

11.3.1 Listado de clases

Clase	Descripción
CBR	Implementa toda la lógica del razonador, es decir, las fases de recuperación, reutilización, revisión y recuerdo. Contiene el algoritmo ID3 para construir el árbol de decisión sobre el que se realiza la búsqueda del caso que más se ajuste al actual.
Case	Estructura que representa una escena de intrusión. Contiene toda la información necesaria y relevante asociada a una escena.
TreeNode	Estructura auxiliar necesaria que compone el árbol de decisión que construye el CBR.

11.3.2 Atributos y procedimientos por clase

CBR	Tipo/Tipo devuelto	Descripción
Atributos		
cases	list<Case>	Lista con todos los casos cargados de la base de conocimiento.
tree	TreeNode*	Árbol de decisión construido mediante el algoritmo ID3 a partir de la lista de casos.
Procedimientos		
groupCases(TreeNode&, list<Case>)	void	Se agrupan los casos de la lista en función del valor del atributo solución y se guardan en el TreeNode.
selectBestAttribute(TreeNode&, list<Case> , set<int>)	int	Selecciona el mejor atributo (aquel que produce máxima ganancia) para desplegar el árbol con los casos de la lista y actualiza el TreeNode
extractCases(list<Case>, int, int)	list<Case>	Extrae todos los casos de la lista

		para un atributo y un valor dados.
id3(TreeNode&, list<Case>, set<int>)	void	Construye un árbol de decisión.
treeQueryRecursive(TreeNode&, Case)	pair<TSolution, list<Case>>	Metodo auxiliary para consultar el árbol de decisión
getValues(int)	list<int>	Devuelve todos los valores posibles de un atributo dado
listOfRelatedCasesR(list<list<Case>> l, TreeNode& tree)	list<list<Case>>	Método auxiliar que devuelve los casos relacionados entre sí.
CBR()		Constructor
~CBR()		Destructor
getTree()	TreeNode&	Devuelve el árbol de decisión
retrieveCases(AnsiString)	bool	Recupera los casos de un árchivo de texto que contiene la base de conocimiento
saveCases(AnsiString path)	bool	Salva la lista de casos en un archivo
exportAllCasesToMatlab(AnsiString path)	bool	Exporta todos los casos a un matriz con formato de matlab en un archivo
query(Case)	pair<TSolution, list<Case>>	Busca los casos relacionados con el consultado
addCase(Case)	bool	Añade un caso al CBR
listOfRelatedCases()	list< list<Case>>	Devuelve todos los casos con las relaciones entre sí
getCases()	list<Case>	Devuelve la lista de casos
constructDecisionTree()	bool	Construye el árbol de decisión
setCases(list<Case> l)	void	Asigna una nueva lista de casos

Case	**Tipo/Tipo devuelto**	**Descripción**
Atributos		
id	int	Identificador de caso
nG, nI	int	Número de guardias y de intrusos
state	TState	Objetivo de los intrusos
door	TDoor	Estado de la puerta
ratio	TRatio	Proporcion entre intrusos y guardias
alarm	TAlarm	Estado de la alarma
dIT, dIE, dGI, dGT, dGE	TDistance	Distancias entre diferentes elementos
solution	TSolution	Solución encontrada para el caso
sG, sI, sT, sE	Point2d	Posición de distintos elementos
result	bool	Resultado de aplicar la solución

Procedimientos		
Case()		Constructor
Case(int i, int nG, int nI, TState, TDoor d, TAlarm, Point2d pG, Point2d pI, Point2d pT, Point2d pE)		Constructor
Case(int, TRatio, TState, TDoor, TAlarm, TDistance, TDistance, TDistance, TDistance, TDistance)		Constructor
~Case();		Destructor
getRatio()	TRatio	Devuelve la proporción entre intrusos y guardias
simRatioGuardInt(TRatio r2)	double	Similitud local para el ratio
simAlarm(TAlarm a2)	double	Similitud entre estados de la alarma
simDistance(TDistance d1, TDistance d2)	double	Similitud entre distancias
simDoor(TDoor d2)	double	Similitud entre estados de la puerta
simState(TState s2)	double	Similtud entre estados del objetivo de los intrusos
stringToEnum(const char*)	int	Convierte un string a un enumerado
similarityW(const Case& c2)	double	Función de similitude global con pesos ajustados
similarityEqual(const Case& c2)	double	Función de similitude global con pesos equiponderados
setId(int i)	void	Asigna un identificador
setDIT(TDistance d)	void	Asigna la distancia entre intrusos y objetivo
setDIE(TDistance d)	void	Asigna la distancia entre intrusos y salida
setDGI(TDistance d)	void	Asigna la distancia entre guardias e intrusos
setDGT(TDistance d)	void	Asigna la distancia entre guardias y objetivo
setDGE(TDistance d)	void	Asigna la distancia entre guardias y salida
setDoor(TDoor d)	void	Asigna el estado de la puerta
setAlarm(TAlarm a)	void	Asigna el estado de la alarma
setNI(int n)	void	Asigna el número de intrusos
setNG(int n)	void	Asigna el número de guardias
setSG(Point2d p)	void	Asigna la posición de los guardias
setSI(Point2d p)	void	Asigna la posición de los intrusos
setST(Point2d p)	void	Asigna la posición de los objetivos
setSE(Point2d p)	void	Asigna la posición de la salida
setState(TState s)	void	Asigna la posición de la dirección de los intrusos
setSolution(TSolution s)	void	Asigna la solución
getNI()	int	Devuelve el número de intrusos

getNG()	int	Devuelve el número de guardias
getSG()	Point2d	Devuelve la posición de los guardias
getSI()	Point2d	Devuelve la posición de los intrusos
getST()	Point2d	Devuelve la posición del objetivo
getSE()	Point2d	Devuelve la posición de la salida
getDoor()	TDoor	Devuelve el estado de la puerta
getAlarm()	TAlarm	Devuelve el estado de la alarma
getState()	TState	Devuelve la intención de los intrusos
getSolution()	TSolution	Devuelve la solución
getId()	int	Devuelve el identificador
getValue(int);	int	Devuelve el estado de la puerta
similarity(const Case&);	double	Función de similitud entre casos
toAnsiString();	AnsiString	Pasa un caso a una cadena de caracteres
cbrInfo();	AnsiString	Pasa los atributos relevantes del CBR a una cadena de caracteres
toMatlabSequence();	AnsiString	Devuelve una secuencia con formato de matlab
toMatlabMatrix();	AnsiString	Devuelve una matriz con formato Matlab
resize(double cW, double cH);	void	Redimensiona las posiciones incluidas en un caso
operator>>(istream& is, Case& c)	istream&	Sobrecarga de los operadores estandar de entrada y salida
operator<<(ostream& os, Case& c)	ostream&	Sobrecarga de los operadores estandar de entrada y salida

TreeNode	**Tipo/Tipo devuelto**	**Descripción**
Atributos		
idAttribute	int	Identificador del atributo
idValueFromParent	int	Valor del atributo del nodo padre por el que se ha llegado hasta el actual
sons	list<TreeNode*>	Nodos hijos que se despliegan del actual
groups[C_TSOLUTION]	list<Case>	Casos compatibles con el nodo y agrupados por solución
leaf	bool	Indica si el nodo es hoja o no
goalCases	list<Case>	Casos solucion (si el nodo es hoja)
numCases	int	Número de casos asociados al nodo
idSol	TSolution	Valor de la solución más probable en ese nodo
Procedimientos		

TreeNode()		Constructor
TreeNode(int)		Constructor
TreeNode(int, int, bool, TSolution)		Constructor
~TreeNode()		Destructor
setIdAttribute(int id)	void	Asigna el identificador
setLeaf(bool b)	void	Asigna una hoja
setNumCases(int c)	void	Asigna el número de casos
setGoalCases(list<Case> gc)	void	Asigna los casos solución
addSon(TreeNode* s)	void	Asigna los hijos
setIdSol(TSolution s)	void	Asigna el identificador de solución
getIdAttribute()	int	Devuelve el identificador de atributo
getLeaf()	bool	Asigna hoja
attribute()	AnsiString	Devuelve el valor del atributo
valueFromParent()	AnsiString	Devuelve el valor del atributo por el que se ha llegado desde el padre
getSons()	list<TreeNode*>&	Devuelve la lista de hijos
getGroups()	list<Case>*	Devuelve la lista de casos agrupados por solución
getGoalCases()	list<Case>	Devuelve los casos solución
getIdSol()	TSolution	Devuelve el tipo de solución
goalCasesToString()	AnsiString	Pasa los casos solución a string
solutionValueString()	AnsiString	Pasa la solución a string
entropy()	double	Calcula la entropía del nodo
sonByValue(int)	TreeNode*	Devuelve el árbol hijo de un valor

El/la abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Desarrollo de un sistema para toma de decisiones en situaciones de intrusión en instalaciones e infraestructuras", realizado durante el curso académico 2008-2009 bajo la dirección de Gonzalo Pajares Martinsanz, y con la colaboración externa de dirección de Ángela Ribeiro Seijas, en el Departamento de Sistemas Inteligentes, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Jesús Mª Conesa Muñoz