TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

Herramienta docente para algoritmos de búsqueda en Inteligencia Artificial.

Autor

José Carlos Martínez Velázquez

Directores

Francisco Gabriel Raúl Pérez Rodríguez

Rocío Celeste Romero Zaliz

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, Junio de 2017

Herramienta docente para algoritmos de búsqueda en Inteligencia Artificial.

José Carlos Martínez Velázquez

Palabras clave: algoritmos, búsqueda, grafos, inteligencia, artificial

Resumen

Este trabajo fin de grado tiene por objetivo el desarrollo de una herramienta docente que permita a los alumnos de la asignatura Inteligencia Artificial conocer los algoritmos de búsqueda informada de una forma amena e interactiva. Se pretende que el alumno cuente con una aplicación siempre disponible, capaz de explicarle qué está sucediendo en cada paso de los diferentes algoritmos de una forma clara y visual. La aplicación permitirá al alumno resolver las dudas que se le planteen rápidamente así como repasar el tipo de ejercicios que se le propondrán en los exámenes de la asignatura.

Teaching tool for searching algorithms in Artificial Intelligence.

José Carlos Martínez Velázquez

Keywords: searching, algorithms, networks, graphs, artificial, intelligence

Abstract

This end-of-degree project aims to develop a teaching tool that allows Artificial Intelligence's students to know how informed searching algorithms work in a fun and interactive way. The purpose of this graduate work is to provide a tool capable of explaining what is happening in each step of each algorithm in a clear and visual way. This software allows students to solve all their knowledge gaps in search algoritms quickly and to review type-exercises that will appear in their artificial intelligence's exams.

D. **Francisco Gabriel Raúl Pérez Rodríguez**, profesor del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

D[a]. **Rocío Celeste Romero Zaliz**, profesora del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Herramienta docente para algoritmos de búsqueda en Inteligencia Artificial.***, ha sido realizado bajo su supervisión por **José Carlos Martínez Velázquez**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 21 de Junio de 2017 .

Los directores:

F. G. Raúl Pérez Rodríguez **Rocío Celeste Romero Zaliz**

Agradecimientos

A la Universidad de Granada en general y al personal docente de la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación en particular, cuya exigencia e intensidad consiguió hacerme volver a sentir útil y orgulloso de formar parte de ella.

A los que quisieron ayudar y lo hicieron, a los que quisieron pero no pudieron y también a los que debieron y no quisieron, porque me han hecho ver que no necesito su apoyo para conseguir lo que me proponga.

A mis tutores, Raúl y Rocío, por estar siempre disponibles para lo que fuese necesario. A mis compañeros y amigos: Benji, Andrés, Antonio, Fran y muchos más que me dejo, por acompañarme y prestarme su ayuda durante estos años de sonrisas y lágrimas.

Y por supuesto a Lali, por aguantar mis idas y venidas durante más de una década y ojalá, el resto de mi vida. Sin ella jamás hubiera escrito estas líneas.

Mi más sincera gratitud.

Índice general

Introducción

Durante los últimos años, la inteligencia artificial ha supuesto una revolución en prácticamente todos los ámbitos de la vida humana. De hecho, tanto se ha instalado en nuestra naturaleza que utilizamos lo que nos ofrece, en la mayoría de los casos, sin darnos cuenta. La inteligencia artificial supone un fuerte impacto en la dificultad de los problemas, ya que todos los problemas que resuelve no son sencillamente resolubles. En palabras técnicas, diremos que la inteligencia se encarga de resolver los problemas denominados NP-Duros, que engloban a los NP-Completos. Que un problema sea NP-Completo significa que el espacio de búsqueda de soluciones es de orden exponencial, dicho de otro modo, no se puede encontrar una solución en tiempo determinístico polinómico. Que un problema sea NP-Duro, significa que puede ser tan difícil o más que un problema NP-Completo [11]. En definitiva, la inteligencia artificial se encarga de resolver problemas para los que no existen algoritmos que encuentran una solución en un tiempo "razonable". El hecho de que la inteligencia artificial se enfrente a problemas que no tienen una solución sencilla, hace que las soluciones encontradas puedan no ser óptimas.

En concreto, el problema de la búsqueda caminos en grafos ha tenido gran repercusión desde los años 1800, cuando Willian Rowan Hamilton definió el de forma general el Problema del Viajante de Comercio [4]. No fue hasta la década de 1930 cuando el Problema del Viajante de Comercio fue estudiado formalmente por matemáticos en Viena y Harvard. El problema consiste en, encontrar una secuencia de nodos del grafo tal que, partiendo desde un nodo origen, se regrese a este habiendo pasado por todos los demás nodos. Esto es lo que se conoce como encontrar un ciclo o camino Hamiltoniano. Para este problema aún no existe ningún algoritmo determinístico polinómico capaz de resolverlo, por lo que resulta de gran interés para la inteligencia artificial. Es por ello que abrió la veda para que surgieran variantes útiles como la que en este documento se aborda, la búsqueda de caminos mínimos en grafos.

El hecho de que en la actualidad se conozcan diversos algoritmos que son capaces de encontrar el camino mínimo en grafos que cumplen diversas propiedades los convierte en un gran atractivo para resolver los problemas que se pueden modelar como grafos. En ocasiones, se hacen grandes esfuerzos por representar problemas como grafos para poder aplicar uno de estos algoritmos, aún cuando la transformación no es intuitiva. En la actualidad, los algoritmos que vamos a tratar en este documento son de gran interés en múltiples aplicaciones, como por ejemplo:

- Juegos: Desde el planteamiento de ciertos juegos, se destacó la gran adaptabilidad de estos algoritmos de búsqueda. Sudoku (clásico), 8-

reinas, e incluso el cubo de Rubik son ejemplos de juegos que han sido resueltos utilizando algoritmos de búsqueda de camino mínimo, que encuentran cómo ganar en el menor número de pasos posible. También se aplica en videojuegos donde, por ejemplo, un enemigo debe encontrar el camino óptimo desde su posición hasta la del jugador para acorralarlo (PacMan fue el primer videojuego en utilizar el famoso algoritmo A*).

- Robótica y automoción: Estos algoritmos son usados en la actualidad para buscar caminos óptimos en sistemas de navegación autónomos.
- Planificación: Uno de los ejemplos cuya conversión no es intuitiva puede ser este. Imaginemos que tenemos que desarrollar una serie de tareas, empezando por una en concreto y finalizar por otra en concreto, debiendo pasar por algunas de ellas, siendo algunas secundarias (de no importancia vital) ¿Cómo organizarlas? Las tareas serán los nodos y las aristas el tiempo que llevará acabar la de nodo origen hasta pasar a la siguiente. A continuación se puede aplicar el algoritmo de búsqueda, obteniendo en respuesta cómo organizar nuestro trabajo. Se utiliza en proyectos software de gran envergadura.

A lo largo de este documento no sólo vamos a ver cómo diseñar un software docente que sea capaz de apoyar al alumno en el aprendizaje de los diferentes algoritmos de búsqueda de camino mínimo en grafos, sino que resumiremos las bases teóricas para conocer las ideas subyacentes.

Análisis y especificación de Requisitos

En este apartado vamos a tratar de identificar las principales funciones que el software debe llevar a cabo y cómo estructurarlas, así como las necesidades del usuario. De todos estos datos podremos extraer los distintos requisitos funcionales y no funcionales [9] [12] que debería implementar el software.

Para obtener las diferentes tareas que debe desempeñar el software deberíamos preguntarnos justo eso, ¿qué funcionalidades debe tener la aplición? De una primera entrevista con el cliente, extraemos que esta aplicación debería permitir crear o introducir un grafo en la aplicación, cambiarlo o editarlo, importar y exportar un grafo, generar funciones heurísticas para el grafo y poder representarlos gráficamente. Aunque en este momento no es importante cómo lo vamos a hacer, es importante anotar que la aplicación debe implementar todas estas funcionalidades. Al responder a la pregunta formulada, hemos obtenido los diferentes **requisitos funcionales**. En un principio, podremos suponer que constituirán módulos independientes en la aplicación y, en algunos casos, se programarán como tales, pero esto lo decidiremos en la fase de diseño. Por ahora, insisto, sólo nos basta con dejar constancia de que la aplicación debe cumplir estos requisitos.

Una vez que tenemos los requisitos funcionales, debemos hacernos otra pregunta: ¿qué características son deseables para las distintas funcionalidades del software? De la respuesta a esta pregunta, extraeremos los diferentes **requisitos no funcionales**. Este tipo de requisitos, generalmente no se consiguen con una entrevista con el cliente, porque en la mayoría de las ocasiones este los presupone. Es más bien una cuestión de sentido común y criterio personal que debería ser estricto. Todo el mundo estaría de acuerdo en que una aplicación debería ser eficaz, eficiente, segura, fiable, escalable y/o mantenible y portable. Estos, entre otros, deberían ser los adjetivos que califiquen a nuestra aplicación tras su implementación y prueba. Debemos incluir requisitos no funcionales que, cuando sean implementados en la aplicación, garanticen todas estas cualidades.

Pasemos pues a detallar los distintos requisitos funcionales y no funcionales.

2.1. Requisitos funcionales

2.1.1. Creación de grafos

En primer lugar, el sistema debe permitir la inclusión de un grafo para poder resolverlo. Para crear un grafo (por ahora) sólo es necesario que el sistema permita añadir nodos y aristas. Estos son los detalles de este requisito funcional:

	Código	Nombre	Fecha	Necesidad
	RF-1.0	Dibujado de Grafos	06/03/2017	Esencial
Descripción	El sistema debe permitir definir y dibujar un grafo desde cero.			
Entradas	Fuente	Salida	Destino	Restricciones
ID y valor heurístico de cada nodo. ID de nodo origen y destino y coste por cada arista.	Formulario de entrada de datos	La representación interna de un grafo	Aplicación	Tanto los valores heurísticos de los nodos como los costes de las aristas deben ser estrictamente positivas (incluido el 0).
Proceso	El usuario de la aplicación tendrá disponible un formulario a través del cual pueda crear el grafo que pretende resolver. Una vez finalizado el proceso, la aplicación tendrá una representación interna del grafo, que permitirá llevar a cabo el resto de operaciones de manipulación.			

El diagrama de caso de uso asociado a este requisito funcional es el siguiente:

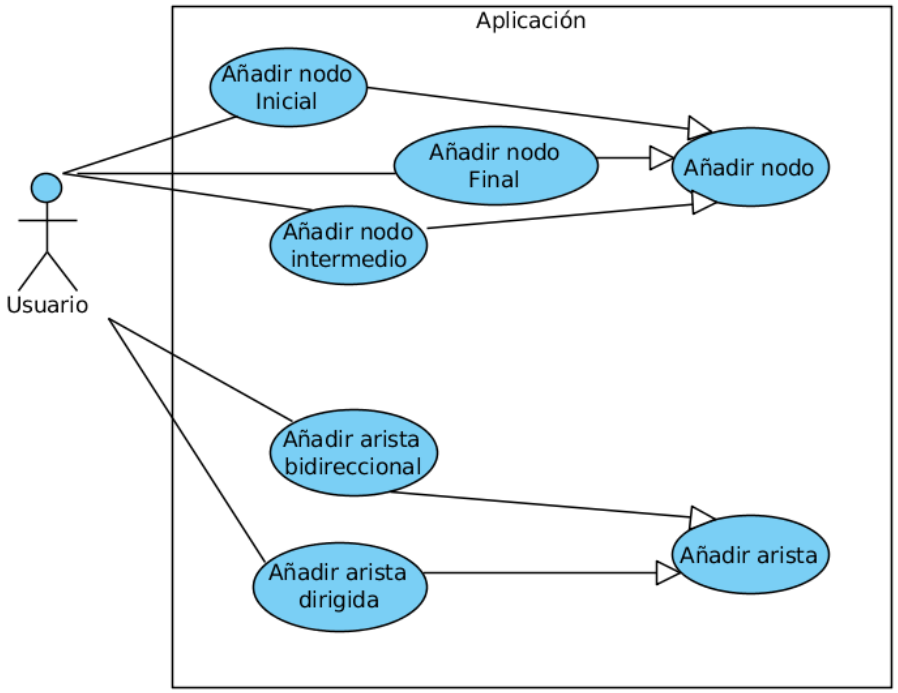


Figura 2.1: Caso de uso: Creación de grafos

El usuario puede añadir un nodo o una arista. Si añade un nodo, podrá decidir si es inicial, final o intermedio, además de su valor heurístico y su ID. Del mismo modo, para añadir una arista, podrá decidir si esta es dirigida o bidireccional, así como los ID de los nodos que une dicha arista y su coste.

2.1.2. Importación/Exportación de grafos

El segundo requisito funcional no es, quizás, uno totalmente obligatorio para que la aplicación cumpla su labor, no obstante, la opción de retomar el trabajo tras cerrar la aplicación es una cualidad deseable en cualquier ámbito. Este requisito funcional pretende hacer que se puedan exportar grafos con los que estábamos trabajando para que, en lugar de dibujar el grafo cada vez que queramos volver a trabajar con él, podamos importarlo con un par de clics. Para ello tenemos que cumplir dos requisitos funcionales: importación (por un lado) y exportación (por otro). Pasamos a detallarlos:

	Código	Nombre	Fecha	Necesidad
	RF-2.0	Exportación de grafo	06/03/2017	Muy recomendable
Descripción	Para evitar tener que dibujar el mismo grafo, si se quieren realizar diversas pruebas con él, podremos exportarlo, pudiéndolo importar en cualquier momento que se requiera.			
Entradas	Fuente	Salida	Destino	Restricciones
La representación interna del grafo	Estado de aplicación	Archivo de texto plano	Usuario	El grafo debe estar en un estado resoluble, esto es, debe existir un sólo nodo inicial y al menos un nodo final.
Proceso	El usuario dispondrá de un botón que le permita exportar un grafo en estado resoluble. Del mismo modo que realizamos cualquier otra descarga, al pulsar el mencionado botón, al usuario se le abrirá un cuadro de diálogo para descargar la representación del grafo localmente a su computadora como un archivo de texto plano.			

	Código	Nombre	Fecha	Necesidad
	RF-2.1	Importación de grafo	06/03/2017	Muy recomendable
Descripción	El sistema debe permitir importar un grafo, de manera que el estado de la aplicación quede modificado en consecuencia.			
Entradas	Fuente	Salida	Destino	Restricciones
Archivo de texto plano	Usuario	La representación interna del grafo	Aplicación	La aplicación debe ser capaz de interpretar el contenido del archivo, lo que significa que el archivo de texto no debió ser modificado cuando se obtuvo la exportación.
Proceso	La funcionalidad de exportación no tendría sentido alguno si no se pudiera importar. El usuario dispondrá de un formulario a través del cual podrá importar un archivo que contenga la representación interna de un grafo previamente exportada.			

El diagrama de caso de uso asociado a estos requisitos funcionales es el siguiente:

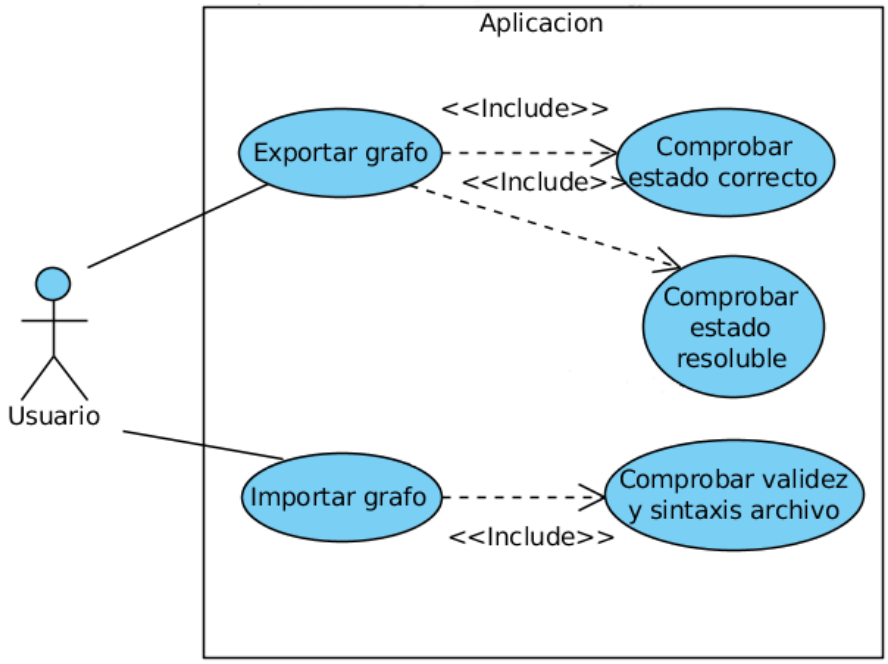


Figura 2.2: Caso de uso: Importación y exportación de grafos

Para exportar un grafo sólo se necesita, tal como indican las restricciones del requisito, que el grafo esté en un estado resoluble y esto debe comprobarse en el proceso de exportación. Del mismo modo, cuando se importa un grafo, se debe comprobar que el archivo sea válido y la sintaxis que contiene el archivo es interpretable por el sistema.

2.1.3. Edición de grafos

El tercer requisito que debe cumplir la aplicación es que el grafo que está tratando la aplicación pueda editarse, a fin de poder probar distintas propiedades del mismo de forma rápida, sin tener que hacer un uso abusivo del módulo de dibujo. Los detalles de este requisito funcional son los siguientes:

	Código	Nombre	Fecha	Necesidad
	RF-3.0	Edición de grafos	06/03/2017	Alta
Descripción	El sistema debe permitir modificar el estado del grafo, esto es: añadir nuevos nodos y aristas, sustituir el nodo inicial, nombrar como final un nodo existente (que aún no fuese final), establecer que un nodo existente deja de ser final (que ya fuese final), modificar el valor heurístico de un nodo, modificar el coste y la orientación de una arista (dirigida/no dirigida).			
Entradas	Fuente	Salida	Destino	Restricciones
La representación interna del grafo	Formulario de entrada de datos	Una nueva representación interna del grafo	Aplicación	
Proceso	Dada una representación interna del grafo, el usuario podrá modificar la representación interna del grafo a través de un formulario. Se podrán insertar nuevos datos y para el caso de borrado y modificación los datos serán recuperados y escritos al formulario para su posterior modificación.			

El diagrama de caso de uso asociado a este requisito funcional es el siguiente:

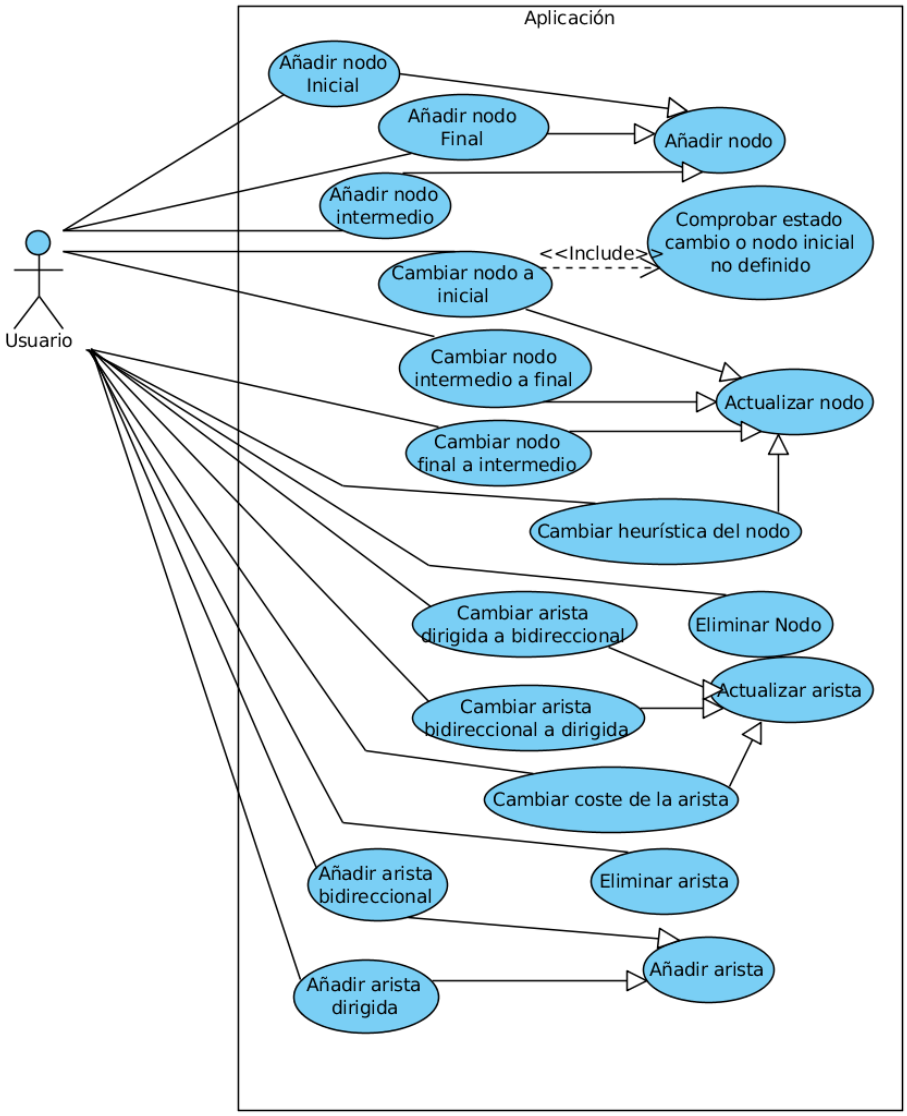


Figura 2.3: Caso de uso: Edición de grafos

Mientras que en el caso anterior sólo era necesario crear nodos y aristas, en este caso se necesita poder actualizar las propiedades de nodos y aristas, ademas de poder borrarlos. Además, el módulo de edición debe permitir añadir nuevos nodos y aristas al grafo, por ello se considera que es una ampliación del módulo de creación de grafo.

2.1.4. Generación de heurísticas

De entre las funcionalidades deseadas para la aplicación se encuentra la generación de heurísticas para el algoritmo A*, de ahí que la importancia de este requisito funcional sea vital. Se debe permitir que el usuario genere heurísticas admisibles o bien que generen un número determinado de cambios en A*. Los detalles de este requisito funcional se expresan en la siguiente tabla:

	Código	Nombre	Fecha	Necesidad
	RF-4.0	Generación de heurísticas	06/03/2017	Esencial
Descripción	El sistema debe permitir generar heurísticas que cumplan ciertas propiedades en función de la necesidad del personal docente.			
Entradas	Fuente	Salida	Destino	Restricciones
La representación interna del grafo (ignorando los valores heurísticos actuales) y las propiedades deseadas.	Estado de la aplicación y formulario de entrada de datos	Conjunto de valores heurísticos que cumplan las propiedades, si es posible encontrarlo. En caso contrario, mensaje de error.	Aplicación	Obviando los estudios teóricos, en función de las propiedades pedidas, puede ser que no sea posible encontrar una heurística que cumpla las mencionadas propiedades. El grafo debe estar en un estado resoluble.
Proceso	Dado un grafo en estado resoluble, el usuario dispondrá de un formulario que le permita generar una heurística que cumpla ciertas propiedades. Si se consigue obtener, la representación del grafo será modificada en consecuencia, en caso contrario, se obtendrá un mensaje de error.			

El diagrama de caso de uso asociado a estos requisitos funcionales es el siguiente:

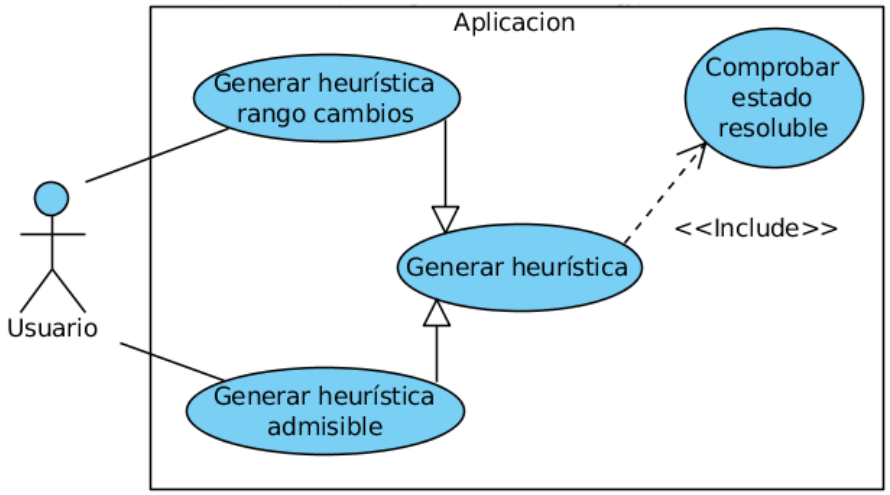


Figura 2.4: Caso de uso: Generación de heurísticas

Como se puede apreciar, el usuario puede generar heurísticas que generen cambios o heurísticas admisibles, pero aunque son procedimientos distintos, vienen a ser casos particulares de un mismo proceso.

2.1.5. Resolución del grafo

Este requisito funcional es el más importante. La aplicación es un resolutor de grafos y, sin este, no tendría sentido la misma. Se debe permitir que el usuario pueda resolver un grafo en estado resoluble. Los detalles del requisito se plasman en la siguiente tabla:

	Código	Nombre	Fecha	Necesidad
	RF-5.0	Resolución del grafo	06/03/2017	Esencial
Descripción	El sistema debe permitir que el usuario elija un algoritmo de resolución y resolver el grafo automáticamente usando dicho algoritmo.			
Entradas	Fuente	Salida	Destino	Restricciones
La representación interna del grafo y algoritmo de resolución	Estado de la aplicación y formulario de entrada de datos	La representación visual los pasos dados por el algoritmo, explicaciones en texto plano de cada paso dado por el algoritmo	Aplicación	El grafo debe estar en un estado resoluble.
Proceso	Dado un estado de la aplicación en que se contenga un grafo en estado resoluble, el usuario elegirá un algoritmo de resolución de entre los disponibles en un formulario y un botón que leerá el algoritmo elegido y lo aplicará a la representación interna del grafo, obteniendo un conjunto de pasos dados por el algoritmo, representados visualmente y un conjunto de explicaciones que justifican los pasos dados.			

El diagrama de caso de uso asociado a estos requisitos funcionales es el siguiente:

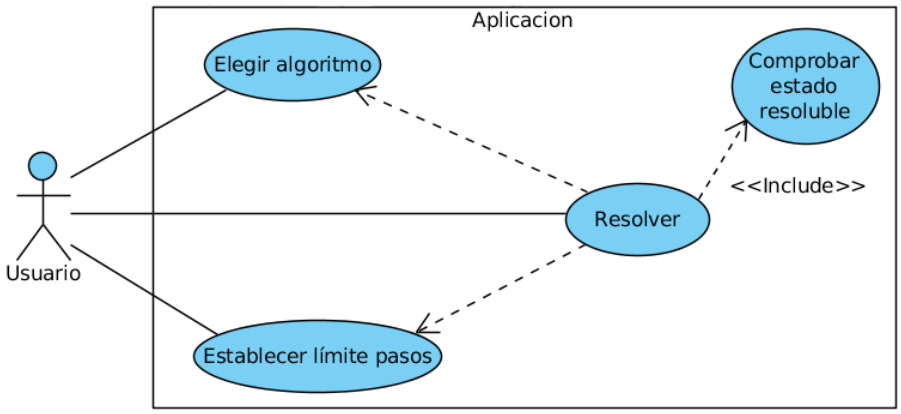


Figura 2.5: Caso de uso: Resolución de grafos

Lo que representa el diagrama es que el usuario puede elegir el algoritmo, establecer el número de pasos límite y resolver el grafo, pero para resolver el grafo, primero tiene que haber hecho las dos cosas anteriores, esto es, la operación de resolver depende de las otras dos. Dentro de la operación resolver se debería comprobar que el grafo se encuentra en un estado resoluble.

2.1.6. Representación gráfica de grafos

Este requisito funcional es de vital importancia para la interpretación visual del grafo. Se debe permitir que, dada una representación interna del mismo, el sistema renderize el grafo para que un humano pueda interpretarlo visualmente. Los detalles del requisito se plasman en la siguiente tabla

	Código	Nombre	Fecha	Necesidad
	RF-6.0	Representación gráfica de grafos	06/03/2017	Esencial
Descripción	El sistema debe permitir, a partir de la representación interna del grafo, dibujar el grafo de manera que sea interpretable visualmente por el usuario.			
Entradas	Fuente	Salida	Destino	Restricciones
La representación interna del grafo	Estado de la aplicación	Por pantalla, representación visual del grafo	Aplicación	
Proceso	Esta funcionalidad debe dispararse durante toda la ejecución de la aplicación. Ya sea por parte de creación de grafos (RF-1.0), importación de un grafo (RF-2.0, RF-2.1), inserción, modificación o borrado de un nodo o arista (RF-3.0, RF-4.0), o cuando se obtenga el conjunto de pasos dados por un algoritmo de resolución elegido (RF-5.0). El usuario obtendrá una representación clara y visualmente interpretable del estado del grafo, pudiendo diferenciar nodos iniciales, intermedios y finales, consultar sus valores heurísticos, así como el tipo y coste de cada arista.			

Este requisito funcional no tiene un diagrama de caso de uso asociado, no obstante la operación de renderizado se debe realizar constantemente por parte de la aplicación.

2.2. Requisitos no funcionales

2.2.1. Requisitos de rendimiento

RNF-1. El sistema debe terminar las tareas en un tiempo finito. Se debe garantizar que tanto la importación, como la generación de características como la resolución por cualquier algoritmo generará una salida en un lapso de tiempo razonable, dependiente del tamaño del grafo.

2.2.2. Requisitos de seguridad

RNF-2. El sistema debe garantizar la seguridad. Dado que el sistema no almacena ningún tipo de información sensible, no es necesario protegerla, pero sí que es necesario proteger la integridad del servidor donde se aloje. Si se subiese algo no debido (un bloque de código malicioso, por ejemplo) en lugar de una representación de un grafo, debería detectarse y no ser procesado.

RNF-3. El sistema debe evitar ataques de denegación de servicio por uso masivo. Para ello, todo el esfuerzo computacional recarerá en el lado del cliente.

2.2.3. Requisitos de fiabilidad

RNF-4. El sistema debe garantizar la detección prematura de bucles infinitos.

RNF-5. El sistema debe garantizar que la importación se realiza correctamente.

RNF-6. El sistema debe estar disponible las 24 horas los 7 días de la semana.

RNF-7. El sistema debe ser robusto frente a fallos críticos.

2.2.4. Requisitos de mantenibilidad

RNF-9. El sistema debe permitir que la adición de nuevos algoritmos sea sencilla, así como cualquier otro cambio. En definitiva, se requiere que el sistema sea escalable.

2.2.5. Requisitos de portabilidad

RNF-9. El sistema debe ser multiplataforma, esto es, que pueda ejecutarse en todos los sistemas operativos disponibles.

Planificación

3.1. Planificación a priori

En lo que a recursos se refiere, este proyecto no es demasiado exigente. Bastaba con un PC y herramientas de programación web. En cuanto a la lista de tareas y la gestión temporal, teniendo en cuenta que no se podía paralelizar el trabajo, por ser una única persona la encargada de hacerlo, se pretendía realizar las siguientes tareas en el siguiente tiempo estipulado:

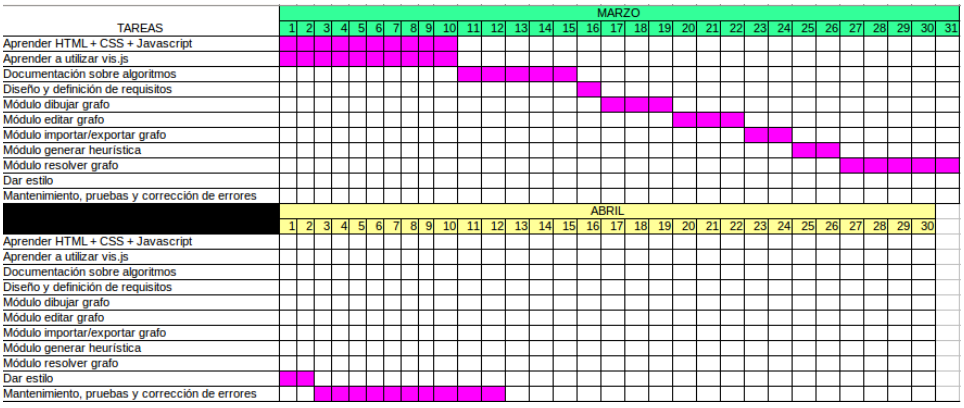


Figura 3.1: Planificacion inicial

Para conseguir estos plazos traté de aplicar una metodología ágil consistente en una variación del modelo de desarrollo en espiral: WIN-WIN [2] [1], tratando de involucrar al cliente (tutores de proyecto) en el desarrollo del mismo, esto es, tratar de conseguir información suficiente sobre qué es lo que desea al principio de un ciclo y obtener un feedback al final del mismo. Los diferentes pasos de la metodología seguida se detallan en la siguiente figura.

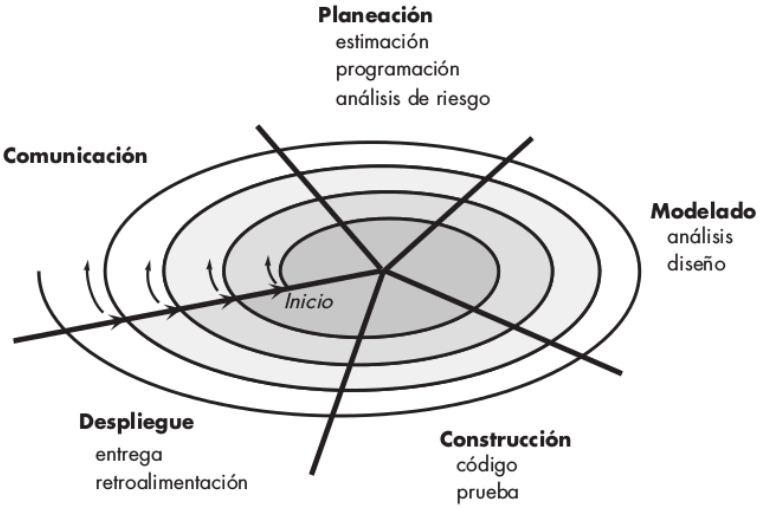


Figura 3.2: Metodología en espiral clásica [9]

3.2. Consecuencias (posteriori)

Lamentablemente, al ser la primera planificación realizada, fui demasiado optimista al asignar los plazos. El primer fallo vino al pensar que podría aprender la biblioteca de dibujado de grafos a la vez que las demás tecnologías de programación. Me resultó dificil comprender su funcionamiento porque jamás había programado web. Al final decidí aprender primero HTML con CSS y JavaScript, que me resultó más sencillo de lo esperado, y cuando me puse de nuevo a aprender la biblioteca de dibujado, no me costó demasiado, sólo lo equivalente a una jornada laboral, con lo que gané un par de jornadas con respecto a la planificación inicial.

Con respecto a la documentación, consideré que no tenía que documentarme tanto, pues había cursado la asignatura Técnicas de los Sistemas Inteligentes el cuatrimestre anterior, donde se estudiaban los contenidos que aquí necesitaban. Nada más lejos de la realidad. Lo que cursamos en esta asignatura es pura teoría, de lo que tuve que repasar demasiados apartados, pero lo que en realidad es necesario, es entender y adaptar los algoritmos a cada estructura de programación. Traté de pensar cómo podría integrar los algoritmos con la parte en la que había que dibujar, una vez que ya sabía cómo funcionaba la biblioteca de dibujado. Aunque esto me llevó más tiempo de lo esperado, pude compensar con lo ganado aprendiendo las herramientas más rápido. Aún así, me retrasé más de lo previsto. Lo mismo ocurrió con la definición de requisitos. A pesar de que ya tenía claro lo que el sistema debía hacer, el formalismo para representarlo me hizo perder bastante tiempo, eso sí, me hizo pensar en las restricciones que debía implementar el sistema para cumplir cada uno de los requisitos que había establecido.

Con todo definido, comencé el desarrollo de la aplicación. En vista a los requisitos establecidos, me propuse dividir el desarrollo en módulos (esto al final no se reflejaría en el diseño), para proceder de forma ordenada con el desarrollo de las diferentes partes de la aplicación. La planificación de esta parte no fue del todo bien. Al comenzar me encontré con muchos problemas que no tenían por qué haber sido necesarios. Traté de configurar un servidor de pruebas que no me sirvió, perdiendo tiempo, al ser novato con el lenguaje, la aplicación no avanzaba lo suficientemente rápido, encontrándome problemas que, en ocasiones tardaba más de una jornada en solucionar, etcétera. Todos estos contratiempos hicieron que, algunos módulos me llevaran mucho más tiempo del previsto, destacando el módulo de dibujado de grafos, el de importación y exportación y del de resolución, donde integrar la implementación teórica de los algoritmos con la biblioteca de dibujado fue bastante más complejo de lo esperado.

Por último, subestimé también la necesidad de hacer pruebas y mantenimiento, así como estar atento a las necesidades del cliente. Cada vez que se prueba el sistema, hay algo que se puede o debe mejorar, o algún cabo suelto que debe corregirse, es por ello que muy posiblemente, después de escribir este documento, aún siga mejorando, probando y corrigiendo el software.

Finalmente, sin contar las tareas de mantenimiento, pruebas y corrección de errores, cometí un error de casi dos semanas desde la planificación inicial con respecto a lo que ocurrió al final. Esta es la planificación que debí haber realizado, que se corresponde con la temporización real que me llevó el proyecto:

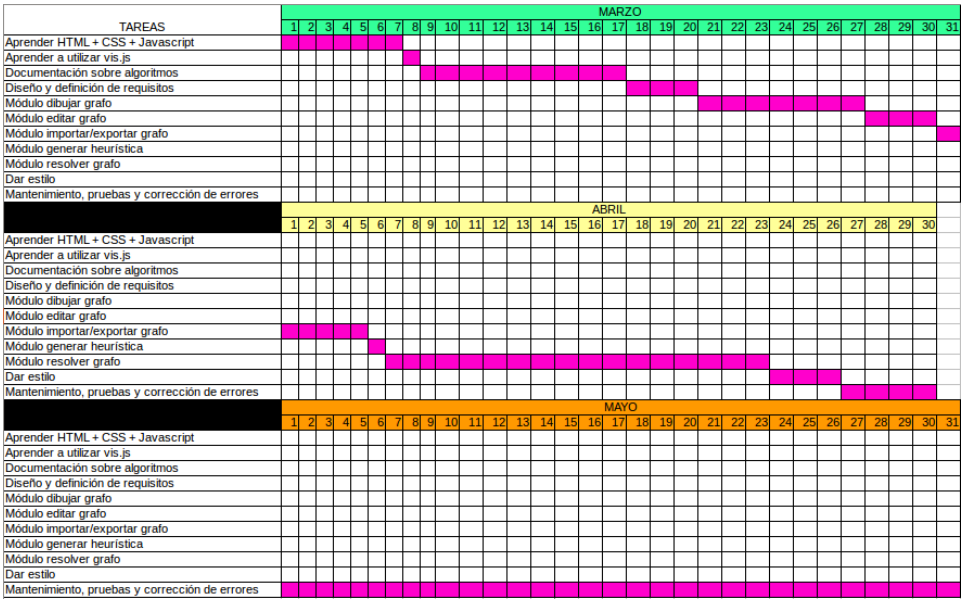


Figura 3.3: Temporización real del proyecto.

Diseño

Se han seguido dos paradigmas de programación para desarrollar este proyecto, que han dado lugar a dos versiones del mismo, cada uno con sus ventajas e inconvenientes, que se detallarán el en apartado de conclusiones. En primer lugar, se ha optado por seguir un paradigma de programación funcional, de forma clásica, es el paradigma que implementan la mayoría de proyectos web desarrollados en JavaScript. En segundo lugar y para poder aplicar ingeniería sobre el proyecto, se ha optado por un paradigma de programación orientada a objetos, que es el que utilizaré para explicar toda la parte de ingeniería de este trabajo.

El diseño de la aplicación pretende enmarcarse en lo que se conoce como patrón Modelo-Vista-Controlador (MVC), esto es, separar la interfaz de usuario de la lógica y gestión interna de los diferentes objetos. La idea básica de la aplicación es construir un grafo y resolverlo mediante diferentes algoritmos, si tenemos en cuenta que queremos utilizar MVC, debemos construir un manejador de interfaz que mande órdenes a los distintos objetos. También parece razonable modelar el grafo como un objeto a parte, de este modo, el manejador de interfaz podría dar órdenes al grafo para modificarlo. Por último, construiremos como objeto una biblioteca de algoritmos, que implementará principalmente los algoritmos de búsqueda para grafos y las funciones auxiliares para éstos. Los algoritmos de búsqueda no son inherentes al grafo, es por ello que se debe separar el concepto de grafo y el concepto de algoritmo de resolución. En lo que a diseño se refiere, ya tenemos las cosas claras: la aplicación será desarrollada como un sistema de tres clases: Interface Handler (o manejador de interfaz), Grafo y Algoritmos. El usuario, por tanto, sólo podrá comunicarse directamente con el manejador de interfaz y este a su vez podrá obtener y cambiar información en el grafo y utilizar la misma para proporcionársela a los distintos algoritmos de búsqueda.

Vamos a tratar de explicar cada clase y definir sus roles.

4.1. Clase Grafo

El objeto grafo es el primero en el que pensamos en esta aplicación y a su vez el que más lejos estará de la gestión directa por parte del usuario, de hecho, el usuario no está gestionando directamente el grafo, aunque se pretenda que tenga la sensación de que lo hace. Necesitamos definir las propiedades de un objeto grafo. Básicamente un grafo tiene nodos y aristas, así que necesitamos definir las estructuras de datos que hagan referencia a esto. Dentro de los nodos, para distintas operaciones, nos interesa saber el

orden en que están los ID de los nodos insertados y sus valores heurísticos, es por ello que, utilizaremos distintos atributos de objeto para este cometido. Del mismo modo procederemos con las aristas. En ciertos instantes tambien es relevante conocer el número de nodos y de aristas que tenemos. Aunque este dato se puede calcular, puede ser eficiente ir calculándolo cada vez que se inserta o borra un nodo o arista, esta operación no es muy pesada, de manera que, al consultar cuántos nodos o aristas tenemos, se devuelva un valor y no haya un proceso de cálculo de longitud cada vez que queramos saber esta cantidad, para ello necesitamos atributos adicionales. Por necesidades de la biblioteca de dibujado, guardaremos en este objeto un atributo que contenga la definición del grafo para la biblioteca, de forma que podamos dibujar automáticamente el objeto sin demasiado esfuerzo. Finalmente, y nuevamente por motivos de dibujado, estableceremos colores para nodos iniciales, finales, estándar y en espera de cambio (seleccionados).

En cuanto a los métodos, añadiremos básicamente los getters y setters para acceder a los distintos atributos mencionados, así como dos métodos que nos permitan cambiar el estado del objeto grafo definiendo sus propiedades. Esto será especialmente útil cuando mostremos la solución, pues cada paso que dibujemos no será más que un estado del mismo grafo, de este modo podremos, con un sólo objeto grafo (grafo solución) representar todos los pasos seguidos en la resolución de un grafo por un algoritmo de búsqueda.

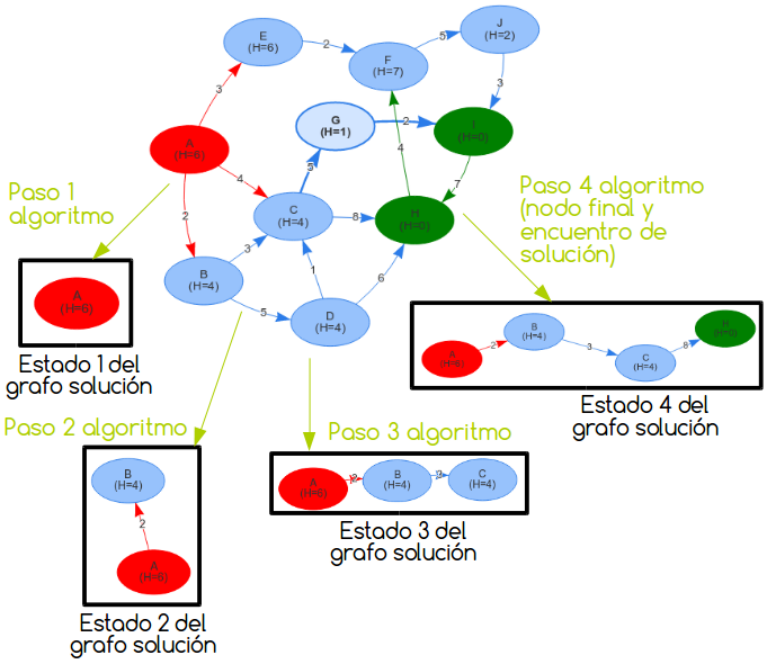


Figura 4.1: Correlación entre pasos de un algoritmo de búsqueda y estados del objeto Grafo solución.

Finalmente añadiremos todos los métodos necesarios para la gestión (insertar, modificar, borrar) de nodos y aristas, que modificarán convenientemente todos los atributos del objeto grafo. También dispondremos de un método que nos permita obtener una descripción detallada del estado del grafo, que el usuario podrá ver en la interfaz. La clase Grafo pues, se modelaría como sigue:

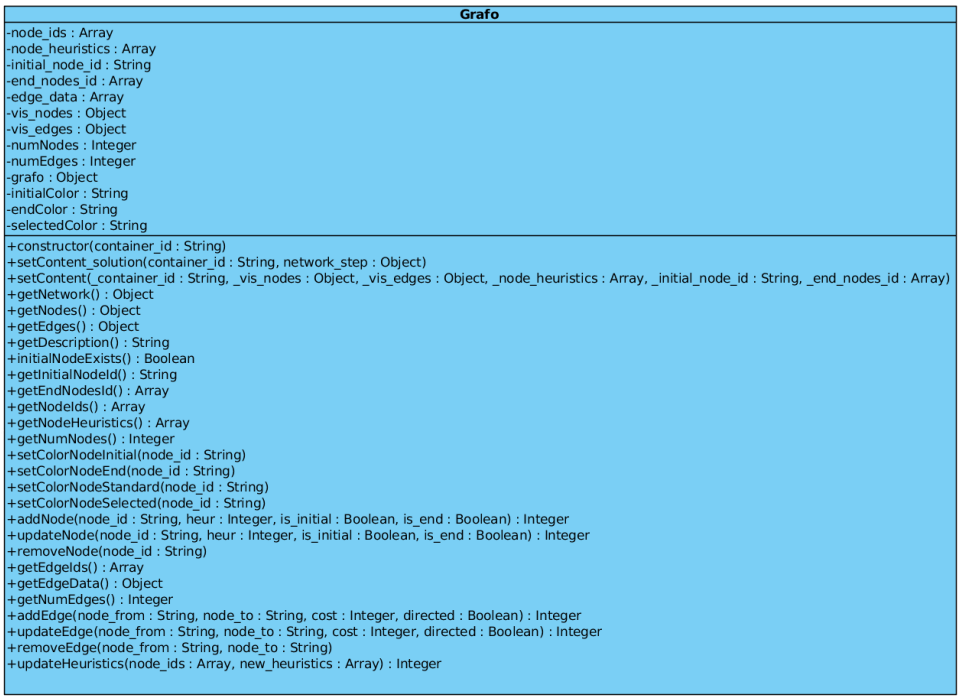


Figura 4.2: Clase Grafo

4.2. Clase Algoritmos

La clase algoritmos será vista como una biblioteca sin más, cuyos atributos serán tres objetos con la misma longitud, enfocados a diferentes estados del grafo solución. Un conjunto de estados del grafo (que pasaremos por parámetro a la clase anterior para definir su estado en cada momento), un conjunto de pasos o cambios en las estructuras de datos utilizadas en los algoritmos (lista de abiertos y cerrados, colas, pilas, etcétera) y un conjunto de explicaciones en lenguaje natural. Cada paso dado por un algoritmo de búsqueda tendrá asociados un "dibujo", una explicación y un cambio en las estructuras de datos, es por ello que si un algoritmo devuelve un array con n "dibujos", entonces devolverá n explicaciones y n cambios en las estructuras de datos. Como es lógico, esta información se utilizará para ser reflejada en la interfaz y ser interpretada por el usuario.

Para llevar a cabo todo lo comentado, la clase algoritmo deberá implementar la definición teórica de cada algoritmo, con la adición de la representación del dibujo, explicaciones y pasos asociados a cada paso dado, además de los métodos auxiliares para cada uno de estos algoritmos. Dado que el módulo de generación de heurísticas requiere realizar cálculos sobre los datos del grafo, como la mayoría de los algoritmos, dotaremos de esta responsabilidad a la clase Algoritmos. Para generar heurísticas para A* que generen un número de cambios concreto en cerrados, requeriremos de una modificación del mismo que cuente este número de cambios, ya que en la versión original este dato es irrelevante. Además, para generar heurísticas admisibles, necesitaremos valernos del algoritmo de Dijkstra completo (no nos vale costo uniforme). Por lo tanto, la clase Algoritmos se implementaría como sigue:

Figura 4.3: Clase Algoritmos

4.3. Clase Manejador de Interfaz

Por último, la clase manejador de interfaz se encargará de mostrar y ocultar los diferentes módulos y menús, presentar información al usuario y configurar parámetros. Esta clase puede considerarse un "mando a distancia" entre el usuario y las clases Grafo y Algoritmos, pues permite que el usuario acceda a estos, pero no directamente. Para describir la clase Interface Handler, habría que describir toda la interfaz y de qué se compone. Por lo que está claro que esta clase y el diseño de la interfaz están estrechamente ligadas. Como se puede apreciar, todos los métodos son accesibles por parte del usuario y ningún método recibe parámetros, esto es porque, al ser un manejador de interfaz, todos los parámetros con los que llama a los métodos de las clases Grafo y Algoritmos son obtenidos de los diferentes componentes gráficos, que son rellenados por el usuario.

Como atributos de la clase, definiremos el grafo principal, al que añadiremos nodos y aristas a través del módulo de dibujado o bien configuraremos su contenido mediante el módulo de importación. Necesitaremos un grafo solución, que irá cambiando su estado conforme los pasos devueltos por cada algoritmo de búsqueda. También necesitaremos una instancia de la biblioteca de algoritmos. El resto de variables nos servirán para controlar el módulo de importación, la creación de eventos para edición rápida sobre el grafo principal y el índice del paso que está visualizando el usuario. La clase Interface Handler entonces, sería implementada como sigue:

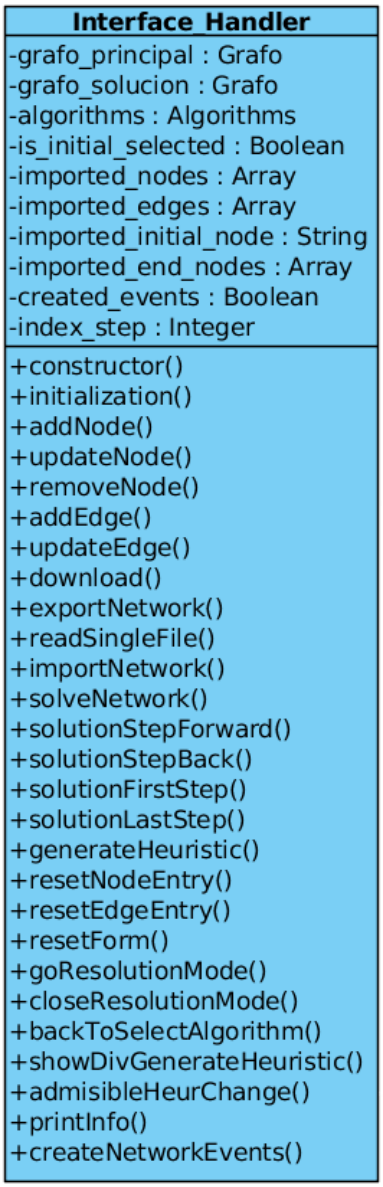


Figura 4.4: Clase Manejador de Interfaz

4.4. Diagrama de clases

Con todo lo explicado, podemos construir el diagrama de clases. La clase Interface Handler implementará exactamente dos instancias del objeto Grafo (grafo principal y grafo solución) y exactamente una instancia de la biblioteca Algoritmos. La clase Grafo hará uso de la biblioteca externa Vis.js [3].

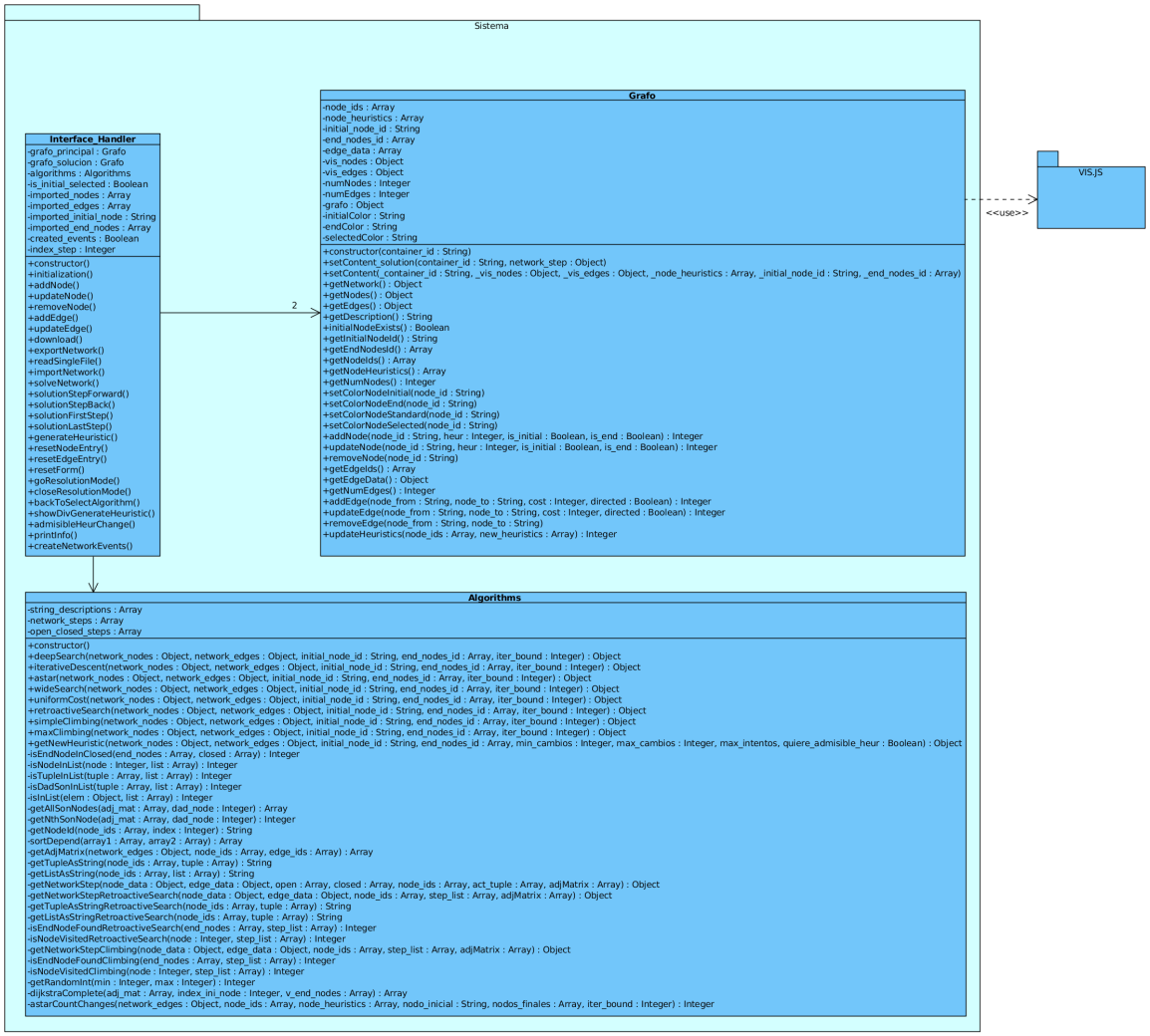


Figura 4.5: Diagrama de clases

4.5. Relocalización de módulos respecto al análisis de requisitos.

En este momento, el diseño ha cambiado totalmente el primer plan de implementación que teníamos con la especificación de requisitos, pero, los módulos siguen estando tras esta estructura de clases. Vamos a tratar de localizarlos y de dejar claro qué clases intervienen en qué módulos:

4.5.1. Módulo de creación de grafos

Este módulo se oculta tras la clase Interface Handler (manejador de interfaz) y la clase Grafo. El Grafo principal se construye inicialmente con

un estado vacío y, a través del manejador de interfaz, se van incluyendo nodos y aristas.

El funcionamiento es el típico del Modelo-Vista-Controlador. El usuario interactúa con el manejador de interfaz, siendo este el encargado de pasar los datos al grafo para que añada nodos y aristas mediante sus métodos de clase, realizando las comprobaciones pertinentes en cada momento. Los métodos usados por este módulo son: `addNode()` y `addEdge()`.

4.5.2. Módulo de importación/exportación de grafos

El módulo de importación y exportación, también queda definido por las clases Interface Handler y Grafo. Un estado del grafo principal en la aplicación puede exportarse si está en estado resoluble, esto es, tiene nodo inicial, final(es) y aristas que los comunican. El proceso de exportación se detalla en el siguiente diagrama de secuencia:

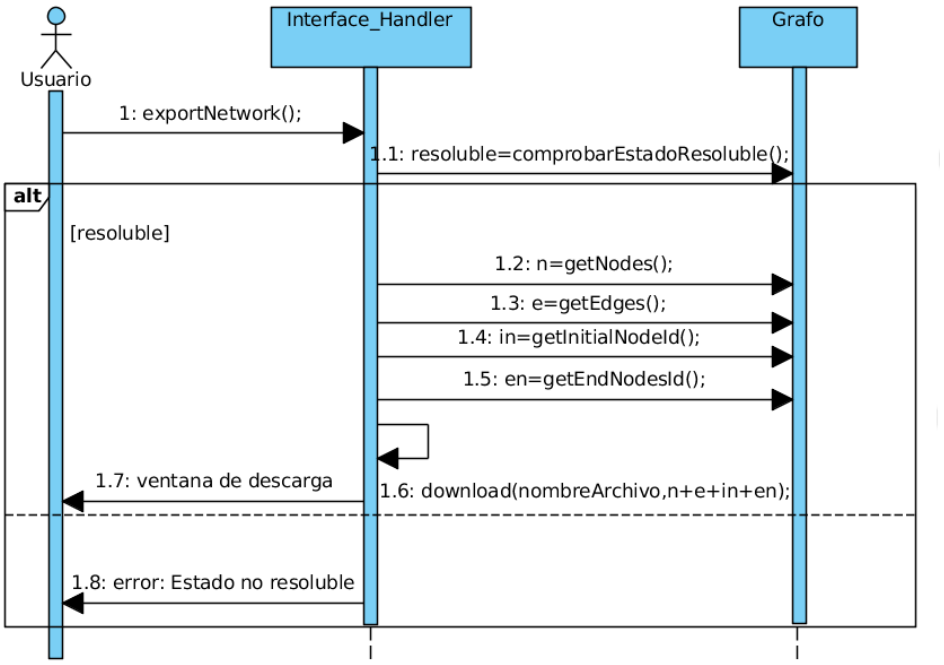


Figura 4.6: Diagrama de secuencia: exportación de grafos

Del mismo modo, para importar un grafo, se requiere que el archivo que se exportó no fuera manipulado, esto es, que el sistema pueda interpretar su sintaxis y que los datos de nodo inicial y final se correspondan con la realidad. Si cuando se lee el archivo introducido por la interfaz, nada de esto ha ocurrido, entonces podemos pasar a establecer el nuevo estado del grafo

al que hemos leído desde el archivo y dibujarlo, siendo esta la salida que recibe el usuario. El proceso incluye todas estas comprobaciones y se detalla en el siguiente diagrama de secuencia:

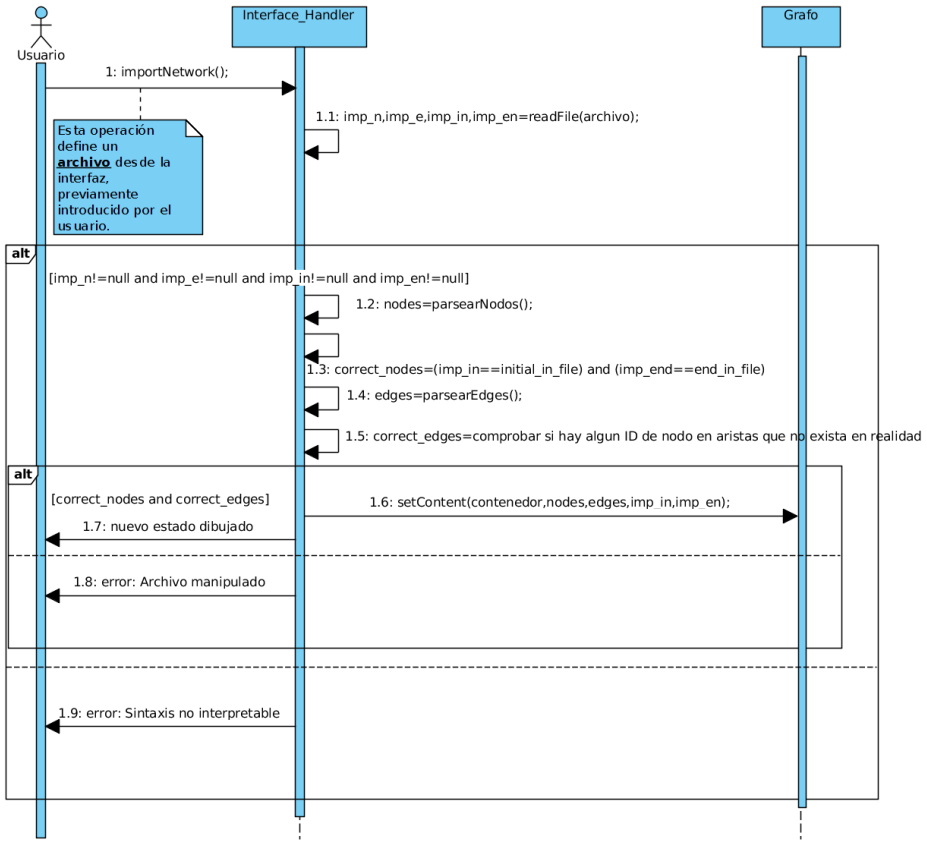


Figura 4.7: Diagrama de secuencia: importación de grafos

4.5.3. Módulo de edición de grafos

El módulo de edición de grafos, tal como veíamos en análisis y especificación de requisitos, sería una ampliación del módulo de creación. Esto no cambia y, de hecho, parte de este módulo son los métodos del módulo de creación. Entonces, al igual que el módulo de creación, este módulo se enmascara tras las clases Interface Handler y Grafo. La forma de proceder es la típica del Modelo-Vista-Controlador y no necesita demasiada explicación. Mediante el formulario el usuario indica a la clase Interface Handler que quiere insertar, actualizar o borrar un nodo o una arista y el controlador se encarga de gestionar los objetos para llevar a cabo la acción solicitada por el usuario. Los métodos usados en este módulo son: `addNode()`, `updateNode()`,

`updateNode()`, `addEdge()`, `updateEdge()` y `removeEdge()`, además de los auxiliares para comprobar existencias y cambiar de color en la representación gráfica.

4.5.4. Módulo de generación de heurísticas

Hay que tener en cuenta que una función heurística es una propiedad y no una característica de un grafo. Lo que quiero decir es que, se debe separar el concepto de función heurística del concepto de grafo. Aunque en esta aplicación los grafos siempre tienen una función heurística, el cálculo de nuevas heurísticas no es una operación que debiera hacer el propio grafo, máxime cuando esta operación conlleva el uso de dos algoritmos de búsqueda: Dijkstra y A*. Si resulta lógico realizar esta separación conceptual, más aún resulta dar la responsabilidad de calcular una nueva heurística a la biblioteca de Algoritmos.

Para generar una nueva heurística, debemos recopilar los datos y propiedades del estado actual del grafo, tales como nodos iniciales, nodos finales, conexiones entre los mismos (matriz de adyacencia) y qué tipo de heurística se desea. Si se desea una heurística que provoque cambios en A*, se debe indicar el rango de cambios que se desea conseguir, en caso de desear una heurística admisible, basta con pasar un booleano obtenido a través de la interfaz, con una casilla de verificación.

Para generar, una heurística admisible, se calcula el algoritmo de Dijkstra desde todos los nodos y se devuelven valores menores o iguales que la heurística ideal para cada nodo. Aunque se podría devolver la heurística de Dijkstra, se prefiere que sea calculada aleatoriamente de este modo, para evitar el determinismo.

4.5.5. Módulo de resolución del grafo

De forma parecida al módulo de generación de heurísticas, el algoritmo de búsqueda es absolutamente independiente del grafo que vayamos a resolver, por lo que también es lógico realizar esta separación. Este módulo, por tanto, se enmascara entre las clases Interface Handler y la biblioteca de Algoritmos. A través de la interfaz de usuario seleccionaremos un algoritmo de resolución, la interfaz recopilará los datos necesarios del grafo y se los pasará al algoritmo elegido para la resolución. De nuevo, la funcionalidad típica del Modelo-Vista-Controlador.

4.5.6. Módulo de representación gráfica

Este módulo está enmascarado en la clase Grafo. La representación gráfica del mismo sólo depende de su estado, por lo que, para dibujar un grafo sólo necesitamos conocer su estado. Este módulo no tiene operaciones con-

cretas ya que cada módulo lo usa de forma constante durante todo el ciclo de vida de la aplicación.

Implementación

5.1. Implementación de la interfaz web

Por el requisito no funcional RNF-6, el sistema debe estar disponible 24/7, por ello, lo más razonable es implementarlo como aplicación web. Implementaremos una aplicación web, dividida en módulos de dibujo, import/export, edición de grafo, generación de heurísticas, selección de algoritmo y visualización de la solución, que juntos aúnan todos los requisitos funcionales descritos y cuya implementación paso a describir por separado en las siguientes secciones.

Todos los elementos de la aplicación serán implementados mediante una combinación de HTML, CSS, JavaScript, JQuery [5] y la biblioteca Vis.js [3] (implementada en JavaScript). La representación interna de grafos se realiza mediante lenguaje DOT [8].

5.1.1. Implementación del módulo de dibujo

El siguiente módulo trata de satisfacer el requisito RF-1.0, para crear un grafo en la aplicación. Definiremos un formulario doble, compuesto por dos subformularios, que nos solicitarán información sobre los nodos y las aristas.

La información que se solicita sobre los nodos es:

- Identificador del nodo: Nos permitirá identificar de forma única al nodo.
- Valor heurístico del nodo: Indica una estimación del coste de llegar desde ese nodo a un nodo final. Esta información es requerida sólo por algunos algoritmos. Para unificar el comportamiento, se pedirá de forma general, pasando a ignorar este valor cuando el algoritmo solicitado no lo requiera.
- ¿Se va a crear un nodo inicial?: Se implementará como una caja de verificación (checkbox) que indica si el nodo que se va a crear es nodo inicial o no. Dado que los algoritmo de búsqueda pueden partir de un solo nodo inicial, una vez definido el nodo inicial esta casilla de deshabilitará.
- ¿Se va a crear un nodo final?: Se implementará como una caja de verificación (checkbox) que indica si el nodo que se va a crear es final o no.

La información que se solicita sobre aristas es la siguiente:

- Nodo origen: Identificador del nodo desde el que parte la arista.
- Nodo destino: Identificador del nodo al que llega la arista.
- Coste: Es un valor numérico que cuantifica el esfuerzo hay que realizar para llegar desde el nodo origen al nodo destino a través de esta arista.
- ¿Se va a crear una arista dirigida?: Se implementará como una caja de verificación (checkbox) que indica si la arista que se va a crear es dirigida o no. Esto significa si la arista que vamos a crear define una forma sólo de ir del nodo origen al destino (dirigida) o la arista define el mismo camino del origen al destino y del destino al origen (no dirigida).

La implementación en HTML es sencilla. Incluiremos los botones Actualizar y Eliminar para incrustar el módulo de edición en el mismo formulacio. Tras dotarlo de estilo con CSS, su aspecto final es el siguiente:

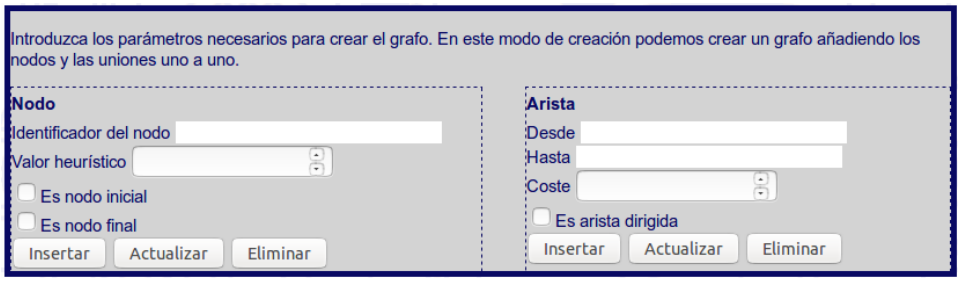


Figura 5.1: Módulo de dibujo

Además de la restricción de que el nodo inicial debe ser uno debemos forzar que tanto el coste de las aristas como el valor heurístico de cada nodo sea positivo, por tanto, forzaremos a que el mínimo valor que se pueda introducir en la parte correspondiente del formulario sea 0, en caso contrario obtendremos un mensaje de error.

5.1.2. Implementación del módulo Importación/Exportación

Para satisfacer los requisitos funcionales RF-2.0 y RF-2.1, se debe incluir un módulo de importación exportación de grafos. Esto es recomendable de cara a retomar los experimentos que se estaban haciendo con un grafo sin necesidad de volver a crearlo. La interfaz del módulo será sencilla: en primer lugar, el módulo de exportar constará de un botón que recopilará todos los datos de la representación del grafo y los escribirá a un archivo que debemos guardar. Se creará una cadena que constará de los datos de nodos y aristas en formato DOT, seguido del ID de nodo inicial y los ID de los nodos finales. La cadena tendrá, por tanto, cuatro bloques delimitados por un separador

tipo cadena. A continuación se empaquetará la mencionada cadena en un archivo de texto plano y se abrirá un cuadro de diálogo para descargar el recurso.

El módulo de importación constará de un formulario, sólo con un botón de tipo "Examinar" y otro para importar grafo. La gestión interna es un poco más compleja. En primer lugar, una vez seleccionado el archivo, una función detectará que había cuatro bloques de datos (delimitamos por los separadores) y en ese caso se permite importar grafo. En caso de no haber cuatro bloques, el sistema detecta que no se trata de un archivo válido pues de algún modo ha sido manipulado. Una vez realizada la comprobación de validez, el sistema, para el primer y segundo bloque debe procesar la sintaxis DOT para extraer los datos de los nodos (ID del nodo, valor heurístico, es nodo inicial o final) y los datos de las aristas (nodo origen, nodo destino, arista dirigida, coste). Para los demás bloques, se comprueba que los ID de nodo inicial y final existan. Con todos estos datos finalmente, se crea la representación interna del grafo.

5.1.3. Implementación del módulo de edición

Para satisfacer el requisito funcional RF-3.0 se debe incluir un módulo que permita editar el grafo. Se puede editar el grafo de dos modos: a través del formulario de creación o a través de la visualización gráfica mediante los eventos de la biblioteca Vis.js.

A través de eventos de Vis.js, vamos a poder cambiar los nodos iniciales y finales de una forma rápida, sólo clicando en ellos. Para establecer un nodo no final como final basta con hacer doble clic sobre él. Del mismo modo se procede si un nodo es final y deseamos que no lo sea. Para cambiar el nodo inicial, primero seleccionaremos el nodo inicial actual, quedando en una especie de modo de espera, a continuación se selecciona un nuevo nodo, que quedará establecido como inicial. Si tras el modo de espera se hace clic sobre el mismo nodo inicial, se vuelve a establacer, si se pulsa sobre un nodo final, obtendremos un error porque se considera que un nodo no puede ser inicial y final a la vez. Obtenemos el mismo error si hacemos doble clic sobre un nodo inicial.

Si se quieren modificar más aspectos del grafo, como los valores heurísticos, costes de las aristas o dirección de las mismas, se debe utilizar el formulario de entrada. El modo de proceder es simple: en primer lugar se selecciona sobre el grafo dibujado el nodo o arista a modificar. A través de una función en JavaScript se obtendrán los datos del nodo seleccionado y se escribirán automáticamente en el formulario de entrada. Dado que la mencionada función ha rescatado el ID del nodo o arista, conocemos qué elemento se debe modificar, así que basta con volver a escribir en el formulario los nuevos valores y hacer clic en actualizar, la función correspondiente se encargará de modificar los valores indicados para el elemento. Lógicamente, este segundo

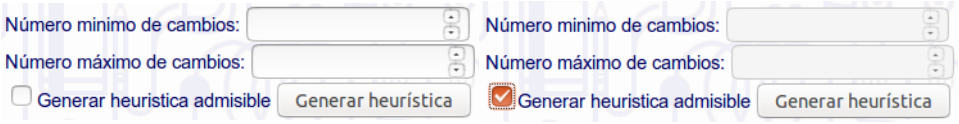
método de edición es más versátil que el primero.

5.1.4. Implementación del módulo de generación de heurísticas

Para satisfacer el requisito funcional RF-4.0 se debe implementar un módulo que permita generar una heurística que cumpla ciertas propiedades. Este módulo tendrá efecto sólo para el algoritmo A*. Servirá para generar ejercicios de exámenes sabiendo previamente cuántos cambios se producirán en la lista de cerrados o los efectos que producirá una heurística admisible (leer la sección dedicada al algoritmo A*).

El personal docente dispondrá de un formulario para generar dos tipos de heurísticas para un grafo dado. En primer lugar, podrá indicar que se desea que se produzca un número determinado de cambios en la lista de cerrados. Para ello tendrá dos campos, uno para poner el número mínimo de cambios y otro el máximo. Lógicamente, si desea un número concreto de cambios, bastará con poner el mismo valor en los dos campos. Se cuenta con comprobaciones de control tales como que el máximo número de cambios no sea menor que el mínimo, que no haya un número negativo de cambios, etcétera. Por otro lado, se podrá generar una heurística admisible y comprobar su efecto, independientemente de los cambios que produzca en cerrados.

El aspecto de este módulo es el siguiente:



(a) Número de cambios en cerrados (b) Heurística admisible

Figura 5.2: Generación de heurísticas

El mecanismo para obtener la heurística es, en primer lugar aplicar el algoritmo de Dijkstra para saber, desde cada nodo, cuál es el coste mínimo a la solución (leer la sección correspondiente a este algoritmo). En caso de que se desee una heurística admisible, se generan aleatoriamente valores aleatorios que no superen el coste de camino mínimo a cada nodo, respetando la definición de heurística admisible. En caso de desear una heurística que provoque cambios, se generan aleatoriamente valores heurísticos, probándolos en el algoritmo A* y contando el número de cambios en cerrados. Cuando se encuentra una heurística que cumple las propiedades, ésta es devuelta. Lo primero es comprobar si la heurística actual cumple con los requisitos solicitados, puesto que si es así, no hay que hacer nada.

5.1.5. Implementación del módulo selección de algoritmo

Para satisfacer el requisito funcional RF-5.0, se debe implementar un módulo que permita seleccionar un algoritmo y posteriormente explique los pasos que ha dado dicho algoritmo sobre el grafo activo. Se implementará un formulario con una lista desplegable, un campo numérico y un botón. La función de la lista desplegable es mostrar los algoritmos disponibles, la del campo numérico es establecer un límite de iteraciones y la del botón es llamar a una función JavaScript que se encargará de recoger los datos del formulario y pasarlos como parámetro al algoritmo seleccionado. Dado que el sistema es de ámbito docente y se van a procesar grafos que se puedan resolver en papel, se establecerá un límite máximo de 1000 iteraciones.

El aspecto de este módulo tiene el siguiente aspecto:

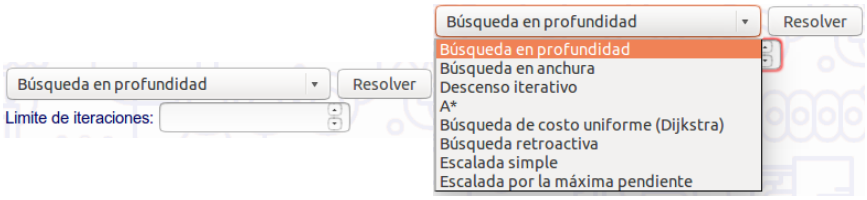


Figura 5.3: Selección del algoritmo

5.1.6. Implementación del módulo de visualización de solución

Para satisfacer el último requisito funcional, RF-6.0 se debe implementar un módulo que permita visualizar el estado actual del grafo. Además servirá para dibujar la solución. Esto se conseguirá con ayuda de la biblioteca Vis.js. No hay más que implementar un elemento de html e indicar a Vis.js que dibuje en *contenedor* un determinado grafo. Esto servirá como cuadro de dibujo donde se irán dibujando las soluciones. Además, dispondremos de un cuadro de texto donde se irán viendo los cambios que sufren los mecanismos de control del algoritmo (lista de abiertos y cerrados, nodos con numero de hijos visitados, etcétera) otro para visualizar las explicaciones de cada paso y cinco botones: paso adelante, paso atrás, ir al principio, ir al final y cambio de algoritmo. Este último nos permitirá volver a seleccionar un nuevo algoritmo. Se alterna entre el módulo de elección de algoritmo y de visualización de la solución con transiciones JQuery.

El aspecto del módulo una vez finalizado es el siguiente:

Figura 5.4: Módulo de visualización de la solución

5.2. Implementación de los algoritmos

5.2.1. Búsqueda en profundidad

El algoritmo de búsqueda en profundidad consiste, básicamente, en expandir el nodo más profundo de entre los que se encuentran abiertos. Esto permite, en cierto modo, aplicar un retroceso, ya que, si la búsqueda queda atrapada en el nodo más profundo actual y éste no tiene nodos hijo, este se descarta y se elige, de entre los abiertos, el siguiente más profundo. Para conseguir esta propiedad de retroceso, se puede implementar el algoritmo con una cola LIFO (Last In, First Out) o, dicho de otro modo, una pila.

Este algoritmo se caracteriza por no necesitar gran capacidad de memoria, almacenando sólo un camino desde la raiz al nodo hoja, además de los nodos hermanos no expandidos (lista de abiertos).

En este algoritmo se pueden producir ciclos infinitos en el caso de grafos no dirigidos (cuando dos nodos están unidos por una arista de ida y vuelta de forma que son padre e hijo el uno del otro). Cuando se expande el nodo hijo, este pasa a ser padre y volverá a expandir como hijo al que era su padre. Detectar este ciclo es sencillo, basta comprobar si ya se visitó el nodo padre. No obstante, continuar por el siguiente hijo correspondería al algoritmo de búsqueda retroactiva, lo que no respetaría la implementación original del algoritmo. Existe una versión del algoritmo denominada BPL (Búsqueda en profundidad limitada) que limita el número de operaciones que se hacen si no se encuentra un camino a un nodo final, esto es, abortar el algoritmo si no hay un nodo final a profundidad p. Esto puede solucionar que el algoritmo no termine en caso de ciclo infinito, pero se puede evitar con detección prematura de ciclos infinitos. La detección prematura de ciclos infinitos es equivalente a limitar la profundidad en el algoritmo de búsqueda, pero sin repetir pasos.

El siguiente ejemplo trata de ilustrar cómo se procede en búsqueda en

profundidad:

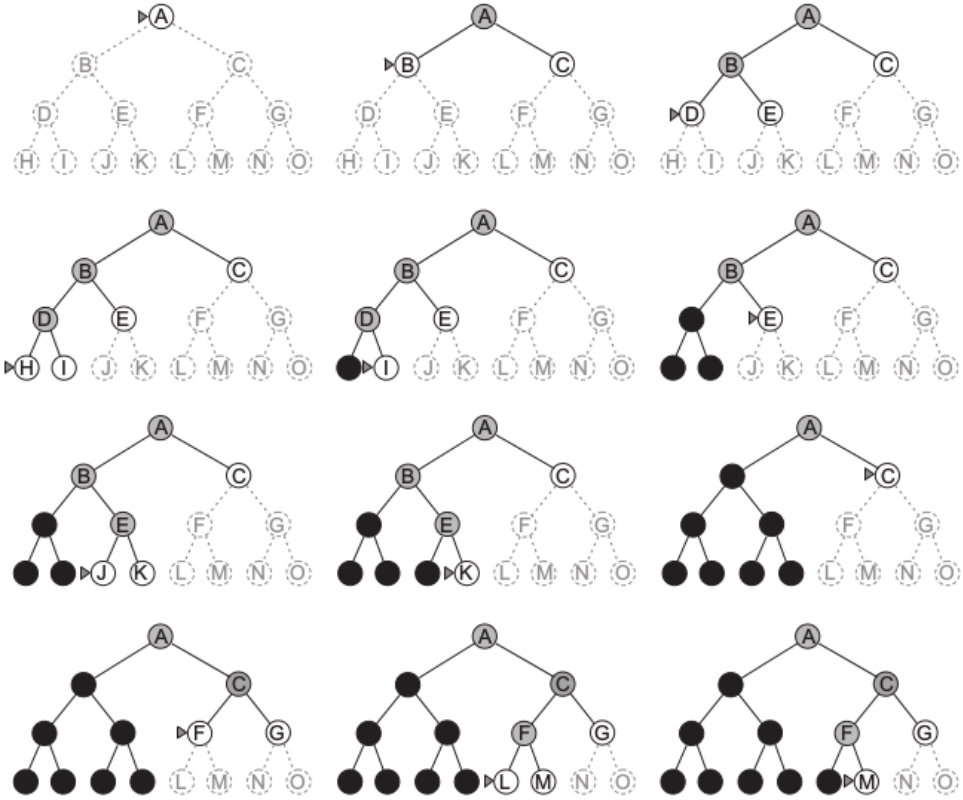


Figura 5.5: Ejemplo de búsqueda en profundidad. [10]

La definición del algoritmo en pseudocódigo es la siguiente:

```
1  Busqueda_profundidad(grafo):
2    introducir en abiertos el nodo inicial
3    nodo_actual=nodo_inicial
4    mientras(un nodo final no este en cerrados):
5      abiertos.add_al_principio(nodo_actual.hijos)
6      abiertos.eliminar(nodo_actual)
7      cerrados.add(nodo_actual)
8      si(abiertos.vacia)
9        devolver error
10     nodo_actual=abiertos.primero
11     si(nodo_actual en cerrados):
12       ciclo infinito detectado
13   camino=reconstruir camino desde cerrados
14   devolver camino
```

Las características de la búsqueda en profundidad son las siguientes:

- Completitud: es completo si usamos grafos finitos y no se limita la profundidad máxima, pues todos los nodos serán expandidos si no se encuentra una solución antes.

- Optimalidad: No es óptimo en ningún caso, pues devuelve la primera solución que encuentra, que puede ser de mayor coste que otra posible solución.
- Complejidad temporal: En el peor caso, el orden de complejidad es $O(h^p)$, siendo h el número de hijos promedio (factor de ramificación) y p la máxima profundidad del grafo.
- Complejidad espacial (coste en memoria): En el peor caso $O(h^m)$, siendo m la máxima profundidad del grafo.

5.2.2. Búsqueda en anchura

La búsqueda en anchura consiste, básicamente, en expandir todos los sucesores de un nodo antes de pasar expandir el siguiente nodo. De forma general, el algoritmo de búsqueda en anchura expande todos los nodos a una profundidad p antes de expandir cualquier nodo del próximo nivel. En cuanto a implementación, la única diferencia con búsqueda en profundidad es que búsqueda en anchura gestiona una cola FIFO (First In First Out), de forma que se asegura que lo primero que se introdujo (los hijos del nodo actual) son los primeros nodos que se expanden.

En este algoritmo, la memoria sí es un problema, ya que se deben almacenar todos los nodos que se generan. Supongamos que cada nodo tiene h sucesores, entonces en el primer nivel tenemos h sucesores, en el segundo h^2, en el tercero h^3, etcétera. Si suponemos que es p la máxima profundidad, supone una complejidad espacial que coincide con su complejidad en tiempo mas uno (por el nodo raíz). Entonces, el número de nodos a guardar es:

$$h + h^2 + h^3 + ... + h^p + \left(h^{p+1} - h\right) = h^{p+1}$$

De la misma forma que la búsqueda en profundidad se pueden generar ciclos infinitos en caso de grafos no dirigidos, pero se detectan procediendo igual, esto es, comprobando si el nodo hijo ya ha sido cerrado.

En el siguiente ejemplo se ilustra cómo funciona búsqueda en anchura:

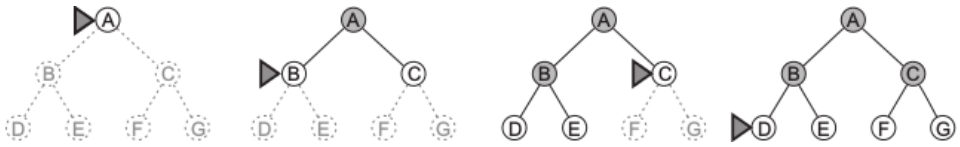


Figura 5.6: Ejemplo de búsqueda en anchura. [10]

La definición del algoritmo en pseudocódigo es la siguiente:

```
1  Busqueda_anchura(grafo):
2    introducir en abiertos el nodo inicial
3    nodo_actual=nodo_inicial
```

```
4     mientras(un nodo final no este en cerrados):
5       abiertos.add_al_final(nodo_actual.hijos)
6       abiertos.eliminar(nodo_actual)
7       cerrados.add(nodo_actual)
8       si(abiertos.vacia)
9         devolver error
10      nodo_actual=abiertos.primero
11      si(nodo_actual en cerrados):
12        ciclo infinito detectado
13    camino=reconstruir camino desde cerrados
14    devolver camino
```

Las características de la búsqueda en anchura son las siguientes:

- Completitud: es completo si usamos grafos finitos, pues todos los nodos serán expandidos si no se encuentra una solución antes.
- Optimalidad: No es óptimo en ningún caso, pues devuelve la primera solución que encuentra, que puede ser de mayor coste que otra posible solución.
- Complejidad temporal: En el peor caso, el orden de complejidad es $O(h^p)$, siendo h el número de hijos promedio (factor de ramificación) y p la máxima profundidad del grafo.
- Complejidad espacial (coste en memoria): En el peor caso $O(h^{p+1})$, tal como se ha razonado anteriormente.

5.2.3. Búsqueda retroactiva

El algoritmo de búsqueda retroactiva es similar al de búsqueda en profundidad, pero hay varias diferencias. En primer lugar, búsqueda en profundidad genera todos sus hijos, añadiéndolos al principio de la lista de abiertos. En búsqueda en profundidad, se producen ciclos si el grafo es no dirigido. Aunque en la búsqueda retroactiva se pueden producir ciclos, *en parte* son evitados porque recuerda, por cada nodo, cuántos hijos expandió ya, de manera que posibilita la técnica denominada *backtracking*, esto es, dar pasos atrás hasta que se puedan expandir más hijos. Dicho de otro modo, backtracking es volver a un estado del pasado e intentar encontrar el camino en otra dirección. En resumidas cuentas, la búsqueda retroactiva es una búsqueda en profundidad mejorada, porque puede recordar en qué hijo se quedó por cada nodo.

Este algoritmo no tiene grandes requisitos de memoria, sobre todo teniendo en cuenta que su forma de proceder es similar a la búsqueda en profundidad. Por cada nodo se va a guardar el ID del nodo qué hijo es el que tendría que expandir a continuación. Si se alcanza un nodo hoja (o un nodo en el que ya se han explorado todos sus hijos), se incrementa en uno el contador de hijo del padre y se expande el hijo siguiente.

Vamos a tratar de ilustrar su funcionamiento con el siguiente ejemplo. Supongamos el siguiente grafo, para el que H es nodo inicial y E es nodo final.

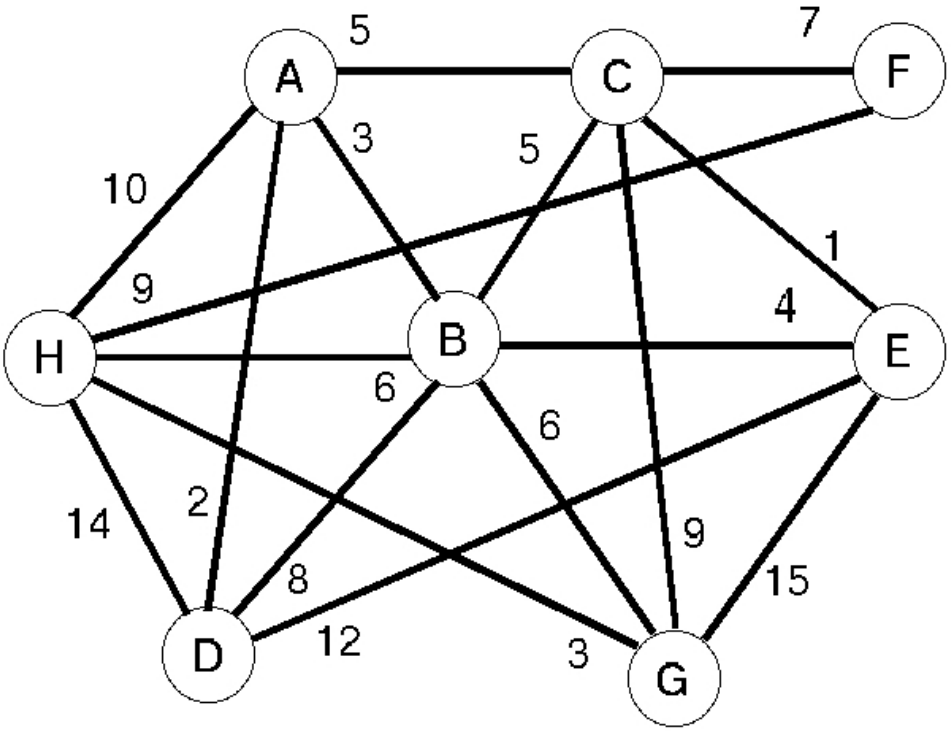


Figura 5.7: Grafo de ejemplo. (*https://upload.wikimedia.org/wikipedia/commons/e/ea/Caminosmascortos.jpg*)

Para resolverlo (usando orden lexicográfico), el algoritmo de búsqueda retroactiva seguiría los siguientes pasos:

1. [(H,0)]
2. [(H,1),(A,0)]
3. [(H,1),(A,1),(B,0)]
4. [(H,1),(A,1),(B,1),(C,0)] No se expande A ni B, pues ya estaban expandidos.
5. [(H,1),(A,1),(B,1),(C,1),(E,0)] El algoritmo acaba por ser E nodo final.

El camino encontrado por el algoritmo sería H-A-B-C-E.
La definición del algoritmo en pseudocódigo es la siguiente:

```
1  Busqueda_retroactiva(grafo):
2    lista=[(nodo inicial,0)]
3    tupla_actual=lista.ultimo
4    mientras(un nodo final no este en cerrados):
5      si( tupla_actual[1] < tupla_actual[0].numero_hijos ):
6        #Se expande el hijo que indique el contador de la tupla
7        lista.add( (tupla_actual[0].hijos[ tupla_actual[1] ],0) )
8        tupla_actual=lista.ultimo
9      si-no:
10       #incremento el contador de hijo a expandir del
11       #nodo padre de la tupla actual
12       lista.eliminar(tupla_actual)
13       si(lista.vacia):
14         devolver error
15       lista.ultimo[1]+=1
16       tupla_actual=lista.ultimo
17
18   camino=extraer los nodos desde el principio al final en la lista
19   devolver camino
```

Las características de la búsqueda en anchura son las siguientes:

- Completitud: es completo si usamos grafos finitos, pues todos los nodos serán expandidos si no se encuentra una solución antes.
- Optimalidad: No es óptimo en ningún caso, pues devuelve la primera solución que encuentra, que puede ser de mayor coste que otra posible solución.
- Complejidad temporal: En el peor caso, el orden de complejidad es $O(h^p)$, siendo h el número de hijos promedio (factor de ramificación) y p la máxima profundidad del grafo.
- Complejidad espacial (coste en memoria): En el peor caso $O(d)$, siendo d la profundidad de la solución más costosa. Esto ocurre porque sólo se guarda el camino hasta el nodo actual.

5.2.4. Descenso iterativo

El algoritmo de descenso iterativo consiste en la aplicación de búsqueda en profundidad limitando en cada iteración la profundidad máxima a la que puede llegar el algoritmo, lo que lo convierte en un híbrido de búsqueda en anchura y búsqueda en profundidad. Inicialmente se establece el límite en cero, que equivale a comprobar si el nodo inicial es final. Como se ha terminado de buscar en todos los nodos de nivel 0, se aumenta el nivel a 1 y se aplica búsqueda en profundidad al grafo hasta dicho nivel de profundidad. Así se continua sucesivamente hasta encontrar un camino a la solución. Lógicamente, encontrará la solución menos profunda pero, si tenemos en cuenta el coste de las aristas, no tiene por qué ser la menos costosa, aunque lo sería si todas las aristas tuviesen el mismo coste o este incrementase con la profundidad.

Este algoritmo aprovecha la ventaja de que necesita poca capacidad de memoria al basarse en búsqueda en profundidad. Además aprovecha la completitud de la búsqueda en anchura al basarse también en esta (cuando el grafo es finito). Aunque la búsqueda por descenso iterativo parece altamente ineficiente en tiempo al visitar estados que ya ha visitado antes, no lo es, puesto que los estados que más se visitan son los del principio que, normalmente no están demasiado ramificado. Suponiendo que p sea el mayor nivel de profundidad y h el número promedio de hijos por nodo, podemos analizar su complejidad en tiempo en el peor caso. Sabemos que los nodos en el nivel más profundo se visitarán una sola vez, si no se encuentra solución antes, entonces se expanden h^p nodos una sola vez. El nivel de profundidad anterior se expande dos veces, entonces tenemos $2 \cdot h^{p-1}$, así hasta llegar hasta p+1 nodos (incluyendo el nodo raíz) que se expande una sola vez. Siguiendo esta lógica, podemos desarrollar la siguiente serie:

$$h^p + 2 \cdot h^{p-1} + 3 \cdot h^{p-2} + 4 \cdot h^{p-4} + ... + (p-1) \cdot h^2 + p \cdot h + (p+1) = \sum_{i=0}^{p} (p+1-i) \cdot h^i$$

Si suponemos un arbol con factor de ramificación constante h, en el peor de los casos el orden de complejidad es $O(h^p)$.

La siguiente figura ilustra un ejemplo de búsqueda por descenso iterativo:

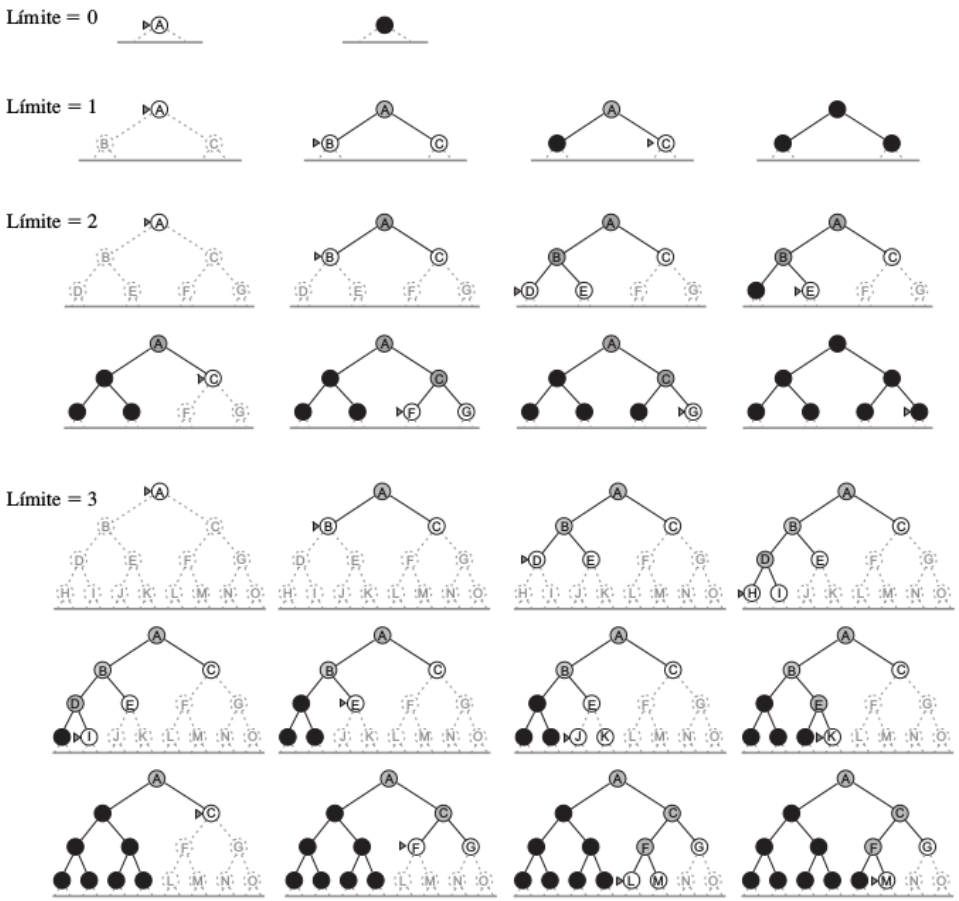


Figura 5.8: Ejemplo de búsqueda por descenso iterativo. [10]

Una posible definición en pseudocódigo del algoritmo de búsqueda por descenso iterativo, aprovechando que hemos definido la búsqueda en profundidad, podría ser el siguiente:

```
1  Descenso_iterativo(grafo):
2    contador_nivel=0
3    subgrafo=nulo
4    camino=nulo
5    mientras(camino!=error y subgrafo!=grafo):
6      subgrafo=extraer_subgrafo(grafo,contador_nivel)
7      camino=Busqueda_profundidad(subgrafo)
8      contador_nivel+=1
9    devolver camino
```

Las características de la búsqueda por descenso iterativo son las siguientes:

- Completitud: es completo si usamos grafos finitos, pues todos los nodos

serán expandidos si no se encuentra una solución antes. (Esto ocurre porque se basa en búsqueda en profundidad y búsqueda en anchura)

- Optimalidad: No se puede garantizar optimalidad en ningún caso al basarse en búsqueda en profundidad y búsqueda en anchura. Recordemos que ninguno de estos algoritmos garantizan optimalidad, ya que devuelven la primera solución que encuentran y no necesariamente es la mejor.
- Complejidad espacial (coste en memoria): En el peor caso $O(h^d)$, siendo d la profundidad de la solución más costosa. En realidad habría que multiplicar por dos porque se guarda también el número de hijo siguiente que debería expandir cada nodo.

Como se puede apreciar, obtenemos las mismas características que una búsqueda en profundidad simple, añadiendo las ventajas de la búsqueda en anchura. Visto esto, podemos asegurar que la búsqueda por descenso iterativo es, como muy mala, igual que la búsqueda en profundidad.

5.2.5. Búsqueda por costo uniforme (Algoritmo de Dijkstra)

El algoritmo de costo uniforme es el primero de los algoritmos que llevamos estudiados que usa información de coste. La idea surge desde la búsqueda en anchura, donde conseguimos un resultado óptimo si la función de costo no es decreciente, pero ¿por qué en lugar de expandir el nodo más superficial no expandimos el nodo cuyo camino desde el nodo origen sea el menor? De este modo conseguiremos que el algoritmo alcance el óptimo independientemente de cómo sea la función de coste. La mecánica del algoritmo, pues, consiste en expandir aquél nodo tal que, desde el nodo actual, el camino recorrido mas el coste de la arista hacia él desde el nodo actual sea menor. Obviamente, la única restricción es que el coste de las aristas no sea negativo, pero es una restricción razonable. Este algoritmo no tiene en cuenta el número de pasos que hay que dar hacia la solución, sino su coste total. Este algoritmo es una especificación del algoritmo de Dijkstra, esto es, mientras que el algoritmo de Dijkstra nos permite saber cuál es el coste mínimo desde el nodo inicial a cualquier nodo, el algoritmo de costo uniforme se queda sólo con el camino y coste desde el nodo inicial al nodo solución. La diferencia es mínima, como veremos en el ejemplo más adelante.

En cuanto a requerimientos, el algoritmo de costo uniforme no puede ser calificado fácilmente, ya que depende de la función de coste de las aristas en lugar de la profundidad a la que se encuentren los nodos finales. Si suponemos que el número de hijos promedio es h, que el coste de la solución óptima es C^* y que todas las aristas tienen un valor estrictamente positivo mayor o igual a ε, entonces la complejidad temporal y espacial del algoritmo es del orden de $O(h^{\frac{C^*}{\varepsilon}})$. Esto puede ser una desventaja, pues si

C^* es lo suficientemente alto y ε lo suficientemente pequeño, la complejidad del algoritmo puede dispararse, siendo mayor que las complejidades de los algoritmos anteriormente estudiados.

Vamos a tratar de ilustrar un ejemplo del algoritmo de costo uniforme y posteriormente veremos una pequeña modificación para obtener el menor coste desde el nodo inicial a cualquier nodo. Sea el grafo de la siguiente figura, con nodo inicial A y nodo final C:

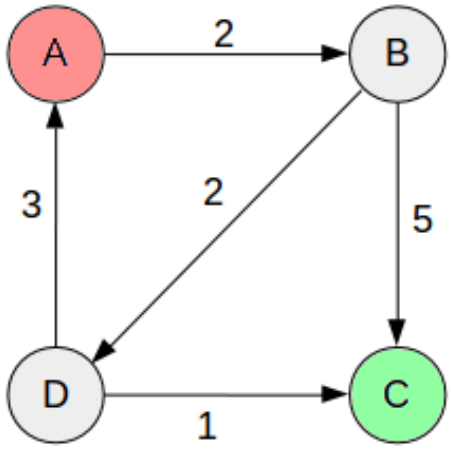


Figura 5.9: Ejemplo de grafo para resolver por Coste Uniforme y Dijkstra

Resolución por Coste uniforme:

1. Abiertos: [(-,A,0)]
   Cerrados: []

2. Abiertos: [~~(-,A,0)~~,(A,B,2)]
   Cerrados: [(-,A,0)]

3. Abiertos: [~~(-,A,0)~~,~~(A,B,2)~~,(B,D,4),(B,C,7)]
   Cerrados: [(-,A,0),(A,B,2)]
   En este paso el camino de B a D cuesta 4 porque llegar de A a B costaba 2 y de B a D otros 2, $2 + 2 = 4$. Lo mismo ocurre con el camino de B a C, $2 + 5 = 7$. De entre los posibles nodos a expandir elegimos el de menor suma, el nodo D, partiendo de B.

4. Abiertos: [~~(-,A,0)~~,~~(A,B,2)~~,~~(B,D,4)~~,~~(B,C,7)~~,(D,A,7),(D,C,5)]
   Cerrados: [(-,A,0),(A,B,2),(B,D,4)]
   En este momento, se deben expandir los hijos de D, que son A y C, ¡pero cuidado! C ya había sido abierto por B. En este momento, debemos evaluar si el nuevo camino hacia C (pasando por D) es mejor que el que ya teníamos (pasando por B). El camino que teníamos era

A-B-C de coste 7 ¿Es mejor si pasamos de B a D y de D a C en lugar del camino directo de B a C? Se puede comprobar que sí: desde B a C directamente obtenemos coste 5, mientras que de B a D, coste 2 y de D a C coste 1, lo que hace un coste de 3, que junto al coste de ir desde A (nodo inicial) hasta B, coste 2, tenemos un coste total de 2+1+2=5. Sustituímos el nodo abierto C con padre B (tachado en rojo) por el nuevo camino encontrado: pasando por D (aparece en verde).

5. Abiertos: [~~(-,A,0)~~,~~(A,B,2)~~,~~(B,D,4)~~,(D,A,7),~~(D,C,5)~~]
   Cerrados: [(-,A,0),(A,B,2),(B,D,4),(D,C,5)]
   Dado que el nodo C es final, el algoritmo termina y conocemos tanto el camino: A,B,D,C y el mínimo coste: 5. Como se puede apreciar es fácil reconstruir la secuencia de nodos (camino) seguida hasta la solución.

Aunque en este ejemplo no se ha dado el caso de actualización en la lista de cerrados, es similar a lo que ocurre en el paso 4, pero en la mencionada lista. Relativo a esto, podemos implementar una fase de propagación. Esta fase tiene por objetivo propagar la reducción del coste del camino a todos aquellos nodos que cuelgan directa o indirectamente del que se actualizó. Obviamente, si el camino desde un nodo n a un nodo n' es actualizado por mejora del coste, esta mejora afecta a todos los caminos que parten desde n'.

En esta fase tiene dos partes importantes:

- Actualización: cada vez que un nodo es introducido en cerrados se reconstruye el camino desde el nodo inicial y se comprueba que el coste es real. En caso contrario, el nodo que se acaba de introducir debe actualizarse con la suma del coste de llegar hasta su padre más el coste desde su padre a él, de manera que el coste del mejor camino conocido sea en todo momento real. Esto debe hacerse para que si un nodo de la lista de abiertos pasa a cerrados después de que un nodo del que depende sufrió una actualización en cerrados, la mejora sea efectiva y se refleje en dicha lista.

- Propagación: Cuando un nodo está en cerrados y se encuentra un mejor camino a él, hay que buscar todos los nodos que cuelgan directa o indirectamente en la lista de cerrados y actualizar el camino a ellos. Para ello guardamos en una lista de propagación todos los nodos hijos del nodo actualizado y actualizamos su camino con la suma que se describe en Actualización. Ahora estos nodos pasan a ser padres y se buscan los hijos de los mismos, actualizando el camino con la misma suma, hasta que no queden nodos en la lista de propagación.

Resolución por Dijkstra:

La resolución por Dijkstra es parecida a la anterior sólo que al elegir el siguiente nodo a expandir hay que barrer el resto de nodos, comprobando si

se puede rebajar el coste del camino pasando por el nuevo nodo expandido. La siguiente tabla resume la aplicación del algoritmo:

B	C	D		Visitados
2	∞	∞		{A}
2	7	4		{A,B}
2	5	4		{A,B,D}
				{A,B,D,C}

La primera fila corresponde al estado inicial, donde colocaremos en visitados el nodo inicial, que se convierte en actual. En el resto de celdas colocaremos la distancia desde el nodo actual hasta cada uno de los nodos restantes. Se coloca ∞ si el nodo no es accesible desde el nodo actual. Para construir los siguientes pasos, se elige el nodo cuya distancia es menor, se marca como mínimo local, se añade a visitados y se establece como nodo actual. A continuación para calcular la nueva distancia entre el nodo inicial y cada uno de los nodos n_i restantes, calculamos:

$$d(n_{ini}, n_i) = min\{d(n_{ini}, n_i), d(n_{ini}, n_{act}) + d(n_{act}, n_i)\}$$

Se repite el proceso hasta que todos los nodos han sido marcados como mínimo local. Cuando esto ocurre, se incluye en visitados el último nodo marcado y, como se puede apreciar, obtenemos el mínimo coste desde el nodo inicial a cualquier otro nodo.

A la hora de generar heurísticas admisibles, utilizaré Dijkstra para conocer el coste del camino mínimo desde cualquier nodo a un nodo final. Para ello, lanzaré el algoritmo de Dijkstra tantas veces como nodos haya, incluyendo los finales, ya que podrían tener aristas que conecten consigo mismos. Por cada resultado del algoritmo de Dijkstra tendré el valor heurístico del nodo que haya establecido como inicial. Si devuelvo valores aleatorios menores que este valor ideal como valor heurístico para cada nodo, obtenemos una heurística admisible.

Una posible definición en pseudocódigo del algoritmo de costo uniforme podría ser la siguiente. Para implementar Dijstra bastaría con añadir una lista de visitados y cambiar la condición de terminación a que todos los estén incluidos en visitados.

```
1  Busqueda_costo_uniforme(grafo):
2    abiertos=[(-,nodo_inicial,0)]
3    coste_recorrido=0
4    tupla_actual=abiertos.ultimo
5    mientras(un nodo final no este en cerrados):
6      si(tupla_actual[1] en cerrados):
7        si(tupla_actual[2] < tupla_en_cerrados[2]):
8          #Actualizo cerrados para mejorar el coste de la tupla
9          cerrados[indice(tupla_en_cerrados)]=tupla_actual
10         propagacion_actualizacion;
```

```
11      si(tupla_actual[1] en abiertos):
12        si(tupla_actual[2] < tupla_en_abiertos[2]):
13          #Actualizo abiertos para mejorar el coste de la tupla
14          abiertos[indice(tupla_en_abiertos)]=tupla_actual
15      #Los dos IF anteriores van calculando los minimos locales
16      si-no:
17        para cada hijo en tupla_actual[1].hijos
18          abiertos.add( (tupla_actual[1],hijo,coste_recorrido +
                coste_arista_padre_a_hijo) )
19        cerrados.add(tupla_actual)
20      abiertos.eliminar(tupla_actual)
21      si(abiertos.vacia)
22        devolver error
23      tupla_actual=abiertos.tupla_menor_suma
24      coste_recorrido += tupla_actual[2]
25    camino=reconstruir camino desde cerrados
26    devolver camino
```

Las características, por tanto, de la búsqueda por costo uniforme son las siguientes:

- Completitud: es completo si usamos grafos finitos y si el costo de cada paso es mayor que algún ε positivo. Todos los nodos serán expandidos si no se encuentra una solución antes.
- Optimalidad: es óptimo con las mismas restricciones establecidas para la completitud.
- Complejidad temporal: $O(h^{\frac{C^*}{\varepsilon}})$ por la explicación vista anteriormente.
- Complejidad espacial (coste en memoria): $O(h^{\frac{C^*}{\varepsilon}})$ por la explicación vista anteriormente.

5.2.6. Escalada simple

El algoritmo de escalada simple (hill climbing) es tan simple como poco eficaz. Es el primero de los que hemos estudiado hasta ahora que usa la información heurística de los nodos. La mecánica es sencilla: se expande el primer hijo del nodo actual que mejore el valor heurístico del padre. El principal problema de este algoritmo es que tiene una gran probabilidad de quedar atrapado en mínimos locales, ya que no posee la capacidad de realizar backtracking. Esto puede traducirse en descender por una rama que no conduzca a la solución, por guiarse únicamente por el valor heurístico y quedar atrapado en un nodo hoja que no sea final o en un nodo cuyos hijos no mejoren la heurística del padre.

En lo que a requisitos se refiere, el algoritmo de escalada simple tiene las mismas propiedades que una búsqueda en profundidad en el peor caso. Igualmente ocurre con la complejidad espacial.

Ilustremos con un ejemplo cómo funcionaría el algoritmo de búsqueda de escalada simple. Consideremos el grafo siguiente donde cada nodo esta

representado por ID(valor heurístico), el nodo inicial es A y los nodos finales son H e I:

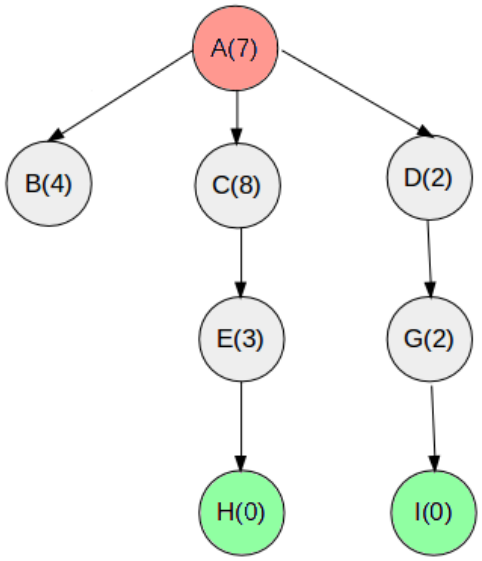


Figura 5.10: Ejemplo de grafo para resolver por Escalada Simple

La resolución sería así:

1. [(-,A,7)]
2. [(-,A,7),(A,B,4)]
   Dado que B no tiene hijos, el algoritmo no puede seguir la búsqueda y queda estancado, por lo que finaliza sin poder encontrar un camino desde el nodo inicial a un nodo final.

Dado que B no tiene hijos, el algoritmo de escalada simple queda atrapado y acaba con estado de error, ya que no fue posible encontrar un camino desde el nodo raíz hasta uno final.

Una definición en pseudocódigo podría ser la siguiente:

```
1  Escalada_simple(grafo):
2    lista=[(-,nodo_inicial,heuristica(nodo_inicial)]
3    tupla_actual=lista.ultimo
4    mientras(no haya un nodo final en la lista):
5      si tupla_actual[1].numero_hijos > 0:
6        para cada hijo de tupla_actual[1]:
7          hijo_mejora=nulo
8          si(heuristica(hijo) < tupla_actual[2]):
9            hijo_mejora=hijo
10           lista.add( (tupla_actual[1],hijo_mejora,heuristica(
                 hijo_mejora)) )
11           tupla_actual=lista.ultimo
12           #Paro de buscar mas hijos que mejoran
```

```
13              romper bucle
14          si(hijo_mejora==nulo):
15            #Ningun hijo mejoro la heuristica del padre
16            devolver error
17        si-no:
18          #Se ha quedado atrapado en una hoja no final
19          devolver error
20      camino=reconstruir camino desde lista
21      devolver camino
```

Las características del algoritmo de escalada simple son las siguientes:

- Completitud: No se garantiza la completitud por la posibilidad de quedar atrapado en nodos hoja no finales.
- Optimalidad: No se garantiza la optimalidad en ningún caso.
- Complejidad temporal: En el peor caso, el orden de complejidad es $O(h^p)$, al igual que búsqueda en profundidad.
- Complejidad espacial (coste en memoria): En el peor caso $O(d)$, siendo d la máxima profundidad del grafo porque a diferencia de búsqueda en profundidad, que guardaba una lista de abiertos y cerrados, en los algoritmos de escalada sólo se guardan los nodos que forman parte del camino desde el nodo inicial al nodo actual. Esto es así aunque el algoritmo quede estancado como en el ejemplo, porque no se puede realizar *backtracking*.

5.2.7. Escalada por la máxima pendiente

La escalada por la máxima pendiente tiene una única diferencia con la escalada simple y es que, en lugar de quedarse con el primer hijo que mejora la heurística del padre, se queda con el hijo cuya heurística es mejor que la de su padre, siendo la mejor de entre la de sus hermanos. Para todo lo demás es exactamente igual que la versión simple.

Vamos a ilustrar la diferencia principal con el mismo ejemplo que usamos en la escalada simple:

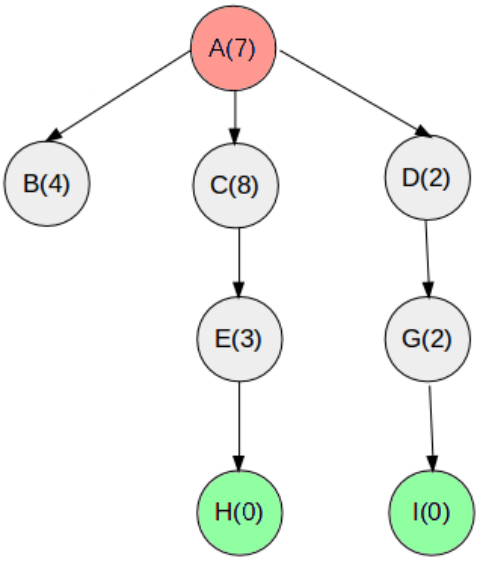


Figura 5.11: Ejemplo de grafo para resolver por Escalada por la máxima pendiente

La resolución sería así:

1. [(-,A,7)]
2. [(-,A,7),(A,D,2)]
   En cada paso, se busca el hijo con mejor valor heurístico, si este, además, es mejor que el de su padre, pasa a ser nodo padre y expandir sus hijos. En los demás pasos se procede de forma análoga.
3. [(-,A,7),(A,D,2),(D,G,2)]
4. [(-,A,7),(A,D,2),(D,G,2),(G,I,0)]

Como se puede apreciar, este pequeño cambio ha permitido encontrar una solución y, evidentemente, es una mejora. La escalada por la máxima pendiente permite evitar situaciones de bloqueo si la heurística es realista (por ejemplo, que el valor heurístico de un nodo sea muy alto si no se alcanza una solución, aunque este mejore el valor heurístico del padre), pero no puede evitar las situaciones de bloqueo en general.

Una definición del algoritmo de máxima pendiente en pseudocódigo sería la siguiente:

```
1  Escalada_max_pendiente(grafo):
2    lista=[(-,nodo_inicial,heuristica(nodo_inicial)]
3    tupla_actual=lista.ultimo
4    mientras(no haya un nodo final en la lista):
5      si tupla_actual[1].numero_hijos > 0:
```

```
6         para cada hijo de tupla_actual[1]:
7           mejor_hijo=nulo
8           si(mejor_hijo==nulo)
9             si(heuristica(hijo) < tupla_actual[2]):
10              mejor_hijo=hijo
11          si-no
12            si(heuristica(hijo) < heuristica(mejor_hijo)):
13              mejor_hijo=hijo
14          si(mejor_hijo==nulo):
15            devolver error
16          lista.add( (tupla_actual[1],mejor_hijo,heuristica(mejor_hijo
                ) ) )
17          tupla_actual=lista.ultimo
18      si-no:
19        #Se ha quedado atrapado en una hoja no final
20        devolver error
21    camino=reconstruir camino desde lista
22    devolver camino
```

Observemos cómo el único cambio es buscar cuál es el mejor hijo en lo que al valor heurístico se refiere.

Visto todo esto, podemos decir que las características del algoritmo son las mismas que la escalada simple, a pesar de la pequeña mejora:

- Completitud: No se garantiza la completitud por la posibilidad de quedar atrapado en nodos hoja no finales.
- Optimalidad: No se garantiza la optimalidad en ningún caso.
- Complejidad temporal: En el peor caso, el orden de complejidad es $O(h^p)$, al igual que búsqueda en profundidad.
- Complejidad espacial (coste en memoria): En el peor caso $O(d)$, siendo d la máxima profundidad del grafo. El razonamiento es el mismo que realizábamos en escalada simple.

5.2.8. A*

El algoritmo A* es el más potente de entre los que estudiamos ya que usa la información heurística junto con la información de costes de las aristas. Este algoritmo está teóricamente muy estudiado y en la actualidad no es posible mejorar a este algoritmo en lo que a búsqueda de caminos se refiere, a pesar de que fue presentado por primera vez en 1968 [6]. Es el algoritmo más eficaz y eficiente en el campo de la búsqueda de caminos y es por ello que, quizás, sea el más famoso de todos los que aquí aparecen. La herramienta que usa el algoritmo A* es lo que se denomina función $f(n)$. Para cada nodo n_i, se puede calcular la función $f(n_i)$, como sigue:

$$f'(n_i) = g(n_i) + h'(n_i)$$

Donde cada uno de estos términos son:

- $g(n_i)$ es el coste real del camino recorrido desde el nodo inicial hasta n_i. Si n_i es nodo inicial, entonces $g(n_i) = 0$.
- $h'(n_i)$ es una estimación del coste, desde n_i, para llegar a un nodo final. Si n_i es un nodo final, entonces $h(n_i) = 0$. Se denomina función heurística o valor heurístico del nodo.
- $f'(n_i)$ es el coste total **estimado** desde el nodo raíz a un nodo final.

Como vemos, A* es la máxima expresión del algoritmo de búsqueda informado. Aunque no es el objetivo de este documento ver todas las demostraciones de por qué A* tiene las propiedades que lo convierten en el mejor algoritmo de búsqueda, vamos a tratar de evaluar su complejidad en tiempo y espacio. En primer lugar, cabe destacar que la complejidad en tiempo de A* está estrechamente ligada con la calidad de la heurística que se utilice, esto es, cuánto de cercana es a la realidad. Si la calidad de la heurística es mala, el algoritmo se ejecutará en tiempo exponencial $O(c^n)$, c constante. Si la calidad de la heurística es perfecta ($h'(n_i)$ coincide exactamente con el coste mínimo desde n_i a un nodo solución), el algoritmo se ejecutará en tiempo lineal $O(n)$. Para que esto ocurra, además de que la heurística sea admisible (el valor heurístico de un nodo jamás supera el coste mínimo desde dicho nodo a un nodo final) sea n_i y n_j con n_j descendiente de n_i, debe cumplirse que:

$$h'(n_i)+g(n_i) \leq h'(n_j)+g(n_j) \Rightarrow h'(n_i) \leq h'(n_j)+g(n_j)-g(n_i) \quad \forall i,j\ i \neq j$$

Esto no significa más que la heurística de un nodo tiene que ser menor o igual que la heurística de su descendiente mas la arista que los une. Esto es lo que se conoce como condición de monotonía (o consistencia):

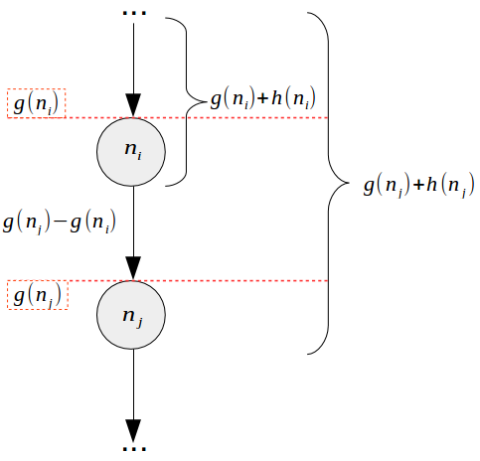


Figura 5.12: Condición de consistencia

El gran problema de A* son sus requisitos en cuanto a memoria. La cantidad de memoria que necesita A* es exponencial con respecto al tamaño del problema, ya que necesita guardar todos los posibles nodos descendientes de cada estado. Para solucionar este problema se han propuesto variantes como RTA*, IDA* o SMA* [7], pero aunque mejoran este aspecto, empeoran en otros, luego no se puede decir que mejoren de forma absoluta a A*.

Vamos a ilustrar el funcionamiento de A* con un ejemplo, usando el mismo grafo de ejemplo que usamos en los algoritmos de escalada, pero prestando atención a los costes de las aristas. La notación de las tuplas será (padre,hijo,$g(hijo)$,$h(hijo)$):

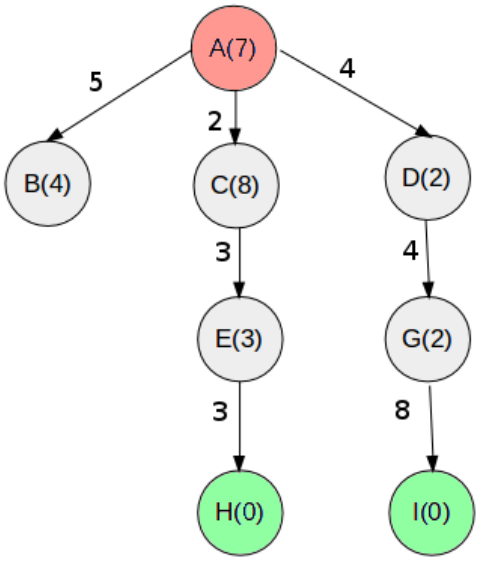


Figura 5.13: Ejemplo de grafo para resolver por A*

1. Abiertos: [(-,A,0,)]
   Cerrados: []

2. Abiertos: [~~(-,A,0,)~~, (A,B,5,4), (A,C,2,8), (A,D,4,2)]
   Cerrados: [(-,A,0,)]

3. Abiertos: [~~(-,A,0,)~~, (A,B,5,4), (A,C,2,8), ~~(A,D,4,2)~~, (D,G,8,2)]
   Cerrados: [(-,A,0,),(A,D,4,2)]
   En este punto aparece el nodo G partiendo de D con coste de camino 8, esto es porque este valor en la tupla debe contener el mejor coste acumulado desde el nodo inicial hasta el padre más el coste desde el padre al hijo. En este caso, el mejor coste acumulado (hasta ahora) desde el nodo inicial a D era 4, desde D a G hay coste 4, luego 4+4=8, que es el valor que aparece. En los demás pasos se procede de forma análoga.

4. Abiertos: [~~(-,A,0,)~~, ~~(A,B,5,4)~~, (A,C,2,8), ~~(A,D,4,2)~~, (D,G,8,2)]
   Cerrados: [(-,A,0,),(A,D,4,2),(A,B,5,4)]
   Dado que los dos nodos a expandir tienen el mismo valor $f'(n_i)$ se aplica como criterio de desempate el orden lexicográfico, eligiendo C.

5. Abiertos: [~~(-,A,0,)~~, ~~(A,B,5,4)~~, ~~(A,C,2,8)~~, ~~(A,D,4,2)~~, (D,G,8,2), (C,E,5,3)]
   Cerrados: [(-,A,0,),(A,D,4,2),(A,B,5,4),(A,C,2,8)]

6. Abiertos: [~~(-,A,0,)~~, ~~(A,B,5,4)~~, ~~(A,C,2,8)~~, ~~(A,D,4,2)~~, (D,G,8,2), ~~(C,E,5,3)~~, (E,H,8,0)]
   Cerrados: [(-,A,0,),(A,D,4,2),(A,B,5,4),(A,C,2,8),(C,E,5,3)]

7. Abiertos: [~~(-,A,0,)~~, ~~(A,B,5,4)~~, ~~(A,C,2,8)~~, ~~(A,D,4,2)~~, (D,G,8,2), ~~(C,E,5,3)~~, ~~(E,H,8,0)~~]
   Cerrados: [(-,A,0,),(A,D,4,2),(A,B,5,4),(A,C,2,8),(C,E,5,3),(E,H,8,0)]
   El algoritmo termina por encontrarse un nodo final en la lista de cerrados. La reconstrucción del camino desde cerrados resulta evidente: A,C,E,H, con coste 8.

Hay algunas consideraciones a tener en cuenta en el algoritmo A*, aunque en este caso no se han dado. En la lista de abiertos y cerrados sólo puede haber un nodo viniendo de un padre, esto significa que hay que hacer revisiones para corregir el mejor camino encontrado hasta ahora. Del mismo modo, se debe tener en cuenta una fase de propagación de las correcciones. No es casualidad que estas actualizaciones recuerden al algoritmo de costo uniforme (Dijkstra) y es que el algoritmo de costo uniforme es un caso particular del algoritmo A*, donde todos los nodos tienen el mismo valor heurístico, por lo que lo único que altera el comportamiento de costo uniforme es la componente $g(n_i)$, esto es, los costes acumulados de las aristas. Dicho esto, podemos dar una definición en pseudocódigo, ampliando el algoritmo de costo uniforme, introduciendo la información que proporciona la heurística:

```
1  Astar(grafo):
2    #Cada tupla es de la forma (padre,hijo,g,h)
3    abiertos=[(-,nodo_inicial,0,heuristica(nodo_inicial))]
4    coste_recorrido=0
5    tupla_actual=abiertos.ultimo
6    mientras(un nodo final no este en cerrados):
7      si(tupla_actual[1] en cerrados):
8        si(tupla_actual[2]+tupla_actual[3] < tupla_en_cerrados[2]+
             tupla_en_cerrados[3]):
9          #Actualizo cerrados para mejorar el coste de la tupla
10         cerrados[indice(tupla_en_cerrados)]=tupla_actual
11         propagacion_actualizacion;
12     si(tupla_actual[1] en abiertos):
13       si(tupla_actual[2]+tupla_actual[3] < tupla_en_abiertos[2]+
             tupla_en_abiertos[3]):
14         #Actualizo abiertos para mejorar el coste de la tupla
15         abiertos[indice(tupla_en_abiertos)]=tupla_actual
```

```
16        #Los dos IF anteriores van calculando los minimos locales
17        si-no:
18          para cada hijo en tupla_actual[1].hijos
19            abiertos.add( (tupla_actual[1],hijo,coste_recorrido +
                  coste_arista_padre_a_hijo, heuristica(hijo)) )
20          cerrados.add(tupla_actual)
21        abiertos.eliminar(tupla_actual)
22        si(abiertos.vacia)
23          devolver error
24        #El criterio de seleccion ahora es el que menor g+h tenga
25        tupla_actual=abiertos.tupla_menor(tupla[2]+tupla[3])
26        coste_recorrido += tupla_actual[2]
27      camino=reconstruir camino desde cerrados
28      devolver camino
```

Ahora que ya conocemos un poco cómo funciona A*, sus características son las siguientes:

- Completitud: Es completo: se recorren todos los nodos si no se encuentra una solución antes, esto es porque siempre se va a elegir el nodo en abiertos cuya suma sea la menor. Aunque quede uno con una suma de $g + h$ muy grande, se explorará si es el único que queda.
- Optimalidad: Se garantiza la optimalidad si la función heurística es admisible y monótona.
- Complejidad temporal: En el peor caso, el orden de complejidad es exponencial, $O(c^n)$, c constante y n el tamaño del problema. En el mejor caso, con heurística ideal, el tiempo es lineal $O(n)$.
- Complejidad espacial (coste en memoria): La complejidad espacial de A* es exponencial $O(c^n)$, c constante.

Pruebas

6.1. Pruebas de caja blanca (PCB)

Las pruebas de caja blanca se centran en los detalles de implementación. Normalmente, quien hace las pruebas debe conocer el código fuente por lo que, generalmente, las pruebas de caja blanca las hace el mismo equipo de desarrollo. En este proyecto, como soy el único programador, diseñaré las baterías de pruebas de caja blanca y las realizaré, detallando los resultados. A pesar de que finalmente la aplicación no ha sido programada por módulos, conceptualmente sí que se pueden ser diferenciados, así que probaré cada módulo por separado para conseguir un orden de procedimiento y tratar de no dejar pasar ningún detalle. Cada una de las siguientes pruebas se realiza de forma indpendiente.

6.1.1. PCB módulo de creación de grafos

Insertar un nodo inicial con valores correctos

Se intenta introducir un nodo inicial que, en principio no presente anomalías. Se inserta en el campo Identificador del nodo un String (A), un valor heurístico numérico (5) y se marca el checkbox "Es nodo inicial".

- **Resultado esperado:** Nodo insertado.
- **Resultado obtenido:** Nodo insertado.

La prueba tiene éxito.

Figura 6.1: Inserción sin anomalías de un nodo inicial.

Insertar un nodo intermedio con valores correctos

Se intenta introducir un nodo intermedio que, en principio no presente anomalías. Se inserta en el campo Identificador del nodo un String (A), un valor heurístico numérico (5) y se no se marca ningún checkbox.

- **Resultado esperado:** Nodo insertado.
- **Resultado obtenido:** Nodo insertado.

La prueba tiene éxito.

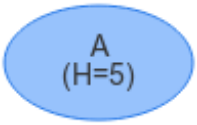


Figura 6.2: Inserción sin anomalías de un nodo intermedio.

Insertar un nodo final con valores correctos

Se intenta introducir un nodo final que, en principio no presente anomalías. Se inserta en el campo Identificador del nodo un String (A), un valor heurístico numérico (5) y se marca el checkbox "Es nodo final".

- **Resultado esperado:** Nodo insertado.
- **Resultado obtenido:** Nodo insertado.

La prueba tiene éxito.

Figura 6.3: Inserción sin anomalías de un nodo final.

Insertar un nodo dejando el formulario vacío.

Se intenta introducir un nodo, dejando el formulario vacío

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

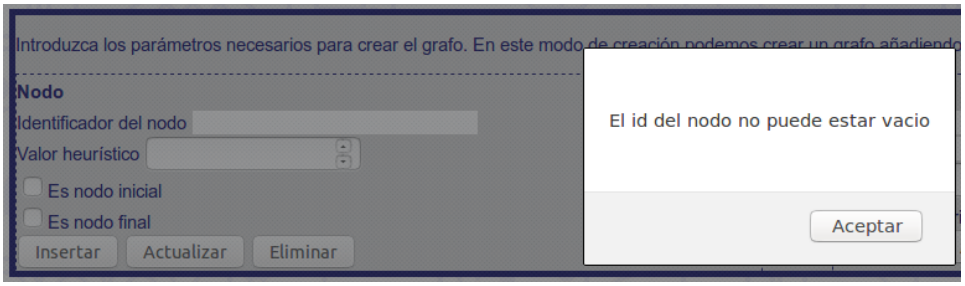


Figura 6.4: Introducción de un nodo, dejando el formulario vacío

Insertar un nodo con un valor heurístico vacío.

Se intenta introducir un nodo cuyo valor heurístico esté vacío.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

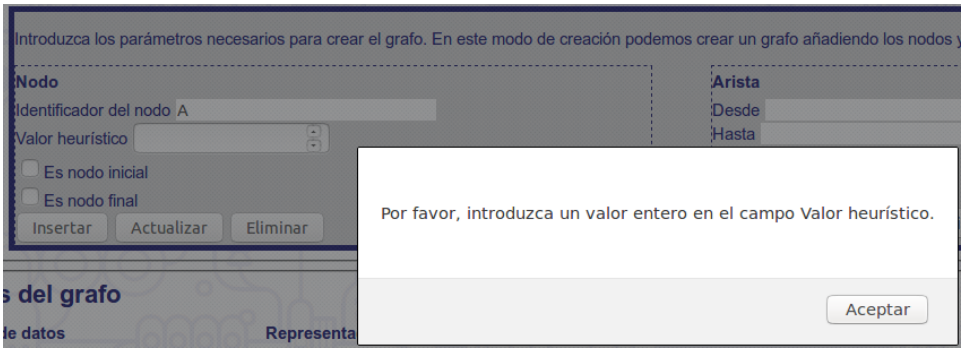


Figura 6.5: Error al dejar el valor heurístico vacío.

Insertar un nodo con un valor heurístico no numérico.

Se intenta introducir un nodo cuyo valor heurístico no sea numérico.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

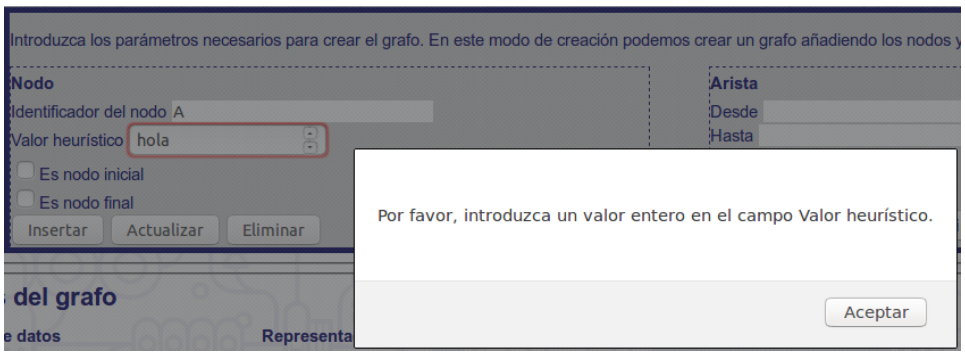


Figura 6.6: Error al introducir un valor heurístico no numérico.

Insertar un nodo con un valor heurístico negativo.

Se intenta introducir un nodo cuyo valor heurístico sea negativo.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

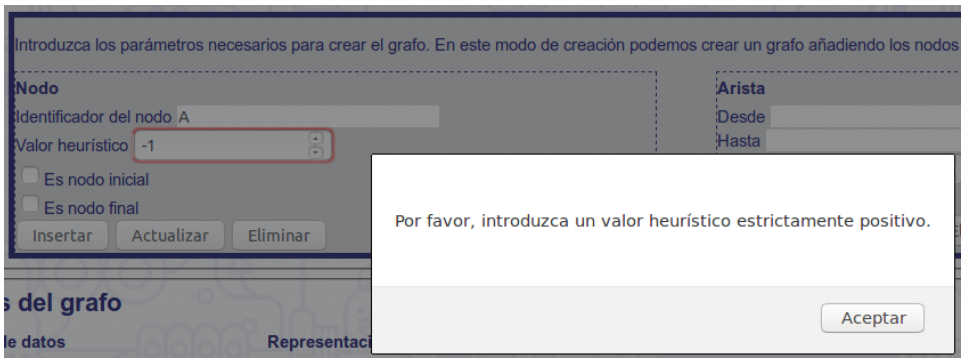


Figura 6.7: Error al introducir un valor heurístico negativo.

Insertar un nodo final e inicial a la vez.

Se intenta introducir un nodo marcando los checkbox de nodo inicial y final a la vez.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

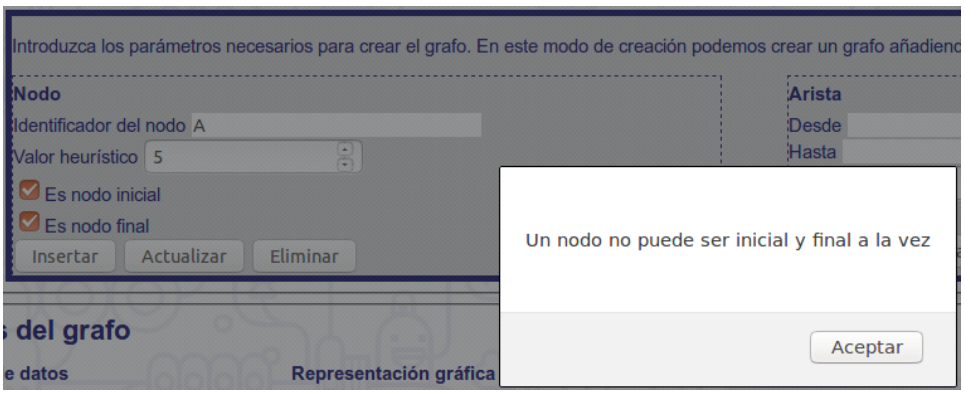


Figura 6.8: Error al introducir un nodo inicial y final a la vez.

Insertar una arista dirigida sin anomalías.

Se intenta introducir una arista dirigida que, en principio no presente anomalías. Se deben crear previamente dos nodos para unirlos mediante esta arista. Se introducirá en los campos "desde" y "hasta" un string igual a los dos nodos que existen. En el campo coste, introduciremos un valor entero (3) y marcaremos el checkbox correspondiente a "Es arista dirigida".

- **Resultado esperado:** Arista creada.
- **Resultado obtenido:** Arista creada.

La prueba tiene éxito.

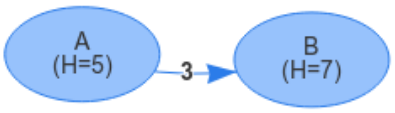


Figura 6.9: Introducción sin anomalías de una arista dirigida.

Insertar una arista bidireccional sin anomalías.

Se intenta introducir una arista bidireccional que, en principio no presente anomalías. Se deben crear previamente dos nodos para unirlos mediante esta arista. Se introducirá en los campos "desde" y "hasta" un string igual a los dos nodos que existen. En el campo coste, introduciremos un valor entero (7) y NO marcaremos el checkbox correspondiente a "Es arista dirigida".

- **Resultado esperado:** Arista creada.

- **Resultado obtenido:** Arista creada.

La prueba tiene éxito.

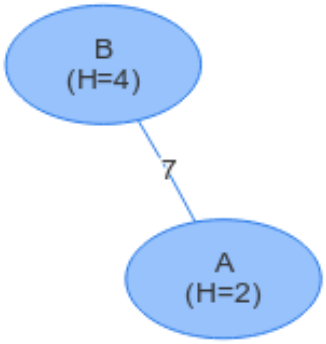


Figura 6.10: Introducción sin anomalías de una arista bidireccional.

Insertar una arista dejando el formulario vacío.

Se intenta introducir una arista dejando el formulario completamente vacío.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

Figura 6.11: Introducción de una arista dirigida dejando en blanco el formulario.

Insertar una arista dejando en blanco los dos nodos.

Se intenta introducir una arista dejando en blanco los campos correspondientes a los dos nodos. Se introduce un valor correcto en el campo coste.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

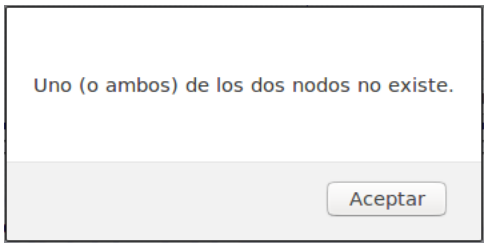


Figura 6.12: Introducción de una arista dirigida dejando en blanco los identificadores de nodos, valor correcto en el coste.

Insertar una arista dejando en blanco uno de los dos nodos.

Se intenta introducir una arista dejando en blanco uno de los campos correspondientes a los nodos. Se introduce un valor correcto en el campo coste.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

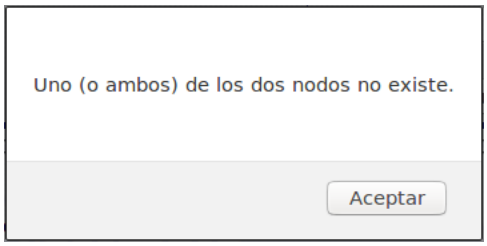


Figura 6.13: Introducción de una arista dirigida dejando en blanco el nodo superior, valores correctos en el resto.

Insertar una arista que une nodos que no existen.

Se intenta añadir una arista, introduciendo en el apartado de nodos, identificadores de nodos que no existen.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

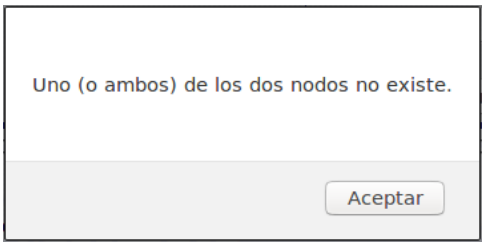


Figura 6.14: Introducción de una arista que une nodos que no existen.

Insertar una arista con valores no numéricos de coste.

Se intenta añadir una arista, introduciendo en el apartado de coste un valor no numérico.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

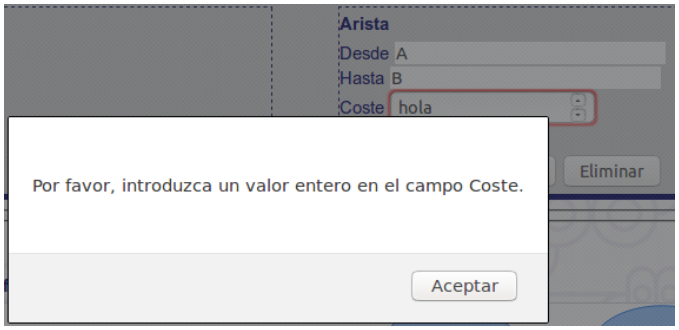


Figura 6.15: Introducción de una arista cuyo coste no es numérico.

Insertar una arista con valores negativos de coste.

Se intenta añadir una arista, introduciendo en el apartado de coste un valor negativo.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

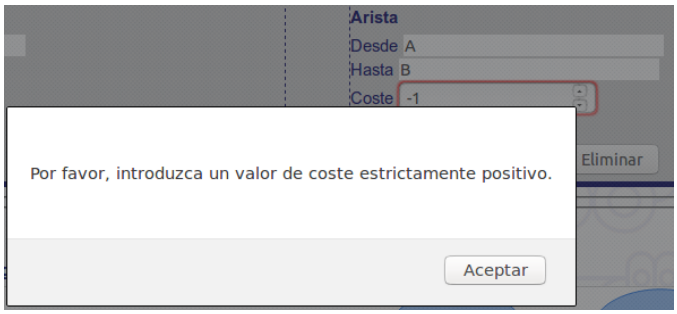


Figura 6.16: Introducción de una arista cuyo coste es negativo.

6.1.2. PCB módulo de importación/exportación de grafos

Exportar un grafo sin anomalías.

Se intenta exportar un grafo con un nodo inicial, un nodo final, un nodo intermedio y aristas que los unen.

- **Resultado esperado:** Archivo de exportación.
- **Resultado obtenido:** Archivo de exportación.

La prueba tiene éxito.

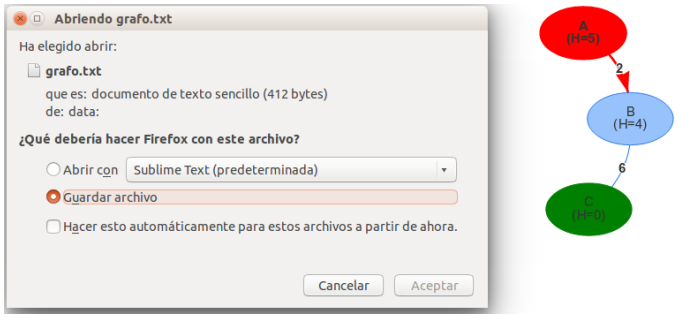


Figura 6.17: Exportación de un grafo sin anomalías.

Exportar un grafo sin nodo inicial.

Se intenta exportar un grafo sin nodo inicial, un nodo final, un nodo intermedio y aristas que los unen.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

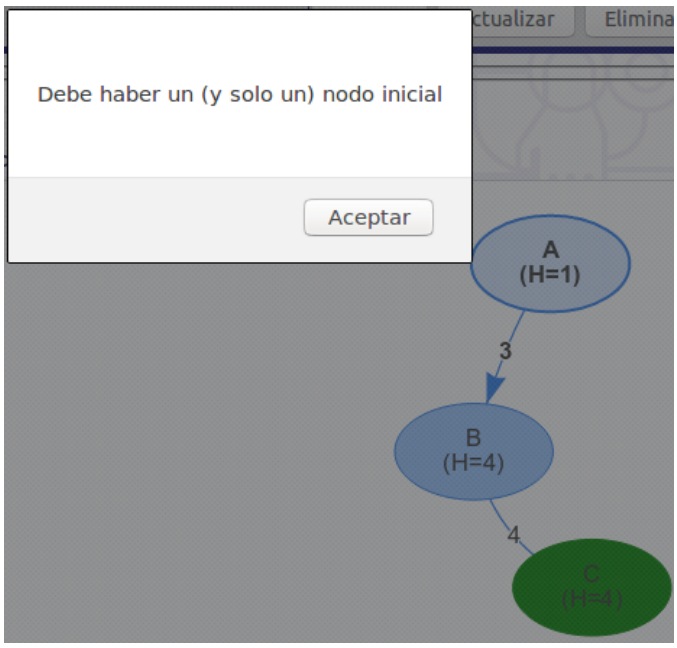


Figura 6.18: Exportación de un grafo sin nodo inicial.

Exportar un grafo sin nodos finales.

Se intenta exportar un grafo con nodo inicial, un nodo intermedio y aristas que los unen, pero ningún nodo final.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

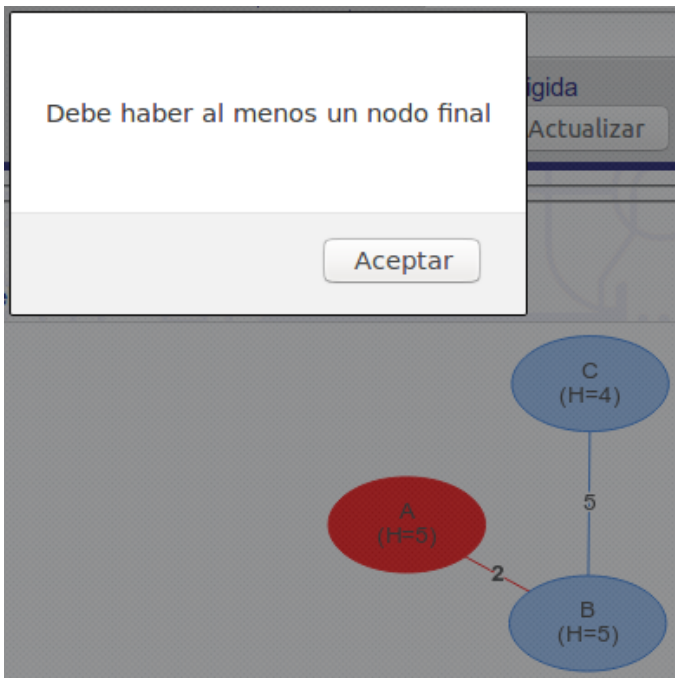


Figura 6.19: Exportación de un grafo sin nodos finales.

Exportar un grafo sin aristas.

Se intenta exportar un grafo con nodo inicial, un nodo intermedio, un nodo final sin aristas que los unan.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

Figura 6.20: Exportación de un grafo sin aristas.

Importar un grafo sin anomalías.

Se intenta importar un grafo, exportado previamente, sin manipular el archivo.

- **Resultado esperado:** Grafo importado.
- **Resultado obtenido:** Grafo importado.

La prueba tiene éxito.

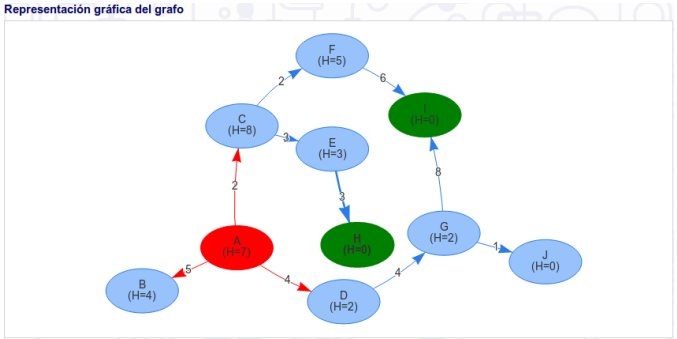


Figura 6.21: Importación de un grafo sin anomalías.

Importar un grafo manipulando los nodos.

Se trata de importar un grafo manipulando el primer bloque del archivo correspondiente

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

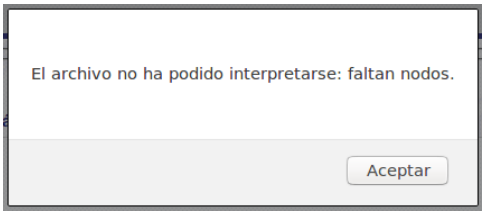


Figura 6.22: Importación quitando uno o varios nodos.

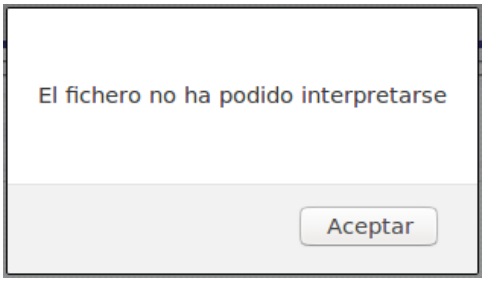


Figura 6.23: Importación quitando aleatoriamente un trozo del primer bloque.

Importar un grafo manipulando las aristas.

Se trata de importar un grafo manipulando el segundo bloque del archivo correspondiente

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

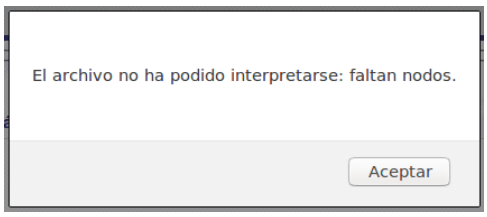


Figura 6.24: Importación quitando una o varias aristas.

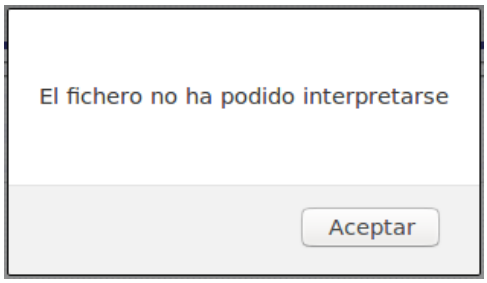


Figura 6.25: Importación quitando aleatoriamente un trozo del segundo bloque.

Importar un grafo manipulando el nodo inicial.

Se trata de importar un grafo manipulando el tercer bloque del archivo correspondiente

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

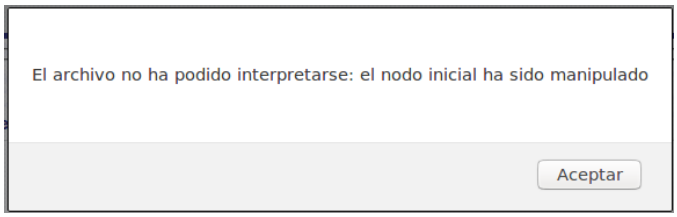


Figura 6.26: Importación quitando el nodo inicial o cambiando su identificador.

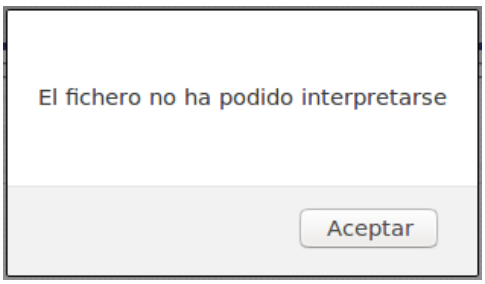


Figura 6.27: Importación quitando aleatoriamente un trozo del tercer bloque.

Importar un grafo manipulando los nodos finales.

Se trata de importar un grafo manipulando el cuarto bloque del archivo correspondiente

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

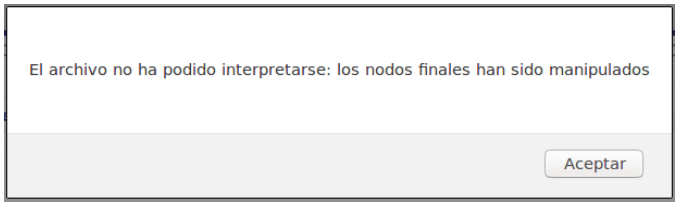


Figura 6.28: Importación quitando el uno o varios nodos finales.

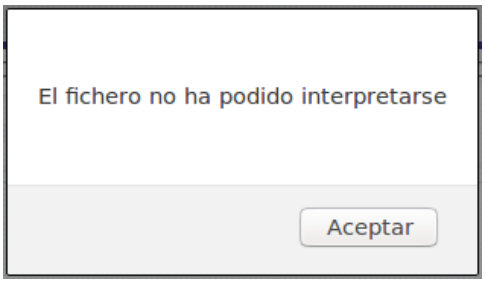


Figura 6.29: Importación quitando aleatoriamente un trozo del cuarto bloque.

6.1.3. PCB módulo de edición de grafos

Las pruebas de caja blanca de este módulo incluyen las del módulo de creación de grafos.

Actualizar un nodo (inicial, final o intermedio) sin anomalías

Se trata de actualizar un nodo cualquiera sin anomalías.

- **Resultado esperado:** Actualización del nodo.
- **Resultado obtenido:** Actualización del nodo.

La prueba tiene éxito.

Actualizar dejando vacío el identificador del nodo

Se trata de actualizar un nodo dejando vacío el identificador del nodo.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

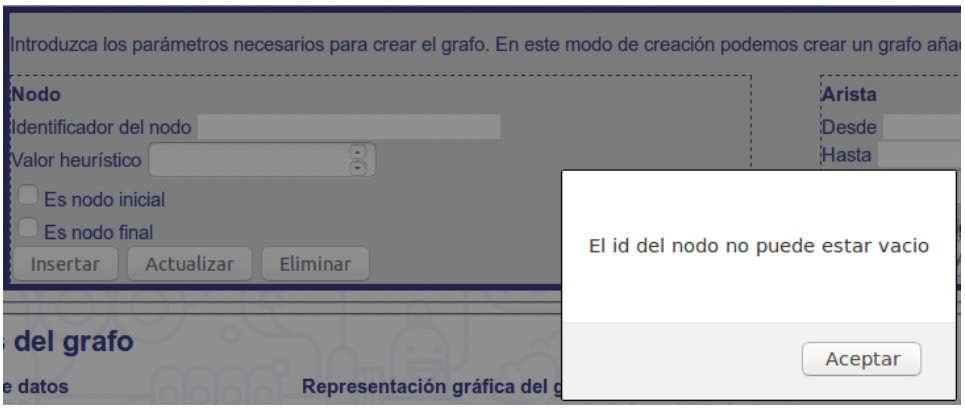


Figura 6.30: Actualización de nodo dejando vacío el campo identificador del nodo.

Actualizar un nodo cuyo identificado no existe

Se trata de actualizar, rellenando el campo identificador del nodo con un identificador que no existe.

- **Resultado esperado:** Error.

- **Resultado obtenido:** Error.

La prueba tiene éxito.

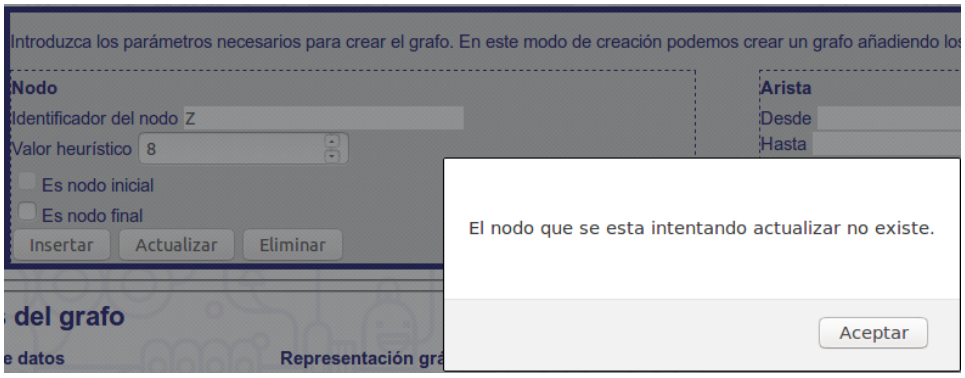


Figura 6.31: Actualización de nodo cuyo identificador no existe.

Actualizar un nodo con un valor heurístico vacío.

Se intenta actualizar un nodo cuyo valor heurístico esté vacío.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

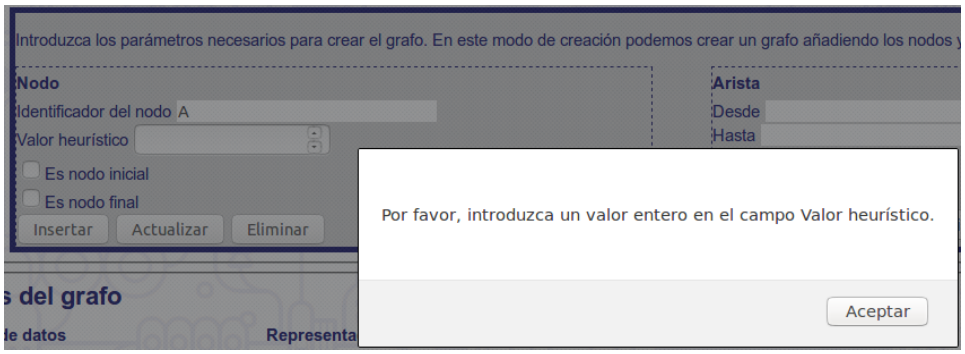


Figura 6.32: Error al actualizar un nodo dejando el valor heurístico vacío.

Actualizar un nodo con un valor heurístico no numérico.

Se intenta actualizar un nodo cuyo valor heurístico no sea numérico.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

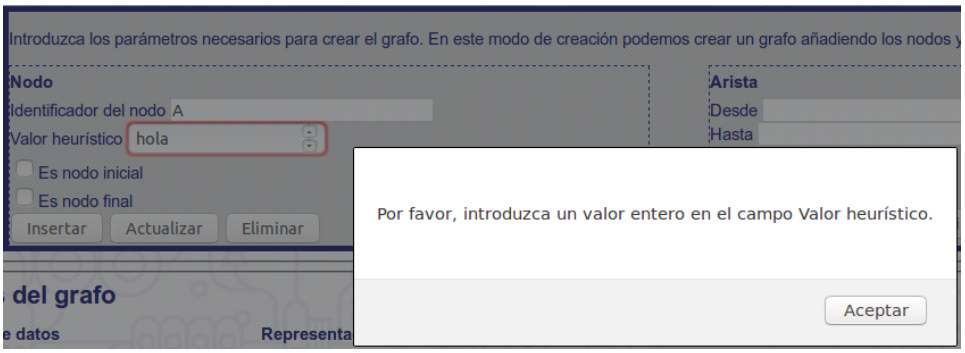


Figura 6.33: Error al actualizar un nodo con un valor heurístico no numérico.

Actualizar un nodo con un valor heurístico negativo.

Se intenta introducir un nodo cuyo valor heurístico sea negativo.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

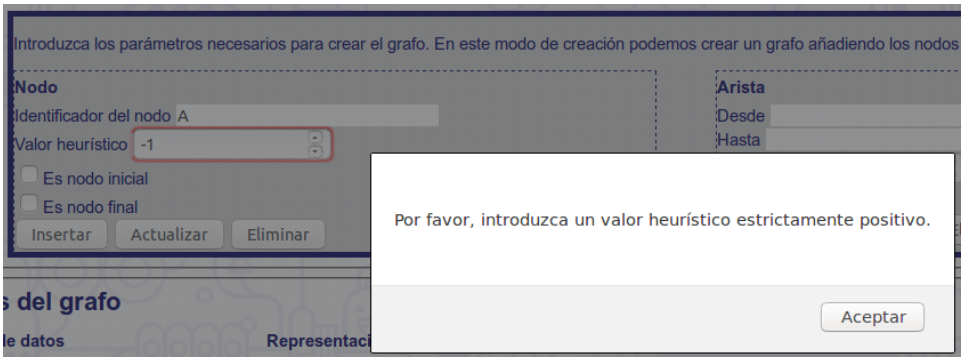


Figura 6.34: Error al actualizar un nodo con un valor heurístico negativo.

Actualizar un nodo final e inicial a la vez.

Se intenta actualizar un nodo marcando los checkbox de nodo inicial y final a la vez.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

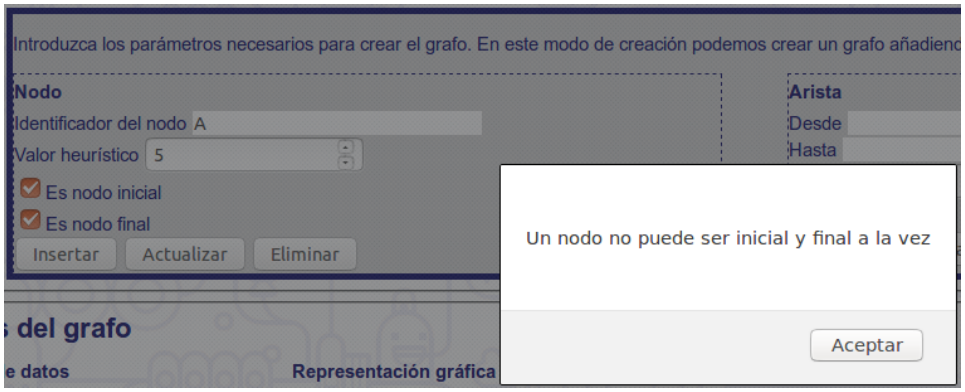


Figura 6.35: Error al actualizar un nodo marcando que es inicial y final a la vez.

Eliminar un nodo cuyo identificador no existe.

Se intenta eliminar un nodo cuyo identificador no existe.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

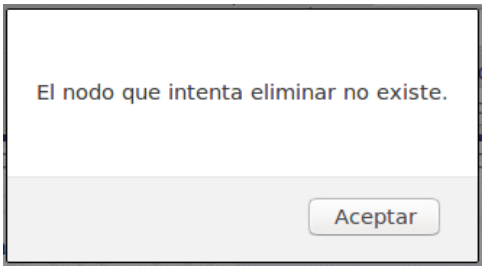


Figura 6.36: Error al eliminar un nodo cuyo identificador no existe.

Actualizar una arista (dirigida o bidireccional) sin anomalías

Se trata de actualizar una arista cualquiera sin anomalías.

- **Resultado esperado:** Actualización de la arista.
- **Resultado obtenido:** Actualización de la arista.

La prueba tiene éxito.

Actualizar una arista dejando vacíos ambos identificadores de nodo

Se trata de actualizar una arista dejando vacíos ambos identificadores de nodo.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

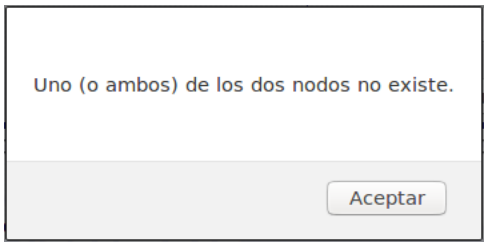


Figura 6.37: Actualización de aristas dejando vacíos los campos identificador de nodo.

Actualizar una arista dejando vacío un identificador de nodo

Se trata de actualizar una arista dejando vacío uno de los campos de identificador de nodo.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

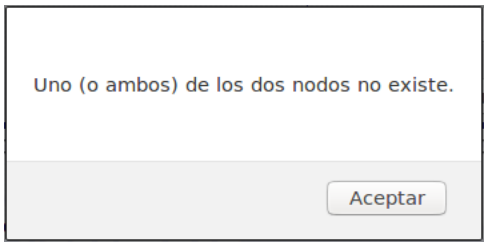


Figura 6.38: Actualización de aristas dejando vacío el campo superior de identificador de nodo.

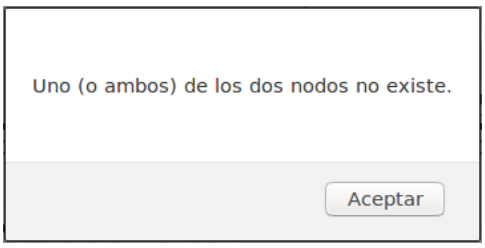


Figura 6.39: Actualización de aristas dejando vacío el campo inferior de identificador de nodo.

Actualizar una arista que no existe

Se trata de actualizar, rellenando los campos de identificador del nodo con identificadores que no existen o bien con identificadores que existen pero no están unidos por ninguna arista.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

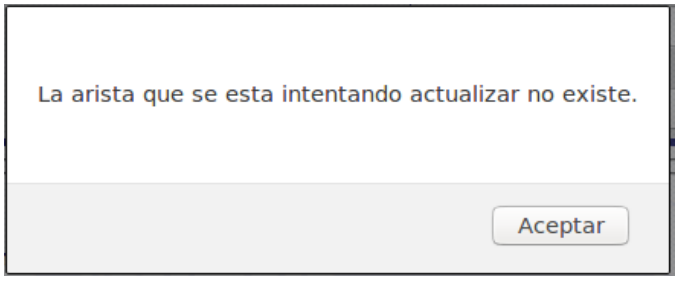


Figura 6.40: Actualización de arista que no existe.

Actualizar una arista con un valor de coste vacío.

Se intenta actualizar una arista cuyo valor de coste esté vacío.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

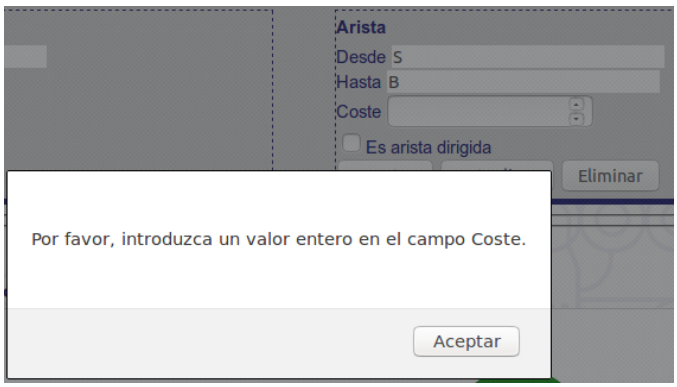


Figura 6.41: Error al actualizar una arista dejando el valor de coste vacío.

Actualizar una arista con un valor de coste no numérico.

Se intenta actualizar una arista cuyo valor de coste no sea numérico.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

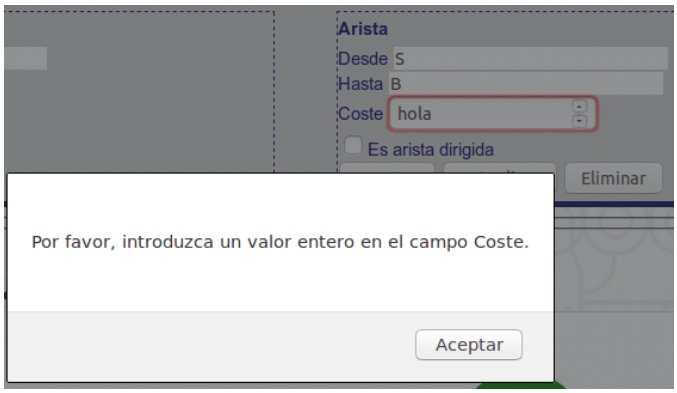


Figura 6.42: Error al actualizar una arista con un valor de coste no numérico.

Actualizar una arista con un valor de coste negativo.

Se intenta introducir una arista cuyo valor de coste sea negativo.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

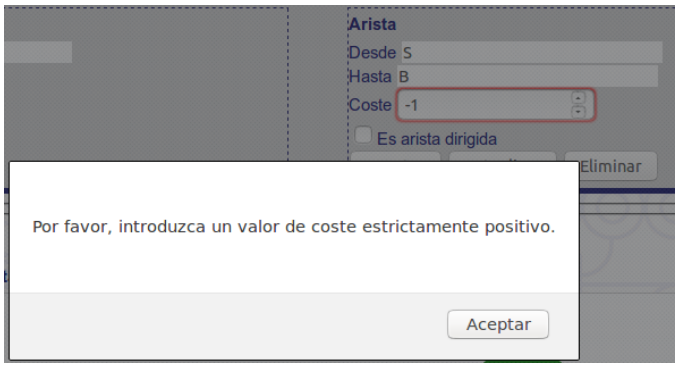


Figura 6.43: Error al actualizar una arista con un valor de coste negativo.

Eliminar una arista que no existe.

Se intenta eliminar una arista con identificadores de nodos inexistentes o identificadores existentes, pero no unidos por ninguna arista.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

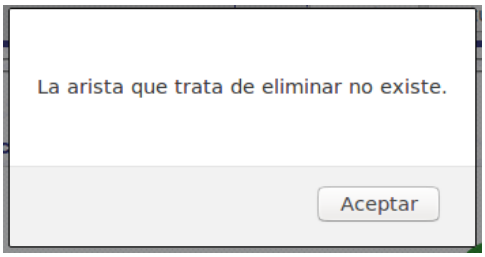


Figura 6.44: Error al eliminar una arista que no existe.

6.1.4. PCB módulo de generación de heurísticas

Para probar este módulo se debe estudiar previamente el grafo que estamos tratando. Para considerar unos parámetros razonables, se debe tener un conocimiento previo del grafo a resolver para saber qué rango de cambios o qué costes se tienen para alcanzar un nodo final desde cualquier nodo. Cuando hablamos de parámetros razonables, hablamos de parámetros para los que se espera obtener una respuesta satisfactoria.

Obtención de una heurística admisible.

Se intenta obtener una heurística admisible para un grafo, marcando el checkbox de heurística admisible.

- **Resultado esperado:** Heurística admisible.
- **Resultado obtenido:** Heurística admisible.

La prueba tiene éxito.

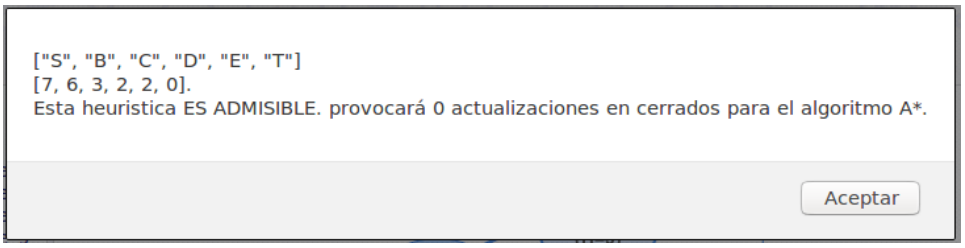


Figura 6.45: Obtención de la heurística admisible.

Obtención de una heurística que produzca cambios, con parámetros razonables.

Se intenta obtener una heurística que provoque ciertos cambios en cerrados para un grafo, rellenando el número de cambios de forma correcta.

- **Resultado esperado:** Heurística que fuerza cambios en cerrados.
- **Resultado obtenido:** Heurística que fuerza cambios en cerrados.

La prueba tiene éxito.

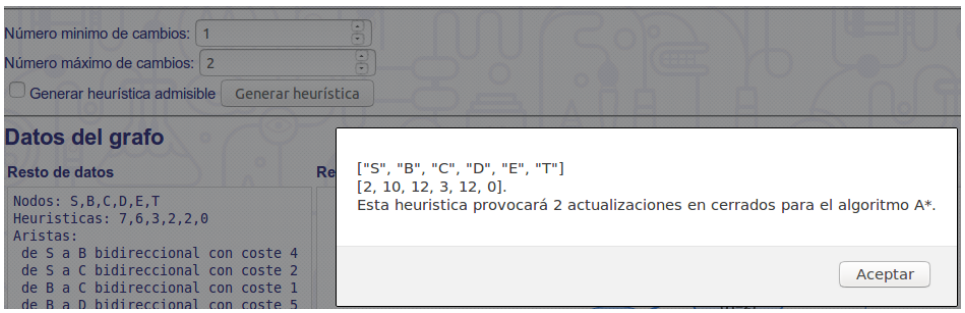


Figura 6.46: Obtención de la heurística que fuerza cambios.

Obtención de una heurística dejando el formulario completamente en blanco.

Se intenta obtener una heurística dejando en blanco por completo el formulario

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

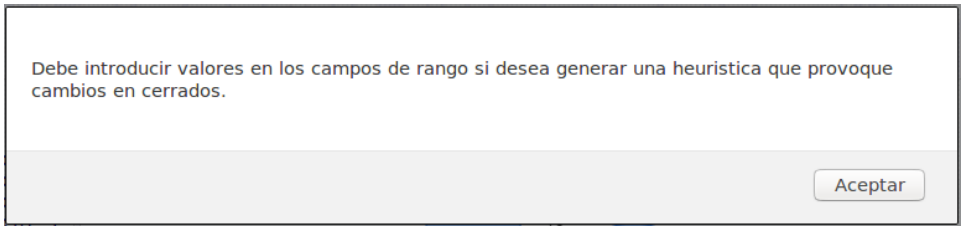


Figura 6.47: Error al obtener una heurística con el formulario en blanco.

Obtención de una heurística colocando valores negativos en los campos de número mínimo y máximo de cambios.

Se intenta obtener una heurística colocando valores negativos en los campos de rango de cambios.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

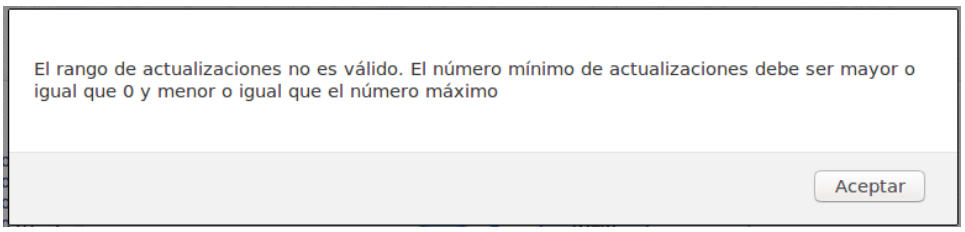


Figura 6.48: Error al obtener una heurística rellenando con valores negativos ambos campos, el campo superior o el inferior.

Obtención de una heurística colocando el número mínimo de cambios mayor que el número máximo.

Se intenta obtener una heurística rellenando el número mínimo de cambios con un valor numérico mayor que el número máximo de cambios.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

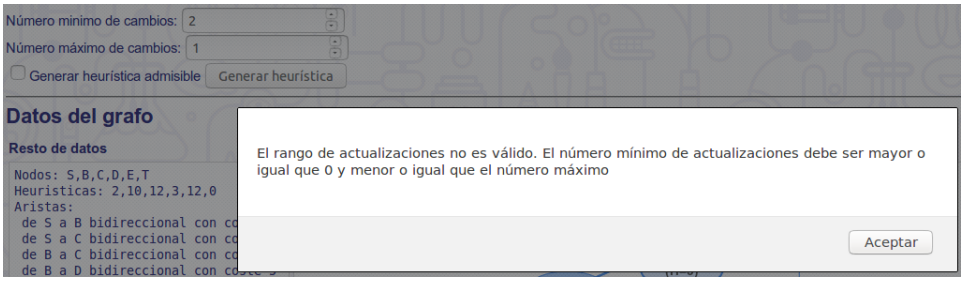


Figura 6.49: Error al obtener una heurística cuando el número mínimo de cambios es mayor que el número máximo.

Obtención de una heurística colocando valores no numéricos en el número mínimo y máximo de cambios.

Se intenta obtener una heurística rellenando el número mínimo y/o máximo de cambios con valores no numéricos.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

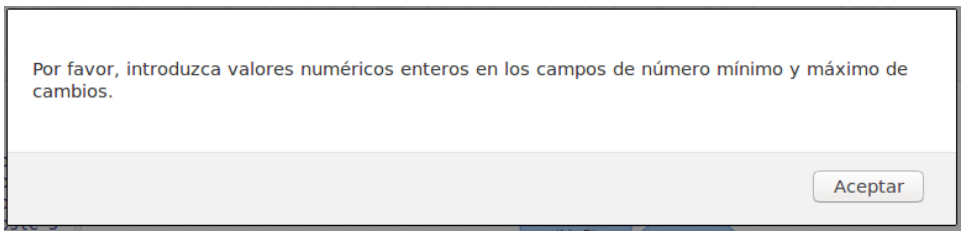


Figura 6.50: Error al obtener una heurística cuando el número mínimo de cambios es mayor que el número máximo.

Obtención de una heurística que provoque cambios con parámetros no razonables.

Se intenta obtener una heurística que provoque un número de cambios que viole el conocimiento previo que se tenía sobre el grafo.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

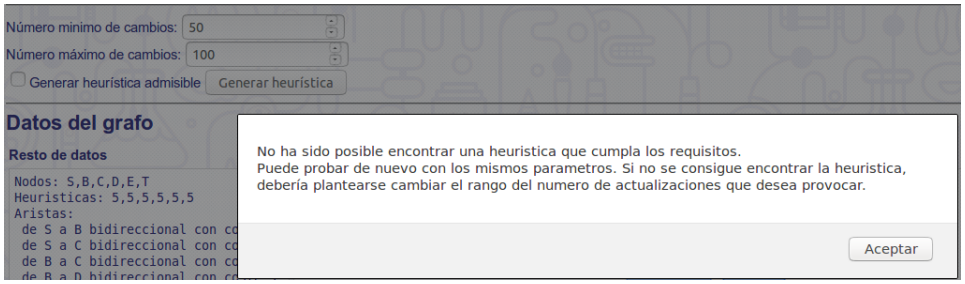


Figura 6.51: Error al obtener una heurística que provoque cambios en cerrados, violando el conocimiento previo.

6.1.5. PCB módulo de resolución del grafo

Resolución por diferentes algoritmos con número máximo de iteraciones correcto.

Se intenta resolver el algoritmo por los distintos algoritmos pasando un valor de máximas iteraciones correcto.

- **Resultado esperado:** Resolución del grafo.
- **Resultado obtenido:** Resolución del grafo.

La prueba tiene éxito.

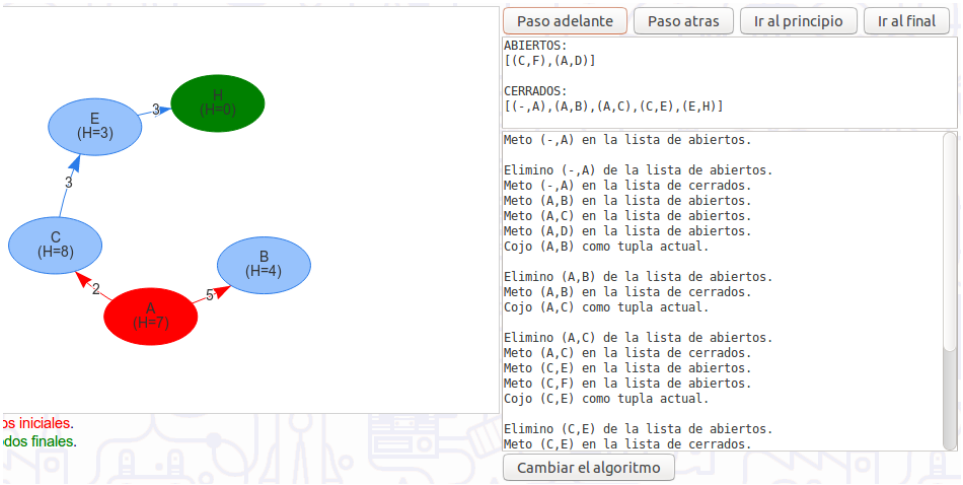


Figura 6.52: Paso a resolución del grafo.

Pasar a resolución dejando vacío el número máximo de iteraciones.

Se intenta resolver el algoritmo dejando vacío el número máximo de iteraciones.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

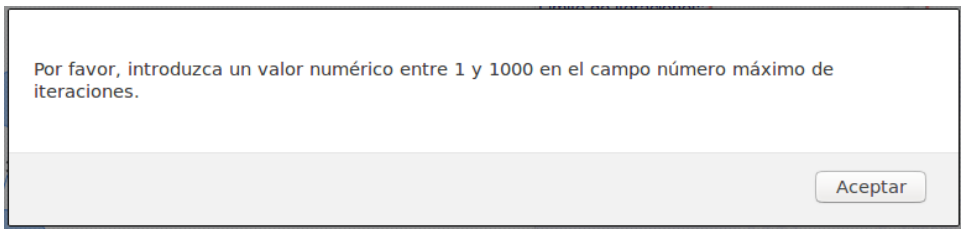


Figura 6.53: Error al pasar a resolución del grafo sin indicar número máximo de iteraciones.

Pasar a resolución pasando un valor no numérico en el número máximo de iteraciones.

Se intenta resolver el algoritmo pasando un valor no numérico en el número máximo de iteraciones.

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

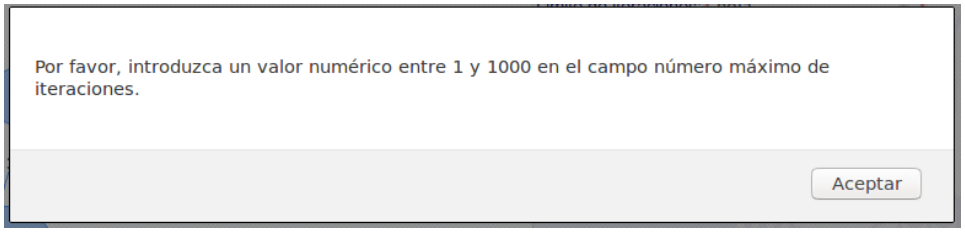


Figura 6.54: Error al pasar a resolución del grafo indicando un valor no numérico en número máximo de iteraciones.

Pasar a resolución pasando un valor que viole los límites en el número máximo de iteraciones.

Se intenta resolver el algoritmo pasando un valor que viole los límites entre 1 y 1000 en el número máximo de iteraciones. Se prueban los valores 0 y 1001. Ya se probaron valores dentro de este intervalo en la primera prueba de este bloque

- **Resultado esperado:** Error.
- **Resultado obtenido:** Error.

La prueba tiene éxito.

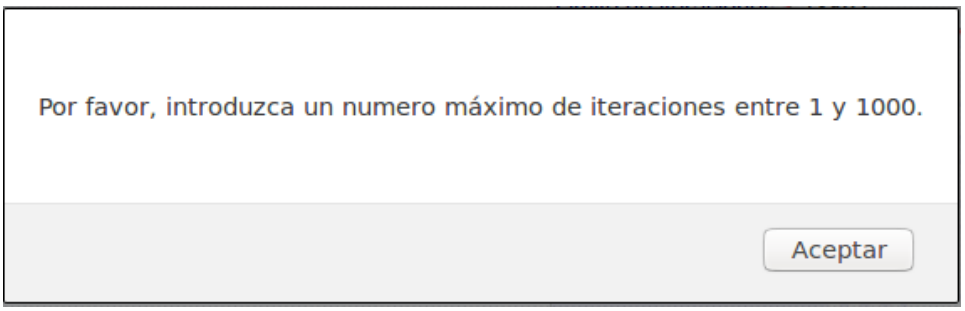


Figura 6.55: Error al pasar a resolución del grafo violando los valores límite de número de iteraciones.

Resolución de un grafo en estado resoluble por Búsqueda en profundidad.

- **Resultado esperado:** Resolución correcta.
- **Resultado obtenido:** Resolución correcta.

La prueba tiene éxito.

Resolución de un grafo en estado resoluble por Búsqueda en anchura.

- **Resultado esperado:** Resolución correcta.
- **Resultado obtenido:** Resolución correcta.

La prueba tiene éxito.

Resolución de un grafo en estado resoluble por Descenso iterativo.

- **Resultado esperado:** Resolución correcta.
- **Resultado obtenido:** Resolución correcta.

La prueba tiene éxito.

Resolución de un grafo en estado resoluble por A*.

- **Resultado esperado:** Resolución correcta.
- **Resultado obtenido:** Resolución correcta.

La prueba tiene éxito.

Resolución de un grafo en estado resoluble por Costo uniforme.

- **Resultado esperado:** Resolución correcta.
- **Resultado obtenido:** Resolución correcta.

La prueba tiene éxito.

Resolución de un grafo en estado resoluble por Búsqueda retroactiva.

- **Resultado esperado:** Resolución correcta.
- **Resultado obtenido:** Resolución correcta.

La prueba tiene éxito.

Resolución de un grafo en estado resoluble por Escalada simple.

- **Resultado esperado:** Resolución correcta.
- **Resultado obtenido:** Resolución correcta.

La prueba tiene éxito.

Resolución de un grafo en estado resoluble por Escalada por la máxima pendiente.

- **Resultado esperado:** Resolución correcta.
- **Resultado obtenido:** Resolución correcta.

La prueba tiene éxito.

Nota: Las pruebas realizadas con los algoritmos pueden presentar tres casuísticas:

- Que el grafo se haya editado tras entrar en el modo resolución y por tanto no se pueda resolver. En este caso se considera una resolución correcta por parte del algoritmo de búsqueda si se obtiene que no se ha podido encontrar un camino.
- Que el grafo, aún estando correcto y en estado resoluble, no sea posible encontrar un camino desde el nodo inicial hasta un nodo final. En este caso también se considera una resolución correcta por parte del algoritmo de búsqueda si se obtiene que no se ha podido encontrar un camino.

- Que el grafo esté correcto y en estado resoluble y se pueda obtener un camino desde el nodo inicial hasta un nodo final. En este caso se considera una resolución correcta por parte del algoritmo si se obtiene el camino encontrado y su coste.

Este criterio es el que se ha utilizado para evaluar las pruebas realizadas con los diferentes algoritmos de búsqueda. Aunque todas las pruebas realizadas han resultado satisfactorias, para garantizar el funcionamiento de todos se deberían probar todas las casuísticas posibles para todos los algoritmos, pero esto es inviable por la gran cantidad de casos que pueden presentarse.

6.1.6. PCB módulo de representación gráfica

Este módulo no requiere de pruebas específicas, pues el resto de módulos dependen de él y se ha probado satisfactoriamente de forma concurrente a los demás módulos.

6.2. Pruebas de caja negra

Las pruebas de caja negra tratan al sistema como una caja cuyo contenido no se conoce. No requiere, por tanto, un conocimiento específico del código fuente, de hecho, se recomienda que no se tenga este conocimiento. Es por ello que, en la mayoría de las ocasiones las pruebas de caja negra son realizadas por un equipo ajeno al que programó el sistema.

En este caso particular, dado que es un sistema de uso docente, se ha decidido proporcionar el sistema al alumnado de la asignatura Inteligencia Artificial y tratar de detectar errores así como obtener críticas y feedback. En esta fase fueron detectados algunos fallos relacionados con la facilidad de uso de la aplicación (intuitividad) y en la resolución de grafos que presentaban casuísticas muy concretas de algunos algoritmos. Tan pronto como fueron conocidos los fallos se procedió a su corrección. Estos fueron los fallos detectados:

- No se encontraban fácilmente los botones para pasar a los modos de generación de heurísticas, importación y exportación y resolución de grafos.
- Actualización incorrecta de nodos cerrados en los algoritmos A* y Costo Uniforme.
- Fallos en la propagación de actualizaciones en los algoritmos A* y Costo Uniforme, debido al punto anterior. Esto sólo afectaba al desarrollo del algoritmo, no a la resolución final.
- Algunas aristas bidireccionales no se dibujaban. Si existe una arista declarada como arista bidireccional de A a B, existe implícitamente otra arista de B a A del mismo coste, pero esto no se representaba correctamente.

Los errores concernientes a los algoritmos y al dibujado fueron solucionados adecuadamente. En el caso de los fallos referentes a la interfaz de usuario, las medidas tomadas para satisfacer la necesidad del usuario fueron rediseñar la interfaz y proporcionar un documento de instrucciones de uso.

Una vez los errores fueron corregidos, se volvió a proporcionar el software a los usuarios y no se reportaron más fallos.

Conclusiones

En este apartado voy a tratar de extraer enseñanzas de la experiencia adquirida desarrollando y aplicando la ingeniería del software a través este trabajo fin de grado. Ante todo, cabe destacar que la primera conclusión que deberíamos sacar es que, al finalizarlo, el trabajo realizado enriquece, de manera que salgo sabiendo un poco más que cuando empecé y que, se interprete como se interprete, esto **siempre** es positivo.

Algo común a todas las fases de desarrollo es que **todas** son necesarias y que, por muy bien que se traten de hacer las cosas, siempre pueden quedar cabos sueltos que sólo se atan con el paso del tiempo y la ganancia de experiencia.

7.1. Conclusiones extraídas de la fase de análisis y especificación de requisitos.

En primer lugar, con respecto al análisis de requisitos, es el primer contacto con la idea de la aplicación. Este proceso es esencial para finalizar de conocer la aplicación, porque, de hecho, debería siempre haber un proceso anterior que constase de un estudio en profundidad de lo que se pretende y espera de la misma, en el que respondiésemos a las preguntas ¿Qué se quiere conseguir con esta aplicación? (Conocer las necesidades del negocio y/o del cliente). ¿Qué debe hacer la aplicación al terminar mi trabajo? (Tener muy claro qué desea el cliente y si podemos proporcionárselo). De aquí obtendremos el conjunto de requisitos funcionales. Además es necesario dotar a la aplicación de propiedades deseables, como seguridad, eficacia, eficiencia... De aquí se extraen los requisitos no funcionales. El análisis es un proceso que lleva tiempo y, en ocasiones, requiere que el equipo de desarrollo se forme en ámbitos hasta ahora desconocidos para ellos. Aunque este no es el caso más radical, he tenido que documentarme en profundidad sobre algoritmos que conocía sólo de forma superficial y, sin este proceso, la aplicación jamás podría haber sido desarrollada.

La fase de especificación de requisitos se debería realizar una vez teniendo muy claro qué debe hacer la aplicación, donde se debe realizar una radiografía en profundidad del sistema, detectando subtareas y dividiendo, siempre que sea posible, en módulos que realicen tareas independientes. Aunque esta no sea quizás la forma definitiva en que se implementará la aplicación, sí nos da una idea sobre qué cosas deberán ser implementadas al final. Cada requisito debe ser detallado minuciosamente mediante un informe y ver qué subtareas debe realizar, qué posibles dependencias tendrán

estas, etcétera. Todos estos datos deben ser plasmados, si procede, en un diagrama de caso de uso por cada requisito.

Dado que este paso son los cimientos de la ingeniería que aplicaremos posteriormente, es de vital importancia que se emplee el tiempo necesario en que este proceso sea desarrollado para obtener los mejores resultados posibles. Como en toda ingeniería, se deberá volver a este punto ya que, en el fondo, seguiremos un procedimiento de prueba y error. Lo que se pretende es volver el menor número de veces posible.

7.2. Conclusiones extraídas de la fase de planificación.

La planificación es una fase importante cuando se realiza de forma correcta. En mi caso no fué calculada con precisión. En esta fase se tiende a subestimar el esfuerzo necesario para llevar a cabo las diferentes tareas necesarias y a ignorar que puedan surgir problemas imprevistos que mermen la velocidad de desarrollo.

Si nos fijamos en la planificación realizada a priori, era muy optimista, pues tardaría en desarrollar el trabajo alrededor de mes y medio. No conté con imprevistos que podrían surgir de desconocer los lenguajes de desarrollo y tratar de aprenderlos en poco tiempo, además el desarrollo de los algoritmos no resultó sencillo, por tener que mezclar la definición teórica con las necesidades gráficas de la aplicación. A pesar de que estos imprevistos fueron resueltos, el tiempo que no se tuvo en cuenta, de cara a un cliente hubiera tenido un impacto muy negativo, pues la aplicación se hubiese retrasado más de un mes con respecto a su fecha de entrega inicial.

Como conclusión y enseñanza de esta fase, me quedo con que es tan malo subestimar la planificación como sobreestimarla, pero siempre es preferible contar con un plazo amplio de tiempo, máxime cuando es autoimpuesto, que tener que solicitar una ampliación o, peor aún, comunicarle al cliente que la aplicación se retrasa por causas ajenas a él.

7.3. Conclusiones extraídas de la fase de diseño.

En la fase de diseño partimos del trabajo realizado en análisis y especificación de requisitos. Debemos analizar si es factible implementar la aplicación como se había planificado en este apartado o bien podemos mejorar en algún aspecto el desarrollo. Este paso es crucial porque de que trabajemos bien aquí dependen muchos de los requisitos no funcionales. Por ejemplo, la escalabilidad de la aplicación depende de forma muy fuerte de que la estructura del proyecto sea la más adecuada. De entre todas las fases, la de diseño es la más importante a mi juicio, porque nos conduce en el siguiente paso a la construcción del producto y, si el diseño es defectuoso, el producto

será defectuoso y todo el trabajo habrá sido en vano, teniendo que volver a repetirlo.

7.4. Conclusiones extraídas de la fase de implementación.

A pesar de que a veces pensamos que el proceso de implementación es el más importante, normalmente no estamos en lo cierto. En una situación ideal, la fase de implementación debería ser trivial si hemos aplicado una buena ingeniería del software. Esto no siempre es así y por eso el proceso de implementación es relevante, porque sólo así nos damos cuenta de cosas que habíamos pasado por alto o que son inviables en la realidad. Si bien es cierto que el proceso de implementación no es el más importante, dado que normalmente no nos encontramos en situaciones ideales, no podemos decir en absoluto, que este proceso sea trivial o irrelevante.

7.5. Conclusiones extraídas de la fase de pruebas.

Esta última fase junto con la de diseño son las más importantes. Si se hizo un diseño de calidad, las pruebas sólo fallarán en pequeñas cosas producidas por la implementación en la mayoría de los casos. Si no se hizo un buen diseño, las pruebas pueden arrojar fallos estructurales graves que puedan acabar en el peor de los casos en un rediseño, reestructuración y (casi) implementar de nuevo el proyecto. Sacar un producto software al mercado sin realizar este paso podría suponer, casi inmediatamente, la quiebra de la empresa que lo hiciese, pues prácticamente ningún software en esta fase está libre de errores.

Sólo probando el producto desarrollado podremos detectar errores que, tras su corrección, se puede empezar a pensar que la aplicación va por buen camino. Las pruebas, además de ir enfocadas a ver que el producto hace lo que debe hacer, deberían concentrarse en comprobar que es robusto a fallos y que no genera fácilmente errores críticos. Este es uno de los requisitos no funcionales que deben especificarse siempre para garantizar un mínimo de calidad. Una fase de pruebas satisfactoria no indica que la aplicación esté libre de errores por completo pues posteriormente a la fase de pruebas (verificación) pasaremos a la fase de validación que normalmente, realiza el cliente, inspeccionando a fondo la aplicación para determinar si la aplicación cumple con lo detallado en la especificación de requisitos.

7.6. Trabajos futuros.

Dado que esta aplicación es un trabajo fin de grado, es susceptible a modificaciones, expansiones y mejoras. Se podría incluir funcionalidad nueva en el futuro que la hiciese más completa, eficiente e intuitiva. Algunas de las modificaciones que podrían llevarse a cabo en el futuro son:

- Inclusión de nuevos algoritmos de búsqueda informada. La estructura de clases elegida facilita la escalabilidad de la aplicación, por lo que incluir un nuevo algoritmo previamente desarrollado sería un proceso bastante sencillo.
- Reutilización del software para incluir algoritmos de juegos con adversario, como por ejemplo minimax con o sin poda α/β.
- Llevar las operaciones al lado del servidor y paralelizarlas. Esta modificación permitiría darle el trabajo computacional al servidor, con la consecuencia de que los cálculos se lleven a cabo de forma más rápida, teniendo en cuenta las capacidades de un servidor con respecto a una máquina cliente media. La paralelización de los cálculos permitiría escalar las posibilidades del sistema, pudiendo resolver cada vez grafos más complejos y más numerosos, teniendo en cuenta que la eficiencia de los algoritmos de búsqueda depende del tamaño de los grafos.
- Dotar a la aplicación de un diseño web adaptativo. Aunque la aplicación puede usarse en cualquier dispositivo, no se garantiza que sea cómodo hacerlo por ejemplo en un smartphone, por culpa del diseño. Si se consiguiera dotar a la aplicación de un diseño responsive, sería cómodo usarlo desde cualquier dispositivo y no sólo desde un PC.

Bibliografía

[1] B. Boehm, A. Egyed, J. Kwan, D. Port, A. Shah, and R. Madachy. Using the winwin spiral model: A case study. *Computer*, 31:33 – 44, 7 1998.

[2] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21:61 – 72, 5 1988.

[3] Almende B.V. Vis.js: A dynamic, browser-based visualization library. `http://visjs.org/`.

[4] David L. Applegate; Robert E. Bixby; Vasek Chvátal; William J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Cambridge University Press, second edition, 2007.

[5] Jon Duckett. *Web Design with HTML, CSS, JavaScript and jQuery Set*. Wiley, first edition, 2014.

[6] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100 – 107, 7 1968.

[7] Nils J. Nilsson. *Artificial Intelligence: A new synthesis*. Morgan Kaufmann Publishers, first edition, 1998.

[8] Massachusetts Institute of Technology. The dot language. `http://web.mit.edu/spin_v4.2.5/share/graphviz/doc/html/info/lang.html`.

[9] Roger S. Pressman. *Ingeniería del Software: Un enfoque práctico*. McGraw-Hill Education, seventh edition, 2009.

[10] P. Norvig S. J. Russell. *Artificial Intelligence: A Modern Approach*. Pearson Prentice-Hall, second edition, 2003.

[11] Boaz Barak Sanjeev Arora. *Computational Complexity: A Modern Approach*. Cambridge University Press, first edition, 2009.

[12] Ian Sommerville. *Software Engineering*. Pearson, ninth edition, 2010.