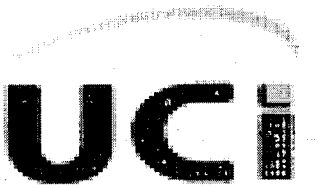
UNIVERSIDAD DE LAS CIENCIAS

INFORMÁTICAS

FACULTAD 5 Entornos Virtuales



Trabajo de Diploma para optar por el Título de

Ingeniero en Ciencias Informáticas

Búsqueda de caminos en entornos virtuales

Autor: Dalila Oconor Labañino

Tutor: MsC. Yuniesky Coca Bergolla

Ciudad de La Habana

Julio 2008

Dedicatoria:

A mi mamá por desearlo tanto como yo, por su gran sacrificio en todo momento para que yo llegara donde estoy.

Resumen:

Una de las problemáticas más vistas en el mundo de los juegos es cómo encontrar el camino mínimo entre dos puntos, este estudio ha comenzado desde los métodos de búsqueda heurística tradicionales que en la mayoría de los casos son incapaces de resolver problemas donde el agente tiene un tiempo limitado para calcular una solución en un entorno inicialmente desconocido hasta los métodos de búsqueda heurística en tiempo real.

Obtener los movimientos necesarios para que un agente pueda alcanzar un objetivo de forma autónoma en un ambiente no estructurado y evitando obstáculos en el camino es esencial en agentes móviles que operan en circunstancias adversas. Alcanzar una solución no es sencillo debido a que existen diversos aspectos que dificultan estas tareas, como lo son, lidiar con un ambiente dinámico, restricciones al movimiento del agente, entre otras.

El presente trabajo está enfocado principalmente a proponer el algoritmo adecuado capaz de solucionar problemas de búsqueda de caminos de manera eficiente, mediante algoritmos de búsqueda heurística en tiempo real. Se presentan varios algoritmos que mejoran el rendimiento de las aproximaciones existentes. Estos se evalúan considerando varias medidas de desempeño, las más importantes son: el coste de la solución, el tiempo total de búsqueda y el tiempo por etapa de planificación.

Palabras clave:

Entornos virtuales, realidad virtual, Inteligencia Artificial, tiempo real, Pathfinding.

Índice:

Índice de figuras.

Introducción:

La vertiginosa evolución de los video-juegos ha llevado a que la Inteligencia Artificial (IA) constituya uno de los aspectos más importantes en el desarrollo de los mismos; es fundamental que los agentes (entidades autónomas) controlados por la computadora se comporten de manera inteligente. Un problema característico es la búsqueda de caminos, que consiste en determinar el camino más conveniente entre una posición inicial y una posición de destino. Si bien el planteo del problema es sencillo, el mismo está lejos de ser ligero debido a la creciente complejidad de los entornos virtuales y los requerimientos de tiempo real de los juegos modernos [Nareyek, 2004].

La búsqueda es una de las técnicas más utilizadas para resolver los problemas de Pathfinding que se presentan en la Inteligencia Artificial de los video-juegos.

De los distintos tipos de algoritmos de búsqueda, los algoritmos de búsqueda heurística completa se encuentran ampliamente difundidos. Sin dudas, el algoritmo A* es el algoritmo de búsqueda heurística más popular.

Los métodos de búsqueda heurística disponen de alguna información sobre la proximidad de cada estado a un estado objetivo, lo que permite explorar en primer lugar los caminos más prometedores.

Los algoritmos heurísticos tradicionales muestran limitaciones importantes cuando el espacio de búsqueda es demasiado grande o existen factores dinámicos. En la búsqueda de caminos de los video-juegos en tiempo real, este problema aparece cuando las rutas a determinar son muy largas, el terreno es modificable o existen muchos objetos móviles. Bajo esas condiciones, los algoritmos de búsqueda clásicos no pueden responder en el tiempo requerido y resultan inadecuados. De esta forma, surge la necesidad de desarrollar nuevas estrategias de búsqueda que se adapten a los requerimientos de tiempo real de los video-juegos y resuelvan adecuadamente caminos en condiciones de incertidumbre sobre terrenos de gran extensión.

La motivación de esta tesis es la resolución de problemas reales de búsqueda de rutas, que no se pueden resolver con los métodos clásicos. En concreto la resolución de problemas en los que la búsqueda está sujeta a restricciones de tiempo en un entorno inicialmente desconocido por el agente. Para esto se proponen un conjunto de algoritmos de búsqueda heurística en tiempo real que se presentan a lo largo de este trabajo.

Un error que con frecuencia cometemos es que a la hora de hablar de los algoritmos de búsqueda heurística en tiempo real se asocia el término "tiempo real" a "rápido". Si se pretende obtener una búsqueda rápida, el

objetivo es minimizar la medida del tiempo de respuesta de cada tarea pero en los algoritmos de búsqueda heurística en tiempo real, el objetivo es que se cumplan las restricciones temporales de cada tarea. Más que ser rápidos, la propiedad más importante que debe cumplir es la de ser predecible, su comportamiento funcional así como temporal debe estar lo suficientemente determinado para poder asegurar que cumplirá las especificaciones del sistema. Lógicamente el hecho de ser rápidos les ayuda a cumplir las especificaciones temporales, pero esto por sí solo no garantiza que sean predecibles. Se llama predecibilidad temporal a la capacidad de predecir a priori que una tarea acabará antes de su plazo máximo de ejecución.

Los video-juegos son un campo reciente y por tanto, conllevan una simulación inteligente mucho más real y necesitan algoritmos más fuertes y de un nivel superior.

En la Universidad de las Ciencias Informáticas se está desarrollando una biblioteca genérica de Inteligencia Artificial para ser utilizada en diferentes aplicaciones, como juegos y simuladores, en la cual no se han implementado técnicas importantes para la búsqueda de caminos.

Para el desarrollo de la Inteligencia Artificial en simulaciones virtuales se han desarrollado varios tipos de modelos cognitivos que permiten a la computadora entender el entorno en que se encuentra. Sobre cada uno de ellos, es muy importante tener un poderoso algoritmo para la búsqueda de caminos, generalmente se implementa el algoritmo para cada tipo de modelo, algo que puede ser costoso y poco práctico.

Para lograr incorporar a esta biblioteca los mejores algoritmos de búsquedas de caminos se necesita una investigación de todos los que se han implementado en aplicaciones afines a las que se realizarían con dicha biblioteca.

Esta investigación está motivada por el siguiente **problema científico**: ¿Cómo encontrar algoritmos de búsqueda de caminos adecuados para ser integrados a la biblioteca de Inteligencia Artificial? Y el **objeto de estudio** son los algoritmos de búsqueda de caminos, se definió como **campo de acción** los algoritmos de búsqueda de caminos eficientes para ser aplicados a entornos virtuales. Como **objetivo general de la investigación** proponer algoritmos de búsqueda de caminos adecuados para ser incorporados a la biblioteca de Inteligencia Artificial de la facultad 5.

Para dar cumplimiento a los objetivos planteados se hace necesario seguir las tareas expuestas a continuación.

Tareas específicas:

_Hacer un estudio profundo de los algoritmos de búsqueda de caminos.

_Hacer alguna propuesta de algoritmos adecuados para incorporar a la biblioteca de IA.

_ Elaborar un documento de consulta con la bibliografía más actualizada sobre el tema.

_Desarrollar alguna variante de los algoritmos de búsqueda analizados en una aplicación demostrativa.

Esperando obtener los siguientes resultados:

_ Propuesta de algoritmos para ser incorporados a la biblioteca de IA.

_Aplicación demostrativa con alguna variante para demostrar su funcionamiento.

_Bibliografía actualizada sobre estrategias de búsqueda de caminos.

Para darle cumplimiento a las tareas se guiará la investigación en los marcos de los métodos científicos de investigación "teóricos y empíricos". Dentro de los métodos teóricos se utilizará el método "Analítico-sintético", con el mismo se podrá analizar cada uno de los elementos de manera independiente, posibilitando una mayor capacidad de comprensión y de síntesis sobre los aspectos más importantes. Por su parte la utilización del método "Análisis histórico-lógico" brindará la posibilidad de analizar toda la evolución del problema que se estará estudiando. Uno de los métodos empíricos que será referenciado es la "Observación" este método permite adquirir información necesaria y puede utilizarse en cualquiera de las fases de la investigación, además ofrece un gran acercamiento a la realidad y permite ver la posible solución del problema desde diferentes ángulos.

El desarrollo de esta investigación estará organizado de la siguiente manera:

En el primer capítulo, **Fundamentación Teórica**, es aquí donde se hace un análisis detallado y se describen las características de algunos de los modelos cognitivos más utilizados en el mundo de los juegos, además se hace referencia a las características de los algoritmos de búsqueda de caminos tradicionales. En el segundo capítulo se tratará las características de los algoritmos posteriores al A* con búsqueda heurística hasta llegar a los de búsqueda heurística en tiempo real. En el tercer y último capítulo es donde se escoge el algoritmo a incorporar

en la biblioteca de Inteligencia Artificial, además se hace una descripción completa del demo realizado, también se detallan el lenguaje y las herramientas con las que se concibió dicho demo.

Capítulo1: Fundamentación Teórica.

1.1 Introducción

Los algoritmos de búsqueda de caminos son usados para encontrar una ruta de navegación entre un punto de inicio y otro objetivo. Esta ruta es generalmente un pequeño subconjunto del mundo. Es por esta razón que en este capítulo se realizará una introducción a los aspectos del tema de la tesis, además de una exhaustiva explicación de las características específicas de algunos de los modelos cognitivos más utilizados en el mundo de los juegos, así como, de algoritmos de búsqueda de caminos tradicionales que servirán de fundamentación teórica a la presente investigación y además ayudarán a entender el desencadenamiento del siguiente trabajo.

1.2 Breve reseña histórica

Con el gran avance que ha tenido el conocimiento del hombre a través del devenir de los años hay momentos que marcan pautas en el desarrollo de la Inteligencia Artificial, fue en la década del 50 donde se utilizó este término y logró captar la atención de muchos para su posterior evolución. La Inteligencia Artificial atiende una rama muy importante y ésta es la búsqueda de caminos.

Las raíces de los algoritmos de búsqueda de caminos hay que asociarlos a la constante evolución de los video-juegos ya que es una de las aplicaciones prácticas más interesantes en este entorno, Son algoritmos por los cuales nuestro objetivo es capaz de eludir los obstáculos y llegar de un punto A a un punto B.

Es a partir de 1959 donde se empieza a buscar variantes para la problemática de la búsqueda de caminos en video-juegos, este proceso ha sido verdaderamente evolutivo ya que se comenzó con algoritmos como Dijkstra, también llamado algoritmo de caminos mínimos luego siguió el algoritmo de búsqueda DFID (Depth-First Iterative-Deepening) éste fue introducido por primera vez en el año 1977 cuando Slate y Atkin presentaron un programa de ajedrez.

Con la constante optimización de los algoritmos de búsqueda aparece el algoritmo de búsqueda de caminos A*, éste fue presentado por primera vez en 1968 por Peter E. Hart, Nils J. Nilsson y Bertram Raphael, A* es un algoritmo de búsqueda heurística y es óptimo siempre y cuando no se sobreestime la heurística.

Más tarde aparecen nuevas variantes derivadas del A* con el fin de lograr mayor eficiencia como es el caso de El IDA* (Iterative-Deepening A*), El SMA* (Simplified Memory-bounded A*), Learning Real Time A*.

Hacia finales de los años 80´ se comenzó a utilizar conceptos de la búsqueda aplicada a juegos de dos jugadores en el desarrollo de estrategias para resolver problemas de pathfinding de un único agente. LRTA* surge en 1990 por Richard Korf, el objetivo de este algoritmo era dar respuesta en tiempo real como su nombre lo indica y es ampliamente utilizado en la simulación y animación, en el mismo año sale a la luz el RTA* que es también un método de búsqueda heurística en tiempo real y surge con el objetivo de mejorar algunas deficiencias del LRTA*, luego de éste surgieron otros tantos como el HLRTA*, seguido del FALCONS y por ultimo el eFALCONS todos con un mismo objetivo que era optimizar al LRTA*.

En Cuba comienza a lograrse un avance en este sentido, los video-juegos y simuladores son un campo reciente y por tanto, conllevan una simulación inteligente mucho más real y necesitan algoritmos más fuertes y de un nivel superior. En estos momentos se desarrolla en la Universidad de las Ciencias Informáticas un módulo genérico de búsqueda de caminos con el fin de lograr un paso más en este sentido.

1.3 Modelos Cognitivos

Para el caso de los agentes de la Inteligencia Artificial (BOTS), se necesita de un modelo cognitivo [Stout, 2000] para visualizar el mundo y poder tener una representación de como está estructurado éste para poder moverse dentro de él.

Sin el modelo cognitivo, el BOT o agente no podría tener claros varios aspectos de la representación del entorno virtual.

El uso de esta técnica reduce los espacios contiguos del terreno a espacios más discretos que los algoritmos de búsqueda puedan interpretar. Como el espacio de búsqueda contiene un número de nodos y aristas, cada nodo representa una localización en el mundo y cada arista representa un camino entre un par de nodos.

Usualmente el modelo cognitivo se usa para que el BOT tenga una versión simplificada del entorno virtual. Si el BOT viviera en el mundo real entonces no sería factible crear un modelo perfecto porque el mundo real es muy complicado.

No siempre un modelo cognitivo es el más adecuado a usarse en un entorno virtual, en algunas ocasiones hay que usar otros modelos para lograr una representación adecuada para el BOT.

En este modelo se incluyen solamente los detalles necesarios como los obstáculos geométricos en el entorno, o sea, muros, aguas o cualquier otro terreno intransitable.

1.4 Tipos de Modelos Cognitivos

En el mundo de los juegos existen muchos tipos de modelos cognitivos que se utilizan frecuentemente, a continuación se ofrece algunos ejemplos de los más usados:

1.4.1 Rejilla Regular

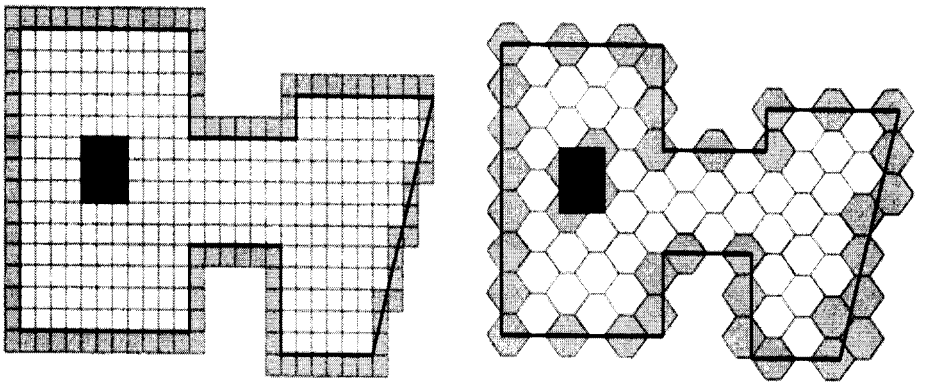


Figura 1: Representación de una rejilla basada en celdas cuadradas y hexagonales.

La rejilla regular [Board, 2002] se utiliza mucho en juegos con sistemas en tiempo real, a veces tiene un complejo ambiente basado en cuadrados o polígonos (triángulos, pentágonos o hexágonos). Esto por consiguiente es racional en el diseño de grafos de navegación.

Este particular método reduce nuestra área de búsqueda a una simple matriz bidimensional. Cada elemento de la matriz representa uno de los cuadrados de la rejilla, y su estado se almacena como transitable o intransitable.

Las rejillas son más útiles en ambientes 2D, no podrían ser usadas en juegos 3D sin alguna modificación.

El valor presente en cada celda representa árboles, murallas, enemigos, el jugador, casas, terrenos entre otras cosas. Por estos valores la Inteligencia Artificial puede planear donde moverse, donde localizar enemigos o cómo evitar obstáculos.

Es importante resaltar que la IA no necesita conocer el color o el tipo de la superficie sino solamente la información que le brindan las celdas, información inteligente que no tiene que contarle las características visuales.

Aún cuando el modelo cognitivo de rejillas es muy simple de usar e implementar, tiene una desventaja y radica en que la búsqueda de espacios puede resultar extremadamente larga. Para un modesto mapa de celdas.

Dado que los juegos con sistemas en tiempo real conllevan regularmente cientos y miles de unidades inteligentes activas cada tiempo, haciendo que haya una demanda de grafos de búsqueda cada vez que se actualiza un paso, creándose un alto procesamiento de datos sin mencionar la cantidad de memoria necesitada para hacerlo.

Por fortuna existen una serie de métodos disponibles para facilitar esta carga.

1.4.2 Puntos de Visión

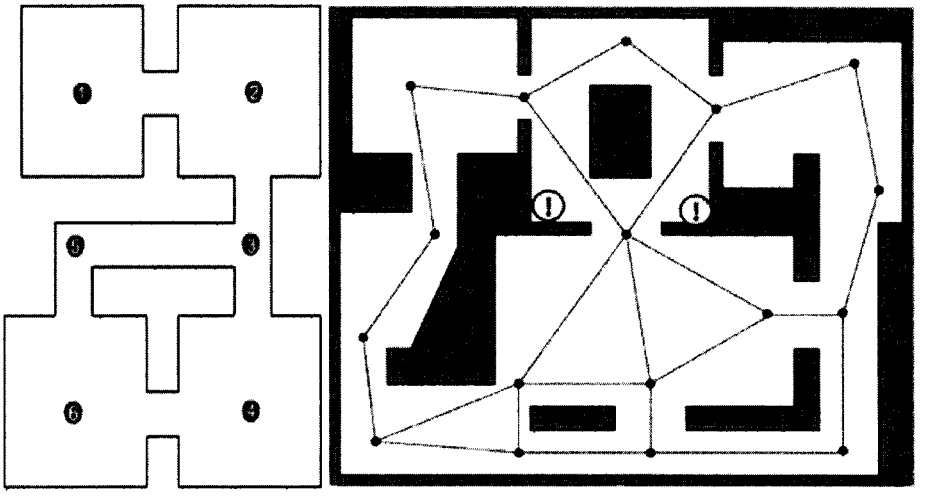


Figura 2: Representación gráfica de cómo ubicar los puntos en un mapa

 --- (puntos ciegos)

Comúnmente, lo usado para los propósitos de la navegación y búsqueda de caminos en los entornos 3D es un sistema de puntos viables (waypoints) [Rabin, 2000].

Básicamente, los waypoints son puntos que colocamos en un mapa que indican lugares a los que podemos ir. Estos waypoints están conectados entre sí de forma tal que forman un grafo. Para los propósitos de navegación las interconexiones entre los nodos tienen propiedades específicas. La propiedad más importante es que se puede viajar con facilidad desde un nodo a otro si están interconectados.

Los waypoint se caracterizan por describir el terreno accesible y los caminos por parte de los actores que interactúan con éste. Describen el movimiento válido y las propiedades del entorno local como las puertas, ventanas, agua, relieve, túneles, tráfico, entre otras.

Además estos se usan para predecir y soportar la búsqueda de pistas, donde a nivel de waypoints se permite encontrar los vecinos cercanos a un punto determinado, buscar el tiempo de viaje hacia los puntos cercanos a un punto en particular y que puntos se pueden ver desde el lugar donde se encuentre parado.

Los grafos entre mayor número de waypoint tengan mayor será la representación del mundo, además de mostrar con mayor claridad todas las propiedades relevantes de la forma más adecuada posible. Todos los lugares accesibles desde un waypoints pueden ser accedidos desde cualquier waypoints transitando a través de uno o más waypoints.

Las razones por las cuales los waypoint son tan atractivos dentro de la IA es que son usados para moverse, encontrar caminos, marcar la presencia de objetos especiales u obstáculos y dar información acerca del diseño del nivel.

La desventaja de utilizar waypoints es que el determinar correctamente si un punto es accesible desde un waypoint, o si un waypoint puede ser accedido desde otro punto, implica cálculos complejos y consumidores de memoria en tiempo de ejecución.

Un grafo de puntos de visibilidad puede ser además problemático si se tiene el plan de incluir algún tipo de característica de generación de mapas por lo que se tiene que desarrollar algún método automatizado de generación de estructuras de grafos de puntos de visibilidad

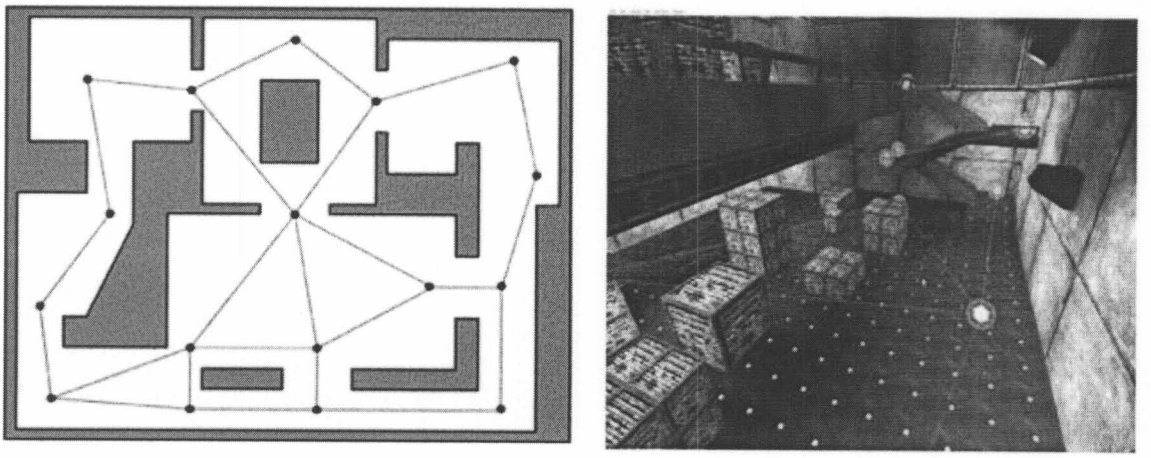


Figura 3: Ejemplo de los puntos de visión en el entorno

Cuando se está diseñando el modelo cognitivo hay que tener mucho cuidado en la inserción de puntos dentro de éste para evitar la inserción de puntos ciegos.

Para determinar una posición específica en el entorno, los algoritmos tienen que encontrar todos los puntos visibles cercanos a partir de aquella posición y el movimiento del agente dentro de la localización, pero puede pasar que para encontrar esta posición deseada no sea posible hallar ningún punto de visibilidad que brinde esta información y a partir de esto aparecen áreas ciegas o puntos ciegos.

Las áreas ciegas no son más que áreas del entorno que quedan fuera del rango de los nodos del grafo de visibilidad. En la figura 2 se muestra con signos de admiración las áreas ciegas.

1.4.3 Mallas de navegación

Una malla de navegación [Snook, 2000], [White, 2002] es una representación que cubre la superficie del mundo con polígonos convexos.

Las mallas de navegación son una de las más poderosas técnicas en los algoritmos del pathfinding en el mundo de 3D, las cuales son conocidas como NavMesh. Usa polígonos complejos para describir el terreno donde se camina en un entorno dado.

Este es un simple y altamente instintivo plan que los caracteres de la Inteligencia Artificial pueden buscar para la navegación y la búsqueda de caminos en el mundo.

Su mayor aplicación está en el mundo de los juegos. Las mallas de navegación solamente se encargan de la parte estática, es el pathfinding el que se encarga de separar las capas y ocuparse de los obstáculos dinámicos. Por lo que podemos decir que el programa funciona bien excepto cuando el medio ambiente cambia mucho.

No puede revelar obstáculos que se muevan o cambien. Si quieres que algo se mueva, tienes que hacer otra capa de polígonos. En general, la idea es que se encuentre el camino más corto de un punto a otro sin atravesar obstáculos como muros, aguas o cualquier otro terreno intransitable.

Una de las características importantes de la malla de navegación es, que el tamaño de la malla no depende del tamaño del mundo aunque sí de las figuras exactas de las barreras o paredes y de la geometría para el entorno.

Adicionalmente como en el interior de un polígono convexo el agente necesita moverse libremente, pues es muy fácil de computar el movimiento en batallas donde el BOT tiene necesidad de moverse repentinamente en muchas direcciones para evitar una colisión o el fuego de un enemigo.

La desventaja de este modelo cognitivo es que es muy difícil y tedioso encontrar todos los polígonos en los cuales se pueda mover o andar para alcanzar los caminos desde polígonos vecinos. Algunas veces este proceso no es acertado en las propiedades físicas de los modelos y esto puede marcar algunos polígonos que pueden ser alcanzados cuando no se está en camino de moverse hacia ellos.

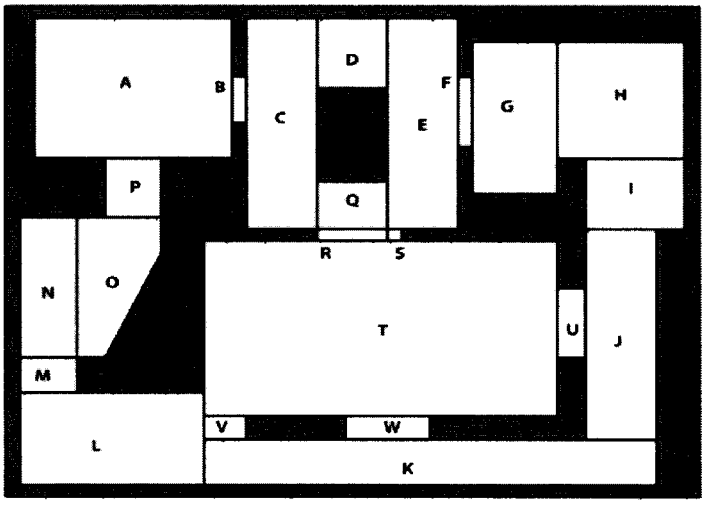


Figura 4: Representación de una malla de navegación común

A continuación se hace mención de dos tipos de mallas de navegación, la primera es la malla de navegación basada en triángulos y la segunda basada en N-lados de polígonos.

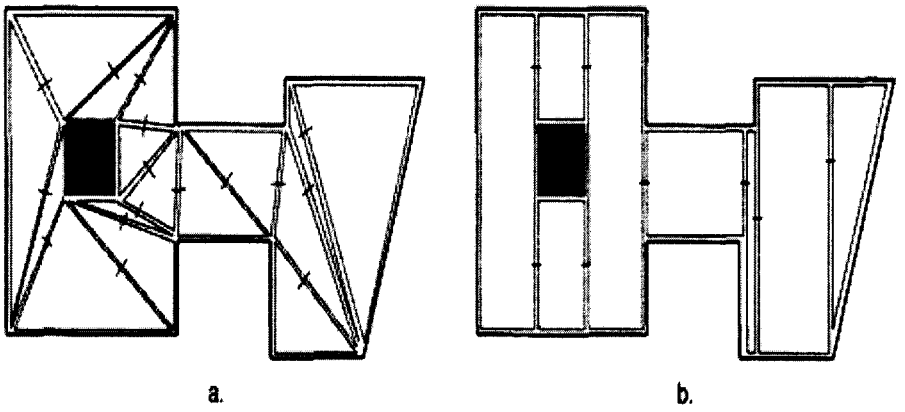


Figura 5: Mallas de navegación basadas en triángulos (a) basadas en N_lados de polígonos (b)

Las mallas de navegación basadas en N-lados de polígonos pueden usualmente representar un espacio de búsqueda más simple que las basadas en triángulos. Esto permite que la búsqueda de caminos sea más rápida y que deje menor huella en la memoria [Tozour, 2002]. Por ejemplo, esto permite un cuarto octogonal usando un simple polígono con 8 lados, mientras que una malla de navegación basada en triángulos por lo menos requerirá de 6 triángulos.

Aunque no podemos dejar de resaltar que las mallas de navegación basadas en triángulos son ampliamente utilizadas ya que su gran número de vértices facilitan la navegación.

1.5 Análisis topográfico

En la mayoría de la ocasiones se hace necesario realizar un análisis topográfico del terreno, aunque un entorno virtual es prácticamente idéntico al mundo real, tiene todas las características de un terreno común, sólo se diferencia de que uno es generado por la computadora con ayuda de personal calificado para esto y el otro por la naturaleza o el hombre.

El análisis topográfico se usa para lograr obtener los datos específicos del terreno, donde a partir de características determinadas se obtienen datos importantes para futuros planes. En el mundo virtual, el análisis topológico brinda otras características y tiene otras bases muy diferentes del mundo real.

El análisis topológico en el mundo virtual se basa fundamentalmente en algoritmos muy sofisticados y eficientes que permiten generar decisiones tácticas e inteligentes y esto se hace de forma común a través de los grafos de navegación que hay en el mundo.

1.6 Aplicaciones de los algoritmos de búsqueda

La investigación de algoritmos tanto exactos como heurísticos para resolver problemas de optimización tiene una vigencia inusualmente importante en estos días, de esta forma, actualmente es acuciante la necesidad de algoritmos muy eficientes que puedan dar respuesta en tiempo real a problemas del usuario o del sistema, y que permitan trabajar con los nuevos sistemas operativos y máquinas.

En el mundo de los video-juegos existen muchas aplicaciones para los algoritmos de búsqueda de caminos. Estos son necesarios siempre que tengamos un escenario o nivel con objetos sólidos o infranqueables en los que queramos que el computador mueva un objeto de un punto a otro del escenario. Como todos sabemos una de las formas más comunes en las que se aplican estos algoritmos es en los juegos de estrategia y de rol, donde les decimos a las tropas donde deben moverse y ellas se las apañan para alcanzar ese punto a través del terreno. Pero también cuando el ordenador necesita mover un enemigo de un punto a otro del escenario necesitará de uno de estos algoritmos.

En el caso del algoritmo A* su principal problema es que no es dinámico, y por tanto si queremos que, por ejemplo, un enemigo persiga a un jugador tendríamos que recalcular el camino a seguir por el jugador cada poco tiempo, lo que puede resultar demasiado costoso, por tanto para darle solución a este problema se utilizan los algoritmos de búsqueda heurística en tiempo real.

Los algoritmos de búsqueda de caminos tienen otras tantas aplicaciones muy útiles para el mundo actual, por ejemplo, algo que tiene mucho éxito en nuestros días son los GPS que poseen los carros de alta tecnología de hoy en día que se utilizan para ayudar al conductor a hallar caminos y sugerir los atajos o caminos más cortos. Esto resulta ser de gran utilidad para el turista que se encuentre en una cuidad desconocida para él.

En el mundo de la medicina también existen aplicaciones para estos algoritmos, los grandes programas hechos para realizar cirugías a pacientes están pensados para hallar el camino más corto y seguro considerando todos los factores como flujos sanguíneos, obstáculos y otros, para llegar a algún lugar decidido previamente por el cirujano dada una entrada. Parte importante de esta técnica está en la definición de la distancia heurística que debe recorrer.

1.7 Conceptos para entender la búsqueda en árboles y grafos

Para entender a cabalidad todo lo relacionado con los algoritmos de búsqueda es necesario conocer algunos conceptos que le serán de gran ayuda, por ejemplo, en problemas de este tipo se utiliza mucho el término **búsqueda**. Las estrategias de búsqueda son criterios que determinan cual es el próximo estado a expandir.

1.7.1 ¿Qué es *búsqueda*?

Método computacional para resolver problemas, cuya solución consiste en una serie de pasos que frecuentemente deben determinarse mediante la prueba sistemática de las alternativas. Desde los inicios de la Inteligencia Artificial, la búsqueda se ha aplicado en diversas clases de problemas como juegos de dos jugadores, problemas de satisfacción de restricciones y problemas de Pathfinding de un único agente.

En la realidad existen tres tipos de búsquedas: la búsqueda a ciegas, la búsqueda heurística y la búsqueda heurística en tiempo real, a continuación una breve explicación de las dos primeras:

1.7.2 Búsqueda a ciegas o sin información

El orden en que la búsqueda se realiza no depende de la naturaleza de la solución buscada, significa que no se tiene información adicional acerca de los estados, la única información es la que proporciona la solución del problema. Sólo generan sucesores y distinguen si han llegado al objetivo o no. La localización de la(s) meta(s) no altera el orden de expansión de los nodos. Las búsquedas no informadas sólo usan la información disponible en la definición del problema, ejemplos de algoritmos que usan esta búsqueda: búsqueda a lo ancho, búsqueda en profundidad, primero en profundidad con profundidad iterativa, búsqueda de profundidad limitada, y búsqueda bidireccional. La desventaja de utilizar este tipo de búsqueda es que pueden acabar recorriendo todo el espacio de búsqueda hasta hallar la solución y esto podría agotar los recursos computacionales.

1.7.3 ¿Qué es heurística?

Para problemas de búsqueda del camino más corto el término tiene un significado más específico. En este caso una heurística es una función matemática, $h(n)$ definida en los nodos de un árbol de búsqueda, que sirve como una estimación de la distancia del camino más económico de un nodo dado hasta el nodo objetivo. Por lo que una estrategia heurística permite saber cuando un estado no es objetivo o si es más prometedor que otro, ejemplos de algoritmos que usan esta búsqueda: Búsqueda primero el mejor, A*, búsqueda en tiempo real.

1.7.4 Algoritmos con búsqueda heurística y búsqueda local

Estos algoritmos utilizan una estimación del coste o de la calidad de la solución para guiar la búsqueda, de manera que se basan en algún criterio heurístico dependiente del problema. Estos algoritmos no son sistemáticos, de manera que el orden de exploración no determina la estructura del espacio de búsqueda sino el criterio heurístico. Algunos de ellos tampoco son exhaustivos y basan su menor complejidad computacional en ignorar parte del espacio de búsqueda.

1.8 Estructuras de datos para la búsqueda de camino

En los algoritmos de búsqueda de caminos las estructuras de datos más adecuadas para representar el espacio de búsqueda son los grafos y los árboles.

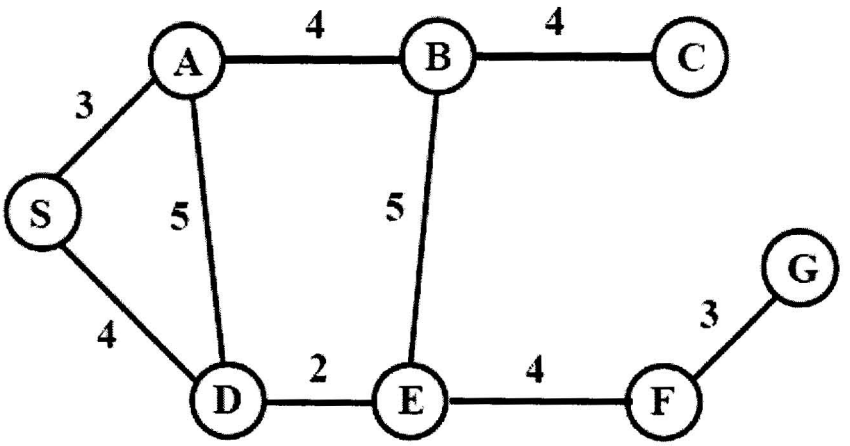


Figura 6: Ejemplo de grafo que representa un espacio de búsqueda.

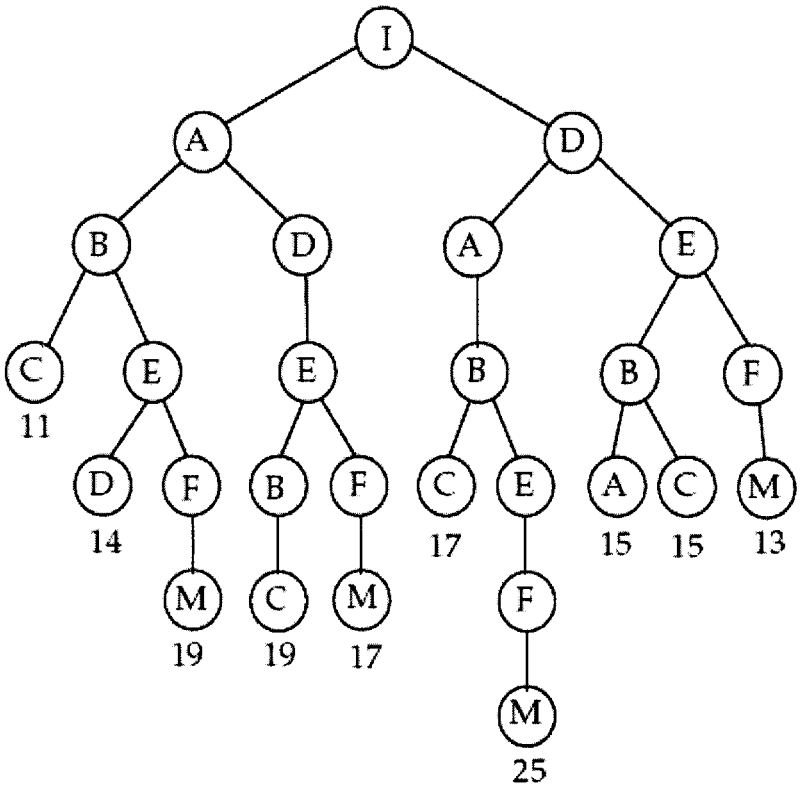


Figura 7: Ejemplo de árbol que representa un espacio de búsqueda.

1.9 Clasificación de algoritmos

Según la búsqueda los algoritmos se pueden clasificar en dos clases:

1.9.1 Los métodos off-line

Este método sigue el siguiente esquema, primero calculan el plan o ruta de mínimo coste de un problema mediante la búsqueda en el espacio de estados y luego ejecutan el plan realizando las acciones o secuencia de operaciones en el entorno. Este esquema, a menudo considerado de forma implícita, supone que ejecutar la solución en el mundo real es un proceso directo y no presenta ninguna dificultad, o sea, primero se realiza una búsqueda completa en el espacio de estados y luego se ejecuta una secuencia de acciones u operaciones en el entorno. En la siguiente figura se muestra su funcionamiento.

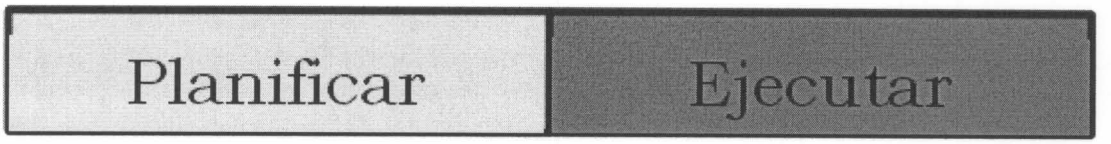


Figura 8: Modo de operar en las búsquedas con métodos off line.

Un ejemplo típico de aplicación de los métodos off-line es la búsqueda de rutas. La búsqueda de rutas consiste en encontrar una ruta en un grafo desde el nodo inicial hasta el nodo objetivo.

1.9.2 Los métodos on-line

En cambio, los métodos on-line intercalan planificación (mediante búsqueda local) y ejecución del plan mientras resuelven un problema. Debido a la propiedad on-line, los métodos tienen una gran flexibilidad para operar en entornos desconocidos y dinámicos. Además, no requieren conocer a-priori todo el espacio de estados, pueden adaptarse a objetos cambiantes y operar en entornos no determinados. Este modo de operación denominado on- line da lugar a los algoritmos de búsqueda en tiempo real. En la siguiente figura se representa el modo de operar de la búsqueda on-line.

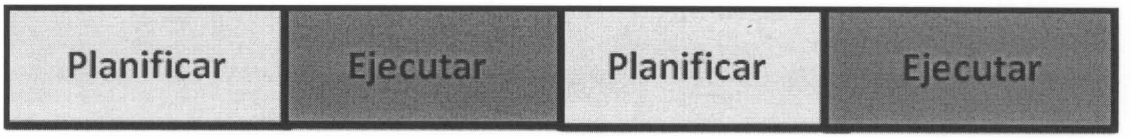


Figura 9: Modo de operar en las búsquedas con métodos on line.

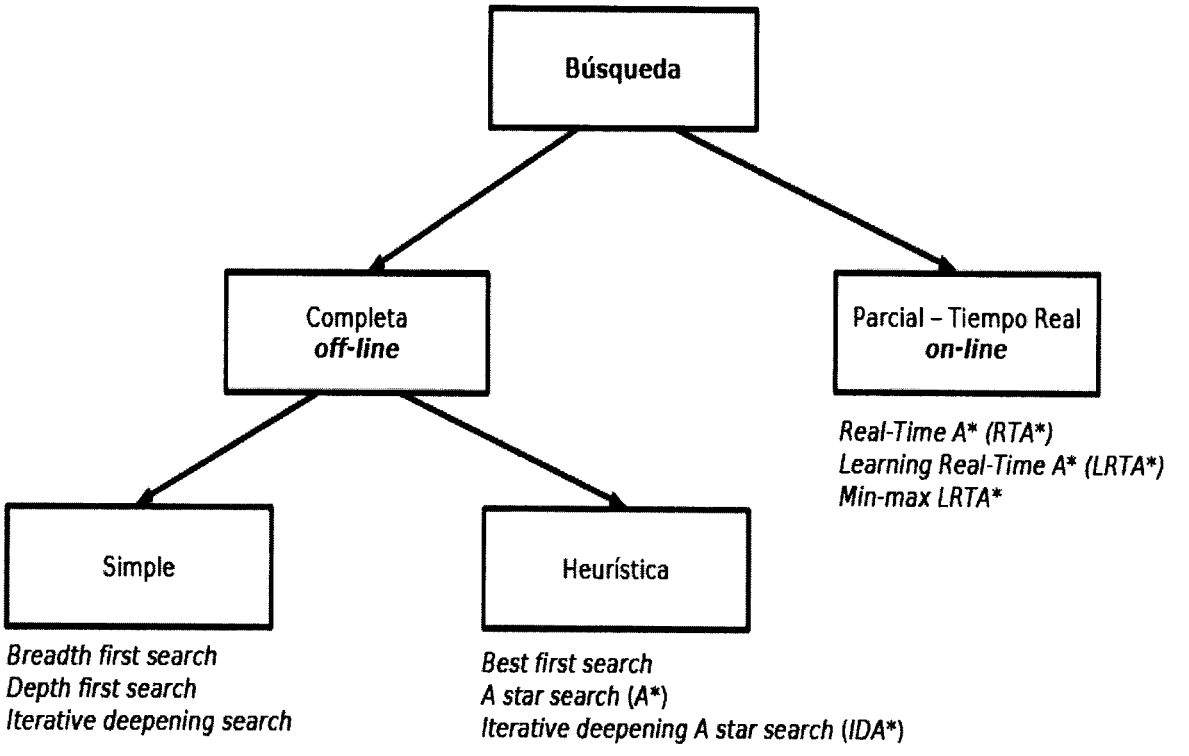


Figura 10: Clasificación de los algoritmos de búsqueda.

1.10 Criterios para evaluar las estrategias de búsqueda

Cada algoritmo de búsqueda de caminos tiene una serie de características que lo identifican y son las que se ofrecen a continuación.

1.10.1 Completitud

Si un algoritmo es completo tenemos la garantía de que hallará la solución, si no lo es, es posible que el algoritmo no acabe o que sea incapaz de encontrarla.

1.10.2 Complejidad en tiempo

Nos indicará el coste temporal de la búsqueda en función del tamaño del problema, generalmente a partir del factor de ramificación y la profundidad a la que se encuentra la solución.

1.10.3 Complejidad en espacio

Espacio requerido para almacenar los nodos pendientes a explorar. También se puede tener en cuenta el espacio para almacenar los nodos ya explorados si es necesario para el algoritmo.

1.10.4 Optimización

Si es capaz de encontrar la mejor solución según algún criterio de preferencia o coste.

1.11 Análisis de Algoritmos

A continuación se realizará un análisis de los algoritmos de búsqueda a ciegas o sin información hasta llegar a algunos de los algoritmos de búsqueda heurística ofreciendo sus características más distintivas.

1.11.1 Depth-First o Búsqueda en Profundidad (LIFO)

Esta búsqueda se centra en expandir un único camino desde la raíz. Siempre se expande el nodo más profundo en la frontera actual. En el caso de llegar a un "callejón sin salida" se retrocede hasta el nodo más cercano (siguiendo al nodo padre) donde se puede tomar una rama alternativa para poder seguir avanzando.

Se puede implementar, mediante el Algoritmo General, considerando la lista, como una pila (cuyas operaciones funcionan en forma LIFO). De esta manera el siguiente nodo a expandir siempre será el último que se haya colocado en la pila de ese nivel, garantizando esto que la expansión vaya aumentando en la profundidad de los nodos.

Es común aplicar esta estrategia mediante un algoritmo recursivo que recorra el árbol en Pre-Orden. Tiene modestos requisitos de memoria. Sólo necesita almacenar un camino, junto con los hermanos restantes no expandidos en cada nodo. De aquí surge otro importante algoritmo, "el backtracking"

1.11.2 Breadth-First search (búsqueda a lo ancho)

La búsqueda en anchura (BFS o Breadth-First search) es un algoritmo para recorrer o buscar elementos en un grafo (usado frecuentemente sobre árboles). Intuitivamente, se comienza en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo) y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol.

Formalmente, BFS es un algoritmo de búsqueda sin información, que expande y examina todos los nodos de un árbol sistemáticamente para buscar una solución. El algoritmo no usa ninguna estrategia heurística.

Este algoritmo de búsqueda en anchura recorre todos los nodos de un árbol de manera uniforme. Expande cada uno de los nodos de un nivel antes de continuar con el siguiente, este algoritmo tiene como ventaja que, al expandir los nodos de forma uniforme garantiza encontrar la mejor solución de un problema de costo uniforme antes que ninguna, de manera que apenas se encuentre una solución, puede devolverse, porque será la mejor, o bien expandir todo el nivel en el cual se hubiere encontrado, para obtener todas las soluciones posibles.

Su principal desventaja es el alto orden de complejidad computacional, que hace que, de no mantenerse muy limitados los parámetros del problema, crezcan rápidamente los requerimientos y se vuelvan inaceptables y que además requiere de mucha memoria.

Básicamente tiene que guardar la parte completa de la red que está explorando. También el tiempo es un factor importante. Fundamentalmente problemas de búsqueda de complejidad exponencial no se pueden resolver, salvo para sus instancias más pequeñas.

La forma de implementarlo es poner los sucesores de cada nodo al final de una cola o agenda, por lo que OPEN (lista de nodos por explorar) se implementa como un stack.

1.11.3 Búsqueda Bidireccional

En la búsqueda bidireccional se llevan a la vez dos búsquedas, una descendente desde el nodo inicial y otra ascendente desde el nodo meta. Al menos una de estas dos búsquedas, debe ser en anchura para que los recorridos ascendente y descendente puedan encontrarse en algún momento. (Tener en cuenta que si tanto el recorrido descendente como el ascendente fueran en profundidad, podría pasar que nunca se cruzaran o encontraran, con lo cual no tendría sentido realizar la búsqueda bidireccional). Por lo demás este método no tiene ninguna dificultad, naturalmente, se podría realizar una búsqueda descendente del nodo meta en anchura y una búsqueda ascendente del nodo inicial en profundidad, alternando la expansión de los nodos entre un tipo de búsqueda y el otro.

Cuando se llegue a un nodo que ya había sido explorado con el otro tipo de búsqueda, el algoritmo acaba. El camino solución es la suma de los caminos hallados por cada búsqueda desde el nodo mencionado hasta el nodo inicial y hasta el nodo meta.

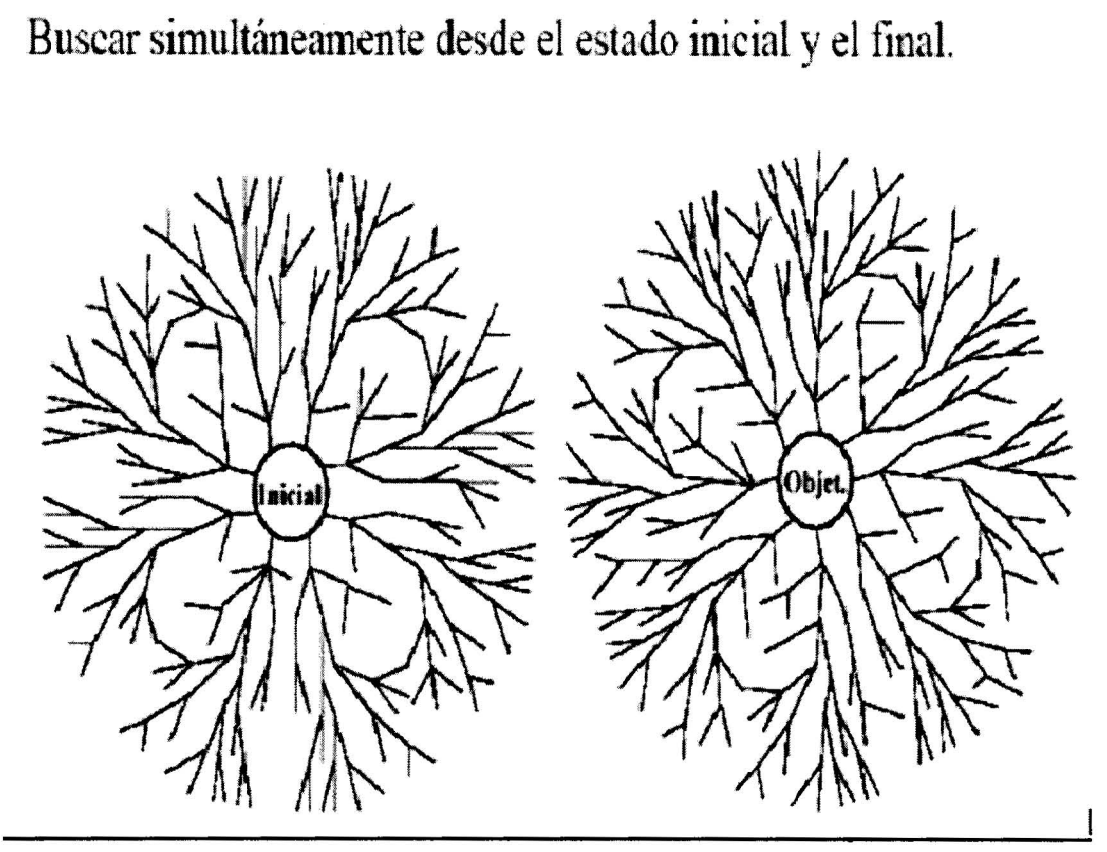


Figura 11: Esquema que representa la búsqueda bidireccional

1.11.4 Búsqueda de Profundidad Limitada

Para evitar caer en caminos de profundidad muy grande, a la mayoría de los algoritmos que usan depth-first se les impone un límite de profundidad máxima. Por tanto es similar a la estrategia de Primero en Profundidad, pero se trata de eliminar el problema de que se puedan generar árboles "ilimitados".

Para ello se establece un límite L a la profundidad del árbol. Todo nodo cuya profundidad sea L, no es expandido (se considera sin sucesores). El problema que puede pasar es si escogemos un valor de L menor que la profundidad del objetivo, en cuyo caso no llegaríamos a la solución.

Si se sabe la profundidad de alguna meta, el algoritmo puede ser completo, pero sigue sin ser óptimo.

1.11.5 Algoritmo DFID (primero en profundidad con profundidad iterativa)

El algoritmo de búsqueda DFID (Depth-First Iterative-Deepening) fue introducido por primera vez cuando Slate y Atkin presentaron un programa de ajedrez.

El algoritmo combina la estrategia de Primero en Profundidad con la de Profundidad Limitada, comenzando con límite igual a 0, aumentándolo de 1 en 1 hasta encontrar el objetivo. Combina las ventajas de las dos estrategias.

Para asegurar el completamiento del algoritmo, el DFID introduce un límite de profundidad en cada una de sus iteraciones. El desarrollo del algoritmo es el siguiente:

En cada iteración el algoritmo realiza una búsqueda limitada en profundidad. Si en dicha iteración no se alcanza la solución óptima, entonces se incrementa dicho límite en una unidad de profundidad y se vuelve a realizar la búsqueda

1.11.6 Backtracking Simple (Vuelta atrás) = Fuerza Bruta

Backtracking es una versión de Depth-First. El término "backtrack" fue acuñado por primera vez por el matemático estadounidense D. H. Lehmer en los años 1950s. Es un método de búsqueda en profundidad que termina encontrando la primera solución sin embargo garantiza encontrar la solución de mínimo coste (óptima), el problema es el tiempo que toma en hacerlo. Su gran desventaja es el no poder incluir información para evaluar cual de los sucesores es el mejor.

A veces los algoritmos de tipo backtracking se usan para encontrar una solución, pero otras veces interesa que las revisen todas (por ejemplo, para encontrar la más corta) .Este método no utiliza información heurística para elegir el sucesor a expandir aunque existe una variante llamada Backtracking Ordenado que sí utiliza información heurística.

Es un método sistemático que itera a través de todas las combinaciones posibles del espacio de búsqueda. Es una técnica general que debe ser adaptada para cada aplicación particular.

El algoritmo backtracking se aplica en la implementación de los lenguajes de programación, tales como, lenguaje de programación Planner y Prolog. Además, se usa en los análisis sintácticos de los compiladores.

El siguiente es un algoritmo genérico de backtracking:

```
Bt (A, k)

1 if Solución? (A, k)

2 then procesarSolucion (A, k)

3 else for each c 2 Sucesores (A, k)

4 do A[k] = c

5 Bt (A, k + 1)

6 if terminar?

7 then return
```

Donde

• Solución? () //es una función que retorna verdadero si su argumento es una solución.

• ProcesarSolucion () //depende del problema y maneja una solución.

• Sucesores () //es una función que dado un candidato, genera todos los candidatos que son extensiones de éste.

• terminar? //es una variable global booleana, inicialmente es falsa, pero que puede ser hecha verdadera por ProcesarSolucion() //en caso que sólo interesa encontrar una solución.

1.11.7 Búsqueda en Profundidad Branch and Bound (Ramificación y Límite)

Este método busca una solución como en el método de backtracking, pero cada solución tiene asociado un costo y la solución que se busca es una de mínimo costo llamada óptima.

Este método imagina el espacio de búsqueda como un árbol de decisión, en cada paso del algoritmo, se toma la decisión de podar las ramas del árbol que parezcan menos prometedoras.

Branch and Bound procede de la forma siguiente: se establece una cota inferior a la posible solución (o superior, si se trata de minimizar), se va siguiendo una rama del árbol, hasta que se encuentra que la solución parcial es menor que esa cota, en ese caso, se descarta la rama completa.

Además de ramificar una solución padre (branch) en hijos, se trata de eliminar de consideración aquellos hijos cuyos descendientes tienen un costo que supera al óptimo buscado acotando el costo de los descendientes del hijo (bound), o sea, utiliza la poda para eliminar las ramas del árbol que no conducen a una solución óptima.

La forma de acotar es un arte que depende de cada problema. La acotación reduce el tiempo de búsqueda de la solución óptima al "podar" (pruning) los subárboles de descendientes costosos.

Puede seguir un recorrido en anchura (estrategia FIFO), profundidad (estrategia LIFO) o cálculo de funciones de coste para seleccionar el nodo más prometedor.

Es un algoritmo muy dependiente de la función heurística. En este algoritmo se manejan términos cómo:

- Nodos vivos, que no es más que un nodo factible y prometedor del que no se han generado todos sus hijos.
- Nodo muerto, es aquel del que no se generan más hijos porque ya se han generado todos sus hijos o no es factible ni prometedor.
- Nodo en expansión, es aquel del que se están generando todos sus hijos en ese instante.

El método funciona de la siguiente forma:

- Se selecciona un nodo del conjunto de nodos vivos.
- Se construyen los posibles nodos hijos del nodo seleccionado anteriormente
- Se eliminan algunos de los nodos creados anteriormente reduciendo así el espacio de búsqueda.

El método finaliza cuando se encuentra la solución o cuando se terminan los nodos vivos.

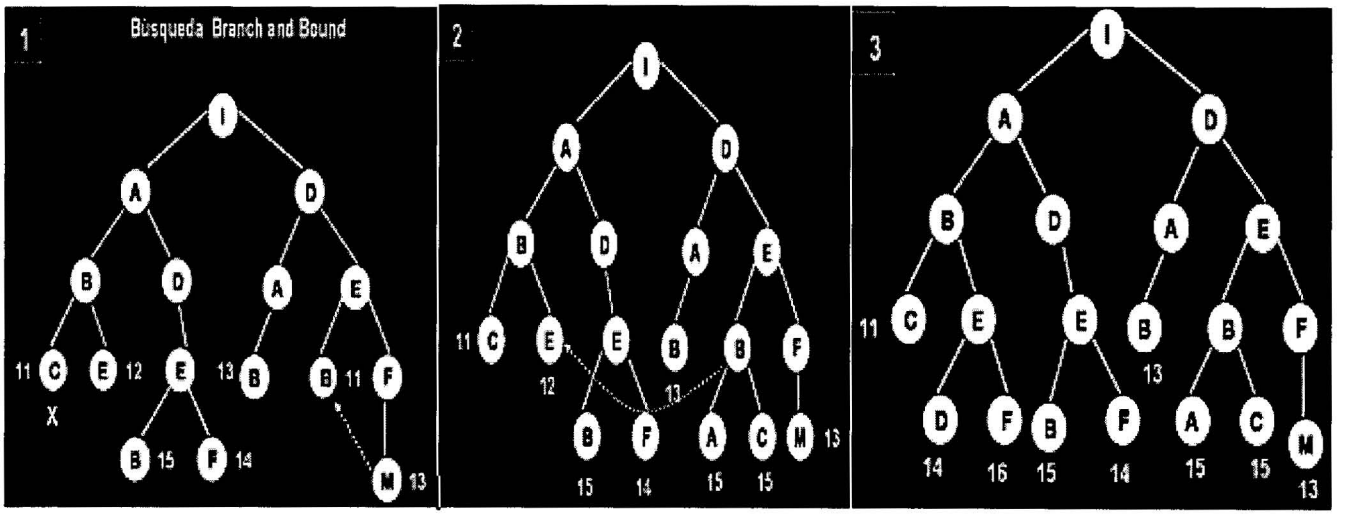


Figura 12: Representación de la búsqueda Branch and Bound.

1.11.8 Algoritmo de Dijkstra

El algoritmo de Dijkstra [Aho, 1983], también llamado algoritmo de caminos mínimos, es un algoritmo para la determinación del camino más corto dado un vértice origen al resto de vértices en un grafo dirigido y con pesos en cada arista.

Su nombre se refiere a Edsger Dijkstra, quien lo describió por primera vez en 1959. Este algoritmo permite computar el camino de costo mínimo de un nodo dado al resto de los nodos de un grafo.

El algoritmo de Dijkstra utiliza una técnica de diseño de algoritmos conocida como greedy (también llamados algoritmos ávidos). Su correcto funcionamiento requiere que los costos asociados a los arcos sean no negativos.

La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices, cuando se obtiene el camino más corto desde el vértice origen al resto de vértices que componen el grafo, el algoritmo se detiene.

El algoritmo es una especialización de la búsqueda de costo uniforme, y como tal, no funciona en grafos con aristas de costo negativo (al elegir siempre el nodo con distancia menor, pueden quedar excluidos de la búsqueda nodos que en próximas iteraciones bajarían el costo general del camino al pasar por una arista con costo negativo).

Pseudocódigo perteneciente a este algoritmo:

```
        DIJKSTRA (Grafo G, nodo _ fuente s)
/* Inicializamos todos los nodos del grafo. La distancia de cada nodo es infinita y los padres son NULL*/
for u que pertenece a V [G] do
   distancia[u] = INFINITO
   padre[u] = NULL
  distancia[s] = 0   //almacenamos todos los nodos del grafo
  Almacenar (cola, V[G])
mientras cola!= 0 do
  /* OJO: Se extrae el nodo que tiene distancia mínima y se conserva la condición de Cola de prioridad*/
   u = extraer_mínimo (cola)
   for v que pertenece a adyacencia[u] do
      if distancia[v] > distancia[u] + peso (u, v) do
         distancia[v] = distancia[u] + peso (u, v)
         padre[v] = u
```

1.11.9 El algoritmo de búsqueda A*

Se clasifica dentro de los algoritmos de búsqueda en grafos. Presentado por primera vez en 1968 por Peter E. Hart, Nils J. Nilsson y Bertram Raphael, A* [Rabin, 2000] es un algoritmo de búsqueda heurística y es óptimo siempre y cuando no se sobreestime la heurística.

Este es un algoritmo de búsqueda de caminos para encontrar el camino más corto entre dos nodos, mientras intenta evitar obstáculos.

El algoritmo A* además de encontrar el nodo objetivo, nos asegura que es el camino más corto.

A* mantiene dos estructuras de datos auxiliares [Higgins, 2002], que podemos denominar *abiertos*, implementado como una cola de prioridad (ordenada por el valor $f(n)$ de cada nodo), y *cerrados*, donde se guarda la información de los nodos que ya han sido visitados.

En cada paso del algoritmo, se expande el nodo que esté primero en abiertos, y en caso de que no sea un nodo objetivo, calcula la función $f(n)$ de todos sus hijos, los inserta en abiertos, y pasa el nodo evaluado a cerrados.

El ciclo en el algoritmo A* consiste en:

- Buscar el nodo más prometedor.
- Analizar a sus nodos adyacentes.
- Poner el nodo analizado en la lista de cerrados.

El algoritmo es una combinación entre búsquedas del tipo primero en anchura con primero en profundidad, mientras que $h(n)$ tiende a primero en profundidad, $g(n)$ tiende a primero en anchura. De este modo, se cambia el camino en la búsqueda cada vez que existen nodos más prometedores.

La clave para determinar que nodos serán visitados está en la siguiente ecuación:

- ***f(n)*** es el costo heurístico asociado a un estado *n*.
- ***g(n)*** es el costo real para alcanzar un estado *n* a partir del estado actual.
- ***h(n)*** es el costo heurístico para alcanzar un objetivo desde el estado *n*.

La relación que existe entre los costos es:

f(n) = g(n) + h(n)

Para garantizar que el algoritmo sea óptimo, la función *h(n)* debe ser admisible, esto es, que no sobrestime el coste real de alcanzar el nodo objetivo. De no cumplirse dicha condición, el algoritmo pasa a denominarse simplemente A, y a pesar de seguir siendo completo, no se asegura que el resultado obtenido sea el camino de coste mínimo.

Una de las principales ventajas y a la vez inconveniente de A* es que explora simultáneamente muchos caminos del espacio de búsqueda, seleccionando siempre el mejor, hasta que se alcanza el objetivo.

Aspecto a tener en cuenta a la hora de implementar:

Sobre lista de abiertos: Suele ser uno de los elementos que más tiempo consume en la función del pathfinding. Cada vez que accedes a la lista de abiertos necesitas encontrar el nodo que tiene el coste ***f(n)*** más bajo.

Hay varias formas en las que podrías hacerlo. Podrías guardar los elementos del camino según los necesites y simplemente recorrer la lista entera cada vez que necesites encontrar el nodo con el coste ***f(n)*** más bajo. Esta es fácil, pero es una manera muy lenta para caminos largos, se podría mejorar manteniendo una lista ordenada y simplemente cogiendo el primer elemento de la lista cada vez que necesites el nodo con el coste ***f(n)*** más bajo o teniendo una cola con prioridad.

Nota 1: Es posible que dos implementaciones de A* con la misma heurística exploren distintos números de nodos.

Nota 2: Es posible que A* agote la memoria del ordenador durante la búsqueda.

A continuación veremos una implementación de dicho algoritmo en pseudocódigo:

```
ABIERTOS:= [INICIAL]          //inicialización

CERRADOS:= []

f'(INICIAL):= h'(INICIAL)

Repetir
```

si ABIERTOS = [] entonces FALLO

si no // quedan nodos

extraer MEJORNODO de ABIERTOS con f' mínima // Cola de prioridad

Mover MEJORNODO de ABIERTOS a CERRADOS

Si MEJORNODO contiene estado _ objetivo entonces

SOLUCIÓN _ ENCONTRADA:= TRUE

Si no

Generar SUCESORES de MEJORNODO

Para cada SUCESOR hacer TRATAR_SUCESOR

Hasta SOLUCIÓN _ ENCONTRADA o FALLÓ

TRATAR _ SUCESOR

SUCESOR.ANTERIOR:= VIEJO // Coste del camino hasta SUCESOR

Caso SUCESOR = VIEJO perteneciente a CERRADOS

Si g (SUCESOR) < g (VIEJO) entonces // (no si monotonía)

SUCESOR.ANTERIOR:= VIEJO // Coste del camino hasta SUCESOR

Caso SUCESOR = VIEJO perteneciente a CERRADOS

Si g (SUCESOR) < g (VIEJO) entonces // (no si monotonía)

// Nos quedamos con el camino de menor coste

VIEJO.ANTERIOR:= MEJORNODO

```
      Actualizar g (VIEJO) y f'(VIEJO)
      Propagar g a sucesores de VIEJO
    Eliminar SUCESOR
    Añadir VIEJO a SUCESORES_MEJORNODO
  Caso SUCESOR = VIEJO perteneciente a ABIERTOS
    Si g (SUCESOR) < g (VIEJO) entonces  // Nos quedamos con el camino de menor coste
      VIEJO.ANTERIOR:= MEJORNODO
      Actualizar g (VIEJO) y f'(VIEJO)
    Eliminar SUCESOR
    Añadir VIEJO a SUCESORES_MEJORNODO
  Caso SUCESOR no estaba en ABIERTOS ni CERRADOS
    Añadir SUCESOR a ABIERTOS
    Añadir SUCESOR a SUCESORES_MEJORNODO
    f'(SUCESOR):= g (SUCESOR) + h'(SUCESOR)
function A*(start, goal)
    var closed: = the empty set
    var q: = make_queue(path(start))
    while q is not empty
        var p: = remove first(q)
```

```
    var x: = the last node of p
    if x in closed
        continue
    if x = goal
        return p
    add x to closed
    foreach y in successors(x)
        enqueue (q, p, y)
return failure
```

Capítulo2: Algoritmos posteriores al A*.

2.1 Introducción

El estudio de este capítulo se centró en el análisis de los algoritmos con soluciones óptimas y con un alto grado de eficiencia, por tanto se detallan y presentan las características principales de los mismos, teniendo en cuenta el análisis que se realizó en el capítulo 1 de algunos de los algoritmos de búsqueda clásica, para de esta forma llegar a seleccionar el algoritmo con mejores condiciones para optimizar el módulo genérico de búsqueda de caminos que está integrado a la biblioteca de Inteligencia Artificial.

También se hará un breve análisis del algoritmo de planificación más idóneo a utilizar con el algoritmo de ejecución RTA*.

Luego de haber analizado en el primer capítulo el conocido algoritmo de búsqueda de caminos A* nos dimos cuenta que a pesar de ser probablemente el más usado en la búsqueda de caminos en juegos, sufre de una clásica limitación en espacios de tiempo. A* puede usar un gran reparto de memoria y tomar un largo tiempo en ejecutarlo.

Ahora nos centraremos en las optimizaciones que se le ha hecho a dicho algoritmo. Los algoritmos que utilizaremos para lograr este objetivo se llaman genéricamente "algoritmos de búsqueda heurística con limitación de memoria". Los más utilizados son el IDA* (Iterative Deepening A*), y SMA* (Simplified memory A*).

2.2 Análisis de los algoritmos posteriores al A*

A continuación se presentarán las características distintivas de algunos algoritmos de búsqueda heurística.

2.2.1 Algoritmo IDA*. (Profundidad Iterativa A*)

El IDA* (Iterative-Deepening A*) [Korf, 1985] es al igual que el DFID un algoritmo basado en la profundización iterativa. La única diferencia entre ambos algoritmos consiste en que mientras el DFID se basa en la profundidad para cada una de sus iteraciones, el IDA* se basa en la información heurística que posee para determinar el siguiente límite de la iteración.

Este algoritmo no funciona de la forma primero el mejor y surge para mejorar la gran limitante del A* que es, que ocupa mucho espacio en memoria, éste por el contrario utiliza muy poca memoria. El tratamiento de esa información se realiza de igual forma que en el algoritmo A*, o sea, mediante la función de evaluación ***f(n)*** introducida anteriormente.

El funcionamiento del algoritmo es el siguiente:

- En cada iteración el algoritmo realiza una búsqueda en profundidad hasta donde se lo permita su límite de coste.
- Cada vez que se visita todo el grafo de búsqueda contenido dentro de ese límite sin hallar la solución entonces, el algoritmo incrementa el límite de coste.
- Ese nuevo límite viene dado por el menor de los límites de coste, o sea, por el menor valor del coste de los nodos que tenían un valor superior en la anterior iteración.

Todo esto se traduce de la siguiente manera, la idea de este tipo de algoritmo es que es iterativo por niveles, en una iteración dada llegan hasta un nivel y te devuelven una estimación heurística del mejor movimiento, en las siguientes iteraciones se va aumentando la profundidad de la búsqueda de nivel en nivel, y se utilizan los valores de las búsquedas anteriores como parámetros para podar en las búsquedas siguientes.

Este tipo de algoritmo cumple con el requisito de poder cortarse en cualquier momento y aún así devolver un resultado.

Como señalamos anteriormente, este algoritmo se considera limitado en profundidad, aunque ésta afirmación no sea estrictamente cierta. La razón de ello viene dado por su eficacia en cuanto al uso de memoria pero como se puede ver, en su funcionamiento no se realiza un control estricto de la memoria.

IDA* es un método de búsqueda completo y óptimo, lo que quiere decir que siempre encuentra una solución si es que ésta existe y además garantiza encontrar la mejor solución de entre todas las posibles.

Este método tiene las mismas ventajas y desventajas que A*, excepto en lo referente al coste espacial. En este aspecto IDA* presenta notables ventajas ya que únicamente necesita un espacio proporcional a la longitud de la ruta más larga que se explore.

Esta limitación en el uso de la memoria resulta beneficiosa pero también tiene sus desventajas, ya que al convertir la búsqueda de la solución en un proceso iterativo se expandirán varias veces los mismos nodos. Esto

es algo muy a tener en cuenta, ya que dependiendo de las características de los problemas a resolver, obtendremos mejores o peores prestaciones.

En el mejor caso el coste temporal de IDA* puede ser muy similar al de A*, e incluso menor, ya que al ser un algoritmo simple y no necesitar de inserciones, borrados y reordenamientos en listas de prioridades tiene una menor sobrecarga por nodo. Este mejor caso ocurrirá cuando tengamos un problema en el que las heurísticas adopten valores aproximados al coste real desde el comienzo de la ejecución, ya que entonces se realizarán pocas iteraciones, expandiendo además pocos nodos en las iteraciones iniciales.

En el peor caso el coste temporal de IDA* se acerca al de un algoritmo de profundización iterativa habitual como el IDS (Iterative Deepening Search), es decir, en cada iteración se profundiza únicamente un nivel más en el árbol.

Esto ocurre en problemas en los que los valores heurísticos son poco acertados, lo que provoca que en cada iteración aumentemos el contorno en sólo uno o dos niveles.

Si comparamos el algoritmo IDA* con A* vemos que los costes temporales son comparables. En el caso de la poda parcial IDA* puede ser hasta más rápido si está bien implementado.

En cuanto al factor de ramificación, A* presenta valores notablemente menores, aunque esto es normal ya que al ser IDA* un proceso iterativo la reexpansión de nodos en cada iteración es inevitable.

En consecuencia, cuando nos enfrentemos a problemas en los que las necesidades espaciales sean importantes, podremos acudir a IDA* sin excesivas preocupaciones.

En los casos en los que dispongamos de memoria suficiente, A* será casi siempre una mejor elección, pero la simplicidad de IDA* y sus buenas propiedades lo hacen una buena elección para una gran variedad de problemas en entornos con memoria limitada, o en los que la velocidad no sea lo más importante.

2.2.2 Algoritmo SMA* (Simplified Memory-bounded A*)

El SMA* aparece en cierta medida debido a los problemas del IDA* en espacios reducidos de memoria. Al contrario de éste, el SMA* realiza un control estricto de la memoria, delimitando desde un principio el máximo de memoria de la que dispone.

De manera general este algoritmo funciona de la siguiente manera:

- Utiliza toda la memoria disponible.
- Evita estados repetidos.

Además lo podemos clasificar cómo:

- Completo si existe suficiente memoria disponible para almacenar el camino más prometedor de la solución.
- Y además es óptimo.

Tiene como inconveniente principal que su implementación podría resultar muy complicada.

Ahora es momento de dominar algunos conceptos importantes que servirán para el mejor entendimiento de este segundo capítulo y que de alguna forma nos ayudarán a escoger la solución más adecuada.

2.3 Búsqueda heurística en tiempo real

En búsqueda heurística en tiempo real (BHTR) [Koening, 2001], el calificativo "tiempo real" se debe a que cada acción individual ha de ser ejecutada en un tiempo limitado. Generalmente, se requiere sacrificar la optimalidad de la solución.

Otra característica importante en BHTR que es el aprendizaje de heurísticas. El aprendizaje fue introducido inicialmente para evitar ciclos infinitos [Korf, 1990].

El proceso de búsqueda en tiempo real se lleva a cabo alternando dos modos de operación bastante distintos:

El primer modo es de planificación: Es donde se simulan los movimientos del agente para evaluar los resultados de las acciones inmediatas. Luego de terminar la búsqueda, en este espacio acotado de posibilidades, el agente pasa al modo de ejecución y realiza efectivamente un movimiento.

El proceso continúa alternando fases de planificación y ejecución hasta que se alcanza el estado objetivo.

La solución óptima no se encuentra la primera vez sino que se espera que la calidad de la solución mejore al repetir la solución de ese problema concreto.

Seria importante tener dominio de dos nuevos conceptos, ¿qué es la fase de planificación y de ejecución? por separado:

2.3.1 Fase de planificación

Durante esta fase es importante limitar la profundidad de búsqueda para poder responder dentro de los límites de tiempo impuestos. Dado que la profundidad de búsqueda es limitada, es necesario contar con un método para evaluar los estados no objetivos. Es muy importante para el funcionamiento de esta estrategia la utilización de heurísticas ya que estas permiten estimar el costo de alcanzar un objetivo desde un estado dado. El proceso de selección de estado funciona muy parecido al del A* pues utiliza las mismas funciones f(n), g(n) y h(n).

La precisión de los valores obtenidos aumenta durante la planificación al incrementar la profundidad de búsqueda, aunque también aumenta el costo de ejecución exponencialmente. De esta forma, se obtiene todo un conjunto de funciones heurísticas que intercambian precisión por costo.

2.3.2 Fase de ejecución

Uno de los principios más importantes de la búsqueda en tiempo real es la utilización de la estrategia de ejecución para los movimientos. Es decir, cada movimiento dependerá de la información obtenida en cada etapa de planificación. La razón es que luego de ejecutar la acción, se supone que la frontera de búsqueda se expandirá, lo cual puede llevar a una elección para el segundo movimiento diferente a la que arrojó la primera búsqueda.

Sin embargo, para realizar una secuencia de decisiones no es suficiente con la información devuelta por el algoritmo de planificación. La estrategia básica de repetir el algoritmo de planificación para cada decisión, resulta inadecuada al ignorar la información relacionada con los estados anteriores. El problema radica en que, al volver a un estado previamente visitado, se entrará en un ciclo infinito. Esto es algo que sucederá con frecuencia, debido a que las decisiones se basan en información limitada y por lo tanto direcciones que al principio parecían favorables pueden resultar equivocadas al reunir más información durante la exploración.

En general, se desea evitar entrar en ciclos infinitos y a la vez permitir volver a estados ya visitados cuando parezca favorable. Por tanto, sólo se deberá regresar a un estado ya visitado cuando el coste sea más favorable que el estado siguiente.

Hoy en día se ha logrado un gran avance en los métodos de planificación y de búsqueda on-line, conocidos como métodos de Búsqueda Heurística en Tiempo Real (BHTR).

Estos métodos, para poder guiar la búsqueda utilizan información heurística. Mientras mejor sea ésta información, o sea, entre más admisible sea la heurística, (que no se sobreestimen los valores reales), menor será el coste de la búsqueda de la solución.

Una capacidad importante de los métodos es el aprendizaje, a través de sucesivas exploraciones del espacio de búsqueda se obtienen valores heurísticos más cercanos al coste real, hasta finalmente converger a los valores óptimos.

2.4 Estrategia de aprendizaje

El aprendizaje fue introducido inicialmente para evitar ciclos infinitos [Korf, 1990]. El aprendizaje en los métodos BHTR expuestos, se hace actualizando el coste heurístico del estado actual con el valor que más conviene para alcanzar el objetivo, obtenido de todos los estados vecinos. Es decir, el aprendizaje de tiempo real sólo se hace sobre el estado que el agente está visitando. El proceso hecho de esta manera no permite que lo aprendido en el estado actual por el agente, se relacione directamente en tiempo real con lo aprendido en uno o más estados atrás de la ruta que ha recorrido. Esta es una estrategia general de aprendizaje en los algoritmos de búsqueda heurística en tiempo real. En la mayoría de los algoritmos, la actualización se limita al estado actual. Típicamente, el aprendizaje se realiza antes de la ocurrencia de ejecución de acciones.

2.5 Recursos Computacionales

En el caso de los algoritmos de búsqueda de caminos clásicos utilizan un alto por ciento de recursos computacionales debido a que deben recalcular en repetidas ocasiones los espacios de búsqueda en terrenos con grandes dimensiones en la fase de planificación, lo que provoca en innumerables ocasiones que la memoria se agote, que además consuma mucho más tiempo del esperado, así cómo, que el coste de la solución sea muy elevado, lo que provoca el fracaso de estos algoritmos. En cambio los algoritmos de búsqueda heurística en tiempo real son capaces de minimizar todas estas desventajas antes expuestas, debido a su capacidad de poder hacer la planificación y ejecución del camino por intervalos, esto trae como ventaja que

los recursos computacionales utilizados por los mismos sean evidentemente menores que los utilizados por los algoritmos tradicionales, otra ventaja es que en terrenos grandes y dinámicos sean estos los más rápidos y apropiados a utilizar, al mismo tiempo de no abusar de la memoria de la máquina; todo esto es lo que justifica la popularidad que tienen en nuestros días los algoritmos de búsqueda heurística en tiempo real. En los epígrafes siguientes se ofrecen características específicas de estos algoritmos.

2.6 Algoritmos que trabajan en tiempo real

Debido a la importante propiedad de poder trabajar en tiempo real, los métodos tienen una gran flexibilidad para operar en entornos desconocidos y dinámicos [Ishida, 1997] [Koening, 1999]. Además, no requieren conocer todo el espacio de búsqueda y pueden adaptarse a objetivos cambiantes. Aunque para probar que convergen a soluciones óptimas debemos probar que los valores heurísticos se mantienen admisibles.

Estas capacidades no son fáciles de implementar con métodos off-line.

2.6.1 Learning Real-Time A* (LRTA*)

Learning Real-Time A* [Korf, 1990], es uno de los métodos originales y más populares de BHTR. Es un método de búsqueda heurística de tiempo real que bajo ciertas condiciones, siempre encuentra una solución a un problema y converge a soluciones óptimas cuando lo resuelve repetidas veces. Alterna entre dos fases: planificar y ejecutar. En la fase de planificación LRTA* decide la acción a realizar considerando la situación actual y teniendo en cuenta todos los posibles caminos que se podrían tomar, luego pasa a la fase de ejecución en la que realiza la acción escogida en la fase anterior. Tenemos que dejar claro que la planificación consiste en seleccionar el próximo estado y la ejecución en moverse a ese estado. La selección del estado siguiente es similar a como se hace en A*.

LRTA* construye y actualiza una tabla que contiene los valores heurísticos. Inicialmente los valores se calculan a partir de una función de evaluación heurística, o se les asigna cero si la función no está disponible. El algoritmo actualiza los valores de la tabla cada vez que realiza el proceso de planificación y ejecución. Si el estado actual es ***s***, actualiza h(s) con en valor de f(s') seleccionado en la fase de planificación.

La actualización permite que a través de sucesivas exploraciones del espacio de búsqueda se aprendan valores heurísticos más cercanos al real hasta finalmente converger a estos. Así, al explorar en varias ocasiones el espacio de búsqueda y partiendo con valores heurísticos admisibles y no negativos, a lo largo de varias

ejecuciones LRTA* almacena los costes reales en la tabla de valores heurísticos de todos los estados de la ruta comprendidos entre el estado inicial y el estado objetivo.

LRTA* es completo si:

- Se cumple que el espacio de búsqueda es finito.
- Existe una ruta desde cualquier estado al objetivo.
- Se parte con valores heurísticos no negativos y admisibles.

Además, bajos las mismas condiciones, converge a rutas óptimas sobre varias exploraciones del espacio de búsqueda. La demostración de estas propiedades se puede encontrar en [Korf, 1990]. La principal característica de este algoritmo es la capacidad de aprendizaje sobre exploraciones sucesivas. La mayoría de los algoritmos de búsqueda heurística en tiempo real desarrollados posteriormente a LRTA*, intentan mejorar sus característica negativas que como ya se expuso anteriormente radica en el tiempo en que da una solución. Inmediatamente se representará de manera sintetizada las características principales de los trabajos realizados hasta la fecha.

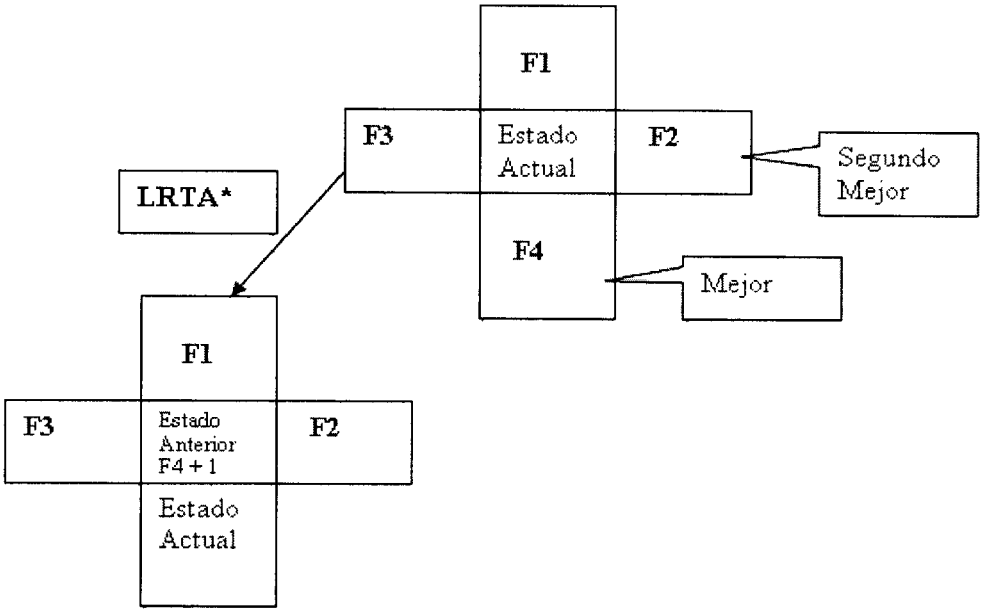


Figura 13: Modo de actualizar los valores en la tabla de hash del LRTA*

2.6.2 RTA* (Real- Time A*)

Es un algoritmo de control de la ejecución. Real-Time A* (RTA*) [Korf, 1990], Es uno de los métodos originales de BHTR. Es un algoritmo para controlar la fase de ejecución de la búsqueda en tiempo real, y es independiente del algoritmo de planificación. Este algoritmo al igual que el anterior actualiza una tabla donde contiene los valores heurísticos, aunque a diferencia utiliza para actualizar el valor heurístico del estado actual el segundo mínimo. La idea de usar el segundo mínimo es que a veces es más conveniente regresar a estados anteriores, que seguir adelante en la búsqueda. ¿Cuando es más conveniente retroceder a un estado visitado anteriormente? cuando la estimación de resolver el problema desde ese estado mas el coste de volver hasta ese estado es menor que el coste estimado de llegar a la meta desde donde estoy hacia adelante.

Esta diferencia hace que RTA* sea más rápido que LRTA* para encontrar la solución la primera vez que explora el espacio de búsqueda y en algún momento los valores heurísticos serán no admisibles, ya que usa el valor del segundo mínimo, que siempre será mayor o igual que el primer mínimo. El mecanismo de actualización de RTA* puede ocasionar que ***h*** se vuelva no admisible, lo que anula la capacidad de aprendizaje. Debido a esto RTA* en ocasiones no garantiza la convergencia a rutas óptimas ya que puede sobrevalorar el coste óptimo.

El algoritmo guarda el valor de cada estado visitado durante la etapa de ejecución en una tabla de hash. A medida que la búsqueda avanza se actualizan estos valores utilizando técnicas derivadas de la programación dinámica.

La propiedad más importante de RTA* es que, si en espacios finitos el estado objetivo se puede alcanzar, se garantiza que se encontrará una solución. Además, el espacio requerido es lineal respecto al número de movimientos realizados, ya que solo lleva una lista de los estados previamente visitados.

El tiempo de ejecución también es lineal respecto a los movimientos ejecutados. Esto se debe a que, aunque el tiempo de planificación es exponencial respecto a la profundidad de búsqueda, el tiempo está acotado por una constante al limitar la profundidad de búsqueda.

En un espacio de estados finito con costes de arcos positivo, valores heurísticos positivos y en donde se puede alcanzar el estado objetivo desde cualquier estado, RTA* es correcto y completo [Korf, 1990] .

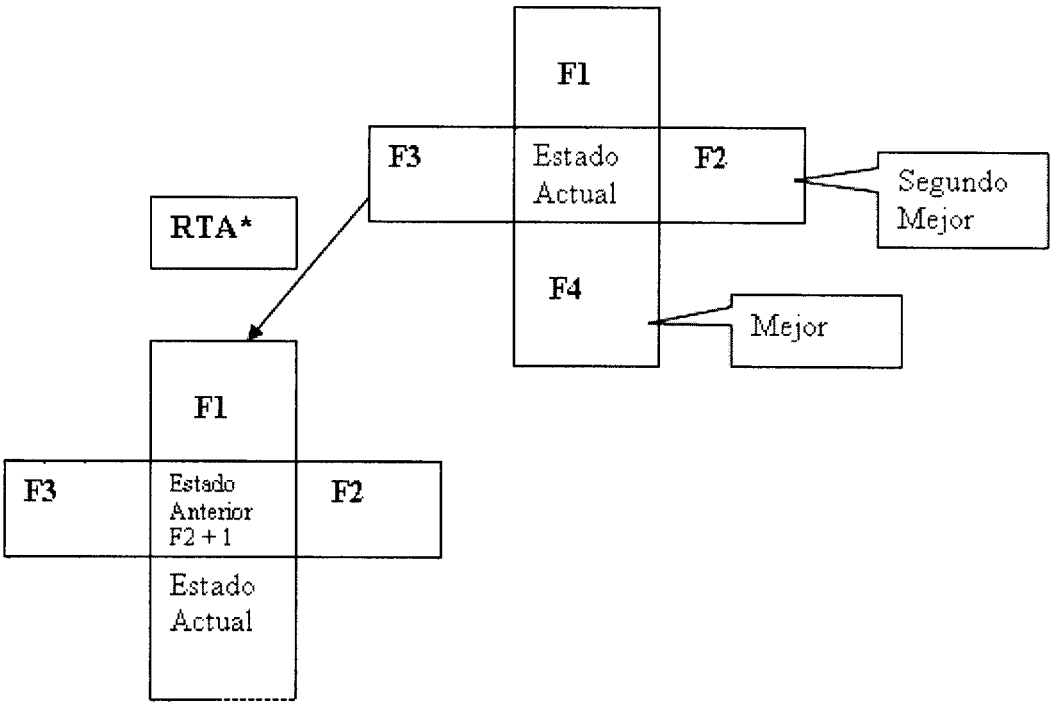


Figura 14: Modo de actualizar los valores en la tabla de hash del RTA*.

2.6.2.1 El algoritmo de planificación: Minimin

El algoritmo empleado para extraer o calcular los planes, ha sido Minimin [Korf, 1990], introducido previamente por Korf en el contexto general de búsqueda heurística en tiempo real. Minimin es un algoritmo planificador de la búsqueda heurística que es capaz de refinar sus soluciones durante la búsqueda usando un criterio dinámico de profundidad limitada.

Este algoritmo es un caso especial del algoritmo Mínimax para juegos, con dos jugadores, es decir, para agentes que compiten o lucha entre adversarios, por lo que el Minimin es una adaptación del Mínimax pero para un único agente.

La búsqueda se hace a partir del estado actual hasta una profundidad determinada y en los nodos de la frontera se aplica la función de evaluación *f*. Para hacer esto se adoptó el coste de la función del algoritmo A*, f(n)=g(n) + h(n).

El valor de cada nodo interno es el mínimo de los valores de los nodos de la frontera del subárbol debajo del nodo.

Seria posible aplicar branch – and bound si la función heurística ***h*** fuera monótona no decreciente (si nunca decrece a lo largo de su trayectoria). Como resultado el algoritmo se llamaría poda alfa por analogía con el algoritmo alfa-beta y ésta consiste en mantener un valor mínimo global en cada búsqueda, de forma que, cualquier nodo que supere este valor será podado. De esta manera, se añade un parámetro ***alfa*** que será el mínimo valor *f* de los nodos de la frontera evaluados hasta el momento. Cuando se genera cada nodo interno se calcula su valor *f,* si éste es igual o mayor que el valor de ***alfa*** se termina la rama de búsqueda correspondiente. Es más, si se ordenan los estados expandidos de modo que los mejores nodos de la frontera sean evaluados primero, será posible reducir considerablemente la búsqueda y por consecuencia mejorar la eficiencia de la poda.

La selección del siguiente estado se hace a través de este algoritmo.

La poda es la técnica de empleo de conocimiento del problema para eliminar durante la búsqueda las partes del espacio del problema que se sabe que no contienen al estado meta. Mediante la poda podemos lograr reducir el tiempo de ejecución del peor caso sin comprometer la búsqueda del estado meta.

2.6.3 HLRTA* (Hybrid Learning Real Time A*)

El algoritmo HLRTA* [Thorpe, 1994], es un híbrido entre el LRTA* y el RTA*. Lo que conllevó a realizar esta combinación entre ambos métodos fue precisamente la necesidad de poder lograr un algoritmo capaz de poseer las ventajas de cada uno.

El RTA* es más eficiente y rápido que el LRTA* la primera vez que se busca el objetivo, pero al no escoger el menor valor de la heurística en su siguiente estado en exploraciones sucesivas pierde la capacidad de aprendizaje, esta última característica resulta ser una gran desventaja para los algoritmos de búsqueda de caminos en el mundo actual. Por otra parte el LRTA* es más lento y por tanto menos eficiente, pero en cambio tiene la capacidad de aprender de sus exploraciones sucesivas. La motivación de HLRTA* es mejorar la capacidad de aprendizaje de LRTA*, usando el segundo mínimo en la actualización, pero manteniendo la admisibilidad de la heurística.

HLRTA* usa dos estimaciones heurísticas por cada estado, h1 y h2, que corresponden a las estimaciones heurísticas de LRTA* y RTA* respectivamente y también almacena la dirección del movimiento del agente en cada estado. Además, si x es el estado actual, mantiene en d(x) la dirección del movimiento desde x, es decir, el estado que el agente selecciona como estado siguiente. El punto interesante de este algoritmo es que

cuando el agente pasa por x y luego regresa a x desde d(x) (esto es, cuando el agente vuelve al estado x por el mismo arco que usó para dejarlo), entonces la estimación h2 es admisible y puede ser usada en lugar de h1 [Thorpe, 1994].

HLRTA* siempre usa heurísticas admisibles, por tanto HLRTA* converge a rutas óptimas de la misma manera que lo hace LRTA*.

El algoritmo en cuestión posee las siguientes características, es completo y converge a rutas óptimas sobre varias exploraciones del espacio de búsqueda si se cumple que:

- Dicho espacio de búsqueda es finito.
- Existe una ruta desde cualquier estado al objetivo.
- Se parte con valores heurísticos no negativos y admisibles.

2.6.4 Algoritmo FALCONS (FAst Learning and CONverging Search)

Este algoritmo [Furcy, 2000] es un sucesor del algoritmo LRTA* y surge por la necesidad de optimizar al mismo ya que FALCONS mejora su fase de planificación, lo que permite que se mejore su capacidad de aprendizaje, para conseguir esto lo que hace es utilizar la información del camino recorrido de manera similar al del A* ,o sea, que para moverse hacia el próximo estado utiliza los costes muy similar al algoritmo A*, es decir, que siempre que escoge la siguiente posición a moverse escoge el de menos coste.

FALCONS utiliza dos tablas para almacenar los valores heurísticos, una denominada h-valores para estimar la distancia de cada estado al estado objetivo, y la segunda denominada g-valores para la distancia estimada desde el estado inicial a cada estado adyacente.

La actualización de las tablas se realiza mientras FALCONS recorre el espacio de estados, al converger las tablas almacenan los valores h* y g*.

FALCONS es completo y converge a rutas óptimas sobre varias exploraciones del espacio de búsqueda si se cumple que:

- El espacio de estado S es finito.
- Los costes de cada estado son positivos.
- Existe una ruta desde cualquier estado al objetivo.

- Se parte con valores heurísticos no negativos y admisibles.

2.6.5 Algoritmo eFALCONS (Even FAst Learning and CONverging Search)

Este algoritmo [Furcy, 2001], es una combinación de FALCONS y HLRTA*. La motivación de eFALCONS es aprovechar las propiedades de HLRTA* y FALCONS que mejoran LRTA*.

HLRTA* mejora la regla de actualización del valor heurístico de LRTA*, lo que permite una búsqueda inicial eficiente y FALCONS mejora la regla de elección del sucesor de LRTA*, aumentando la capacidad de aprendizaje. eFALCONS usa la regla de actualización del valor heurístico de HLRTA* para los g-valores y h-valores y la regla de elección del sucesor de FALCONS.

El algoritmo eFALCONS es completo y converge a rutas óptimas sobre varias exploraciones del espacio de búsqueda si se cumple que:

- El espacio de estado S es finito.
- Los costes de cada estado son positivos.
- Existe una ruta desde cualquier estado al objetivo.
- Se parte con valores heurísticos no negativos y admisibles.

Capítulo3: Propuesta de solución.

3.1 Introducción

En este último capítulo se explicará de manera detallada el proceso de selección del algoritmo teniendo en cuenta el análisis que se hizo en el capítulo anterior. Además se ofrecerán detalles del demo realizado, el mismo proporciona un entorno agradable, del demo en cuestión también se darán sus características principales, cómo fue concebido, el modo en que se realizó, entre otras cosas. El entorno de navegación consiste en un espacio discreto y sencillo de sólo dos dimensiones en forma de matriz, denominado rejilla regular. Al utilizar un entorno tan simple se evita el análisis de terreno, que consiste en la extracción de información útil y manejable de un modelo complejo de terreno.

- Para el proceso de selección del algoritmo base de la aplicación se tuvo en cuenta principalmente los recursos computacionales utilizados por cada algoritmo además del tiempo en encontrar una solución.

Es indudable que el algoritmo óptimo encontrado hasta el momento y sin discusión alguna, es además el más popular es el algoritmo de búsqueda heurística A*, sin embargo tiene una gran deficiencia, y es que sólo opera en terrenos estáticos y que debido a que pertenece al grupo de los algoritmos clasificados off-line el proceso de planificación en un terreno grande podría tomar un tiempo muy extenso o lo que es peor podría agotar todos los recursos computacionales lo cual resulta ser una gran desventaja. Para darle solución a este gran problema surgen los algoritmos en tiempo real, estos algoritmos dada su habilidad de operar en tiempo real son clasificados algoritmos on-line lo que permite que puedan ser utilizados en terrenos dinámicos, inicialmente desconocidos o cuando el agente tiene un tiempo limitado para planificar sus movimientos. Básicamente lo que hacen es intercalar el proceso de planificación y ejecución en repetidas ocasiones, en dependencia del nivel de profundidad que el usuario desee hasta alcanzar al fin una solución que aunque no garantizan ser completamente óptimas sus soluciones si convergen a la optimalidad. En cuanto a la profundidad a la que se hizo mención anteriormente se podría decir que a medida que la misma se aumenta se supone que se pueda evaluar estados más cercanos (o no) a la solución, y por lo tanto obtener mejor información de qué camino o acción es más adecuada. Es decir, aumentando la profundidad, se obtendría más y mejor información para decidir qué hacer en cada paso. La desventaja sería en que si hacemos eso entonces como consecuencia se deberá invertir más tiempo en planificar, posiblemente perdiendo la oportunidad de moverte, o sea, se demostró que al aumentar el horizonte de búsqueda (profundidad), se mejora la convergencia, pero, el tiempo de ejecución aumenta exponencialmente.

Por tanto el desafío consiste en saber qué balance entre planificación y movimientos (poco informados) es la más conveniente para cada problema. A veces es más conveniente moverse sin saber con seguridad para donde hacerlo, ya que a fin de cuentas, eso puede terminar ofreciendo más información que quedarse mucho tiempo planificando.

El riesgo, por supuesto, es que se podría desviar hacia una dirección equivocada y por consecuencia se pierda tiempo volviendo sobre tus pasos.

3.2 Off-line versus On-line

Existe cierto debate sobre cual es la opción más conveniente a utilizar. Naturalmente realizar la planificación y seguidamente la ejecución una única vez, que es la manera de operar de los algoritmos off-line, produce trayectorias óptimas y de mejor calidad. Sin embargo, el tiempo de CPU usado en la planificación de este modo generalmente es mayor que el usado por la forma donde se intercala repetidas veces la planificación y la ejecución de los movimientos, que es el modo de operar de los algoritmos on-line, porque todo el esfuerzo dedicado a la anticipación produce un solo movimiento. Por esto, planificar varios movimientos es una opción atractiva que ha sido investigada desde diferentes enfoques [Koening, 2004] .

Hacer la planificación de manera off-line es una estrategia conservadora. El agente puede calcular la mejor trayectoria hacia un estado en la frontera del espacio local de búsqueda, pero desde una perspectiva global no está seguro si esta trayectoria efectivamente lo acerca al estado objetivo, o lo lleva por una ruta errónea hacia un obstáculo (la ruta errónea se conoce después de algunos movimientos, a medida que el entorno es descubierto). En esta situación puede ser adecuada una estrategia de mínimo compromiso y planificar una sola acción: moverse desde el estado actual hacia el mejor de los sucesores inmediatos.

Por otro lado si la planificación la realizamos de manera on-line resulta ser una estrategia arriesgada. Dado que el espacio local de búsqueda es pequeño respecto a todo el espacio de estados, no está claro si la mejor trayectoria dentro del espacio local de búsqueda también es buena a nivel global. Seguir la mejor trayectoria en el espacio local es arriesgado porque puede no ser buena a nivel global, y si finalmente es errónea, debido a que incluye varios movimientos, se requerirá cierto esfuerzo en volver atrás. Por otro lado, si la trayectoria es buena, realizar varios movimientos por etapa de planificación es mejor que realizar un único movimiento, porque el agente se aproxima más al estado objetivo.

3.3 Rendimiento de los algoritmos

En la primera ejecución, RTA* exhibe el mejor desempeño seguido por HLRTA* y LRTA*. Esto se debe a, como ya habíamos visto en el capítulo anterior el RTA* debido a su característica de escoger el segundo mínimo para actualizar su tabla de hash, la misma le da la posibilidad de ser mucho más rápido en comparación con los otros dos aunque esta particularidad le anula completamente la capacidad de aprendizaje, esto resulta ser una gran desventaja para este algoritmo, seguidamente viene el algoritmo HLRTA* que debido a su característica de ser un algoritmo híbrido, es decir, una combinación entre el RTA* y el LRTA* al tomar la característica del RTA* de escoger el segundo mínimo para actualizar la tabla también adquiere al igual que el mismo la posibilidad de ser un algoritmo rápido, aunque nunca más que el primero, este rasgo distintivo provoca que se mejore la fase de planificación del algoritmo HLRTA* manteniendo su capacidad de aprendizaje, por último está el LRTA* debido a que al escoger siempre el mínimo absoluto como candidato para actualizar la tabla de hash esto le da como beneficio una original capacidad de aprendizaje y una exitosa convergencia a la solución óptima pero como desventaja, esta característica lo convierte en un algoritmo más lento, lo cual hace que surjan otras versiones para minimizar este evento tan poco favorable. Por tanto podemos concluir diciendo que en convergencia hacia soluciones óptimas LRTA* exhibe un mejor desempeño que HLRTA*. [Thorpe, 1994], [Furcy y Koening, 2000], [Shimbo e Ishida, 2003]. Teniendo en cuenta el análisis realizado en el capítulo 2 de los algoritmos de búsqueda heurística en tiempo real hasta hoy conocidos, se puede llegar a la conclusión que aún cuando los algoritmos que surgieron después del LRTA* presentan algunas ventajas más favorables que el mismos, por ejemplo, son superiores en cuanto a su fase de planificación así cómo también de su velocidad, la propuesta del algoritmo para ser incorporado a la biblioteca de Inteligencia Artificial se inclina al LRTA* debido a que como es un algoritmo simple, es decir no está concebido por medio de ninguna combinación de otros algoritmos, además de presentar algo muy importante en este tipo de algoritmo que es el aprendizaje ya que posibilita que tienda a converger de manera original a soluciones óptimas a pesar de su gran limitante que es su velocidad. El LRTA* debe su capacidad de aprendizaje a su manera única de hacer sus actualizaciones en la tabla de hash del estado anterior, el mismo lo realiza escogiendo el mejor nodo, es decir, el que contenga el coste menos elevado.

Este algoritmo resulta ser fácil de implementar debido a que a diferencia del RTA* incluye las dos fases necesaria para realizar el recorrido, me refiero a la fase de planificación y la fase de ejecución respectivamente. Además, a diferencia del algoritmo HLRTA* que utiliza dos valores heurísticos así como el algoritmo FALCONS que utiliza dos tablas para actualizar los valores heurísticos, el LRTA* utiliza sólo un único valor heurístico lo cual facilita el trabajo del programador.

3.4 Estabilidad del proceso de convergencia a soluciones óptimas

En [Shimbo e Ishida, 2003], se critica la inestabilidad en el proceso de convergencia a soluciones óptimas en LRTA*. Experimentalmente se observa que la convergencia no es monótona: una ejecución puede requerir un mayor número de acciones o movimientos que la ejecución anterior, sobre el mismo ejemplar de problema. Esto se traduce de la siguiente manera, debido a que el algoritmo puede comenzar inicialmente con valores heurísticos no admisibles necesitaría realizar repetidas exploraciones sobre el espacio de búsqueda, aunque debido a su característica de poseer capacidad de aprendizaje, esto de algún modo me garantizaría que a medida que vaya recorriendo todo el espacio aprenda y esto haría que se llegue a alcanzar la convergencia a valores óptimos que es precisamente lo que esperamos de él, aunque el tiempo para hallar estos valores está sujeto a los valores que toma la heurística en la primera exploración. Todo esto nos llevaría a pensar que este algoritmo no es la mejor opción, pero podemos garantizar que es una conclusión errónea ya que como asegura que aprende de sus exploraciones entonces asegura además que siempre va a convergir a la optimalidad. A continuación se hará un análisis de todos los algoritmos en tiempo real con el algoritmo LRTA* para que quede justificada la elección del algoritmo.

3.4.1 LRTA* versus RTA*

RTA* debido a su característica de tomar el segundo mínimo para actualizar su tabla de funciones heurísticas realiza un mejor desempeño que el algoritmo LRTA* en su primera fase de ejecución aunque después de repetidas exploraciones no garantiza la convergencia a soluciones óptimas, no así, el LRTA* que debido a su capacidad de aprendizaje nos garantizará la convergencia a soluciones óptimas. Otro punto a favor del LRTA* es que requiere menos recursos computacionales que el algoritmo RTA*.

3.4.2 LRTA* versus HLRTA*

El algoritmo HLRTA* cuando realiza la búsqueda en el espacio de estado si parte de valores heurísticos admisibles garantiza al igual que el LRTA* la convergencia a soluciones óptimas. Debemos tener en cuenta que en la primera fase de planificación del HLRTA*, éste realiza un mejor desempeño que el LRTA*, esto se debe a que presenta características similares al RTA* ya que es un híbrido entre LRTA* y RTA*, aunque requiere de una mayor cantidad de repeticiones que el LRTA* para lograr la convergencia a soluciones óptimas, esto se puede ver con mas detalles en [Furcy and Koening, 2001].

3.4.3 LRTA* versus FALCONS

El algoritmo FALCONS [Furcy and Koening, 2000] al igual que el LRTA* garantiza la convergencia a soluciones óptimas debido a que su fase de planificación la realiza muy similar al algoritmo A*, ya que utiliza la función de evaluación heurística f(n), calculándola de la siguiente manera f(n) = h(n) + g(n), lo que garantiza que FALCONS mejore la convergencia pero a su vez éste puede realizar una gran cantidad de exploraciones en fases anteriores, lo cual trae como consecuencia una perdida de tiempo y de recursos computacionales.

3.4.4 LRTA* versus eFALCONS

El algoritmo eFALCONS [Furcy and Koening, 2001] como vimos en el capítulo anterior es un híbrido entre HLRTA* y FALCONS. Este algoritmo al igual que HLRTA* y FALCONS converge a soluciones óptimas, este algoritmo realiza una cantidad más pequeña de acciones en fases anteriores que el FALCONS, por tanto garantiza realizar un mejor desempeño que el mismo en las primeras fases de ejecución, pero puede realizar una cantidad de acciones en sus anteriores fases superior a las realizadas en LRTA*.

3.5 Resultados del análisis

Hasta el momento la idea de desarrollar todos los algoritmos a los que anteriormente se le hizo mención fue la de mejorar tanto el rendimiento de la primera ejecución como la capacidad de aprendizaje del algoritmo LRTA*. Lo resultados expuestos en [Furcy and Koening, 2001] muestran que HLRTA* seguido del RTA* , es el algoritmo que realiza el menor número de acciones en la primera ejecución, es decir, que en la primera fase de ejecución/planificación éste muestra un mejor desempeño que el LRTA*, esto queda evidenciado en el capítulo 2 donde se hace referencia a todos los algoritmos de búsqueda de caminos en tiempo real pero además también podemos constatarlo el la comparación que se realizó en el capítulo 3 de estos algoritmos con el LRTA*, también se pudo llegar a la conclusión de que FALCONS y eFALCONS son los que realizan el mejor aprendizaje, debido a que gracias a varias técnicas aplicadas como la combinación de otros algoritmos se pudo lograr una mejora en la estrategia de actualización de la tabla de valores heurísticos, por lo que estos dos algoritmos tienen un número similar de acciones para converger. También nos pudimos dar cuenta de que los algoritmos analizados mejoran su convergencia al aumentar la profundidad aunque el tiempo de ejecución aumenta exponencialmente.

Por tanto podemos decir que los algoritmos surgidos posterior al LRTA* mejoran alguna de sus característica, pero en ningún caso se encontró un algoritmo capaz de mejorar todas a la vez, por lo que queda demostrado una vez más su superioridad.

A continuación se ofrece una gráfica para entender mejor todo lo explicado anteriormente.

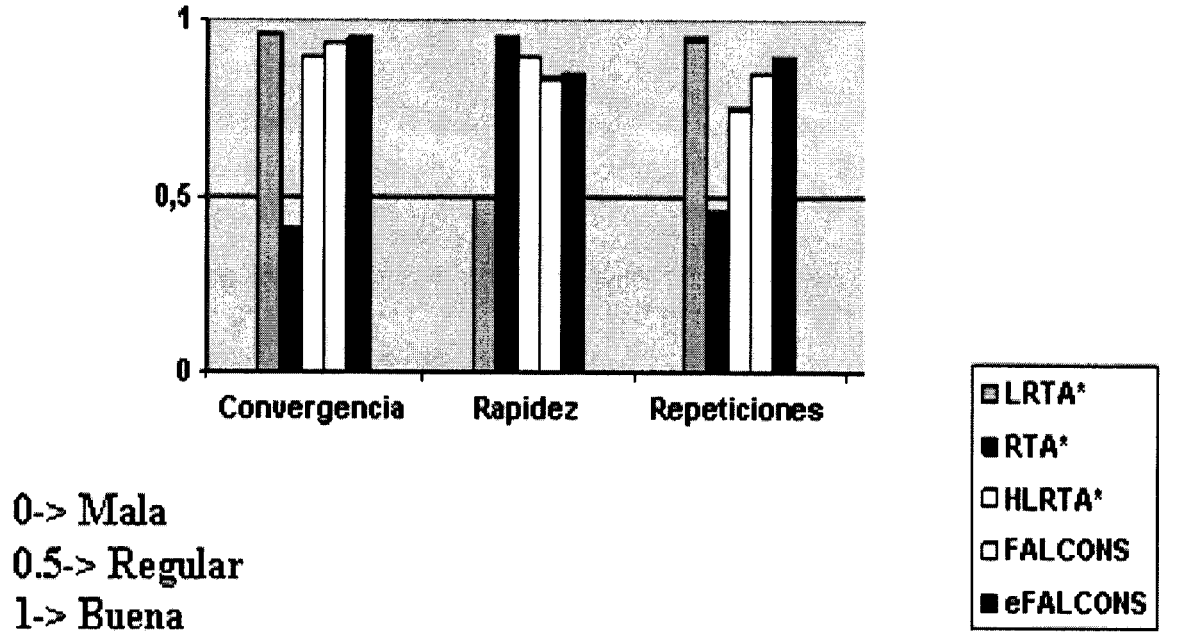


Figura 15: Representación del rendimiento de cada algoritmo

3.6 Descripción de la aplicación

El modelos cognitivo que se utilizó en la aplicación fue la rejilla regular, ya que es la ideal para ser utilizada en un entorno virtual 2D, además de tener las características de ser ampliamente utilizado en juegos con sistemas en tiempo real (STR), es muy bueno en el diseño de grafos de navegación.

Al dividir el área en polígonos nos da una mejor información sobre cada celda, donde cada una de ella representa un valor, que será, transitable o intransitable, donde el valor intransitable contendrá la información del tipo de obstáculo, que podrían ser, árboles, murallas, enemigos, el jugador, casas, terrenos entre otras

cosas. Por estos valores la Inteligencia Artificial puede planear donde moverse, donde localizar enemigos o cómo evitar obstáculos.

El ambiente de la aplicación tendrá la siguiente apariencia:

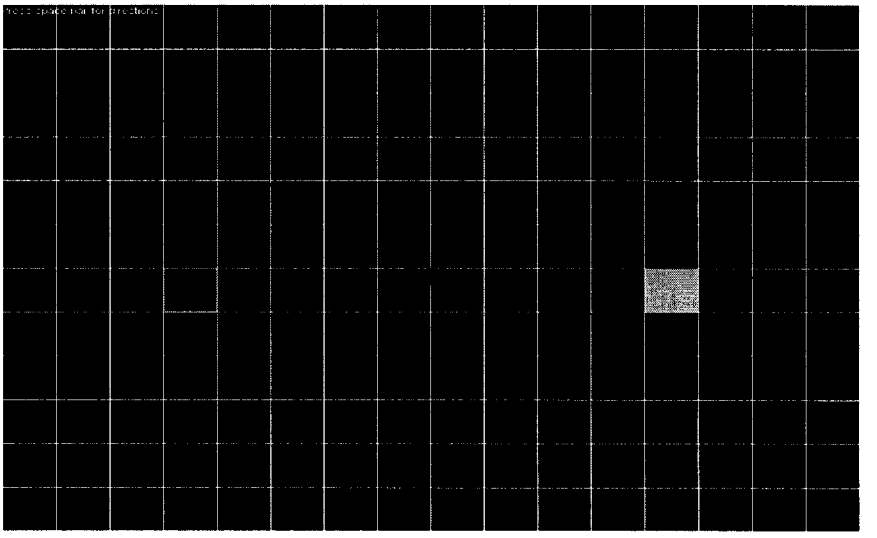


Figura 16: Representación de la aplicación al ser iniciada

Los obstáculos en el ambiente serán ubicados a conveniencia del usuario al dar un clic sobre cualquier celda libre del ambiente y se eliminarán al dar otro clic sobre el obstáculo ya creado. Al finalizar tendrá una apariencia similar a la de la siguiente imagen:

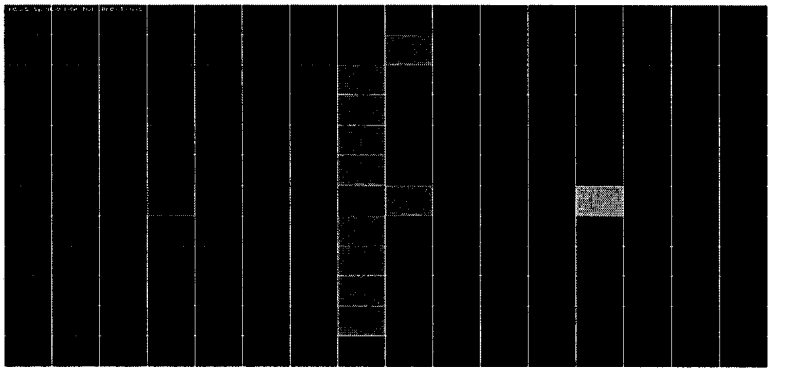


Figura 17: Representación de la aplicación luego de ubicar los obstáculos

De esta manera queda argumentada la elección en cuanto al modelo cognitivo.

Aunque se había escogido el algoritmo LRTA* como el más eficiente de los algoritmos analizados, el que se seleccionó para la aplicación fue el reconocido AStar o A*, ya que de esta forma estaría probando que si el algoritmo escogido para la aplicación funcionó de manera eficiente entonces si se hubiese realizado con el LRTA* la eficiencia hubiese sido mayor debido a todas las características que presenta el algoritmo y que fue explicada anteriormente.

El objetivo principal de este trabajo fue incorporar a la biblioteca de Inteligencia Artificial de la facultad 5 el algoritmo de búsqueda de caminos más adecuado y eficiente, luego de un intenso estudio de las características de cada uno de ellos se llegó a la conclusión de que es el algoritmo LRTA* el ideal para ser incorporado a dicha biblioteca, además de proponer este algoritmo se implementó en la aplicación demostrativa el algoritmo A*, con el fin de que ambos sean incorporados, ya que como nuestra facultad se especializa en la rama de la realidad virtual y se ha trazado como propósito la creación de juegos virtuales, es importante que en dichos juegos se halle en momentos determinados el camino más corto, por tanto se propone la idea de utilizar el LRTA* en todo momento, pero cuando se necesite en un momento determinado hallar un camino más preciso se utilizará el A*, ya que posee la característica de ofrecer una solución óptima.

La solución del algoritmo sobre la aplicación escogida quedaría de la siguiente manera:

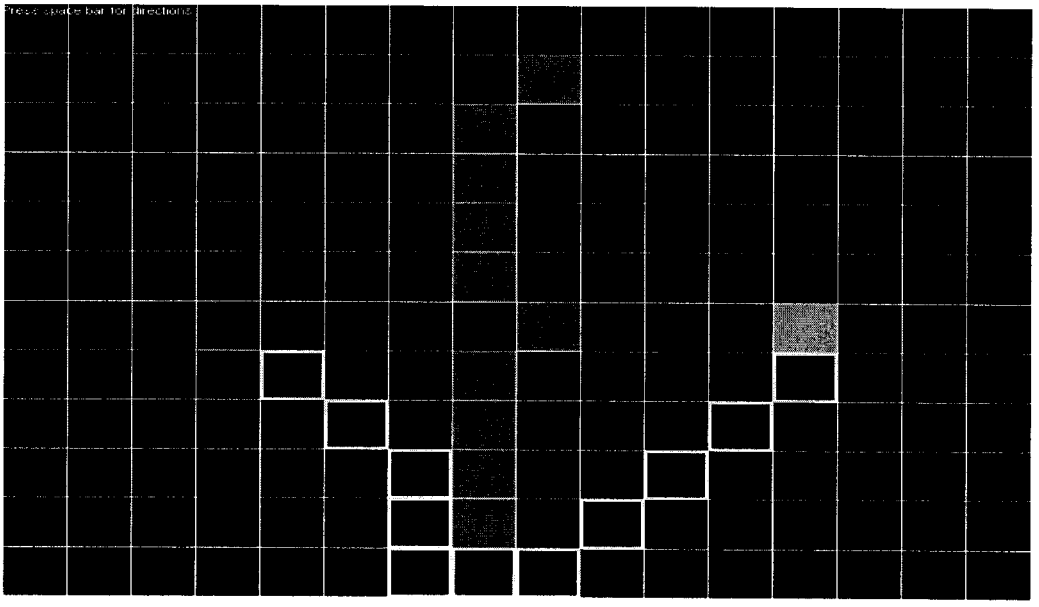


Figura 18: Representación de la aplicación luego de hallar una solución

3.7 Características específicas de la aplicación

- El demo encuentra el camino más corto entre el cuadrado rosado que representa el origen y el verde que representa el destino.
- Con el clic izquierdo del Mouse se ubican las paredes u obstáculos a necesidad del usuario.
- La búsqueda comienza al presionar la tecla **enter.**
- Después que se encuentra el camino, se puede resetear el entorno con un clic del Mouse, y crear nuevamente con el Mouse nuevas paredes u obstáculos y hallar una vez más el camino más corto.
- Presionando el número 1 hace una búsqueda paso a paso. Continúe presionando 1 para cada nuevo paso. Presione **enter** para terminar la búsqueda paso a paso todos a la vez.
- Con el clic izquierdo crea nuevos cuadrados grises que representan las paredes u obstáculos.
- Borra los obstáculos creados al presionar nuevamente clic izquierdo sobre ellos.
- Mueve el cuadrado verde u objetivo alrededor de la pantalla con sólo presionar clic derecho.
- Mueve el cuadrado rosado u origen alrededor de la pantalla al presionar la letra "g" del teclado junto al clic izquierdo.

Al realizar todas estas instrucciones la aplicación responde de la siguiente manera:

- El camino más corto es marcado por puntos azules.

3.8 Lenguaje y herramientas utilizadas en la aplicación

La aplicación se implementó en C++ debido a que es el lenguaje utilizado en todas las aplicaciones que serán incorporadas en la biblioteca de Inteligencia Artificial de la facultad 5. Además por ser un lenguaje eficiente, presenta gran flexibilidad, debido a que tiene relacionado los procedimientos que manipulan los datos con los datos a tratar, por tanto cualquier cambio que se realice sobre ellos quedará reflejado automáticamente en cualquier lugar donde estos datos aparezcan, presenta estabilidad debido a que permite un tratamiento diferenciado de aquellos objetos que permanecen constantes en el tiempo, sobre aquellos que cambian con frecuencia, permite aislar las partes del programa que permanecen inalterables en el tiempo. Otras de las grandes ventajas del C++, es que es un lenguaje multi-nivel, es decir, puedes usarlo tanto para programar directamente el hardware (dependiendo del sistema operativo), como para crear aplicaciones tipo Windows identificadas por poseer una misma interfaz.

Como entorno de desarrollo, el Visual Studio, ya que es una herramienta poderosa y fuerte que tiene una gran integración de varios lenguajes, entre ellos, el C++, C#, y Asp.Net. Tiene la posibilidad de implementar aplicaciones para soluciones integrales que aprovechen de manera óptima la ventaja de cada lenguaje, además de tener una interfaz amigable con el usuario.

3.9 Pruebas realizadas

La PC donde se probó la aplicación presenta las siguientes características:

Procesador: 2.40 GHz

Tarjeta: asrock-p4i65g

Video: 64 MB

RAM: 248 MB

SO: Microsoft Windows XP

Aunque no presenta limitaciones para ser probado en cualquier máquina sea cuales sean sus características.

Conclusiones:

Al finalizar esta investigación quedaron cumplidos satisfactoriamente los objetivos trazados en la misma, ofreciendo como resultado una bibliografía abundante y actualizada sobre las estrategias de búsqueda de caminos que podrá ser utilizada en investigaciones posteriores. Se propuso un algoritmo para ser integrado a la biblioteca de Inteligencia Artificial de la facultad 5, además se realizó una aplicación demostrativa con uno de los algoritmos analizados, el A*, sobre una rejilla regular, para demostrar su funcionamiento, con el fin de que éste también sea incorporado a dicha biblioteca.

Recomendaciones:

Para futuros trabajos se recomienda:

Realizar la implementación del algoritmo LRTA* para ser incorporado a la biblioteca de Inteligencia Artificial.

Actualizar la investigación realizada con los nuevos algoritmos de búsqueda que surjan, teniendo como referencia los de propagación acotada.

Referencia Bibliográfica:

[Aho, 1983] Aho, Alfred V.; Hopcroft, John E.; Ullman, Jeffrey D., Data Structures and Algorithms, Addison-Wesley Publishing Company, 1983

[Board, 2002] Board, Ben, and Ducker, Mike, "Area Navigation: Expanding the Path-Finding Paradigm," Game Programming Gems 3, Charles River Media, 2002.

[Furcy, 2000] D. Furcy, S. Koening. "Speeding up the Convergence of Real-Time Search." In Proceedings of the National Conference on Artificial Intelligence. 891-897. 2000.

[Furcy, 2001] D. Furcy, S. Koening. "Combining Two Fast-Learning Real-Time Search Algorithms Yields Even Faster Learning." In Proceedings of the 6th European Conference on Planning. 2001.

[Furcy y Koening, 2000] D. Furcy y S. Koenig. "Speeding up the convergence of real-time search". In Proceedings of AAAI. 2000.

[Furcy y Koening, 2001] D. Furcy y S. Koenig, "Combining two fast-learning real-time search algorithms yields even faster learning". In Proceedings of the 6th European Conference on Planning. 2001.

[Higgins, 2002] Higgins, Daniel F. "How to Achieve Lightning-Fast A*". AI Game Programming Wisdom. Charles River Media, 2002.

[Ishida, 1997] T. Ishida. "Real-time search for learning autonomous agents". Kluwer Academic Publishers. 1997.

[Koening, 1999] S. Koening. "Exploring Unknown Environments with Real-Time Search or Reinforcement Learning". In Advances in Neural Information Processing Systems. Volume 11. 1003-1009. 1999.

[Koening, 2001] S. Koening. "Agent-Centered Search". Artificial Intelligence Magazine. Winter. 2001.

[Koening, 2004] S. Koenig. "A comparison of fast search methods for real-time situated agents." In Proceedings of the 3rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS), 2004.

[Korf, 1985] Korf, R., Depth-first iterative deepening: an optimal admissible tree search. Artificial Intelligence 27, pp. 97–109. 1985.

[Korf, 1990] R. Korf. "Real-time heuristic search". Artificial Intelligence 42(2-3):189-211. 1990.

[Nareyek, 2004] Nareyek, Alexander, "AI in Computer Games", Game Development, Vol 1, 10, 2004.

[Rabin, 2000] Rabin, Steve, "A* Speed Optimizations" and "A* Aesthetic Optimizations," Game Programming Gems, Charles River Media, 2000.

[Shimbo e Ishida, 2003] M. Shimbo y T. Ishida. "Controlling the learning process of real-time heuristic search." Artificial Intelligence, 146(1):1–41, 2003.

[Snook, 2000] Snook, Greg, "Simplified 3D Movement and Pathfinding Using Navigation Meshes," Game Programming Gems, Charles River Media, 2000.

[Stout, 2000] Stout, Bryan, "The Basics of A* For Path Planning," Game Programming Gems, Ed. Mark DeLoura, Charles River Media, 2000.

[Thorpe, 1994] P. Thorpe. "A Hybrid Learning Real-Time Search Algorithm." Master's thesis, Computer Science Department, University of California at Los Angeles, Los Angeles (California). 1994.

[Tozour, 2002] Tozour, Paul, "Building a Near-Optimal Navigation Mesh," AI Game Programming Wisdom, Charles River Media, 2002.

[White, 2002] White, Stephen, and Christensen, Christopher, "A Fast Approach to Navigation Meshes," Game Programming Gems 3, Charles River Media, 2002.

Bibliografía Citada:

Alexander Nareyek (Febrero 2004) "AI in Computer Games " Nareyek (Game Development Vol 1, 10) David M. Bourg, G. S. (2004). AI for Game Developers, O'Reilly.

Fernández, M. O. (2003). "Algoritmos de búsqueda heurística en tiempo real. Aplicación a la navegación en los juegos de videos." 21.

Higgins Daniel F. (2002). "Generic Pathfinding", AI Game Programming Wisdom.

Higgins, Daniel F. (2002) "How to Achieve Lightning-Fast A*", AI Game Programming Wisdom.

Rabin Steve (2000) "A* Speed Optimizations" Game Programming Gems, Charles River Media.

Richard E. Korf. (Marzo 1990) "Real - Time Heuristic Search " (Artificial Intelligence, 42).

http://www.iiia.csic.es/~chernan

http://deadchannel.blogsome.com/2007/07/15/modelo-vi-pathfinding-y-aplicaciones

http://www.policyalmanac.org/games/aStarTutorial.htm

http://www.gameai.com

http://www.cip.ifi.lmu.de/~proescho/progs/ia/p3/Enunciado_practica_3.pdf

http://www.cristalab.com/tips/43162/pathfinding-en-actionscript-usando-waypoints

http://www.ldc.usb.ve/~meza/ci-2617/a-j2006/proyecto2.pdf

http://yobtrams.wordpress.com/2007/10/25/algoritmos-de-busqueda

Glosario de términos:

Arista: Segmento que une los nodos conectados en la estructura "Grafo".

Convergencia: Es la acción de aproximarse al valor real esperado luego de seguir repetidas acciones.

Entidades Autónomas: Agentes dentro de un entorno virtual que perciben y actúan sobre el mismo.

Grafo: Brinda información acerca de la estructura del entorno, es ideal para la aplicación de algoritmos de Inteligencia Artificial como la búsqueda de caminos.

Inteligencia Artificial: Se denomina Inteligencia Artificial a la rama de la informática que desarrolla procesos que imitan el comportamiento y la inteligencia de los seres humanos.

Nodo: Elemento que representa una posición en la estructura "Grafo".

Pathfinding: El término Pathfinding se refiere tanto al problema general de encontrar una serie de pasos para llegar a un estado objetivo partiendo de un estado inicial, en el contexto de un problema cualquiera, como al problema específico de la navegación.

Realidad Virtual: Es una representación de las cosas a través de medios electrónicos, que nos da la sensación de estar en una situación real en la que podemos interactuar con lo que nos rodea.

Tiempo Real: Los algoritmos de búsqueda en tiempo real también se conocen como algoritmos de búsqueda de profundidad limitada, que abarca a todos los algoritmos que intercalan búsqueda y ejecución, y son aquellos que responden en tiempo constante.