UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN

El algoritmo Fringe Search como solución superior a A* en la búsqueda de caminos sobre gráficos de malla

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

Licenciado en Informática

PRESENTA:

Jesús Manuel Mager Hois

ASESORA:

Mtra. GARCÍA VARGAS ADRIANA

México, D.F. 2015

Universidad Nacional
Autónoma de México

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Índice general

Índice de Algoritmos **3**

Índice de figuras **4**

Introducción **7**

1. Marco teórico **15**

1.1. Teoría de gráficos y definiciones formales 15

1.2. La búsqueda de caminos sobre gráficos desde la inteligencia artificial 17

1.2.1. Búsqueda no informada 18

1.2.2. Búsqueda informada 20

1.3. El Algoritmo Dijkstra 21

2. Marco Contextual **24**

2.1. El algoritmo A* 24

2.1.1. Planteamiento formal 24

2.1.2. La estimación heurística 25

2.1.2.1. Distancia Manhattan 26

2.1.2.2. Distancia de Chebyshov 27

2.1.2.3. Distancia Euclidiana 27

2.1.2.4. Algunas notas sobre la función heurística 28

2.1.3. El algoritmo 28

2.1.4. Los tipos de datos usados para el algoritmo 30

2.1.4.1. Listas ordenadas 31

2.1.4.2. Binary Heap 31

2.1.4.3. Skiplist 32

2.1.4.4. Tablas hash y arreglos . . . 32
2.1.5. Implementación mixta . . . 32
2.1.6. Algunas notas sobre la implementación . . . 33
2.1.7. Análisis del código fuente . . . 33
2.2. Fringe Search . . . 39
2.2.1. IDA* . . . 39
2.2.2. Mejorando IDA*: Fringe Search . . . 40
2.2.2.1. Heurística . . . 41
2.2.2.2. Apuntes sobre los caminos generados . . . 41
2.2.3. Implementación . . . 42
2.2.4. Análisis del código fuente . . . 43

3. Método y técnica **50**
3.1. El método . . . 50
3.1.1. El método dialéctico . . . 50
3.1.2. Tipo de estudio . . . 52
3.1.3. Fuentes . . . 52
3.1.4. Indicadores y variables . . . 53
3.2. La técnica utilizada . . . 53

4. Resultados **55**
4.1. Comparación entre A* y Fringe Search . . . 55
4.1.1. La experimentación . . . 55
4.1.1.1. El entorno de experimentación . . . 55
4.1.1.2. El experimento . . . 56
4.1.1.3. Presentación visual . . . 57
4.1.2. Comportamiento . . . 58
4.1.2.1. Sobre un mapa de una sola medida . . . 58
4.1.2.2. Sobre mapas aleatorios de medidas diversas . . . 59
4.1.2.3. Regresiones . . . 60
4.1.3. Análisis . . . 62
4.1.3.1. Velocidad . . . 62

4.1.3.2. Camino . . . 64
4.2. Aplicaciones en casos prácticos . . . 66
4.2.1. Búsqueda de mejores caminos . . . 66
4.2.2. Caminos en el mundo real . . . 68
4.2.2.1. Problemas de transporte . . . 69
4.2.2.2. Robótica . . . 70
4.2.3. Juegos de vídeo . . . 72
4.2.4. Otros usos . . . 73

Conclusiones **74**

Anexos **80**

Índice alfabético **99**

Bibliografía **102**

Índice de Algoritmos

1. Depth-First Search (DFS) 19
2. Breadth-first search (BFS) 20
3. Dijkstra 23
4. A* 29
5. Fringe Search 49

Índice de figuras

1.1. Gráfico de Lattice o de malla 16

1.2. Algoritmo Djikstra 22

2.1. Implementación de las listas *now* y *later*. 43

4.1. Camino encontrado por A* en un gráfico 100x100. 57

4.2. Camino encontrado por Fringe Search en un gráfico 100x100. 58

4.3. Comportamiento de A* y Fringe Search con respecto al número de nodos del gráfico y al tamaño del camino. 59

4.4. Comportamiento del tiempo de ejecución de los dos algoritmos con respecto al tamaño del gráfico. 60

4.5. Residuos y datos relevantes para A*(arriba) y Fringe Search(Abajo) 61

4.6. Comparación del tiempo de ejecución entre A* y Fringe Search sobre los mismos gráficos. 63

4.7. Tabla ANOVA: relación entre la velocidad y los dos algoritmos 64

4.8. Comparación de los caminos generados A* y Fringe Search en los mismos gráficos. 65

4.9. Tabla ANOVA: Relación entre el tamaño del camino y los dos algoritmos 65

4.10. Búsqueda de caminos en MapQuest usando OpenStreetMap. La imagen fue tomada de la wiki de OSM 70

4.11. Búsqueda del mejor camino de robots esclavos 71

4.12. Agentes del juego libre TuxHistory buscando el mejor camino para llevar la cosecha del campo al granero. 73

Dedicatorias

A mi madre Elisabeth Mager por haberme ofrecido todas las oportunidades, apoyo y cariño para la optimización en este intrincado gráfico llamado vida.

Agradecimientos

Agradezco al pueblo mexicano que ha hecho posible que estudie en la Universidad Nacional Autónoma de México, institución pública y gratuita, de una calidad extraordinaria y que aún se encuentra al servicio de todos; a todos los profesores, trabajadores y alumnos de la UNAM que han sido la comunidad que me ha formado desde el CCH Naucalpan, la FES Acatlán y la Facultad de Contaduría y Administración; pero sobre todo a aquellos que han puesto sus esfuerzos en conservar su espíritu crítico y abierto.

Quiero hacer mención especial del Lic. Carlos Pineda, profesor de la Facultad de Estudios Superiores Cuautitlán que me ha dado dirección en el estudio del tema de búsqueda de mejores camino, sobre todo en el área paralela, a mi asesora de tesis Mtra. Adriana García Vargas que ha dedicado un gran esfuerzo en apoyarme en este trabajo, al Dr. Iván Vladimir Meza Ruiz del IIMAS que aportó su orientación en la definición del tema, además de correcciones al trabajo y a mi madre la Dra. Elisabeth Mager que me enseño desde joven a realizar investigaciones exhaustivas y los métodos para alcanzar los objetivos académicos.

También agradezco a la comunidad de Software Libre en el mundo, y en especial al proyecto Tux4Kids que me introdujo en este increíble mundo de la inteligencia artificial aplicada a juegos que sirven en la enseñanza para niños en todo el mundo. También quiero agradecer al grupo de Software Libre de la FES Cuautitlán y el apoyo con el diseño gráfico por parte de Rebeca Guerrero.

Introducción

En un mundo virtual, que puede representar un espacio de la realidad, existen posibles movimientos, y encontrar los mejores caminos para realizar un desplazamiento dentro de este entorno es un tema que ha sido intensamente estudiado por la inteligencia artificial y las matemáticas discretas. El primer trabajo que marcó permanentemente esta área de estudio fue el algoritmo elaborado por el físico Edsger Wybe Dijkstra(Dijkstra, 1959) en 1959, donde soluciona el problema del camino más corto con un nodo de inicio a todos los nodos dentro de un gráfico, conocido como "Algoritmo Dijkstra".

Hoy en día el problema se encuentra resuelto con algoritmos secuenciales que han trabajado para encontrar un camino óptimo entre dos nodos de un gráfico, siendo A* el algoritmo estándar para este problema. Sin embargo, el tiempo de cálculo, incluso de las soluciones óptimas, sigue ocupando una parte relevante(si no es que crítica) de la ejecución en las aplicaciones en que se usan obligando a investigar nuevas mejoras en su desempeño. El problema se acentúa por la constante necesidad de trabajar en gráficos más cercanos a la realidad y con mundos virtuales más grandes. El desarrollo de los procesadores en materia de velocidad se ha reducido considerablemente, por lo que se hace necesario mejorar los algoritmos ya existentes. Si se hace un recuento de 1986 a 2002 se nota que promedio los procesadores incrementaron su rendimiento un 50 %, pero de 2002 a 2005 la velocidad de mejora se ha reducido a 20 %, y con tendencia a reducirse aún más.(Pacheco, 2011, p. 1).

Problemática.

Desde el primer planteamiento del problema del mejor camino entre dos nodos sobre un gráfico se ha trabajado por encontrar el algoritmo óptimo, lo cual se logró principalmente con A*, el cual fue planteado por primera vez en 1968 por Hart, Nilsson y Betram (Hart, Nilsson, y Raphael, 1968), y ha demostrado ser, hasta ahora, el mejor algoritmo en obtener de la manera más rápida un camino de costo mínimo, si es que este existe. Lo hace con el uso de funciones

heurísticas que permiten evaluar el costo de llegar a un nodo dado y el costo del camino que aún falta recorrer para llegar al objetivo. Para lograr escoger los mejores candidatos entre los nodos propuestos lleva acabo un ordenamiento de los mismos en una lista de prioridad. Mantener ordenada esta lista es una operación costosa y la parte más discutida del algoritmo. La práctica ha demostrado que es necesario buscar soluciones superiores a la existente para mejorar el rendimiento de las aplicaciones.

Gran parte de los trabajos realizados hasta la fecha se han centrado en encontrar mejoras para A*, sin realmente alcanzar un avance significativo. Fue en 2005 cuando Böjrnsson de la universidad de Reykjavik y Enzenberger, Holte y Schaeffer de la Universidad de Alberta desarrollaron un algoritmo secuencial llamado Fringe Search(Björnsson, Enzenberger, Holte, y Schaeffer, 2005) que supera a A* de 10 % a 40 % (basado en sus propios datos) en casi todos los experimentos pragmáticos sobre gráficos de dos dimensiones, con la desventaja de no obtener el camino más corto en todos los casos.

A pesar de la mejora aparente expuesta por los creadores de Fringe Search, la industria aún no ha adoptado a este de una manera amplia, tomando A* cómo la opción por defecto. Resta plantear la cuestión de comprobar el referido incremento en el rendimiento de Fringe Search, saber si es significativo y si el camino resultante del mismo es aceptable para ser implementado en la práctica como un sustituto de A*.

Justificación.

La búsqueda del mejor camino para un agente en un gráfico, intentando llegar de un nodo de salida a un nodo de llegada, es una tarea común en las simulaciones, la robótica, los juegos, y en las aplicaciones de mapas. Todas estas implementaciones necesitan una cantidad considerable de su tiempo de ejecución para poder llevar acabo la búsqueda por lo que es considerado una de las partes que más intensamente utiliza la CPU. Mejorar su rendimiento es un objetivo al que se aspira constantemente y se intenta mediante optimizaciones, precalculo de caminos o incluso con nuevos algoritmos.

Con Dijkstra que tiene una complejidad de $O(n^2)$ en implementaciones no optimizadas y $O(|E| + |V| \log |V|)$ con un heap de Fibonacci(Barbehenn, 1998), hasta A* que llega a $O(bd)$ en el mejor de los casos(Heineman, Pollice, y Selkow, 2008) se ve que es necesario mejorar los resultados para desarrollar aplicaciones adecuadas a los requerimientos actuales.

Poder encontrar mejoras o algoritmos nuevos que permitan optimizar el tiempo de ejecución daría la oportunidad a las plataformas de juegos interactuar con mundos más extensos, detallar los sistemas de información geográfica (GIS) y el movimiento en ellos. Por ello, el estudio del problema de mejorar la búsqueda de caminos es de gran importancia para la ciencia de la computación.

Objetivos

Esta tesis, partiendo de la importancia que representa la discusión de la búsqueda del mejor camino para el desarrollo de la ciencia de la computación, y entendiendo que es necesario mejorar el rendimiento de las soluciones existentes, tomando en cuenta la delimitación del tema, pretende alcanzar los siguientes objetivos:

Objetivo principal: Encontrar una vía para poder mejorar el rendimiento en cuanto a tiempo de ejecución de la búsqueda de caminos en un gráfico de malla.

Objetivos secundarios:

- Exponer las soluciones existentes para tratar el problema de encontrar un camino entre dos nodos dentro de un gráfico de malla, principalmente A* y Fringe Search.
- Analizar los dos algoritmos, sus limitaciones y ventajas.
- Implementar las versiones secuenciales discutidas.
- Llevar acabo experimentos sobre diversos gráficos, exponer los resultados y plantear posibles ventajas de alguno de los algoritmos.
- Abrir posibles líneas de investigación prometedoras sobre la búsqueda de caminos, tanto en el campo secuencial como en el paralelo.

Hipótesis

Como hipótesis general esta tesis plantea que: *el algoritmo Fringe Search es una solución que reduce el tiempo de ejecución significativamente en comparación con A* y el camino que genera no varía significativamente con respecto al óptimo en la solución del problema de la búsqueda de caminos sobre gráficos de malla.*

El desarrollo de gran parte de la investigación en algoritmia que ha trabajado sobre la búsqueda de caminos en gráficos lo ha realizado basados en A*, por lo que la visión actual de que *A* es el algoritmo idóneo para encontrar el mejor camino en gráficos de malla* será la hipótesis alternativa.

Alcances y limitaciones. El presente texto se centrará en trabajar sobre un gráfico de malla, donde cada nodo tiene ocho arcos, cada uno de estos enlazados nuevamente a otros nodos con ocho arcos, a excepción de aquellos que se encuentran en la delimitación de la malla. La malla, también llamada gráfico de *lattice*, tendrá un tamaño de x por y nodos, creando un mapa de dos dimensiones perfecto para representar diversos agentes y sus movimientos generales. Este trabajo se centrará en dos algoritmos que trabajan sobre este tipo de gráficos: A* y Fringe Search, los cuales se compararán tanto en el campo teórico como en el aspecto práctico. El trabajo únicamente analizará los algoritmos de búsqueda de camino más corto(o del que tienda a ser el más corto) entre dos nodos, mencionando a otros únicamente como referencia o de manera explicativa. También se centrará el trabajo en el aspecto de la velocidad en tiempo de ejecución y su aplicación a diversos problemas, pero no se tocará el tema del consumo de memoria, para poder poner la velocidad en el centro del análisis.

Aportaciones

En primer lugar el texto presenta un amplio panorama de la búsqueda de caminos y sobre todo de A*, su heuristica y las cuestiones teóricas que vienen con ello. En la investigación no se encontró ningún texto que presentara de esta manera el problema y el algoritmo, por lo que la recompilación del material presentado es una aportación relevante. La experimentación del uso con *skiplist* también no ha sido realizada por alguna implementación abierta.

En segundo lugar, el algoritmo Fringe Search casi no ha sido explorado por la ciencia de la computación, únicamente se encontró el artículo en el cual se propone, además de algunas referencias al mismo en otros. Tampoco se pudo encontrar una codificación abierta del mismo, los cual lleva a que la presentación e implementación del algoritmo es importante a tomar en cuenta.

El estudio que compara los dos algoritmos, utilizando los métodos estadísticos del capítulo de resultados, no ha sido realizado con anterioridad, y por lo tanto el debate sobre los dos que se lleva acabo en esta tesis es la principal aportación de la misma. Con estos resultados también

se presentan opciones y una discusión sobre el correcto uso de ambos.

Por último, la creación del una implementación de los dos algoritmos bastante optimizada, presentada con una licencia GPL v.3 [1] permite a cualquier interesado estudiar, mejorar y utilizar los dos algoritmos, lo cual representa un importante aporte a la sociedad.

Estructura capitular

El trabajo consiste de cinco capítulos los cuales primero presentan un acercamiento teórico, exponiendo las bases de las matemáticas discretas y de la inteligencia artificial referente al tema que esta tratando para posteriormente entrar en la discusión de los dos algoritmos. Una vez concluida este análisis será necesario tomar, con base a la experiencia empírica decisiones sobre su comportamiento para poder pasar a la aplicación en programas y sistemas computaciones. Ahora se presenta el contenido de cada capítulo.

Marco Teórico En este capítulo inicial se presentarán las bases, las definiciones y los conceptos en la teoría de gráficos de manera formal para poder hacer un acercamiento al problema concreto. Así mismo se realizará una breve exposición del problema desde el punto de vista de la inteligencia artificial y se presentará la diferencia entre las búsquedas informadas y las no informadas, retomando como ejemplos para las primer los algoritmos BFS y DFS, mientras que para las segundas presentaré Dijkstra. Los mencionados algoritmos serán de utilidad para A* y Fringe Search, ya que los dos se basan en ellos.

A* El capítulo hace una exposición del algoritmo A* en la cual se plantearán los términos formales del mismo, dentro de la cual se explicará la manera en que este realiza la estimaciones heurísticas y presentaré las funciones heurísticas más usadas en los gráficos de malla. En la parte de la implementación se expone los diversos tipos de datos que se pueden usar sobre las cuales se llevarán acabo las operaciones del algoritmo y se explicará el porqué en esta implementación se usaron algunas por encima de otras. Por último se presenta el código fuente explicado para dar una idea más detallada al lector de su comportamiento.

Fringe Search Gran parte de los conceptos generados por A* serán también utilizados por Fringe Search, sin embargo, será necesario explicar primero IDA* el cuál plantea las bases del algoritmo a discutir. Encontradas las virtudes y limitaciones de IDA* se presenta Fringe Search y cómo logra superar los mismos, a la vez de que explico el funcionamiento del mismo.

[1] El código fuente se puede encontrar en `https://github.com/pywirrarika`

La implementación tendrá una atención importante, porque gran parte de la velocidad que se pretende generar también depende de las estructuras de datos usadas. Finalmente, al igual que con A*, se realizará un recorrido por las partes más importantes de código fuente para ilustrar su desarrollo.

Comparación entre A* y Fringe Search La experimentación empírica será necesaria para poder entender las ventajas y desventajas de los dos algoritmos, así cómo el poder llegar a conclusiones sobre las hipótesis planteadas, por lo que en este capítulo se plantean los experimentos a realizar junto al entorno de experimentación con el que se llevan acabo, para pasar a una exposición sobre el comportamiento del mismo a través de regresiones. Por último se aplican dos ANOVA tomando la "variables tiempo de ejecuciónz "tamaño del caminoçon respecto al tipo de algoritmo aplicado al generarse estos datos para determinar si estos influyen significativamente en los resultados.

Aplicación en casos práctios Realizo en ese apartado un breve recorrido por algunas aplicaciones comunes de los algoritmos de búsqueda del mejor camino. Para ello muestro la aplicaciones donde los algoritmos Djikstra, Bellman-Ford(Bellman, 1958)(Ford Jr., 1959) y Warshall-Floyd(Warshall, 1962)(Floyd, 1962), los cuales se estudiarán en esta tesis, se deben aplicar y por lo tanto Fringe Search ni A* son pertinentes. Una vez descartados los casos donde otros algoritmos serán necesarios se presentan casos donde A* y Fringe Search deben usarse y en donde Fringe Search puede ser explotado al máximo. Con estos ejemplos quiero hacer incape que la efectividad de cada algoritmo únicamente puede realizarse si se aplica al problema indicado.

Resumen

El problema de la búsqueda del mejor camino sobre gráficos de malla entre dos nodos es resuelto por el algoritmo A*, solución que utiliza la búsqueda informada sirviéndose de funciones heurísticas para estimar los caminos idóneos. Sin embargo, desde su creación el problema se ha quedado estancado y no se ha logrado superar sus tiempos de ejecución en un mundo que ha visto un incremento gigantesco del volumen de datos a procesar. Por lo tanto, es necesario encontrar nuevas mejoras para poder reducir la cantidad de recursos utilizados, sobre todo el de procesador, para incrementar el rendimiento de las aplicaciones ya existentes.

En este trabajo tengo el objetivo de encontrar una forma en la cual se pueda mejorar el desempeño de la resolución del problema. Una opción ha sido el algoritmo Fringe Search que ha planteado la posibilidad de mejorar el rendimiento de A* en cuestiones del tiempo de ejecución con el costo de no encontrar el camino óptimo, pero acercándose de manera importante a el. Por lo tanto, la hipótesis de este trabajo se centra en que Frienge Search es superior a A* en velocidad, sin tener una pérdida significativa de calidad en su camino.

Con base en la experimentación que se ha realizado de los dos algoritmos y su implementación optimizada se logró llegar a la conclusión de que Fringe Search definitivamente es superior a A* en velocidad, y que la penalización que sufre su camino generado no es significativa. Esto lo hace perfecto para implementarse en todas las plataformas de juegos que requieran la movilidad de agentes o de robots(pero no reduciéndose a ellas) que deben navegar en espacio dado, mejorando su rendimiento. Otras aplicaciones seguirán requiriendo caminos óptimos por lo que A* es el algoritmo idóneo.

Capítulo 1

Marco teórico

1.1 Teoría de gráficos y definiciones formales

En un primer momento se darán algunas definiciones básicas de la teoría de gráficos útiles para el trabajo. En su artículo Hart, Nilsson y Raphael(Hart y cols., 1968) expresan la siguiente notación y definiciones, que también usaré en el texto. " Un gráfico G es definido como un conjunto de $\{n_i\}$ elementos llamados nodos y un conjunto de $\{e_{ij}\}$ segmentos de línea dirigidos llamados arcos"(Hart y cols., 1968, p. 101). En algunos textos en español se nombra a los nodos como vértices y a los arcos como aristas(Johnsonbaugh, 2005, p. 320). "Si e_{pq} es un elemento del conjunto $\{e_{ij}\}$, entonces decimos que es un *arco* del nodo n_p al nodo n_q y que n_q es un *sucesor* de n_p o *incidente* sobre n_p. Estos arcos pueden tener costos o no, sin embargo, el texto se centrará en los arcos con costos. Se usará c_{ij} para representar el costo del arco e_{ij}. Si existe un arco de n_i no implica la existencia de un arco de n_j a n_i; pero si ambos arcos existieran, en lo general $c_{ij} \neq c_{ji}$. Los gráficos que se estudian con los algoritmos de caminos son llamados *gráficos* δ, donde $\delta > 0$ y cada arco de G es mayor que δ"(Hart y cols., 1968).

Es posible que no se declare abiertamente un gráfico como un conjunto de nodos y arcos, pero en vez de esto se realiza de manera implícita por medio de un conjunto de nodos de origen $S \subset \{n_i\}$ y un operador Γ definido en $\{n_i\}$ devolviendo un conjunto de pares $\{(n_{ij}, c_{ij})\}$ lo cual llevará a varios n_j(Hart y cols., 1968). Aplicando Γ a los nodos de origen y sucesivamente a sus sucesores hasta que nuevos nodos puedan ser generados resulta nuevamente en el gráfico que se ha definido anteriormente.

Si se toma en cuenta que: $G' = (\{n'_i\}, \{e'_{ij}\})$ es una subgráfica de G si $\{n'_i\} \subseteq \{n_i\}$ y $\{e'_{ij}\} \subseteq \{e_{ij}\}$ y si para todo arco $e' \in E'$ e' incide en n'_p y n'_q, entonces $n'_p, n'_q \in N'$(Johnsonbaugh, 2005, p. 330); se puede asegurar que "la subgráfica G_n de cualquier nodo $n \in \{n_i\}$ es el gráfico

definido implícitamente por un nodo de origen n único y cierto Γ definido en $\{n_i\}$. Por lo tanto se dice que cada nodo en G_n es accesible desde n"(Hart y cols., 1968).

Ahora para poder entender el concepto de camino se plantea que: "un *camino* de n_1 a n_k es un conjunto ordenado de nodos $(n_1, n_2, \ldots n_k)$ con cada n_{i+1} como un sucesor de n_i. Existe un camino desde n_i hasta n_j si y sólo si n_j es accesible desde n_i"(Hart y cols., 1968).

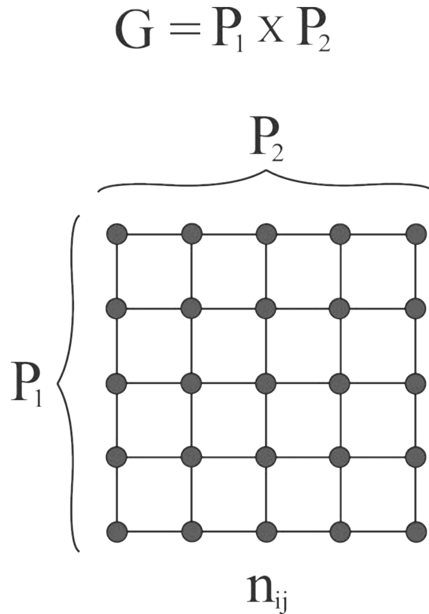

Figura 1.1: Gráfico de Lattice o de malla

Como se ha mencionado en la delimitación del tema, el trabajo se centrará en gráficos de malla de dos dimensiones. Estos gráficos regulares, también son conocidos como *gráficos lattice* y se definen como el gráfico G que es el producto cartesiano de los caminos $P_1 \times \ldots \times P_n$ y se denota $i = [(i)_1, \ldots, (i)_n]$ como un nodo de G donde el componente $(i)_k$ identifica la posición del nodo en el $k$seavo camino(Notarstefano y Parlangeli, s.f., p. 6). Lo cual lleva a que n es la cantidad de nodos en el camino y por lo tanto las dimensiones de nuestra malla. En este caso de estudio únicamente se usarán gráficos que surgen a partir de dos dimensiones de la forma $m \times n$ que generan un gráfico rectangular, con costos de sus arcos que varían entre uno y cero, para representar en el caso de tener el valor uno una zona de tránsito, y cero una zona que impide el paso del agente.

1.2 La búsqueda de caminos sobre gráficos desde la inteligencia artificial

En el campo de la inteligencia artificial un agente puede resolver un problema a partir de su planteamiento y llegar a una solución. Para hallar las soluciones requeridas los agentes podrán tener carencia total acerca de como se aproximan a la solución, los cuales son algoritmos clasificados como *búsquedas no informadas*; o tener cierta información que les facilite llegar a resolver el problema en cuestión que se denominan *búsquedas informadas*.

La búsqueda del mejor camino, es un problema planteado sobre un gráfico, pero este puede representar una abstracción de otros problemas, por lo que no necesariamente es un mapa o un espacio físico. Entendiendo esto, como primer paso, se plantea un *objetivo* que se encuentra representado en el gráfico al cual el agente pretenda llegar. El siguiente paso es llamado *formulación* del proceso en el cual se decide que acciones y estados se deben de tomar en cuenta. A esta parte sigue la *búsqueda* de la solución: "en general, un agente con distintas opciones inmediatas de valores desconocidos puede decidir qué hacer, examinando las diferentes secuencias posibles de acciones que le conduzcan a estados de valores conocidos, y escoger la mejor secuencia."(Russell y Norvig, 2004, p. 69) Por lo tanto, un algoritmo de búsqueda tomará como entrada un problema dado, y devolverá una sucesión de acciones representadas como conjunto de nodos ordenados, para llegar a la solución de dicho problema. Como ultima fase en la búsqueda, se debe reproducir la solución, también llamada *ejecutar*.

Para poder formalizar un problema de búsqueda se plantean cuatro componentes según Russell y Norvig(Russell y Norvig, 2004, p. 70):

- **Estado inicial**: El estado en el que un agente se encuentra, representado por el nodo n_s dentro del gráfico G sobre el cual se realizará la búsqueda.

- **Función sucesor**: Es la función Γ definida en $\{n_i\}$ que indicará como poder avanzar de un nodo a su sucesor.

- **Test objetivo**: Se entiende este test como una función Λ donde como entrada se especifica un nodo n_p y se devuelve un valor booleano, donde *cierto* indica haber alcanzado el o los objetivos y *falso* el no haberlo/s hallado.

- **Costo del camino**: El costo de una solución es la suma de costos por acción denotada

como $C = (c_1 + \ldots + c_n)$; mientras que el de cada acción se denota como $c(a, x, y)$ que se evalúa por una acción a, y un estado x al estado y. Una vez encontrada una o más soluciones se calculan los diferentes costos hasta hallar la solución con el costó mínimo.

1.2.1 Búsqueda no informada

Estas búsquedas incluyen todos aquellos algoritmos que no cuenten con información sobre la relación de los estados con la solución, y se basan sólo en la del planteamiento del problema y en la función Λ, pasando de un nodo a otro mediante Γ hasta encontrar la solución o terminar de buscar en todo el gráfico, lo cual significaría que no existe solución.

Si se cuenta con la información necesaria sobre un lugar, o en este caso de un gráfico, es necesario iniciar la búsqueda aleatoriamente, pero sin una manera informada de generar una hipótesis se debe estar dispuestos a perder mucho tiempo(Tanimoto, 1987, p. 150) en la ejecución del algoritmo. Dos ejemplos muy claros de este tipo de búsqueda son los algoritmos *Depth-First Search (DFS)* o Búsqueda de fondo en español y *Breadth-First Search (BFS)* o búsqueda a lo ancho. Estos dos algoritmos son muy parecidos el uno del otro, por lo que se puede analizarlos en conjunto.

Se explorarán los dos ejemplos para poder entender posteriormente los algoritmos heurísticos que son la materia de este trabajo. *Depth-Fist Search* que es posible apreciar en el listado, con una complejidad de $O(V + E)$, funciona basado en una pila, donde cada nodo nuevo se va agregando a la misma mediante la función *empujar*. Posteriormente se *jalará* el elementos mas reciente para poder ser trabajado a la vez de que será marcado en una tabla hash como visitada para evitar estados repetidos. Todos los nodos que pueden ser accedidos mediante la función sucesor Γ y que no hallan sido visitados serán empujados a pila. Al usar una pila o la filosofía de *último que entra primero que sale (LIFO)* se expandirá un árbol de búsqueda por entre el gráfico, que explorará primero a profundidad, llegando a una solución que no necesariamente tiene que ser la óptima(Tanimoto, 1987, p. 152). Para poder reconstruir el camino por entre el gráfico será necesario agregar otra tabla hash, en la cual se iría registrando el nodo antecesor, de tal manera que cada nodo pueda acceder al nodo que lo invocó. En cada visita a un nodo, se realizará también una comprobación de Λ para ver si se ha llegado al objetivo(Heineman y cols., 2008, p. 144).

Empero, el algoritmo pasado aún no puede resolver el problema de encontrar el mejor camino.

Algoritmo 1 Depth-First Search (DFS)

Basado en (Heineman y cols., 2008)

Entrada: G, n_s, n_g
Salida: G'
DFS(G, n_s, n_g)
  Pila $S \leftarrow \{\}$
  Visitados = Hash
  Visitados[n] $\leftarrow$ *null* **para todo** n en G
  Memoria = Hash
  Memoria[n] $\leftarrow$ *null* **para todo** n en G
  Empujar n_s a S
  Mientras S no está vacío:
    $n \leftarrow$ jalar desde S
    Si $\Lambda(n)$ **entonces**: **regresa** n
    Si Visitados[n] = *null* **entonces**:
      Visitados[n] $\leftarrow$ *verdadero*
      Para todo $n_i \in \Gamma(n)$:
        Si Visitados[n_i] **entonces**:
          empujar n_i a S
          Memoria[n_i] $\leftarrow n$
        Fin para todo
      Fin si
    Fin mientras
  Fin DFS

La variante BFS logra resolver esta limitante de DFS, la cual expandirá los nodos del árbol de búsqueda a lo ancho, llegando así a un camino con el nivel más bajo de expansiones. Esto se logra utilizando una lista de nodos, en la cual con cada iteración se agregará los nodos indicados por Γ al final de la lista, y se visitará el inicio de la misma en cada iteración. Si bien es posible que DFS puede llegar a encontrar el camino más corto por coincidencia y por lo tanto logre resolver el problema mas rápidamente que BFS, esto no será de manera general ni será garantía de que el camino encontrado sea el más corto. También habría que tomar en cuenta que si el tamaño del camino es menor, BFS será más rápido, caso contrario si el camino es más largo (Tanimoto, 1987, p. 153). La complejidad del algoritmo es de $O(V + E)$ en todos los casos(Heineman y cols., 2008, p. 150).

Algoritmo 2 Breadth-first search (BFS)

Basado en (Heineman y cols., 2008)

Entrada: G, n_s, n_g
Salida: G'
DFS(G, n_s, n_g)
  Lista $S \leftarrow \{\}$
  Visitados = Hash
  Visitados[n] $\leftarrow$ *null* **para todo** n en G
  Memoria = Hash
  Memoria[n] $\leftarrow$ *null* **para todo** n en G
  Agregar al final n_s a S
  Mientras S no está vacío:
    $n \leftarrow$ Tomar primer elemento de S
    Si $\Lambda(n)$ **entonces**: **regresa** n
    Si Visitados[n] = *null* **entonces**:
      Visitados[n] $\leftarrow$ *verdadero*
      Para todo $n_i \in \Gamma(n)$:
        Si Visitados[n_i] **entonces**:
          Agregar al final n_i a S
          Memoria[n_i] $\leftarrow n$
        Fin para todo
      Fin si
    Fin mientras
  Fin DFS

1.2.2 Búsqueda informada

La búsqueda informada es una estrategia que utiliza información que no se encuentra definida en el mismo problema y utiliza esta información para resolver el problema de una manera más eficiente(Russell y Norvig, 2004, p. 107). Los anteriormente estudiados algoritmos no informados buscan cual ciegos por la respuesta, no tienen sentido alguno acerca de si se encuentran cerca del objetivo o no. “Una función f es una *función de evaluación* que mapea cada nodo a un número real que sirve ya sea para estimar un beneficio relativo o el costo de continuar con la búsqueda en la misma dirección desde ese nodo”(Tanimoto, 1987). De manera alternativa $f(N)$ puede estimar la distancia entre el nodo ingresado y el nodo de destino. La función de estimación también es llamada *heurística* en gran parte de la literatura.

Las búsquedas informadas tienen por lo general una rendimiento mayor que sus contra partes, y por lo tanto se ha extendido su uso, hasta convertirlas en la opción por defecto en casi

todas las aplicaciones sobre gráficos de malla. Los algoritmos *A** y *Fringe Search* en los cuales se centra el texto, son de la familia de las búsquedas informadas y a continuación analizaré su funcionamiento.

En el capítulo pasado se ha expuesto los fundamentos de la teoría de gráficos para poder analizar de manera formal el problema y los elementos con los cuales se trabajará en el resto del texto; además de haber abordado el problema de la búsqueda de caminos para agentes en el terreno de la inteligencia artificial. Para ello se ha servido de dos algoritmos explicativos llamados *Breadth-First Search* y *Depth-First Search*. Los dos son ejemplos muy sencillos de estos procedimientos y funcionando de manera no informada. En el presente capítulo se explorarán a detalle los dos mejores algoritmos para la resolución del problema del mejor camino de un nodo a a un nodo b: la evolución heurística A* de Dijkstra y un algoritmo innovador llamado *Fringe Search* que genera resultados que tienden a ser el mejor camino, sin llegar a serlo. Pero para poder comprender de manera correcta A* será necesario estudiar antes Dijkstra, algoritmo del cual proviene.

1.3 El Algoritmo Dijkstra

El algoritmo Dijkstra es el algoritmo que resuelve el problema de la búsqueda del mejor camino de con un único nodo de partida hacia todos los nodos del gráfico, y es hasta el momento el algoritmo más veloz, para este caso con optimizaciones posteriores. En 1959 Dijkstra (Dijkstra, 1959) escribe su algoritmo en la revista "Numerische Mathematik"que tendrá un rendimiento, según sus análisis, de $O(n^2)$ en el peor caso donde n son el número de nodos del gráfico(Schrijver, 2010). Sin embargo, posteriormente se comenzaron a usar listas de prioridad para mejorar el rendimiento del algoritmo con *binary heaps*(montículo binarios) y *Fibonacci heaps*(Fredman y Tarjan, 1987)[1]. Con este aporte se logró mejorar la complejidad a $O(|E| + |V| \log |V|)$, usando en concreto un *heap* binario (Barbehenn, 1998). El algoritmo tiene también el supuesto de que el gráfico sobre el que se va a trabajar es de peso y el valor de cada uno de sus arcos debe ser positivo, o mayor que cero, de lo contrario se devolverá resultados erróneos o incluso pueden generarse ciclos infinitos(Heineman y cols., 2008). Para gráficos que requieren utilizar arcos negativos se puede usar algoritmo *Bellman-Ford*.

[1]Los *fibonacci heaps* son un desarrollo de las colas binomiales.

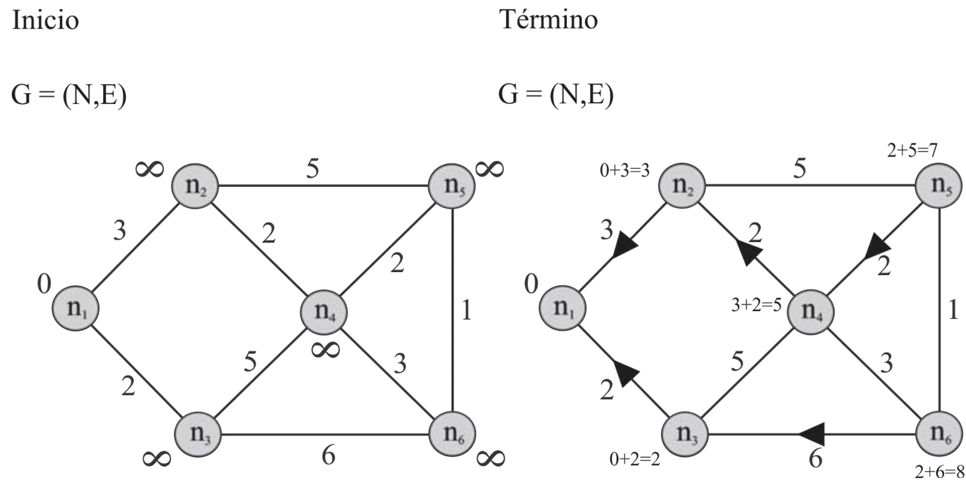

Figura 1.2: Algoritmo Djikstra

El trabajo se centrará en esta solución ya que para su desarrollo interesa el problema de encontrar el camino mas corto entre dos nodos. Dijkstra busca el camino más corto de un nodo de salida a todos los nodos del gráfico que puedan ser accesibles lo cual lo lleva a usar muchos más recursos de cómputo que los que se analizan(Millington, 2006). Sin embargo, $A*$ fue desarrollado a partir de Dijkstra y por lo tanto este último es de relevancia para el trabajo.

Como se puede apreciar en el algoritmo 1.3 los datos de entrada que recibe Dijkstra son un gráfico G y un nodo de partida $n_s \in N$, donde cada $e \in E$ tiene un peso positivo. Como resultado produce dos arreglos computados donde $dist[]$ representa la distancia desde el nodo de inicio a cada uno de los nodos computados; y un arreglo de punteros que indica el antecesor de cada nodo, según se halla explorado, para poder reconstruir el camino(Heineman y cols., 2008).

El funcionamiento del algoritmo es el siguiente: se asigna una distancia infinita y se especifica como nulo el puntero de memoria a cada nodo del gráfico, con excepción del nodo fuente, al cual se le asigna una distancia de cero y se agrega cada nodo a una lista de prioridad, en la cual en un inicio el nodo de origen será su cabeza. Posteriormente se itera sobre la lista de prioridad tomando el menor elemento para removerlo de la misma y visitar cada nodo sucesor Γ, calculando su distancia con respecto al nodo de origen y asignando el nodo antecesor. La expansión llevará a la creación de un árbol en el cual se obtendrá el camino de costo mínimo de cada nodo del gráfico hacia el inicial. Se puede alterar la implementación para incluir una función Λ para buscar el mejor camino entre dos nodos, aunque esta opción no es la más eficiente y será tema de A*.

Algoritmo 3 Dijkstra

1.3 basdo en (Dijkstra, 1959)

Entrada: G, n_s
Salida: Distancia[], Memoria[]
dijkstra(G, n_s)
  ColaDePrioridad $P \leftarrow \{\}$
  Distancia[n_s] $\leftarrow 0$
  Para todo n en N:
    si $n \neq n_s$ **entonces**:
      Distancia[n] $\leftarrow \infty$
      Memoria[n] $\leftarrow$ *null*
    fin si
    agregar n a P
  fin para todo
  Mientras S no este vacío:
    $n \leftarrow min(P)$
    eliminar n de P
    Para todo $n_i \in \Gamma(n)$:
      $c \leftarrow$ Distancia[n] + costo(n, n_i)
      si $c <$ Distancia[n_i]
        Distancia[n_i] $\leftarrow c$
        Memoria[n_i] $\leftarrow n$
      Fin si
    Fin para todo
  Fin mientras
  Devuelve Distancia[], Memoria[]
Fin Dijkstra

Capítulo 2

Marco Contextual

2.1 El algoritmo A*

Uno de los grandes problemas de los algoritmos no informados es su necesidad de expandir cada nodo que encuentran, sin poder discernir entre los nodos más prometedores y los que no lo son. Esto lleva a que su rendimiento sea bajo y por ello no se considere una opción viable para su implementación en una gran cantidad de aplicaciones, sobre todo si G es muy grande. Para poder expandir el menor número de nodos posible se deben realizar la mayor cantidad de decisiones informadas posibles para llegar a un camino óptimo(Hart y cols., 1968). Realizar la expansión de nodos que no prometen estar en el camino óptimo sería una pérdida de tiempo, como es el caso de realizarlo sin tener información. Por otro lado, si se ignoran sistemáticamente los nodos que se encuentran en el camino óptimo se llegará a un resultado que no necesariamente sería el adecuado.

A* es un algoritmo que logra utilizar los pasados planteamientos *heurísticos* y por lo tanto se ha convertido casi en el sinónimo de búsqueda de caminos(Millington, 2006, p. 223). A* es utilizado para una gran variedad de tareas, desde aplicaciones sobre mapas, hasta búsqueda en gráficos de juegos. El algoritmo necesita de la entrada de dos nodos que deben ser conectados y regresará al finalizar su ejecución un camino entre estos dos.

2.1.1 Planteamiento formal

El problema de encontrar el camino más corto entre dos nodos se concentra en la "subgráfica G_s que parte desde un único *nodo origen* s. Definimos como un conjunto de nodos no vacíos T en G_s como *nodos de destino*. Para todo nodo n en G_s, un elemento $t \in T$ es un nodo de destino *preferente* de n si y sólo si el costo de un camino óptimo de n a t no excede el costo

de cualquier otro camino de n a cualquier otro miembro de T"(Hart y cols., 1968). Para poder expresar de una manera más simple se utilizará $h(n)$ lo cual significa(Hart y cols., 1968).:

$$h(n) = \min_{t \in T} h(h, t) \tag{2.1}$$

2.1.2 La estimación heurística

Se tomará una función de evaluación $f(n)$ con la cual se puede calcular cualquier nodo $n \in G$, la cual dará el costo optimo de un camino óptimo restringido a pasar por n desde s a un nodo objetivo en N(Hart y cols., 1968). La clave de este planteamiento es que se intenta forzar el camino sobre el nodo n, lo cual no quiere decir que n esté en ese camino óptimo. Si se partiera del camino óptimo entonces $f(s) = h(s)$ para todo nodo en el; la función, por lo tanto, al evaluar el costo de un nodo n que no se encuentra en este camino será superior al del camino óptimo: $f(n) > f(s)$. Pero como no se conoce el camino *a priori* la discusión se centrará en intentar encontrar una función de estimación $\hat{f}(n)$(Hart y cols., 1968). Gran parte del comportamiento del algoritmo como tal se centra en lo correcto o incorrecto de la función que se describe.

La función heurística $f(n)$ que plantea el algoritmo A* se compone de dos partes:

$$f(n) = g(n) + h(n) \tag{2.2}$$

Donde $g(n)$ es el costo que tiene el camino óptimo desde el nodo de origen s al nodo actual que se está analizando n y $h(n)$ es el costo de camino óptimo del nodo actual n al destino preferido en n. Para poder avanzar con la construcción de la función de estimación tomemos a $\hat{g}(n)$ como la función de estimación de $g(n)$. Para esto, se usará el costo mínimo que el algoritmo ha encontrado hasta ese momento para llegar al nodo n, lo cual resultará en: $\hat{g}(n) \geq g(n)$ (Hart y cols., 1968).

La segunda parte de la ecuación $h(n)$ consiste en la evaluación del camino mínimo desde n al destino, donde se tendrá que encontrar la función de estimación $\hat{h}(n)$. Esta estimación se encuentra íntimamente relacionada con la cuestión del problema, por lo que no se puede plantear una solución general para todos. Es posible que, como en este tipo de gráfico, se utilice distancias geométricas para poder crear una función de estimación que ayude al algoritmo. Gran parte de la eficacia de A* recae en encontrar una función correcta para el problema en cuestión.

Ahora se planteará la función de evaluación, la cual se compone, al igual que la ecuación 2.2 de dos partes:

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n) \tag{2.3}$$

Donde $\hat{g}(n)$ es el camino de mínimo coste estimado hasta ese momento por el algoritmo desde el nodo inicial a n y $\hat{h}(n)$ es el estimado del costo de un camino óptimo desde n a un nodo objetivo en N(Hart y cols., 1968). En el *Teorema 1* de su trabajo Hart, Nilsson y Raphael(Hart y cols., 1968) expresan que como condición para que sea admisible la estimación anterior debe cumplir con $\hat{h}(n) \leq h(n)$ para todo n.

¿Pero que función debemos plantear para $\hat{h}(n)$ en una gráfico de malla que representa un mapa? Esta pregunta muy usada en robótica, en inteligencia artificial de juegos y en general en gráficos que representan lugares en un espacio geográfico y tiene soluciones que refieren a la geometría. A continuación se motrará las funciones de estimación apropiadas para cada tipo de problema.

2.1.2.1 Distancia Manhattan

En gráficos cuyos nodos internos tienen cuatro arcos, representando una cuadrícula, se recomienda el uso de la *distancia Manhattan*(Patel, 2014) planteada por Minkowsky en el siglo XIX que calcula la distancia a partir de la suma de las diferencias absolutas de las coordenadas de dos puntos. La diferencia entre estos puntos hace recorrer la distancia no en línea recta como en la geometría euclidiana, sino en forma de cuadrícula, forma que tiene el gráfico de malla antes descrito. La fórmula aplicada a dos dimensiones, que tiene nuestra malla, y adaptada para su uso en el estimador h es:

$$\hat{h}(n) = D(|x_n - x_g| + |y_n - y_g|) \tag{2.4}$$

donde n es el nodo a evaluar y g el nodo de destino, los cuales se expresan en su forma de puntos en un plano $(x_n, y_n)(x_g, y_g)$. El valor de la constante D corresponde a la escala que se desee asignar con relación a todo el gráfico y el costo de avanzar cada nodo: avanzar más cerca al nodo objetivo debe incrementar a g en D y debe decrementar h por D.

2.1.2.2 Distancia de Chebyshov

Si el gráfico de malla tiene nodos de ocho arcos para estar conectados de manera vertical, horizontal y diagonal con otros se necesita la *distancia de Chebyshov*(Patel, 2014). Planteado por el matemático ruso Pafnuti Lvóvich Chebyshov donde la distancia entre dos vectores es la mayor de las diferencias de entre las coordinadas de la dimensión. En este caso donde el plano se representará con movimientos en ocho direcciones la distancia se expresa de la siguiente manera:

$$\hat{h}(n) = D(\text{máx}(|x_n - x_g|, |y_n - y_g|)) \tag{2.5}$$

Las variables tienen la misma connotación que en la ecuación 2.4. Sin embargo, esta fórmula no resulta correcta si el costo de moverse en diagonal es distinta que la de moverse en vertical o en horizontal. Si el costo de un movimiento en diagonal tiende a $D\sqrt{2}$ entonces se puede utilizar el siguiente método para plantear la estimación:

$$\begin{aligned} dx &= |x_n - x_g| \\ dy &= |y_n - y_g| \\ \hat{h}(n) &= D_r(dx + dy) + (D_d - 2D_r)\,\text{mín}(dx, dy) \end{aligned} \tag{2.6}$$

donde, como ya se ha mencionado, D_d es el costo del movimiento en diagonal y D_r es el costo del movimiento en horizontal o vertical(Patel, 2014).

2.1.2.3 Distancia Euclidiana

Si el agente puede moverse en cualquier dirección dentro del espacio es necesario usar la *distancia euclidiana*, con la desventaja de que la operación es una costosa función de raíz cuadrada y la función $\hat{g}(n)$ no coincidirá del todo con $\hat{h}(n)$. A pesar de esto, aún se puede conseguir el camino más corto, pero con un fuerte castigo al tiempo de ejecución(Patel, 2014). La función de distancia que se ha comentado puede verse en la ecuación 2.7.

$$\begin{aligned} dx &= |x_n - x_g| \\ dy &= |y_n - y_g| \\ \hat{h}(n) &= D\sqrt{(dx)(dy) + (dx)(dy)} \end{aligned} \tag{2.7}$$

2.1.2.4 Algunas notas sobre la función heurística

El comportamiento de A* varía dependiendo del resultado de $\hat{h}(n)$:

- Si $\hat{h}(n)$ adquiere el caso extremo de devolver siempre 0, entonces el algoritmo comienza a funcionar como Dijkstra, al sólo ser tomado en cuenta $\hat{g}(n)$ para la valoración del mejor camino.

- Si $\hat{h}(n)$ es igual o menor que el costo de moverse de n al nodo objetivo entonces se garantiza el mejor camino, pero esto puede traer una cantidad grande de nodos visitados y por lo tanto una ejecución más lenta. Entre más cerca el valor de $\hat{h}(n)$ se encuentra del valor de $h(n)$ menos nodos se expandirán.

- En el caso de que $\hat{h}(n) = h(n)$ A* solo seguirá el mejor camino y por lo tanto no expandirá nodos extra. Esto no se puede asegurar ya que no se conoce $h(n)$ a priori.

- Si $\hat{h}(n)$ es en ocasiones mayor que $h(n)$ entonces A* no encontrará forzosamente el mejor camino, pero su tiempo de ejecución bajará.

- En el otro extremo, si el valor de $\hat{h}(n)$ es muy muy grande, A* se convertirá en un algoritmo Best-First-Search.

Este comportamiento de A* es muy útil para la implementación práctica en el mundo real, donde se puede dar preferencia a una ejecución más rápida o a un camino más exacto manipulando los valores que arroja la función $\hat{h}(n)$(Patel, 2014). Aprovechando los casos pasados también es posible realizar algunas optimizaciones con el precalculo de los valores de costos, registrando los valores desde todos los nodos del gráfico a otros nodos, de tal manera que siempre se podrá encontrar $\hat{h}(n) = h(n)$.

2.1.3 El algoritmo

*A** básicamente es un derivado de *Dijkstra* donde se incluye una función heurística para discernir entre posibles nodos en los cuales buscar. Como se ve en el algoritmo 2.1.3 se utilizan dos estructuras de datos principales, que son una lista cerrada C y una lista de prioridad A que llamaremos lista abierta, la cual se ordena con el valor que arroja $h(n)$.

Primero se establecen como vacía la lista cerrada y a la abierta, además de establecer la función $g(n)$ con los valores de 0. Posteriormente agregamos el primer nodo a la lista abierta, para pode iniciar las iteraciones sobre esta lista. Se repetirá el ciclo principal hasta que A esté completamente vacía, lo cual indicaría la falta de una solución, o hasta que se haya encontrado el camino que se buscaba.

Algoritmo 4 A*

Referencia: 2.1.3 con base en (Heineman y cols., 2008) (Hart y cols., 1968) (Patel, 2014)

Entrada: G, n_s, n_g
Salida: G_g
astar(G, n_s, n_g)
  Cola de prioridad $A \leftarrow \{\}$
  Hash A $C \leftarrow \{\}$
  agregar n_s a A
  Mientras A no esté vacío:
    $n \leftarrow min(A)$
    eliminar n de A
    agregar n a C
    si $n = n_s$:
      regresar solución G_s
    Fin si
    Para todo $n_i \in \Gamma(n)$:
      $c \leftarrow g(n)+$ costo(n, n_i)
      si n_i en A y $c < g(n_i)$
        eliminar n_i de A
      fin si
      si n_i en C y $c < g(n_i)$
        eliminar n_i de C
      fin si
      si n_i no está en C y c no está en A
        $g(n_i) \leftarrow c$
        $f(n_i) \leftarrow g(n_i) + h(n_i)$
        agregar n_i a A con valor de $f(n_i)$
        establecer n como pariente de n_i
      fin si
    Fin para todo
  Fin mientras
Fin astar

En cada iteración sobre el ciclo principal se eliminará el elemento de costo mínimo de la lista abierta, al ser el nodo con las mejores posibilidades de encontrarse sobre el camino óptimo

se agrega a la lista cerrada, la cual contendrá todos los nodos que ya han sido visitados; se comprueba si este nodo es la solución al problema y en caso contrario se comienza una iteración sobre los nodos sucesores que se obtienen mediante la función $\Gamma(n)$.

Al recorrer los diferentes nodos sucesores del nodo principal, se establece un nuevo costo, que resulta de sumar el costo pasado que se tiene registrado de llegar hasta el con el costo de avanzar del nodo principal al nodo sucesor, estableciendo así el valor de $\hat{g}(n)$. Con ese valor, identificado como costo, se busca si el nodo se encuentra en la lista abierta y cerrada. Si el valor del costo actual es inferior al del estado registrado anteriormente se elimina de la lista y se reinserta a la lista abierta con el nuevo costo y un nuevo cálculo heurístico. Por último se establece como padre del nuevo miembro de la lista abierta al nodo principal.

El proceso se realiza constantemente para cada nodo en la lista abierta, examinando cada nodo sucesor, hasta encontrar el mejor camino. El algoritmo, como se ha mencionado, encuentra el mejor camino de esta manera y lo convierte, bajo el entendimiento general, en el mejor para resolver el problema de encontrar el camino óptimo entre dos nodos, tanto en garantizar el camino como en el rendimiento al realizar la búsqueda, afirmación que discutiré en este trabajo. Su complejidad es de $O(n)$ en el mejor caso y $O(n^d)$ en el peor.

2.1.4 Los tipos de datos usados para el algoritmo

En el algoritmo se describen diversas operaciones que se realizarán sobre las dos estructuras de datos, una lista abierta y otra cerrada. Para llevar acabo el algoritmo se necesita la función de pertenencia, inserción y eliminación de la lista cerrada; y en la lista abierta se requerirán las anteriormente mencionadas más una función de "jalar" que arrojará el estado con el menor valor y lo eliminará. También interesan funciones triviales como obtener información sobre el estado del tipo de dato, así como si se encuentra vacío y una para determinar el número de miembros de la lista.

Tabla 1: Complejidad de las estructuras de datos y sus operaciones

Operación	Lista ordenada	Binary heap	Skiplist	Implementación combinada
Pertenencia	$O(F)$	$O(F)$	$O(F)$	$O(1)$
Inserción	$O(F)$	$O(\log F)$	$O(\log F)$	$O(\log F)$
Búsqueda	$O(F)$	$O(F)$	$O(\log F)$	$O(\log F)$ / $O(1)$
Eliminación	$O(F)$	$O(F)$	$O(\log F)$	$O(\log F)$
Pop	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Cabría aclarar que en la función de cambio de lugar y eliminación se presupone que no se conoce el lugar que ocupa el estado en la estructura de datos. A continuación se hará una breve exposición de distintos tipos de datos que pueden usarse para implementar los conceptos de lista abierta y cerrada que se muestran en la table anterior.

2.1.4.1 Listas ordenadas

Este tipo de datos es el de más fácil implementación, pero es poco recomendable para A* ya que consume una gran cantidad de tiempo de ejecución en la parte más crítica del algoritmo. Según Patel(Patel, 2014) las listas ordenadas tienen costos de $O(F)$ para pertenencia; mientras que insertar, eliminar y cambiar de posición únicamente significa $O(1)$ si se conoce el lugar donde llevar acabo la operación; realizar esta búsqueda tiene una complejidad de $O(F)$ convirtiéndolo en un tipo de dato altamente costoso en tiempo de ejecución.(Langsam, Augenstein, y Tanenbaum, 1997, p. 3).

2.1.4.2 Binary Heap

Los *Binary heap* son la elección por defecto de la mayoría de las implementaciones de A* por ser de fácil implementación y muy eficientes en tiempo de ejecución. La operación de inserción y de remover el mejor elemento es de $O(\log F)$, mientras que la pertenencia es de $O(F)$ por tener que recorrer el total de la estructura. Para hacer un cambio de miembro se necesita encontrarlo y posteriormente se necesita $O(\log F)$ para realizar el cambio, con el problema de que esta operación no se encuentra incluida en la mayoría de las bibliotecas que ofrecen *Binary Heaps*(Patel, 2014). La operación puede ser evitada al agregar simplemente un nuevo elemento a la lista con la consecuente penalización en consumo de memoria.

2.1.4.3 Skiplist

Las listas de salto, o *Skiplists* son una alternativa probabilística a los árboles balanceados con la ventaja de no tener que depender de inserciones aleatorias como si lo requieren los árboles, inventada por W. Pugh en 1989(Pugh, 1990). Esta estructura de datos funciona con una jerarquía de nodos que enlazan los niveles superiores para avanzar de manera rápida por la lista, y bajar de nivel para aumentar la precisión de la consulta, hasta llegar al nivel más bajo donde se encuentra una lista ordenada normal. Su velocidad es igual a la de un *binary heap* mejorando su rendimiento en estructuras con más de diez mil elementos(Patel, 2014). En la implementación se usa una skiplist con el objetivo de utilizar estructuras lo más grandes posibles para poder explotar al máximo el análisis de gráficos de gran tamaño.

2.1.4.4 Tablas hash y arreglos

Las tablas *Hash*, a diferencia de un arreglo, utiliza mucho menos memoria(Patel, 2014) para poder manejar los diversos conjuntos, ya sea para la tabla abierta o como auxiliar de la cerrada. Una tabla hash es una estructura de datos que requiere de una llave *hash* (*dispersión*) para poder acceder de manera directa a cierto dato, lo cual se realiza en $O(1)$(Drozdek, 2007, p. 219), al igual que un arreglo. En la implementación se utilizarán arreglos, asumiendo la penalización en memoria a favor de una simplificación en la implementación y con un ligero incremento en la velocidad, reutilizando las tablas de nodos ya existentes.

2.1.5 Implementación mixta

En la implementación realizada de A* se ha optimizado el comportamiento del algoritmo utilizando varios tipos de datos en la lista abierta y en la cerrada, con lo cual se puede reducir casi todas las llamadas a $O(1)$ y $O(\log F)$. Como se ha mencionado, la estructura del gráfico sobre el cual se trabajará tiene una forma de malla, por lo cual se puede representar con matrices $N \times M$ en forma de un plano cartesiano y representado como un arreglo. La coordenada (x, y) será nuestra llave para localizar el nodo en nuestra malla con $O(1)$ de complejidad. El arreglo se utilizará en la lista abierta y su acceso se hará con gran velocidad.

En el caso de la lista cerrada utilizaré dos tipos de datos, una *skiplist* para tener tiempos de inserción, eliminación y en general cualquier operación que requiere una búsqueda con $O(\log F)$ utilizando como clave de ordenamiento el valor de $\hat{f}(x)$ de nuestra función antes descrita. Sin

embargo, para la búsqueda basada en una llave de coordenada como lo es la función de pertenencia skiplist requeriría $O(F)$, lo cual es demasiado costoso. En este caso se complementa la *skiplist* con un segundo arreglo que servirá como soporte para el acceso mediante coordenadas a la lista cerrada reduciendo la complejidad de cualquier operación que utilice pertenencia a $O(1)$.

2.1.6 Algunas notas sobre la implementación

Para reducir el uso de memoria se utilizó una única tabla $X \times Y$ que contiene todos nodos los cuales poseen la información necesaria para poder funcionar en el algoritmo como se verá en la sección de código fuente. Los nodos de la *skiplist* únicamente apuntan a la tabla, mientras que no es necesario implementar dos arreglos separados para la lista abierta y la cerrada, ya que se puede introducir un campo en la estructura del nodo en la cual se especifica la pertenencia o no a cada una de las listas. Se intentado optimizar al máximo la ejecución del algoritmo para poder compararlo con una versión semejante de Fringe Search.

En la parte heurística se decidió utilizar la distancia *Chebyshov* para los estimadores $\hat{g}(n)$ y $\hat{h}(n)$ ya que se utiliza una estructura de malla donde cada nodo interior tiene ocho arcos hacia sus nodos vecinos. Esta estrategia en las funciones logra optimizar el rendimiento acercándolo lo más posible a una forma realista de la representación de un mapa de terreno.

2.1.7 Análisis del código fuente

Para tener una compresión mayor del funcionamiento de A* será pertinente una breve exposición a las partes más importantes del código fuente de la implementación del algoritmo que se realizó. El programa inicia su secuencia desde el archivo *main.c*, donde se leen los argumentos desde la linea de comandos, los cuales se componen del nodo de partida en (x, y), un nodo de destino (x, y), el nombre del archivo de imagen del cual se obtendrá el gráfico que será usado para generar el gráfico G, el nombre de una imagen de salida sobre la que se dibujará el resultado de la búsqueda y un argumento final que activa el texto de debugging en el caso de ser positivo.

Con los datos recabados desde la línea de comandos se llenarán las variables iniciales con las cuales se invocará la función *read_map(char *)* la cual revisará si el archivo BMP indicado existe y si este es válido, con ello se tendrá cargado en memoria los datos del gráfico y se llenarán las variables globales de tamaño del gráfico. La información será guardada en un arreglo que

contendrá la información necesaria para la ejecución del algoritmo.

```c
typedef struct anode{
    int             index;
    int             val;
    int             deph;
    int             g, h;
    int             coord;
    int             count;
    apoint          point;
    int             x,y,f;
    int             inopen, inclosed;
    struct anode    *parent;
    struct anode    *pointer;
    struct skiplistnode *skip;
}anode;
```

Definición del nodo

Como es posible apreciar el nodo contiene toda la información necesaria para ser trabajado por el algoritmo. Contiene la información heurística en forma de valores f, g, h un puntero a su nodo padre y la capacidad de localizarse tanto en un espacio geográfico, como en la *skiplist*. Se genera un arreglo dinámicamente $x \times y$ el nodo antes descrito. Todo trabajo posterior se realizará mediante punteros a los nodos generados de esta estructura.

Será también necesario mostrar la estructura de datos que unirá todas las variables necesarias para el funcionamiento de A*.

```c

typedef struct aobj{
    Skiplist        *openlist;
    anode           **cache;
    apath           path;
    anode           *item;
    char            *output;
    int             open_empty, closed_empty;
}aobj;
```

Definición del controlador de A*

El arreglo anteriormente generado será apuntado desde *anode **cache* que servirá al mismo tiempo como lista cerrada. El puntero *Skiplist *openlist;* utiliza la estructura de datos de skiplist para utilizarla como la lista abierta.

Al tener contenida toda la información relevante, la estructura de datos que se ha descrita será utilizada como argumento en la llamada a la función principal de la siguiente manera:

```
    ai_shortes_path(astar, 0, source, goal))
```

Al iniciar la secuencia de la función principal se comprobará si los nodos de inicio y de destino son nodos válidos, entendiendo como nodos no válidos aquellos que se encuentren fuera del gráfico o aquellos que han sido descritos con valor de falso al cargar el mapa. A continuación se iniciará la lista abierta y la cerrada y se cargará a la lista abierta el nodo inicial.

```
item->deph = 0;
item->point = source;
item->x = source.x;
item->y = source.y;
item->h = hdist(item->x, item->y, goal.x, goal.y);
item->g = 0;
item->f = item->g + item->h;
item->parent = NULL;

if(!openlist_insert(astar, item))
    ...
```

Se implementó la función heurística utilizando la distancia de *Chebyshov* la cual fue descrita anteriormente, de la siguiente manera.

```
int hdist(int p1x, int p1y, int p2x, int p2y)
{
    int xDistance = abs(p1x-p2x);
    int yDistance = abs(p1y-p2y);

    if (xDistance > yDistance)
    {
        return 14*yDistance + 10*(xDistance-yDistance);
    }
    else
    {
        return 14*xDistance + 10*(yDistance-xDistance);
    }
}
```

Una vez teniendo en la lista abierta el nodo de inicio, a partir del cual se expandirán todos los demás nodos que compondrán el árbol de búsqueda se pasará a declarar el ciclo principal que se mantendrá activo hasta que se encuentre vacío o que la función Λ haya encontrado un resultado. Como primer paso se remueve de la lista abierta el elemento más prometedor y por

lo tanto el menor que se encuentra en el tope de nuestra lista ordenada, siento el puntero *item* el nodo sobre el cual se trabajará en este ciclo.

```
while(!openlist_isempty(astar))
{
    item = openlist_getmin(astar);
    ...
    if(!closedlist_insert(astar, item))
    ...
    if(item->x == goal.x && item->y == goal.y)
    {
        ...
```

A continuación se insertará el elemento removido de la lista abierta a la cerrada y se realizará una prueba con la cual podrá comprobarse si el nodo actual *item* es el nodo objetivo o no. Si no lo fuera se seguirá adelante con la búsqueda, de lo contrario, se reproduce el camino encontrado, se genera la variable camino en el objeto astar y se interrumpirá la ejecución de la función *astar()*.

Posteriormente se expandirán todos los nodos con los cuales *item* tiene algún arco mediante la función que se ha descrito como Γ haciendo un recorrido de las posibles direcciones en los cuales se puede expandir; cada dirección será evaluada para discernir si existe o no y por último se verá si es un nodo transitable o no. El nodo recién expandido será sujeto a un cálculo de $\hat{g}(n), \hat{h}(n), \hat{f}(n)$, además de apuntar al nodo padre, que en este caso es *item*. Cabe destacar que para cada nodo que tenga un arco vertical u horizontal se usará una distancia regular que tendrá un costo de diez y los arcos que se encuentren en diagonal tendrán un costo de catorce.

```c
for(a = 0; a < NUM_DIRS; a++)
{
    vector = get_vector(item->point, a);
    if(vector.x != -2 && vector.y != -2)
    {

        pt.x = vector.x + item->point.x;
        pt.y = vector.y + item->point.y;
        if(ai_valid_tile(unit, pt))
        {
            next->deph = item->deph + 1;
            next->point = pt;
            next->h = hdist(next->point.x, next->point.y, goal.x, \
                    goal.y);
            next->x = next->point.x;
            next->y = next->point.y;
            if( a == ISO_N ||
                a == ISO_S ||
                a == ISO_W ||
                a == ISO_E)
                next->g = item->g + G;
            else
                next->g = item->g + G+4;
            next->f = next->g + next->h; // F = G + H
            next->parent = item;
            ...
```

Ahora será necesario ver si el nodo actual se encuentra ya en la lista cerrada o no, de ser así hay que comprobar si el nuevo valor es menor que el anterior que ya está en la lista cerrada. En el caso de que así fuera se removerá de la lista cerrada y se marcará una bandera que indicará que no se encuentra en la cerrada. En caso contrario, se mantendrá el nodo que se encuentra en la lista cerrada y se continuará con el flujo.

```c
if((old = closedlist_search(astar, next)) != NULL)
{
    if(next->f < old->f)
    {
        if(!closedlist_remove(astar, old))
        {
            astar->item = NULL;
            return 0;
        }
        closed_flag = 0;
    }
    else
    {
        closed_flag = 1;
    }
}
```

Como se ha comprobado la existencia del nodo en la lista cerrada, será necesario también comprobar si existencia en la lista abierta. En dado caso de que esto sea afirmativo se debe nuevamente comprobar si el nuevo elemento es mayor que el ya existente, para que si fuese así eliminarlo de la lista abierta. Al igual que con la lista cerrada, si el elemento nuevo es mayor que el ya existente únicamente se indica en una bandera adecuada.

```
if((tmp_item = openlist_search(astar,next))!=NULL)
{
    if(tmp_item->f > next->f && closed_flag == 0)
    {
        open_flag = 0;
        if(!openlist_rm(astar, tmp_item))
            return 0;
    }
    else
    {
        open_flag = 1;
    }
}
```

Por último, si las dos banderas son falsas se parte de que ya no existe el nodo en ninguna de las dos listas, y por lo tanto lo se inserta en la lista abierta y se podrá el puntero item nuevamente a nulo para su uso en el nuevo ciclo.

```
if(open_flag == 0 && closed_flag == 0)
{
    if(!openlist_insert(astar, next))
    {
        astar->item = NULL;
        return 0;
    }
}
else
{
    if(next != NULL)
    {
        free(next);
        next = NULL;
    }
}
```

Las partes expuestas el el código más relevante que permite explicar el comportamiento del algoritmo en forma de código. El código se encuentra a disposición bajo la licencia GPL v3 en internet para todo aquel que guste explorar más a fondo la implementación pueda hacerlo.

Además en el apartado de anexo también puede ver el archivo principal donde se expone el código fuente de la lógica del algoritmo.

2.2 Fringe Search

Fringe Search(FS) es el desarrollo de un algoritmo planteado en 1985 por Richard E. Korf llamado IDA*(Korf, 1985) que reduce el uso de memoria de A*, pero que adolece de algunos defectos importantes como lo es la posibilidad de generar ciclos en el espacio de búsqueda, estados repetidos y un uso demasiado simple de la búsqueda de derecha a izquierda(Björnsson y cols., 2005). El algoritmo Fringe Search(Björnsson y cols., 2005) elaborado en 2005 por Björnsson, Enzenberger, Holte y Schaeffer de las universidades de Rekyavik y Alberta es un derivado de IDA*, con la ventaja de eliminar gran parte de las deficiencias que presenta este, teniendo como gran ventaja llegar a caminos que tienden a ser óptimos, sin llegar a serlo, y con una mejora de rendimiento con respecto a A* en el tiempo de ejecución de 10 % a 40 % según los autores, lo cual se ha logrado comprobar mediante la implementación que se ha realizado para este trabajo.

A continuación presentaré el algoritmo IDA* para poder posteriormente continuar con la exposición del Fringe Search y plantear las cuestiones de implementación.

2.2.1 IDA*

En su artículo Korf(Korf, 1985) plantea el *Depth-First Iterative-Deeping* (DFID) cómo un algoritmo basado en DFS, pero que logra resolver gran parte de sus limitaciones que ya se ha descrito antes. Este algoritmo consiste en primero realizar una búsqueda DFS hasta el nodo uno, para después descartar los nodos generados en la primer búsqueda. Realizado esto vuelve a realzar un DFS en el siguiente nivel, en ese caso el nivel dos. El proceso se repite de nivel en nivel hasta que se alcanza el nodo objetivo(Korf, 1985). DFID expande todos los nodos de un nivel antes de avanzar, por lo que esto le garantiza encontrar el mejor camino y por no expandir profundidades mayores a la de la solución su complejidad es de $O(d)$.

Para desarrollar IDA* se utiliza el planteamiento heurístico de A* que tiene una forma de búsqueda semejante al BFS y se combina con DFID. Se incrementa con cada iteración no la profundidad de niveles sino el costo total del camino y con el comportamiento siguiente: ”en cada iteración se realizará un DFS que se interrumpirá si el costo total $(g + h)$ excede una variable memoria dada. Esta memoria comienza en el costo estimado del primer estado y se incrementa

por cada iteración del algoritmo. En cada iteración la memoria que se utiliza para la próxima iteración es el mínimo de los costos de todos los valores superiores a la memoria actual"(Korf, 1985). Explicando el algoritmo debe recalcarse que sigue conservándose la condición $\hat{h}(n) \leq h(n)$ para que sea admisible $\hat{h}(n)$. El autor de IDA* además plantea que el algoritmo encuentra el camino óptimo y que expande el mismo número de nodos que A*.

De lo antes expuesto se debe remarcar que A* realiza una búsqueda BFS informada, mientras que IDA* realiza DFS, pero A* crea árboles de búsqueda menores ya que se beneficia de usar un registro de almacenamiento en forma de listas abiertas y cerradas, lo cual no realiza IDA*. Esto conlleva a que IDA* utilice menos espacio de memoria al realizar sus búsquedas, y no gasta recursos en mantener estas dos listas. Sin embargo, el prescindir de un almacenamiento trae consigo fuertes desventajas y son las siguientes(Björnsson y cols., 2005):

- No puede detectar estados repetidos ya que no cuenta con un almacenamiento de estados visitados a diferencia de A*.
- A* mantiene una frontera de búsqueda mediante la lista abierta donde se almacenan los estados visitados, lo cual no puede realizar IDA*.
- A* utiliza como frontera una lista ordenada expandiendo los nodos en un orden, mientras que IDA* tiene que utilizar una búsqueda de izquierda a derecha simple.

Fringe Search plantea superar estas limitaciones, lo cual le permite aprovechar las ventajas de IDA* y resolver las cuestiones expuestas, lo cual lo lleva a superar en varios aspectos a A*.

2.2.2 Mejorando IDA*: Fringe Search

A diferencia de IDA* el nuevo algoritmo planteado por Böjrnsson, Enzenberger, Holte y Schaeffer en su artículo *Fringe Search: Beating A* Pathfinding on Game Maps*(Björnsson y cols., 2005) en 2005 plantea conservar una memoria de los nodos visitados. En una tabla hash se conserva $\hat{g}(n)$ y $\hat{h}(n)$ de cada nodo visitado, además de crear dos listas, llamadas *ahora* (now) y *después*(later) en forma de listas ligadas. El valor mínimo de f se guardará en cada iteración en la variable memoria f_{limit} la cual en un inicio será infinita. Posteriormente se iniciará a recorrer los nodos por la cabeza de la lista *now*. Los nodos que sean mayores en costo que la variable memoria f_{limit} serán agregados al final de la lista *later* y los que sean menores se expandirán, agregando sus sucesores después del nodo. En cada comparación se conservará una variable que

llamaremos $f_{\text{mín}}$ con el costo mínimo que se tiene hasta el momento en la estimación de $\hat{f}$, y se comparará cada nuevo valor con el existente.

Una vez esté vacía la lista *now* se terminará la iteración, se asignará el valor de la variable de $f_{\text{mín}}$ a f_{limit} y se volverá a iniciar una nueva iteración sobre la lista *later*. El proceso se llevará acabo hasta que la lista F esté vacía, lo cual indicará que no existe un camino al nodo destino, o al encontrar el nodo destino.

La gran ventaja de esta búsqueda transversal de izquierda a derecha es el hecho de que no se requiere una lista ordenada o cualquier otro tipo de ordenamiento, lo cual acelera en gran medida la búsqueda, pero si bien es más lento Fringe Search que IDA*, el nuevo algoritmo evita los estados repetidos y mejora el camino resultante a través de la tabla hash en la cual almacena el estado de todos los nodos visitados.

En el algoritmo 2.2.2 se puede observar a detalle la lógica de Fringe Search, utilizando los elementos mencionados anteriormente. La posibilidad de utilizar una lista ligada para realizar la búsqueda, en vez de una lista ordenada, plantea una mejora significativa de su desempeño como se discutirá en los siguientes capítulos.

2.2.2.1 Heurística

Fringe Search también es un algoritmo de búsqueda informado al utilizar las funciones heurísticas que se ha expuesto en A*. La fórmula para poder calcular el costo del camino $f(n) = g(n) + h(n)$ sigue conservándose al igual que las partes de la misma. También se debe de estimar el camino óptimo mediante $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$ por la cual se conocerá el probable costo. Esta información será de utilidad para diferenciar los elementos relevantes a ser evaluados mediante f_{limit} y los que no lo son. Cada nodo que supere el valor de estimación de f_{limit} será pasado a la lista *later*. Esta lista no será expandida ni se pretenderá llevar acabo operación alguna sobre ella, y será guardada hasta la próxima iteración, en la cual quizá ya haya cobrado relevancia. El valor mínimo entre el valor estimado $\hat{f}$ y $f_{\text{mín}}$, el que se guardará nuevamente en $f_{\text{mín}}$ será, al final de ciclo, el nuevo valor utilizado en f_{limit}.

2.2.2.2 Apuntes sobre los caminos generados

Es importante poner en claro que por ningún motivo se puede considerar al camino encontrado por Fringe Search como óptimo, por lo que cualquier intento de utilizar el algoritmo con

el motivo de encontrar específicamente este camino llevará a un resultado erróneo. En problemas que implican la cuestiones como decisiones financieras, caminos óptimos en mapas reales, entre otros, los resultados pueden llevar a pérdidas no deseables para el usuario, pero en todos problemas donde el camino no es necesariamente el óptimo donde el tiempo de procesamiento es vital y se busca intensamente reducir los tiempos de ejecución el algoritmo Fringe Search sería ideal. Si se toma en cuenta que los caminos tienden a ser óptimos, en una juego de vídeo, el generar un camino en un juego de mapas de una manera más rápida será un gran avance. El tiempo de cálculo podría ser utilizado en otros procesos más importantes, como por ejemplo en gráficos de mejor calidad o en inteligencia artificial más desarrollada o incluso mundos más grandes. El usuario por otro lado, en aplicaciones así, rara vez podrá notar alguna diferencia entre el comportamiento de los agentes a la hora de desplazarse por el mapa.

2.2.3 Implementación

Al igual que en el texto de los autores(Björnsson y cols., 2005) se utilizó para la implementación de las listas *now* y *later* una única lista doblemente ligada, la cual es una serie de punteros a una tabla que contiene todos los registros de los nodos del gráfico. Esta lista tiene punteros al inicio y al final de la misma, al igual que un puntero al nodo actual sobre el que se está trabajando. Este nodo es el inicio de la lista *now* y todos los elementos que se encuentran a su izquierda serán los pertenecientes a *later*, mientras que todos aquellos nodos (incluyendo al mismo nodo cabecera) que se encuentren a la derecha se encontraran en la lista *now*. Los nodos que se agreguen a la lista *now* mediante la función $\Gamma(n)$ se agregarán directamente a la derecha del nodo cabecera, de tal manera que serán evaluados inmediatamente. Si el valor f del nodo cabecera es mayor que f_{limit} entonces el puntero cabecera apuntará al siguiente nodo a la derecha. Esto permitirá reducir al mínimo las operaciones sobre la lista. Si se usaran dos listas distintas para realizar las operaciones, por mínimo que sea el costo de estas operaciones, la cantidad de las mismas perjudicaría al desempeño de la implementación.

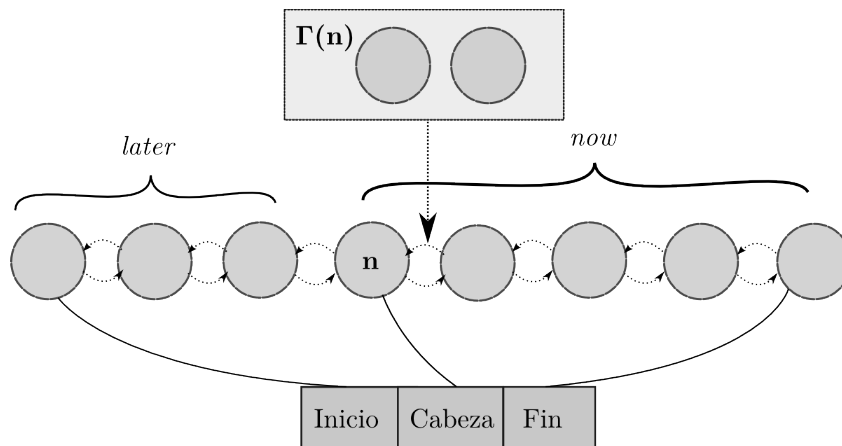

Figura 2.1: Implementación de las listas *now* y *later*.

Por otro lado, se utilizó una tabla en forma de arreglo, ya antes mencionada donde se contiene los datos de f y g, además de las banderas si el nodo se encuentra o no en la lista, si ha sido visitado y si es válido. El índice de dos dimensiones de la tabla funciona como la calve de una tabla hash, pero el valor de f y g, además de las banderas, también podrían ser guardadas en una tablas hash perfecta para los gráficos que no sean de una estructura de malla.

2.2.4 Análisis del código fuente

Para finalizar la discusión de Fringe Search se abordará el código fuente de la implementación que fue realizada para probarla en contra de A*. En primer lugar será necesario mostrar las estructuras de datos. El nodo de la estructura Fringe Search será conocido como *fnode* y consta de todas las variables necesarias para su trabajo, entre ellas el puntero a su nodo padre y un puntero al elemento de la lista doblemente ligada *now-later*. Se generará un arreglo dinámico según el tamaño del gráfico que será leído de con el mismo mecanismo que A*.

```c
typedef struct fnode{
    int                 valid;
    int                 visited;
    int                 inlist;
    char                id[ID_STRING_LENGTH];
    int                 index;
    int                 deph;
    long                f, g, h;
    int                 coord;
    int                 count;
    int                 x,y;
    struct fnode        *parent;
    dlnode              *lnode;
}fnode;
```

Estructura del Nodo Fringe Search

Como se puede apreciar, es posible acceder a cualquier nodo a una velocidad de $O(1)$ si se conoce el lugar por (x, y) y a su vez identificar por este nodo su valor f, g, h además de saber si ha sido visitado, si se encuentra en la lista, si es válido y recuperar sus coordenadas.

Este arreglo dinámico se puede acceder desde el puntero **cache* desde el objeto Fringe Search llamado *fobj*. También contiene este objeto punteros hacia la lista doblemente ligada, y hacia la secuencia de resultado. Como datos complementarios se conoce mediante esta estructura el nodo de salida, el nodo objetivo, el tamaño del arreglo, y otras banderas necesarias.

```c
typedef struct fobj{
    dlnode              *listnode_table;
    dllist              *list;
    fnode               *cache;
    fnode               *result;
    int                 xgoal, ygoal;
    int                 xsource, ysource;
    int                 xsize, ysize;
    int                 size;
    int                 compleat_path;
    int                 debug;
    char                output[21];
}fobj;
```

Estructura del objeto Fringe Search

Una vez definidas las estructuras de datos se llamará a la creación del nuevo objeto Fringe Search con los argumentos de la línea de comandos, que en este caso es casi el mismo que A*. Con el objeto creado a partir del archivo principal *main.c* se puede llamar la función principal de Fringe Search *finge_search(fobj *)*.

```
fringe_master = fs_create(source.x,
                          source.y,
                          goal.x,
                          goal.y,
                          x_tildes ,
                          y_tildes ,
                          argv[6]);
...
fringe_search(fringe_master);
```

Creación del objeto y llamado a la función de inicio

A continuación será realizado el mismo procedimiento que con A* localizando los punteros para el nodo de inicio y el nodo objetivo, seguido de una comprobación para saber si los nodos se encuentran o no dentro del arreglo, al mismo tiempo se valida si es un nodo que se pueda usar.

```
goal = &(fringe->cache[COORD(fringe->xgoal, fringe->ygoal)]);
start = &(fringe->cache[COORD(fringe->xsource, fringe->ysource)]);

n = fs_getnode(fringe, fringe->xgoal, fringe->ygoal);
```

Será agregado el nodo de salida a la lista abierta y sera creado f_{limit} con $\hat{h}(n)$ del inicio de nodo. Cabe resaltar que poner todos los nodos del arreglo en NULL ya fue hecho desde la función de creación del mismo por lo que ya no será necesario hacerlo nuevamente aquí.

```
// Fringe<-(s)
fs_movtonow(fringe, fringe->xsource, fringe->ysource);
start->inlist = 1;

// Cache C[start]<-(0, null)

fringe->cache[COORD(fringe->xsource, fringe->ysource)].g = 0;
fringe->cache[COORD(fringe->xsource, fringe->ysource)].parent = NULL;

// C[n]<-null for n != start
// is already done in fs_create()

//flimit<-h(start)
flimit = h(fringe, start);

//found<-false
found = 0;
...
```

Una vez que estén listas todas las variables y punteros necesarios, se dará inicio al ciclo principal el cual terminará en el momento en que se encuentre el nodo objetivo, o cuando la lista doblemente ligada se encuentre completamente vacía. Al inicio de cada iteración se definirá un nuevo $f_{\text{mín}}$ con el valor de infinito, además de utilizar el primer nodo de la lista doblemente ligada guardando su valor en la variable *nnode*.

```
while(!found && !dllist_isempty(fringe->list))
{
    //fmin<-infinte
    fmin = INF;
    nnode = fringe->list->start;
    ...
```

El segundo ciclo apuntará al nodo guardado en la variable *nnode*, obtendrá el valor guardado de $\hat{g}(n)$ y calculará la función heurística para el nodo en turno mediante $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$.

```
do
{
    n = (fnode *)(nnode->data);
    ...

    //(g, parent)<-c[n]
    g = n->g;

    // f <- g + h(n)
    f = g + h(fringe, n);
    ...
```

En dado caso de que $\hat{f}$ resultante sea superior al límite que se ha establecido para el ciclo actual, entonces el segundo ciclos se interrumpirá para continuar al iniciar el proceso con un nuevo nodo, este nuevo nodo será apuntado desde *nnode->next* y se contendrá en el ya existente *nnode*. Esta operación es una de las partes centrales de la idea de Fringe Search, ya que se limitará a recorrer los nodos existentes pero sin un costoso de ordenamiento, simplemente se avanzará de un nodo al otro de la lista.

Por otro lado, es hora de revisar si el nodo actual es el nodo objetivo, siendo así se interrumpe la función y se establece la bandera *found* como verdadera. De esta manera al salir de la función se podrá comprobar si la búsqueda tuvo éxito.

```
if(f>flimit)
{
    fmin = min(f, fmin);
    nnode = nnode->next;
    continue;
}
if(n == goal)
{
    found = 1;
    break;
}
```

A continuación se llevará acabo la función $\Gamma(n)$ para encontrar todos los nodos sucesores del nodo activo, los cuales serán evaluados para ver si son nodos válidos y de ser la respuesta afirmativa el nuevo valor de $\hat{g}(n)$ el cual permitirá tomar decisiones en el algoritmo. La función $\Gamma(n)$ utiliza las ocho posibles direcciones, asignando a cada una de ellas un número de cero a siete, y este número será interpretado por la función *get_iso_vector(int)* para devolver un vector que exprese el desplazamiento que debe realizarse. El costo del desplazamiento por otro lado será calculado por una función llamada *cost(vector)* la cual permite evaluar el nuevo costo según si el desplazamiento es horizontal o vertical.

```
for(i = 7; i >= 0; i--)
{
    vect = get_iso_vector(i);
    s = fs_getnode(fringe, (vect.x + n->x), (vect.y + n->y));
    ...
    gs = g + cost(vect);
    ...
```

Si el nodo sucesor indicado por la función $\Gamma(n)$ ya ha sido visitado con anterioridad será necesario llevar acabo algunas pruebas. El valor anterior del nodo sucesor s será alojado en una variable temporal y comparada con el nuevo costo de llegar a el estipulado en $\hat{g}(n)$. En dado caso de que el nuevo costo sea superior o igual que el nuevo costo se interrumpirá el ciclo actual y se continuará con el siguiente nodo encontrado. Pero si el nuevo nodo también se encuentra en la lista y cumple con la condición de que su costo estimado g sea menor al anterior, entonces se indagará si se encuentra en la lista, de ser así se removerá de ella.

```
if(s->visited)
{
    gi = s->g;
    if(gs >= gi)
    {
        continue;
    }
}
if(s->inlist)
{
    dllist_removefromlist(fringe->list, s->lnode);
    s->inlist = 0;
}
```

Una vez evaluado las pertenencias anteriores será necesario calcular $\hat{f}(n)$, asignar el nodo padre y designar como visitado para que posteriormente pueda ser insertado en la lista doblemente ligada en el lugar inmediato derecho al nodo actual.

```
s->visited = 1;
s->g = gs;
s->parent = n;
s->f = gs + h(fringe, s);
dllist_insertafter(fringe->list, s->lnode, n->lnode);
s->inlist = 1;
```

Por último, y habiendo agregado todos sus nodos sucesores se eliminará el nodo actual de la lista para que se pueda seguir adelante en los ciclos antes mencionados.

```
dllist_removefromlist(fringe->list, nnode);
n->inlist = 0;
nnode = tmp_node;
```

Para concluir con esta breve exposición acerca de la implementación Fringe Search será adecuado mencionar que todas las operaciones que se ha podido apreciar no son costosas, no implican listas de prioridad de alto consumo de tiempo de procesamiento como es el caso de los usados en A* y tiene en lo general una implementación más sencilla. Estas ventaja habría que considerarse en estudios posteriores donde se puede explotar las ventajas de las estructuras de datos usados en operaciones de concurrencia tanto paralelos como distribuidos, temas en los que los algoritmos paralelos de A* se han estancado durante los últimos años.

Algoritmo 5 Fringe Search

2.2.2 con base en (Björnsson y cols., 2005)

Entrada: G, n_s, n_g
Salida: Camino
fringe_search(G, n_s)
  Fringe $F \leftarrow (0, null)$
  C[n] $\leftarrow$ nulo **para todo** $n \neq g_n$
  $f_{limit} \leftarrow h(g_n)$
  encontrado $\leftarrow$ falso
  Hasta encontrado = verdadero **o** F esté vacío:
    $f_{\text{mín}} \leftarrow \infty$
    Itera sobre los nodos $n \in F$ desde izquierda a derecha:
      $(g, pariente) \leftarrow C[n]$
      $f \leftarrow g + h(n)$
      Si $f > f_{limit}$
        $f_{\text{mín}} \leftarrow \text{mín}(f, f_{\text{mín}})$
        continuar
      Fin si
      Si $n = n_g$
        encontrado $\leftarrow$ verdadero
        romper
      Fin si
      Itera sobre $s \in \Gamma(n)$ desde izquierda a derecha:
        $g_s \leftarrow g + costo(n, s)$
        Si $C[s] \neq$ nulo
          $(g', pariente) \leftarrow C[s]$
          Fin si
          Si $g_s \geq g'$
            continuar
          Fin si
        Fin si
        Si $s \in F$
          Remueve s de F
        Fin si Insertar s en F después de n
        $C[s] \leftarrow (g_s, n)$
      Fin iterar
      Remueve n de F
    Fin iterar
    $f_{limit} \leftarrow f_{\text{mín}}$
  Fin hasta
Fin Fringe Search

Capítulo 3

Método y técnica

En este trabajo se utilizará el método científico de índole analítico sintético utilizando la lógica dialéctica. Para poder realizar el trabajo se sigue el camino de una amplia investigación documental con la cual se recaba la extensa investigación existente sobre el tema, para pasar al aspecto práctico de la implementación de los algoritmos y experimentar con su comportamiento para posteriormente poder realizar nuevamente una reflexión teórica . La contradicción entre los postulados generalizados de que A* es el algoritmo idóneo y la posibilidad de encontrar la solución con nuevos algoritmos puede llevar a una nueva visión del problema. Esta contradicción es el centro del trabajo, y por lo tanto la investigación es adecuada para el uso de una lógica dialéctica.

3.1 El método

A continuación se realizará una breve discusión sobre los aspectos metodológicos y sobre su aplicación en este trabajo, con lo cual se podrá tener un panorama más amplio del procedimiento que se tuvo en este trabajo.

3.1.1 El método dialéctico

Para poder estudiar el tema, será necesario plantear que las matemáticas discretas no son una creación de la mente humana, del pensamiento puro, que puede vivir aislado de la realidad y su entorno, sino que son, en última instancia, una abstracción del mundo material en el cual nos encontramos y producto de una necesidad. Como dijo Engles “la matemática pura tiene como objeto las formas especiales y las relaciones cuantitativas del mundo real, es decir, una materia muy real. El hecho de que esa materia aparece en la matemática de un modo sumamente

abstracto no puede ocultar sino superficialmente su origen en el mundo externo. Para poder estudiar esas formas y relaciones en toda su pureza hay, empero, que separarlas totalmente de su contenido, poner éste aparte como indiferente"(Engels, 1976, p. 26). El método para acercase al problema, para plantearlo y para resolverlo no puede estar alejado de esa realidad, limitando su existencia a esquemas cuantitativos o cualitativos, si no se debe buscar una concatenación de los fenómenos de la teoría con la práctica concreta. Esto sirve, a su vez para disernir entre los aspectos verdaderos o falsos de la hipótesis y la hipótesis alternativa. La práctica como criterio de verdad es comprobado en el método científico a través de la experimentación, y se comprueba día a día en el uso de las aplicaciones concretas de los productos del pensamiento. "La verdad es subjetiva en el sentido de que construye un conocimiento humano, pero es objetiva por cuanto el contenido del conocimiento verdadero no depende ni del hombre ni de la humanidad"(Kopin, 1966, p. 159)

Por lo tanto, un gráfico es una abstracción de la realidad y fue la necesidad de resolver uno o varios problemas la que ha llevado a abstraerlo. Por ello, si bien la discución sobre los algoritmos son propios de esa abstracción, forzosamente deberán volver a la realidad para su aplicación: "el pensamiento no puede jamás obtener e inferir esas formas de sí mismo, sino sólo del mundo externo. Con lo que se invierte enteramente la situación: los principios no son el punto de partida de la investigación, sino su resultado final, y no se aplican a la naturaleza y a la historia humana, sino que se abstraen de ellas; no es la naturaleza ni el reino del hombre los que se rigen según los principios, sino que éstos son correctos en la medida en que concuerdan con la naturaleza y con la historia."(Engels, 1976, p. 22) Tanto en aplicaciones sociales como naturales, la teoría de gráficos es usable, con sus representaciones se ha logrado aplicar algoritmos generales.

Por último, como ya se ha referido que este mundo material y su abstracción teórica se encuentran en constante movimiento, ya que en ellos se encuentran contradicciones internas y externas que se dan con también hacia el mundo exterior. Entender que estas contradicciones mueven el desarrollo de las ciencias es muy importante, y es parte fundamental del estudio que se realiza en este trabajo. "La infinitud es una contradicción y está llena de contradicciones. Ya es una contradicción el que una infinitud tenga que estar compuesta de honradas finitudes, y, sin embargo, tal es el caso. La limitación del mundo material lleva a no menos contradicciones que su limitación, y todo intento de eliminar esas contradicciones lleva, como se ha visto, a nuevas y peores contradicciones. Precisamente porque la infinitud es una contradicción, es in-

finita, un proceso que se desarrolla sin fin en el espacio y en el tiempo. La superación de la contradicción sería el final de la infinitud."(Engels, 1976, p. 39) Por lo tanto "si nos empeñamos en atenernos exclusivamente a un punto de vista, considerándolo como absoluto por oposición al otro, o si, respondiendo a las necesidades momentáneas del razonamiento, saltamos del uno al otro, permaneceremos cautivos de la estrechez del pensamiento materialista; no captaremos la concatenación y nos embrollaremos en una contradicción tras otra."(Engels, 1974, p. 26) Es así que la hipótesis y la hipótesis alternativa se enfrentan en este estudio comparativo para poder brindar luz sobre la contradicción de los dos algoritmos.

3.1.2 Tipo de estudio

El estudio que se realiza es de comparación de dos soluciones en el cual se pretende resolver el mismo problema. Al plantear la comparación entre ellos será necesario ver las contradicciones internas de cada uno de ellos, así como las que presentan para con el problema en general, y entre ellas. A su vez, por ser un proceso de abstracción, en que los gráficos son capaces de representar fenómenos vinculados a la realidad, no pueden ser expuestos únicamente contemplando su abstracción, sino que deben ser vinculados nuevamente a la práctica concreta. En este estudio se tiene una implementación, y casos concretos de laberintos, además de un enlace con aplicaciones, lo cual permitirá retomar el enlace con el origen, y criterio de verdad.

3.1.3 Fuentes

Para poder recabar el marco teórico se oarte desde la teoría general de gráficos de las matemáticas discretas, hasta los algoritmos de mejor camino desarrollados en proposiciones de algoritmos concretos y versiones de los mismos. Es necesario reunir los artículos de revistas y ponencias donde se expusieron originalmente estos, y una bibliografía considerable para la exploración de los temas ya ampliamente aceptados. La investigación documental, sin embargo, se centra principalmente en artículos, tomando en cuenta que gran parte de la discusión al respecto se realiza en estos foros, y no se encuentra aún expuesto en forma de libros. Se ha puesto un especial interés en buscar los textos originales en los cuales los algoritmos discutidos son expuestos, complementados por otros que analizan su complejidad, variables y alternativas.

3.1.4 Indicadores y variables

En el estudio se utilizan dos variables dependientes, que son el tamaño del camino y el tiempo de ejecución, mientras que se utiliza una variable independiente que es el tamaño del gráfico. En cada ejecución se buscará el camino más corto desde los dos extremos más alejados del gráfico para poder obtener datos útiles al estudio. Como se ha mencionado en la introducción se ha dejado fuera la medición de la cantidad de memoria utilizada, dejando esta variable para un estudio posterior.

También se cuenta con una serie de constantes, las cuales son el equipo en el cual se probó la ejecución, los mapas aleatorios (que fueron generados una única vez) sobre los cuales se realizaron todas las pruebas, así como el equipo de cómputo utilizado.

3.2 La técnica utilizada

En la implementación de los algoritmos se decidió por la utilización del lenguaje de programación C, con el cual se puede controlar completamente el comportamiento de la aplicación explotando sus cualidades de bajo nivel y velocidad. Lenguajes de alto nivel no son idóneos para este trabajo por incrementar los tiempos experimentales de una manera muy importante y no permitir trabajar directamente con las estructuras de datos a utilizar y así hacer casi imposible la tarea de la optimización del código.

Para poder agilizar el trabajo se utilizaron algunas bibliotecas de terceros bajo licencias de código abierto o libre, las cuales se incorporaron al código que se encuentra anexado. El código se encuentra disponible bajo la licencia GPL v.3[1] que decidí utilizar con el propósito de compartir el conocimiento con todo aquel interesado en extender la investigación sobre el tema. Con el fin de agilizar el tiempo de desarrollo y hacerlo multiplataforma se utilizó la herramienta *autotools* como entorno de producción, la cual genera archivos *Make* para diferentes compiladores y sistemas operativos.

En la implementación de A* se utilizó como lista cerrada una tabla hash perfecta en forma de arreglo bidimensional que representa cada nodo posible y de esta manera agilizar el acceso al estado del mismo. En la lista abierta, la cual representa el aspecto que más tiempo de cálculo consume, se utilizó una *Skiplist*(Pugh, 1990) en vez del tradicional montículo binario (*Binary*

[1] Licencia Pública General (o General Public Licence en inglés) es la licencia desarrollada por la Free Software Foundation para mantener el código libre. Su versión tres fue publicada en 29 de junio de 2007.

Heap). Los dos tienen un comportamiento de $O(\log F)$(Patel, 2014)(en el texto se explicará el costo por operación) con la ventaja para *Skiplist* de comportarse mejor en listas grandes, por lo cual se usarán en este trabajo para medir su comportamiento en gráficos grandes.

Para Fringe Search se utilizó una lista doblemente ligada tanto para la lista ahora (*now*) como para la después (*later*), las cuales se diferencian con un puntero a la cabeza de la lista *now* y que recorrerá la lista de izquierda a derecha, y una tabla hash perfecta en forma de arreglo bidimensional que llevará los registros.

Capítulo 4

Resultados

4.1 Comparación entre A* y Fringe Search

Como se ha visto, los dos algoritmos A* y Fringe Search tratan de resolver el problema de búsqueda del mejor camino entre dos nodos en un gráfico. Se sabe también que A* garantiza el mejor camino entre ambos puntos, mientras que Fringe Search no. Lo que haré en este capítulo es plantear el problema sobre el terreno práctico en el cual se abordará la cuestión de la velocidad de ejecución y el tamaño del camino para poder generar conclusiones sobre los mismos.

4.1.1 La experimentación

En esta sección serán expuestos de manera breve algunos aspectos del ambiente, la forma y los detalles de la experimentación que servirán para poder comprender mejor los resultados a los cuales se pudo llegar en este trabajo.

4.1.1.1 El entorno de experimentación

Para llevar acabo los experimentos se utilizó una computadora GNU/Linux Mint 15 olivia (x86-64), con un kernel 3.8.0-35 corriendo sobre un procesador Ingel Core i7-3517U con cada CPU a 1.90GHz y 3.8 GiB de memoria. Los dos algoritmos fueron escritos en ANSI C99 y construidos con el compilador GCC 4.7.3. Además se utilizaron en la generación de los archivos ejecutables las herramientas autotools.

Para la generación de los datos se usaron una serie de scripts en lenguaje Bash versión 4.1 y para el análisis estadístico se utilizó R Versión 2.15.2 para 64 bits.

4.1.1.2 El experimento

Para poder comparar los dos algoritmos se generaron treinta y cuatro mapas aleatorios guardados en un BMP cada uno, los cuales serán transformados a gráficos por el mismo ejecutable del algoritmo. Estos mapas fueron creados por un script python que utiliza el algoritmo DFS con algunas modificaciones.

El siguiente código muestra la manera en que se realizó la recolección de los datos de tiempo de ejecución, tamaño del gráfico y nodos del camino. Para poder medir el tiempo se utilizó el comando *time* de Unix; para medir el tamaño retomo el número de pixels de los 34 mapas, los cuales parten de 100x100 incrementándose en 100 cada ciclo hasta alcanzar el tamaño de 3400x3400, lo cual implica 11 560 000 nodos; y por último para medir el tamaño del camino se toma la información que genera el ejecutable del algoritmo.

```bash
#!/bin/bash
ulimit -Sv 3000000
for a in `seq 1 34`; do
  ((x=$a*100));
  ((X=$x-1));
  FILE="img/maze1_${x}x${x}.bmp";
  FILE2="img/m1_res_${x}x${x}.bmp";
  rm res.txt
  (/usr/bin/time -f "real \t %e" ./astar 1 1 $X $X $FILE $FILE2 0 ) &>> res.txt
  p="$(awk '/^Size/ {print $5}' res.txt)"
  awk -v size=$x -v path=$p '/^real/ {print $2 " " size " " path}' res.txt
done
```

Script Bash de medición de tiempo de ejecución

Los datos obtenidos a través de este sistema son considerados muestras de una población desconocida al no ser posible acceder a una población finita. Las pruebas podrían repetirse una cantidad indeterminada de veces sin llegar a cubrir todos los resultados posibles. Se parte de que cada mapa tiene su propio camino óptimo de un punto al otro, de que existen una gran cantidad de posibles pares, a su vez que existen una infinita cantidad de mapas posibles, y que en cada ejecución factores externos al algoritmo influyen en el tiempo de ejecución de los mismos. Por lo tanto fueron realizadas 34 muestras de mapas y veinte ejecuciones de los mismos, con un camino probado sobre cada mapa para poder obtener datos que garanticen una inferencia estadística

adecuada.

4.1.1.3 Presentación visual

Para poder entender mejor el problema se presenta un ejemplo gráfico de un mapa 100x100 el cual ha sido resuelto por A* y Fringe Search como es posible ver en las figuras 4.1 y 4.2. El resultado de cada ejecución es a su vez un BMP con el mapa original, una área gris trazada sobre el y un camino denotado en azul.

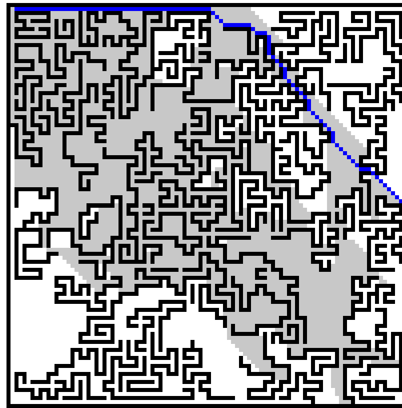

Figura 4.1: Camino encontrado por A* en un gráfico 100x100.

En la figura 4.1 se aprecia como A* comienza a buscar centrándose en la posibilidad de que exista, según la heurística, un camino directo al objetivo, pero al no hallar mejores resultados en esa dirección, comparadas con otras que ya había explorado tiene que retomar elementos antes descartados. De esta manera examina primero la opción más prometedora que es un recorrido en linea recta llegando a un punto en el que no hay paso poco antes de llegar a su objetivo. Los nodos que han recorrido y se han integrado a la lista cerrada son la mayor parte del gráfico. Como se ha expresado si $f(s) = h(s)$ y por lo tanto todo nodo en $h(s)$ estuvieran sobre el mejor camino no se tendría un sólo elemento en la lista cerrada. Pero como en este caso nuestra función heurística no pudo determinar un camino cerrado antes de llegar al objetivo ha tenido que explorar gran parte del gráfico.

Una vez analizar el resultado gráfico generado por A* se inicia una comparación gráfica con Fringe Search cuyo comportamiento es visible en la figura 4.2. El camino es en lo general casi el mismo así como el espacio que ha explorado.

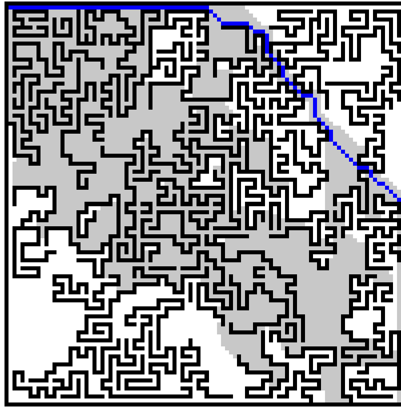

Figura 4.2: Camino encontrado por Fringe Search en un gráfico 100x100.

Pero cada nodo que ha recorrido, que en este caso es el espacio en gris, ha costado mucho menos expandirlo dado a que no es necesario agregar cada nodo a un lista de prioridad. Cabe destacar que a pesar de lo que podría creerse a simple vista, el camino encontrado no es el óptimo hallado previamente por A* que es de 121 nodos, comparado con los 123 encontrados con FS. La diferencia que representa tan sólo el 1.65 % del camino será un tema de discusión en este apartado. También hay que considerar la velocidad a la que se generó este camino: A* tardó 0.03 segundos en encontrarlo; mientras que FS 0.02 segundos. Presento los datos de esta muestra únicamente con el propósito de ilustrar la discusión que a continuación se llevará acabo, donde inferiré sobre los resultados obtenidos y así generar conclusiones sólidas.

4.1.2 Comportamiento

Una vez presentados los aspectos del experimento pasaré a examinar los datos que se ha podido recopilar y encontrar cualidades del comportamiento que tienen los dos algoritmos, primero en un caso sencillo con un mapa pequeño, para pasar a un caso generalizante con mapas aleatorios, a lo cual describiré las regresiones que logré inducir.

4.1.2.1 Sobre un mapa de una sola medida

Analicemos primero los datos descriptivos de un experimento de mil muestras del comportamiento de los dos algoritmos sobre un mapa de tamaño 100x100. La media encontrada es de 0.0094 segundos en A* y de 0.0068 segundos para FS, con una mejora del 72 % para FS. Agregamos a esto que el tiempo de ejecución de A* es más variable que su contra parte mostrando una desviación estándar de 0.006 segundos contra 0.0046. Realizando una prueba t con una hipótesis

alternativa de que la media de A* es mayor con un 95 % de grados de confianza, aplicando una corrección Welch se deshecha la nula, con el resultado de que ciertamente se puede afirmar que FS en el caso de un mapa 100x100 de estas características es superior a A*.

4.1.2.2 Sobre mapas aleatorios de medidas diversas

Sin embargo, para avanzar en nuestra afirmaciones será pertinente analizar el comportamiento de los dos algoritmos con mapas aleatorios de diversos tamaños. Usando los datos obtenidos mediante el estudio antes descrito se podrá apreciar gráficamente el comportamiento de los algoritmos de manera más clara en la figura 4.3.

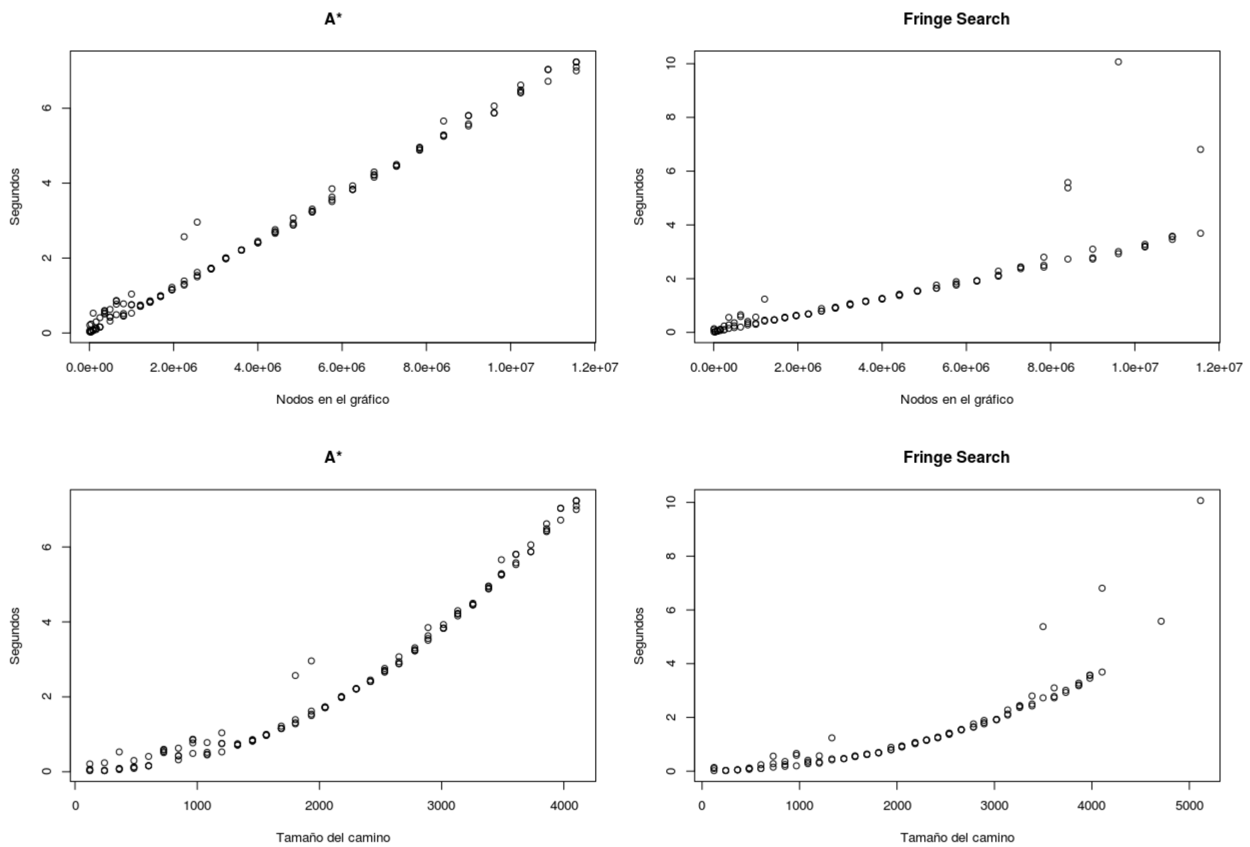

Figura 4.3: Comportamiento de A* y Fringe Search con respecto al número de nodos del gráfico y al tamaño del camino.

En este caso, para A* la media de tiempo de ejecución es de 2.5010 segundos, contra 1.456 para FS, lo cual muestra que la tendencia se conserva en ambos muestreos. Sin embargo, el tiempo máximo que tardó un algoritmo para concluir en FS fue de 10.07 segundos comparados con los 7.2s de A*. A su vez se tiene que la mediana de FS también es menor que A* con

un 1.02 contra 1.98, donde entre el primer cuantil y el tercero se encuentra en un intervalo de 0.71s a 4.10s en A* y de 0.41s a 1.45s en FS, lo cual corresponde a una dispersión mayor de los resultados en A* que en FS.

4.1.2.3 Regresiones

Retomando la figura 4.3 se observan los valores aislados en FS que salen del comportamiento regular del algoritmo, los cuales corresponden a problemas en la solución del mapa y caminos mucho más largos que el óptimo. También se observa que puede existir una correlación lineal en el crecimiento del número de nodos en un gráfico con respecto al tiempo de ejecución. Trazando los datos correspondientes a esta regresión en la figura 4.4 se ve en rojo la correspondiente a A* y en azul la que se genera a partir de FS.

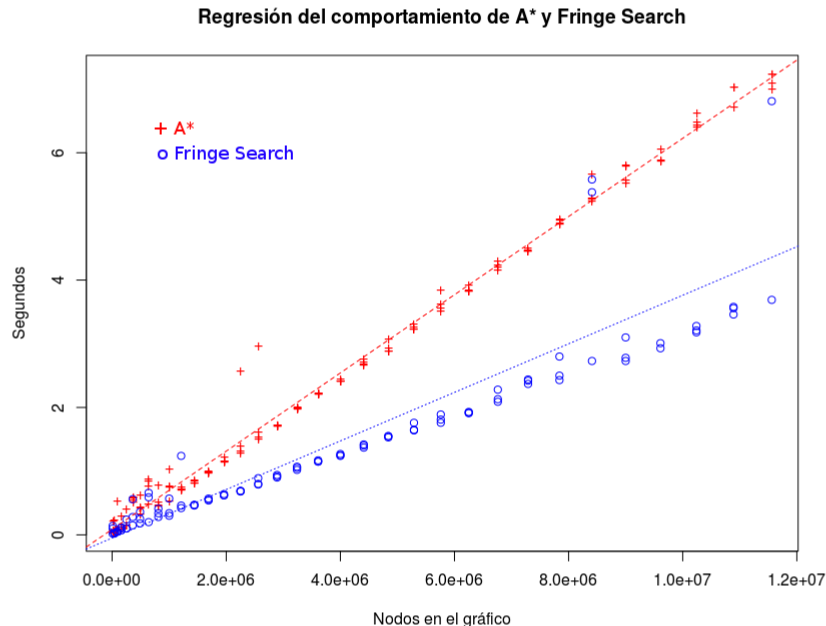

Figura 4.4: Comportamiento del tiempo de ejecución de los dos algoritmos con respecto al tamaño del gráfico.

En la regresión para los datos generados por A* se obtiene una fórmula $y = 6{,}140e - 07x + 8{,}6443e - 02$ que se ajusta muy bien a los datos con un estadístico R^2 de 0.9911. La correlación con un nivel de confianza del 95 % son aceptados en una prueba t.

Para la regresión para los datos generados por Fringe Search se obtiene una fórmula $y = 3{,}812e - 02x - 5{,}008e - 02$ que se ajusta de manera aceptable a los datos con un estadístico

R^2 de 0.72. En una prueba t con el 95 % de confianza acepta los datos de la correlación, sin embargo, para explicar el valor de R^2 debo mencionar que los datos extremadamente altos de los datos relevantes llevan a plantear una linea colocada en una posición más alta, lo cual se puede apreciar en la figura 4.5.

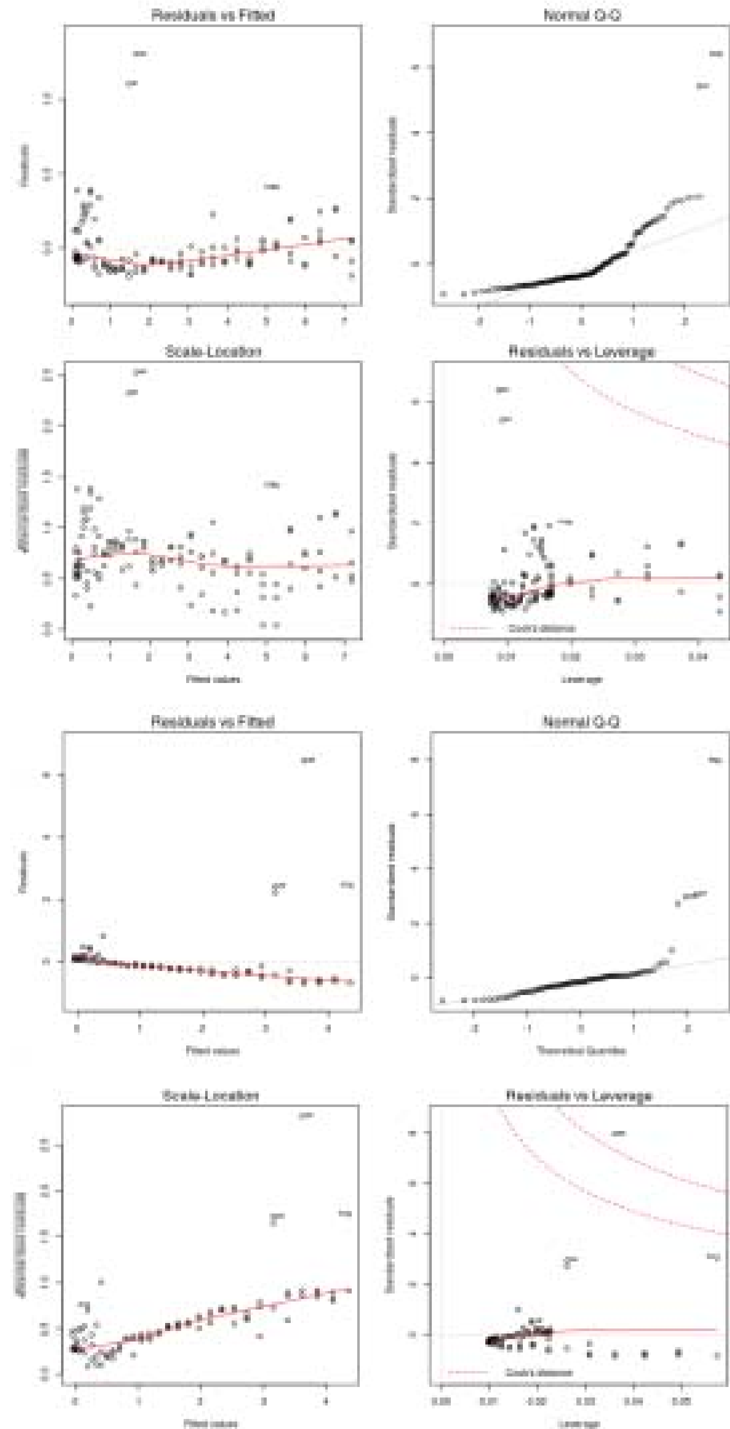

Figura 4.5: Residuos y datos relevantes para A*(arriba) y Fringe Search(Abajo)

En las figuras 4.6 se aprecia como los datos de los residuos son superiores en A* que en FS, explicando la desviación estándar de 2.155639 segundos en comparación con Fringe Search que tiene una desviación estándar de 1.565729. El estadístico R^2 por lo tanto no se explica a partir de la dispersión general de los datos obtenidos por FS si no por algunos datos relevantes que se puede apreciar en la gráfica de *Residuales contra relevantes* donde un dato de FS cae incluso en el tercer círculo. En lo general, con excepción de algunos datos aislados, existe un apego bastante bueno a la recta de regresión.

Con las regresiones pasadas se ha visto como el comportamiento general que se ha inferido por parte de los dos algoritmos muestran una velocidad de ejecución superior de FS al de A*. En las figuras 4.6 se aprecia como los datos de los residuos son superiores en A* que en FS, explicando la desviación estándar de 2.155639 segundos en comparación con Fringe Search que tiene una desviación estándar de 1.565729. El estadístico R^2 por lo tanto no se explica a partir de la dispersión general de los datos obtenidos por FS si no por algunos datos relevantes que se puede apreciar en la gráfica de *Residuales contra relevantes* donde un dato de FS cae incluso en el tercer círculo. En lo general, con excepción de algunos datos aislados, existe un apego bastante bueno a la recta de regresión.

Con las regresiones pasadas se ha visto como el comportamiento general que se ha inferido por parte de los dos algoritmos muestran una velocidad de ejecución superior de FS al de A*.

4.1.3 Análisis

Una vez visto el comportamiento de los dos algoritmos en un plano general pasaré a plantear las pruebas necesarias para desechar la hipótesis de esta tesis o aceptarla. Con base a los resultados de las pruebas empíricas comenzaré a desarrollar dos ANOVA, una enfocada a ver la relación que guarda la velocidad con las categorías A* y Fringe Search, y la segunda la relación de los algoritmos con el tamaño del camino.

4.1.3.1 Velocidad

El objetivo principal es encontrar un algoritmo con una velocidad de ejecución menor para así aumentar el rendimiento de las aplicaciones donde se usa A* actualmente. Los datos generados arrojan, como ya se ha visto, que la media de las muestras para A* es de 2.5010s con una desviación estándar de 2.15563s y una media de 1.456s con una desviación estándar de 1.5s. Los

datos se muestran de manera gráfica en la figura 4.6 donde se comparan los dos datos para a velocidad.

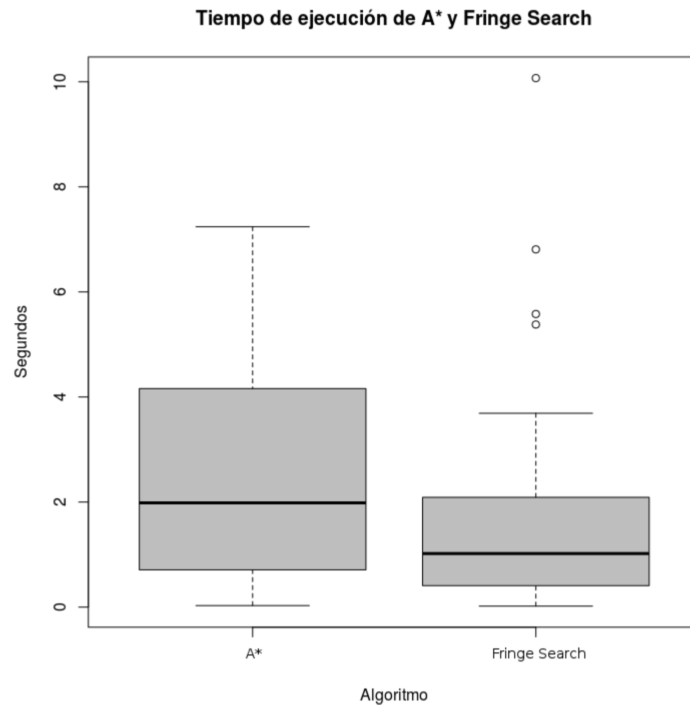

Figura 4.6: Comparación del tiempo de ejecución entre A* y Fringe Search sobre los mismos gráficos.

Aplicando una ANOVA que contempla la velocidad de ejecución con respecto al tipo de algoritmo se ve que la influencia es grande con 95 % confianza. Por lo que se puede concluir que el tipo de algoritmo define la velocidad de la solución del problema, haciendo válida nuestra hipótesis de que Fringe Search ofrece una solución más rápida al problema de la búsqueda de mejor camino con respecto a A*.

```
Analysis of Variance Table

Response: v
           Df Sum Sq Mean Sq F value    Pr(>F)
a           1  62.88  62.877  16.973 5.271e-05 ***
Residuals 233 863.17   3.705
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

Figura 4.7: Tabla ANOVA: relación entre la velocidad y los dos algoritmos

4.1.3.2 Camino

Ahora que se ha demostrado la velocidad superior de Fringe Search con respecto a A* falta ver si la diferencia de los caminos entre FS frente a A* es significativo. Para ello se tiene el siguiente resumen.

La media de caminos de A* es de 2084 nodos y 2120 de FS, con una mediana de 2046 nodos en A* y 2050 en FS. El primer cuartil en A* es de 1081 y el tercero de 3013, mientras que para FS en el primer cuartil 1086 y 3136 en el tercer. Las desviaciones estándar son de 1224.66 para FS y de 1175.901 para A*. Estos valores se pueden representar en la figura 4.8 donde se aprecia muy poca variabilidad de los caminos, con una notoria presencia de valores relevantes en FS. Esto se explica en que el máximo valor encontrado para FS de camino es de 5115 nodos en comparación con A* el cual tiene un camino máximo de 4105.

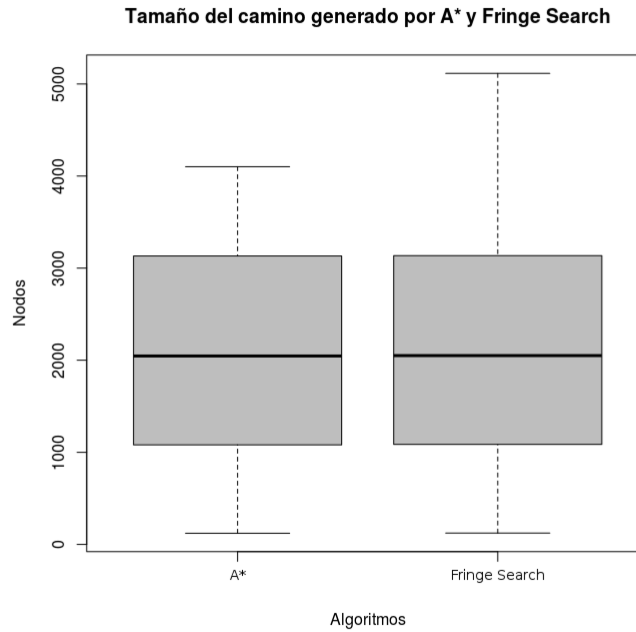

Figura 4.8: Comparación de los caminos generados A* y Fringe Search en los mismos gráficos.

A partir de estos datos, se realizará una ANOVA en la cual se prueba si es significativa la relación entre el camino y la categoría de algoritmo con un 95 % de confianza. Con base en la tabla ANOVA de abajo se concluye que no existe relación significativa del tamaño del camino con el tipo de algoritmo usado, por lo que es posible afirmar que la diferencia del camino generado por Fringe Search no representa una limitante importante que evita utilizar este algoritmo en vez de A* comprobándose así a integridad la hipótesis planteada en la introducción.

```
Analysis of Variance Table

Response: p
           Df    Sum Sq Mean Sq F value Pr(>F)
a           1     74676   74676  0.0521 0.8196
Residuals 233 333884031 1432979
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

Figura 4.9: Tabla ANOVA: Relación entre el tamaño del camino y los dos algoritmos

4.2 Aplicaciones en casos prácticos

Una vez que se ha visto a profundidad los algoritmos Fringe Search y A* es necesario explorar el horizonte de oportunidades en las cuales estos dos procedimientos pueden ser implementados, mostrando cómo diversos algoritmos serán usados en problemas muy específicos, las ventajas de su uso en uno u otro caso, para de esta manera poder usar de la mejor manera posible los resultados que presenta este texto.

Las ventajas que presenta Fringe Search pueden ser fácilmente mal usadas en problemas que tendrán requerimientos diferentes y harán que el rendimiento de los programas no sea óptimo, o que no logre generar una solución correcta.

Habría que recordar que además de los ejemplos típicos también se puede representar una máquina abstracta no determinista en forma de un gráfico para determinar una serie de decisiones a tomar para llegar a un cierto objetivo. Un ejemplo de esto es la manera en que un cubo de Rubik puede resolverse.

4.2.1 Búsqueda de mejores caminos

Varios problemas de mejores caminos no pueden ser resueltos por A* o Fringe Search ya que estos únicamente tratan el problema de encontrar el camino más corto de un nodo a otro. Para todos los demás problemas existen una serie de algoritmos que en este caso únicamente mencionaré para dar una breve idea de su existencia y su uso, pero no explicaré ninguno de ellos.

- **Dijkstra**(Dijkstra, 1959)

  Soluciona el problema del camino más corto con un único punto de origen, excepto si los arcos son negativos. Es muy usado en problemas de planeación, así cómo de precalculo, ya que inspecciona todos los nodos del gráfico y su relación con el nodo de inicio, con la respectiva penalización en el tiempo de ejecución. Los detalles de este procedimiento se han analizado en el capítulo uno de este trabajo.

- **Bellman-Ford**(Bellman, 1958)(Ford Jr., 1959)

  Para todos aquellos gráficos que en sus planteamientos tienen un gráfico con arcos negativos, y que necesitan encontrar el camino más corto a partir de un único nodo de origen,

no podrán realizarlo con Dijkstra, y necesitarán usar *Bellman-Ford*, que tiene una complejidad de $O(|V||E|)$. Es muy usado en el uso de protocolos de *Routing* para el protocolo IP. Sin embargo, tiene los defectos de no escalar bien; no tener una buena capacidad de adaptarse a cambios en las topologías de las redes al tener que volver a calcular los caminos.(Heineman y cols., 2008)

- **Floyd-Warshall**(Floyd, 1962)(Warshall, 1962)

  Este algoritmo encuentra todos los caminos entra cada uno de los pares de nodos del gráfico, sin importar si sus arcos sean negativos o positivos con la condición de que no deben contener ciclos negativos. La ejecución de este algoritmo tendrá una salida del tamaño de los caminos más cortos pero en su versión básica no presentará mayores datos.

  La complejidad del algoritmo es algo a tomar en cuenta ya que su comportamiento es de $O(|V|^3)$ lo cual lo hace uno de los algoritmos más costosos. Suele aplicarse este algoritmo en la resolución de expresiones regulares, para encontrar la raíz inversa de matrices reales y soluciones precalculadas en redes de transporte.

- **Johnson**(Johnson, 1977)

  Este algoritmo resuelve también el problema de encontrar el mejor camino entre todos los pares de nodos en un gráfico sobre arcos negativos, no negativos, con la condición de que no existan ciclos negativos. Pero a diferencia del anterior si arroja el camino mínimo, lo cual logra usando *Bellman-Ford* para computar una transformación del gráfico a arcos no negativos y así después utilizar Dijkstra. Tiene una complejidad de $O(|V|^2 \log |V| + |V||E|)$(Heineman y cols., 2008).

Para poner un caso práctico se tomará la investigación de operaciones que es un campo de estudio donde se aplica con gran frecuencia búsqueda de soluciones óptimas y por lo tanto es posible aplicar los mejores caminos. Si casi cualquier situación real es representable con gráficos, las operaciones también lo serán. Se eligieron dos ejemplos muy importantes para la informática administrativa que son ilustrativos de las aplicaciones posibles.

El problema del máximo rendimiento. En este problema se representa un gráfico donde cada nodo equivale una serie de estados o medidas y los arcos un cierto rendimiento. Es necesario plantear un rendimiento alto entre toda la red de posibilidades. En el procedimiento se determina

un camino dirigido de máximo beneficio o rendimiento del nodo inicial a todo otro nodo del gráfico donde cada arco está asociado con una medida sobre el rendimiento. (Ravindran, 2008, p. 44). Normalmente se utilizará algoritmos como Dijkstra o *Bellman-Ford* para poder realizar estas tareas, ya que se necesita encontrar un camino más idóneo de un punto de salida a todas las posibilidades en el gráfico.

El problema del reemplazo. La entrada es el total del flujo monetario y salida de dinero por comprar el equipo, usando el equipo un cierto período de tiempo y finalmente vendiéndolo. Esto puede ser representado por una estructura de red asumiendo que los nodos representan el tiempo y que los arcos representa una decisión de reemplazo. El costo del arco entre los dos nodos separados por k años es el costo total de comprar, mantener y vender el equipo por k. El objetivo de este tipo de problemas es determinar la mejor política de reemplazo sobre un periodo de T años. (Ravindran, 2008, p. 44)

Además de la investigación de operaciones existen muchos más usos para este tipo de algoritmos, como por ejemplo en el Análisis de Redes Sociales, para buscar la cercanía de dos personas entre los contactos de otras personas, donde es posible utilizar *Loyd-Warshall* o *Johnson*, con el fin de encontrar la relación de todos los pares en un gráfico, en este caso una red social, y generar de manera numérica la relación entre cada par de esta misma. Se utiliza en las sugerencias de amistades en *Facebook* o en notificaciones de seguridad sobre posibles redes criminales o persecuciones de grupos políticos.

4.2.2 Caminos en el mundo real

El mundo real es una espacio que puede ser representado en dos dimensiones o se pueden utilizar las tres dimensiones de las que se dispone, sin tomar el cuenta el tiempo (el cuál también podría incluirse). Por lo tanto, todo el puede ser representado a partir de un gráfico de malla para generar un mapa de terreno completo, o un gráfico que únicamente plantee algunos nodos en forma de poblaciones, puntos de tránsito, etc. y arcos en forma de caminos transitables entre estos nodos. Los dos llegarán ser abstracciones del mundo real para poder computar soluciones, tiendo en cuenta que gran parte del problema también depende de la calidad con que se abstrae el mundo real en el gráfico a tratar.

4.2.2.1 Problemas de transporte

"El problema de la búsqueda del camino óptimo es un importante estudio en el área de la transportación. ... Dependiendo de que cómo se asuma el tiempo del viaje en la red el problema del camino óptimo puede ser clasificado en dos: deterministas y estocásticos"(Zhou, 2008, p. 7). Dentro de los deterministas se encuentran los algoritmos como Fringe Search, A* y Dijkstra ya que calculan el mejor camino partiendo de hechos concretos y encuentran la opción más adecuada con relación a ellos, mientras que los estocásticos plantean probabilidades de encontrarse con tráfico, bloqueos, cierres de rutas, etc., además de los caminos y obstáculos ya conocidos.

La búsqueda del mejor camino en los problemas de transporte es comúnmente usado para resolver problemas como gasolina limitada, costo de casetas, penalizaciones por tráfico, mejores rutas de una ciudad a otra, encontrar la vía más idónea para llegar a cierta calle, etc. Ejemplos de aplicaciones conocidas son *Google Maps*, *Mapquest* que se puede ver en la figura 4.10(OpenStreetMap, 2014) o su motor *OpenStreetMap*[1] o la página de *CPUFE* para encontrar la ruta de casetas más económica. Los que han utilizado diversas de estas aplicaciones sabrán con seguridad que es muy importante especificar la optimización requerida. Existen ocasiones en que se pretende optimizar el tiempo de tránsito sin tomar el cuenta el costo de casetas o gasolina y viceversa.

Como *OpenStreetMap* es una aplicación abierta es posible tener un acercamiento más estrecho a su funcionamiento y por ello en la página wiki(OpenStreetMap, 2014) de referencia especifican el uso de algoritmos en diversas aplicaciones de escritorio que utilizan sus mapas con el fin de solucionar problemas particulares. *Simple Map Routing* es una implementación con un A* doble, *IMPORTIS* es una aplicación para optimizar rutas de transporte para diversos vehículos usando A* en C#, mientras que para dispositivos móviles *Spatialite* hace la búsqueda de caminos mediante Dijkstra o A*. Muchas más implementaciones pueden ser encontradas en la página que se cita en las referencias.

[1] Es un servicio abierto de mapas que pueden ser descargados o utilizados mediante una API, dentro de la idea de una base de datos libre. Puede ser accedida en: `https://www.openstreetmap.org/`

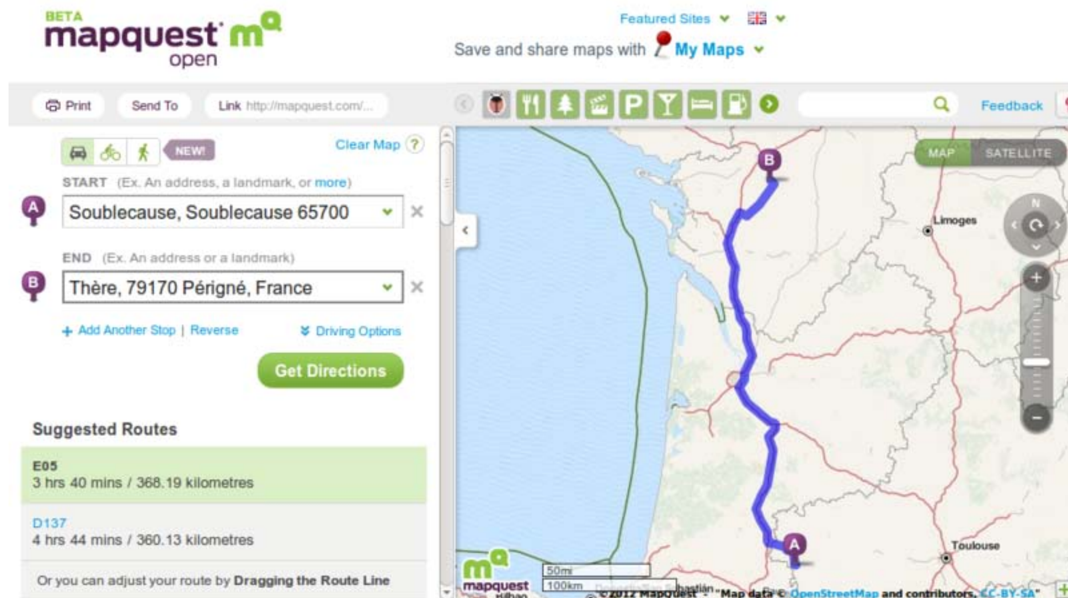

Figura 4.10: Búsqueda de caminos en MapQuest usando OpenStreetMap. La imagen fue tomada de la wiki de OSM

Dentro de esta rama se pueden desprender una gran variedad de usos, como la implementación de un sistema automatizado de estacionamiento. Sería necesario utilizar un sistema de reconocimiento de imágenes para encontrar los espacios libres, posteriormente reconocer el tipo de estacionamiento que es igual que el tipo de coche, para guardar los datos en una base de datos. Posteriormente se usaría el algoritmo A* o Fringe Search para encontrar el camino más corto y se representaría en forma de coordinadas(Yamani, Noorzaily, Zaidi, y Nor, 2005). También otras implementaciones se han propuesta para el uso de este tipo de algoritmos, donde A* es el estándar por defecto propuesto como lo hacen Idris y Tamil(Idris, Tamil, Razak, Noor, y Kin, 2009).

4.2.2.2 Robótica

Los agentes autónomos en un mundo real necesitan navegar en el, y para realizarlo se debe comprender al mundo mediante sensores, que se debe abstraer hasta llegar a un gráfico idóneo, con el cual el agente puede calcular sus movimientos. En esta navegación se utilizan algoritmos cómo A*, Fringe Search y Dijkstra, sin limitarse a ellos.

El área de la navegación de robots móviles es extensa y abarca diversos pasos y procedimientos, los cuales se engloban en tres partes: los métodos de localización, la planificación de

caminos y la evaluación de obstáculos. A* y Fringe Search son aplicables en el apartado de planificación de caminos(Ortiz, 2014).

En lo general es posible utilizar un mapa sin rutas establecidas previas, con lo que sería necesario usar una gráfica de malla y cuando se logres establecer ciertas rutas previamente recorridas de un área conocida se podrá establecer un gráfico donde los nodos representen puntos de importancia en ese recorrido. Para robots de servicio como el *Golem*[2] será necesario tomar en cuenta que cualquier lugar del recorrido puede ser un punto al cual se deba llegar, ya que puede localizarse tareas o una localización física específica, que no necesariamente tiene que ser dentro gráfico que presente los lugares más relevantes de un espacio. Por lo tanto, un gráfico de malla sería la opción que más se acerca a la realidad. Tomando como ejemplo al robot mexicano *Golem* se indica que el robot utiliza un plano cartesiano por conjuntos de pares ordenados (x, y) y son analizados don Dijkstra(Luis A., 2013). A* y Dijkstra son los algoritmos más usados para estos casos y por lo tanto sería posible mejorar el rendimiento de los mismos usando Fringe Search.

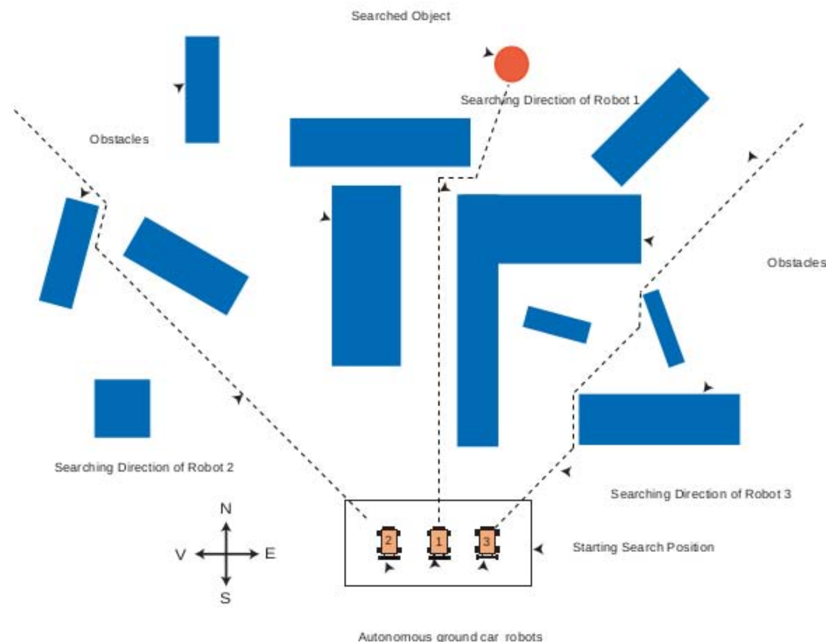

Figura 4.11: Búsqueda del mejor camino de robots esclavos

En su caso de implementación Popiralan y Dupac(Claudiu y Mihai, 2009) realizan un carro

[2]El proyecto *Golem* desarrollado por el IIMAS es un robot de servicio para atender las necesidades del humano en el hogar.(En una entrevista con el Dr. Ivan Vladimir Meza Ruiz)

robot como un agente autónomo que tiene que en un camino óptimo sin choques y en el menor tiempo posible, moviéndose entre varios obstáculos, pero además con las dinámicas propias de la cinemática. Para generar el mejor camino dos tipos de agentes son usados, un agente esclavo que utiliza A* cómo se muestra en la figura 4.11(Claudiu y Mihai, 2009) y el segundo tipo es un agente móvil que se comunica mediante un módulo de comunicación. El agente esclavo manda las coordenadas presentes y las coordenadas objetivo al agente móvil, con lo que el agente móvil planea la dirección basado en el mapa parcialmente generado del entorno. Los agentes esclavos utilizan el A* sobre un entorno dividido en una gráfico de malla(Claudiu y Mihai, 2009).

En aplicaciones como las que se han visto anteriormente los caminos cien por ciento óptimos pueden ser fácilmente sustituibles por caminos que tienden a ser óptimos, por lo que Finge Search es una alternativa muy importante, para reducir recursos dedicados a la búsqueda de caminos y dedicarlos a otras tareas más relevantes para el autómata.

4.2.3 Juegos de vídeo

Una de las aplicaciones donde el algoritmo es más usado es en el cálculo de caminos para los agentes dentro de un juego de vídeo. En todos aquellos juegos donde se tenga un movimiento libre de los agentes y no una secuencia preestablecida será necesario planear la forma en la cual estos agentes se moverán dentro del mundo virtual creado para ello. Los mundos pueden ser representados tanto en dos dimensiones, a pesar de contar con tráficos en tercera dimensión, o en tercera dimensión para juegos que implica agentes voladores.

La gran mayoría de los juegos utiliza representaciones en dos dimensiones y por lo tanto utilizan gráficos de malla. Desde juegos deportivos donde los jugadores de un equipo tienen que correr en una dirección, pasando por actores que disparan en contra de otros en juego de tirador de primera persona(FPS[3]), hasta los juegos de estrategia en tiempo real (RTS[4]).

En juegos de vídeo se calcula la ruta entre dos puntos para el desplazamiento de agentes, por lo que se el mejor método es A* o Fringe Search, dejando casi relegado Dijkstra por su alto costo. Es muy común pode encontrar el desplazamiento libre de manera más evidente en los juegos RTS y donde se puede encontrar títulos como *Dune II*, *Warcraft* y *Age of Empires*. También en algunos juegos de estrategia por turnos como *Civilization*. En un clon de *Age of Empires* libre dedicado al sector educativo llamado *TuxHistory*(Mager, 2014) se ve en el código el uso de A*

[3]First Person Shooter
[4]Real Time Strategy Games

cómo algoritmo que trata el tema. En la figura 4.12 se aprecia el área de juego, la localización de los agentes que se mueven de un punto a otro del mapa para realizar ciertas funciones, como es la recolección de recursos o el movimiento de tropas, la movilización de trabajadores para la construcción de un edificio, entre otras posibilidades.

Figura 4.12: Agentes del juego libre TuxHistory buscando el mejor camino para llevar la cosecha del campo al granero.

También en estos casos es completamente posible la utilización de Fringe Search sobre A* al presentarse movimientos que no necesitan optimización total de sus rutas, e importa más la velocidad a la cual se realizan las tareas de cálculo. En su trabajo inicial los autores de Fringe Search plantearon su algoritmo únicamente aplicado a mapas de juegos de vídeo(Björnsson y cols., 2005). Esta aseveración sería limitada, pero el algoritmo ciertamente debe ver más uso en la inteligencia artificial de juegos.

4.2.4 Otros usos

En materia de redes se utiliza los algoritmos de mejor camino entre dos puntos para realizar *routing* entre dos nodos. Por ejemplo, y sólo para citar una aplicación, Rana y Zaveri(Keyur y Mukesh, 2011) utilizan A* para poder desarrollar una red de sensores inalámbricos con poco uso de memoria. Se utiliza en este caso A* para evitar un uso desmedido de procesamiento que generaría un gran consumo de recursos energéticos. En este tipo de problemas, es posible

utilizar A* o Fringe Search, donde la mejora en el tiempo de cálculo de la ruta se complementa con la mejora en el uso de memoria y resulta en una mayor velocidad en la red.

También, en investigación de operaciones se pueden utilizar este tipo de algoritmos, como por ejemplo para poder encontrar las decisiones menos costosas de construir una nueva fábrica o de realizar una cierto proyecto. Sin embargo, en estos casos no es tan importante el tiempo que se dedica a la generación del camino, sino el camino en si, por lo que encontrar la mejor ruta sería absolutamente necesario para optimizar al máximo el proyecto. A* debe ser la opción, y Fringe Search no es un candidato para resolver este tipo de problemas.

Conclusiones

La inteligencia artificial en computación es un campo relevante en varios aspectos de la vida digital cotidiana (Russell y Norvig, 2004) hasta el punto en que parece completamente natural. Clasificadores de spam en los correos electrónicos, recomendaciones de artículos en las tiendas electrónicas, navegadores GPS, enemigos virtuales en los juegos de computadoras, entre muchos otros ejemplos son completamente cotidianos para nosotros.

Para lograrlo se necesitan distintos tipos de abstracciones de la realidad representadas normalmente en gráficos. El mundo concreto que se pretende representar no es más que la unidad de un conjunto de elementos que pueden ser representados en un gráfico, es como diría Marx la unidad de aspectos múltiples. “Lo concreto es concreto por ser la síntesis de muchas definiciones, o sea, la unidad de aspectos múltiples. Aparece por tanto en el pensamiento como proceso de síntesis, como resultado y no punto de partida, aunque es el verdadero punto de partida y también, por consiguiente, el punto de partida de la contemplación y representación. El primer procedimiento ha reducido la representación plena a definiciones abstractas; con el segundo, las definiciones abstractas conducen a la representación de lo concreto por medio del pensamiento.”(Marx, 1989). El proceso de síntesis descrito en la anterior cita es lo que el diseño del problema conlleva, y de este se realizan una serie de representaciones en forma de gráficos. La pregunta central en este momento sería, que permitiría evaluar lo correcto de los resultados a los que se ha llegado a partir de esta abstracción matemática. “El problema de si al pensamiento humano se le puede atribuir una verdad objetiva, no es un problema teórico, sino un problema práctico. Es en la práctica donde el hombre tiene que demostrar la verdad, es decir, la realidad y el poderío, la terrenalidad de su pensamiento. El litigio sobre la realidad o irrealidad de un pensamiento que se aísla de la práctica, es un problema puramente escolástico.”(Marx, 1888). Toda abstracción que se realiza mediante una computadora únicamente puede ser correcta si es comprobada en la práctica. El algoritmo aplicado puede ser correcto para el punto de vista teórico, pero al no corresponder la abstracción del gráfico a la realidad que pretende resolver, o

al no centrarse en los aspectos principales de la contradicción los resultados llevarán a errores o a fracasos. En este texto se ha trabajado sobre una abstracción generalizada de la realidad, para encontrar entre ésta el mejor camino de un estado a otro, pero en ningún momento quiere decir que forzosamente esto se cumpla sin una correcta abstracción e implementación de la función heurística por lo que gran parte del resultado aún depende de la implementación concreta del mismo.

Ahora bien, para poder encontrar el camino óptimo ente dos nodos de un gráfico las ciencias de la computación gracias al aporte de Bart, Nilsson y Raphael(Hart y cols., 1968) han logrado hallar la solución a este problema utilizando una búsqueda informada que toma una función heurística, que se supone adecuada, para adivinar un probable camino más corto durante su desarrollo. Las diversas pruebas que hace el algoritmo en en transcurso de su ejecución asegurarán encontrar el camino perfecto entre estos dos puntos. Este algoritmo ha sido utilizado ampliamente en toda la industria que requiere resolver problemas semejantes y ha logrado llegar a grandes acuerdos sobre las funciones heurísticas en las aplicaciones sobre todo tipo de mapas. Sin embargo, y a pesar de la solución alcanzada, aún es necesario encontrar mejoras en su tiempo de ejecución para mejorar el desempeño en las que se utiliza, sobre todo sobre gráficos de gran tamaño. Por ello es que en 2005 Björnsson, Enzenberger, Holte y Johnathan desarrollaron un algoritmo basado en IDA*(Korf, 1985) llamado Fringe Search(Björnsson y cols., 2005) que logra superar en tiempo de ejecución a A*, pero que tiene el defecto de no encontrar el mejor camino, a pesar de que el camino encontrado tiende a serlo. En la práctica, en varias aplicaciones la diferencia no tiene ninguna desventaja y, en comparación, una considerable mejora en su rendimiento. En una primer supervisión visual la diferencia en el tamaño del camino de manera gráfica es casi imperceptible o en la mayoría de los casos, imperceptible.

En los experimentos realizados, y basado en el análisis estadístico expuesto en este trabajo afirmo que Fringe Search, con un 95 % de confianza, es más veloz que A* al ser ejecutado sobre los gráficos de estudio, llamados gráficos de malla. Las ventajas son evidentes y mejoran el tiempo de ejecución de manera notable. Sin embargo, este rendimiento no se encuentra exento de eventos externos al mismo, por lo que el tiempo que se registró en cada prueba tuvo una dispersión en los dos algoritmos. Se puedo encontrar que existe menos variación en el tiempo de ejecución en FS que A*, con excepción de ciertos datos relevantes que existe más frecuentemente en Fringe Search. En ocasiones los caminos generados por FS son demasiados alejados del óptimo. Pero

los tiempos de ejecución de FS son más regulares que los de A* por lo general.

Por otro lado, en el análisis de los caminos generados por los dos algoritmos, tomando en cuenta que A* genera caminos mínimos óptimos, se ha demostrado que la diferencia entre los dos no es significativa realizando una prueba ANOVA sobre la relación que guarda el haber utilizado A* y Fringe Search en los datos. Esto indica que en gran parte de los problemas no es relevante la pequeña diferencia de caminos generados por Fringe Search. En lo que toca a la evolución del camino se debe mencionar que la variación del tiempo de ejecución de A* y Fringe Search no aumenta proporcionalmente con el tamaño del camino, donde un camino corto de FS tiene de 2 a 8 nodos más que A*, pero estos mismos números se conservan en caminos mucho mas largos, llevando por ello a la diferencia del camino encontrado por Fringe Search con respecto a A* a ser irrelevante en gráficos grandes, tendiendo a una igualdad.

Sin embargo, Fringe Search para búsqueda de caminos en donde estos requieren ser necesariamente óptimos no es recomendable ya que no garantiza encontrarlos. En el capítulo de aplicaciones se presentan algunas áreas en las cuales no será apropiado utilizar Fringe Search, sobre todo en aquellas donde el tiempo de cálculo no es tan relevante, como suelen ser caminos precalculados, pero también en aquellos donde el camino debe ser optimizado completamente para alcanzar un resultado perfecto, como puede ser en la toma de decisiones en temas económicos o financieros, como se realiza en la investigación de operaciones. Pero, la mejora alcanzada es adecuada para aquellas aplicaciones que requieren calcular caminos en tiempo real, sobre todo sobre gráficos muy grandes. Por eso es recomendable utilizar Fringe Search en vez de A* en la inteligencia artificial de juegos que requieren un desplazamiento de un agente de un punto al otro, y que además de la búsqueda de caminos tienen otras operaciones bastante costosas. Resumiendo, en problemas donde la calidad del camino es más importante que calcular el tiempo de su creación A* es preferible, mientras que en caminos largos que son muy costosos de calcular y donde no es tan importante su calidad es mejor utilizar Fringe Search.

Los objetivos planteados en un inicio han sido cumplidos donde se han expuesto las soluciones para encontrar el mejor camino entre nodo y nodo; se analizó los dos algoritmos con mejores perspectivas para solucionar el problema que son Fringe Search y A* obteniendo una comprensión íntegra de su planteamiento formal, su funcionamiento e implementación; se han implementado los dos algoritmos en lenguaje C poniendo especial énfasis en su optimización para contar con los mejores ejemplos y tomar conclusiones basados en ellos; y con estas imple-

mentaciones fueron hechos los muestreos necesarios para poder llegar a una conclusión; y se ha logrado presentar diversas líneas de investigación con las cuales se puede continuar el estudio en este tema.

También se ha llegado a la conclusión, al igual que Amit(Patel, 2014), que la implementación de los algoritmos juega un papel muy relevante en su desempeño, donde la optimización de los mismos es muy importante y recae principalmente en los tipos de estructuras de datos que utilizan. A* que se basa en una lista ordenada que utiliza gran parte de su tiempo de procesamiento depende en gran medida de utilizar una buena opción. *Skiplist* es un tipo de datos muy efectivo, que puede sustituir sin problemas a las *Binary Heaps* pero es ventajoso por no perder efectividad en agrupaciones de datos muy grandes. También plantea una ventaja en lo que a uso de memoria se refiere. Como un campo de estudio posterior sería adecuado poder implementar otra opciones en vez de una *skiplist* como son las colas HOT(*Heap-on-Top*) planteadas por Boris V. Cherkassky y Andrew V. Goldberg(Cherkassky y Goldberg, 1996) con lo que los autores plantean tener una complejidad esperada de $O(m + n(\log C)13 + ffl)$ en un Dijkstra o un *Splay tree*(Bui-Huu, 2014) que según Amit(Patel, 2014) tiene una complejidad de de $O(\log F)$.

En lo que se refiere al consumo de memoria en este texto no se centró a su estudio, pero para poder mejorar el rendimiento en consumo de memoria se debe utilizar *tablas hash* dinámicas perfectas como la que presenta Fredman, Michael y Koml(Fredman, Komlós, y Szemerédi, 1984) con un tiempo de acceso de $O(1)$, que son superiores a los arreglos. Las tablas hash con cierto grado de colisiones repercuten de manera muy importante en el rendimiento de los algoritmos por lo que se deben evitar las que presenten algún grado de colisiones.

Fringe Search mejora a A* en el rendimiento en caminos que se acercan a ser óptimos, por lo tanto debo subrayas que este únicamente debe ser utilizado a casos que lo permiten. Cada algoritmo tendrá su campo de aplicación, el cual debe ser comprendido muy bien para no tener severas consecuencias. El uso de las estructuras de datos y el uso de la función heurística indicada también son esenciales a la hora de definir si el algoritmo es efectivo o no.

Sin embargo, el reto previo que se tiene para que las soluciones sean correctas es plantear de la mejor manera posible los gráficos sobre los cuales trabajarán todos los algoritmos. Estos tráficos podrán ser creados ya sea por humanos o por otros algoritmos que analicen cierto tipo de información obtenida de ese mundo material. También es importante usar el gráfico

correcto para resolver un cierto tipo de problema. Los algoritmos mencionados A* y Fringe Search no son los únicos existentes para encontrar mejores caminos, existen otros casos como el encontrar el mejor camino de un nodo a todos los demás nodos de gráfico y el de todos los nodos a todos los nodos de un gráfico, que deberán ser usados de manera correcta. Algoritmos como Dijkstra(Dijkstra, 1959), *Bellan-Ford*(Ford Jr., 1959)(Bellman, 1958), *Floyd-Warshall*(Warshall, 1962)(Floyd, 1962) y *Johnson*(Johnson, 1977) para sólo mencionar los más importantes

En la cuestión de la aplicación se ha logrado explorar que si bien los autores de Fringe Search lo limitaron a mapas de juego, las aplicaciones de este algoritmo son más extensas. Sus campos de trabajo se relacionan principalmente con el mundo real de mapas en forma de coordenadas dentro de un plano cartesiano, como es la navegación dentro de las funciones de la robótica móvil. A* ha sido estudiado ampliamente no únicamente en el terreno secuencial, si no que existe un intenso trabajo en el campo paralelo(Pacheco, 2011). Sin embargo, FS únicamente se menciona en pocos trabajos, lo cual permite abrir un amplias posibilidades para seguir la investigación. Para comenzar con los planteamientos de paralelización se debe comenzar con Dijkstra que lo ha hecho bastante bien (Crauser, Mehlhorn, Mayer, y Sanders, s.f.). Pero A* ha planteado problemas serios. Se ha intentado mediante una búsqueda en dos direcciones(Oliveira Rios y Chaminowicz, s.f.) o con trabajos que no han resuelto completamente la cuestión (Cohen y Dallas, s.f.)(Smith, 1992). Otros intentos plantean un A* de manera distribuida como *Hash Distributed A**(Yoshikazu, Akihiro, y Osamu, 2011). Algoritmos como Best-First Heuristic(Burns, Lemons, y Ruml, 2010) Search crean nuevas formas en las cuales encuentran caminos en máquinas multicore. Quiero hacer una especial mención de los trabajos de Birrada y Brand(Sandy y Birrada, 2012) que han planteado Fringe Search paralelo, que hasta donde sé, es el primer intento de llevar este innovador algoritmo al campo paralelo. También estudios sobre la paralelización de las estructuras de datos son campos que pueden ser desarrollados aún más. En el caso de nuestra aplicación de A* se usa Skiplist la cual ha visto cierto trabajo en este campo con Lotan y Nir(Italy y Shavit, 2000). El plantear una abstracción correcta es central en lograr buenos resultados, al igual que utilizar el algoritmo correspondiente. Pero gran parte también depende de la implementación en cuanto a la codificación de las estructuras de datos, que a su vez es un tema que tiene una gran oportunidad de estudio. Por último, también se plantea la posibilidad para estudios posteriores avanzar en la paralelización

de Fringe Search ya que existe una fuerte ausencia de estudio de este algoritmo en esta área.

Para concluir, gracias a los datos experimentales, se toma como cierta la hipótesis de que Fringe Search es superior a A* en el tiempo de ejecución y este mayor rendimiento es significativo. Por el otro lado, también se confirma la hipótesis planteada en esta tesis de que la diferencia entre los caminos generados por A* y Fringe Search no es significativa y por lo tanto el incremento en la velocidad planteado por Fringe Search no se ve demeritado en la muchos casos por la pérdida de calidad del camino generado. También se ha planteado la necesidad de delimitar el área de uso de cada algoritmo a los casos prácticos correspondientes, limitando el uso de Fringe Search en aquellos casos en que se requiere un camino perfecto como tal, dejando el campo abierto para todas las implementaciones que tienen mayor necesidad de contar con tiempo de ejecución menor y donde la necesidad de un camino mínimo no es tan imperante. Con ello se concluye el presente trabajo.

Anexos

Código fuente de A*: base.h

```
/*
 * base.h
 *
 * Description: Definition of the basic data types and include
 *              of headers.
 *
 * Author: Manuel Mager <fongog@gmail.com> (C) 2013
 * Copyright: GPL v3 or later
 *
 */

#ifndef BASE_H
#define BASE_H

#include <stdlib.h>

#include "skiplist.h"

typedef enum {false, true} bool;

#define min(a,b) (((a) < (b)) ? (a) : (b))
#define max(a,b) (((a) > (b)) ? (a) : (b))

// Free memory
#define FREE(p) do{ free(p); (p) = NULL; } while(0)

//Number of elements of a array
#define NUM_ELEMENTS(x) (sizeof (x) / sizeof (*(x)))

// Comments
#ifndef COMMENT
#define COMMENT if(debug){
#define COMMENTEND }
#endif

//Basic type definitions for the A* implementation.

typedef struct apoint{
    int                 x;
    int                 y;
}apoint;

typedef struct avector{
    int                 x;
```

```
45     int                  y;
46 }avector;
47
48 typedef struct apath{
49     apoint               **path;
50     int                  size;
51 }apath;
52
53 typedef struct aagent{
54     int                  id;
55     int                  x,y;
56 }aagent;
57
58 typedef struct anode{
59     int                  index;
60     int                  val;
61     int                  deph;
62     int                  g, h;
63     int                  coord;
64     int                  count;
65     apoint               point;
66     int                  x,y,f;
67     int                  inopen, inclosed;
68     struct anode         *parent;
69     struct anode         *pointer;
70     struct skiplistnode *skip;
71 }anode;
72
73 typedef struct aobj{
74     Skiplist             *openlist;
75     anode                **cache;
76     apath                path;
77     anode                *item;
78     char                 *output;
79     int                  open_empty, closed_empty;
80 }aobj;
81
82 int agent_counter;
83 int num_of_agents;
84 int debug;
85
86 #endif
```

Código fuente de A*: astar.c

```
1  /*
2   * astar.c
3   *
4   * Description: The A* algorithm implementation.
5   *
6   * Author: Manuel Mager <fongog@gmail.com> (C) 2013
7   * Copyright: GPL v3 or later
8   *
9   */
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <malloc.h>
```

```
13 #include <string.h>
14 #include <math.h>
15
16 #include "base.h"
17 #include "graphs.h"
18 #include "astar.h"
19
20 #include "openlist.h"
21 #include "closedlist.h"
22
23 // Defining local macros
24
25 // Absolute value of a diference of two numbers
26 #define ABSD(a, b) (a<b) ? (b-a) : (b-a)
27
28 // Hueristic distance between to points
29
30 // Manahttan distance
31 #define HDIST(x1, y1, x2, y2) (((x1<x2)?(x2-x1):(x1-x2) + \
32             ((y1<y2)?(y2-y1):(y1-y2)))*10)
33
34 // Chebyshev distance
35 #define HDISTC(x1, y1, x2, y2) 10*(abs(x2-x1) + abs(y2-y1))+\
36     (13-2*10)+(abs(x2-x1)<abs(y2-y1))?(abs(x2-x1))*10:(abs(y2-y1))*10
37
38 // Euclidean distance
39 #define HDISTE(x1, y1, x2, y2) (int)(sqrt(((x2-x1) * (x2-x1)) + \
40             ((y2-y1) * (y2-y1)))*10)
41
42 // No Hueristic function
43 #define HDISTN(x1, y1, x2, y2) 10
44
45
46
47 #define G 10
48
49 // Local Function definitions
50 void    ai_free_path(apath path);
51 int     hdist(int p1x, int p1y, int p2x, int p2y);
52 int     ai_valid_tile(int unit, apoint coords);
53
54
55 // Local function implementations
56
57 // Modified Chebyshev distance to get a precise distance
58 // for NW, NE, SW and SE direccions.
59 int hdist(int p1x, int p1y, int p2x, int p2y)
60 {
61     int xDistance = abs(p1x-p2x);
62     int yDistance = abs(p1y-p2y);
63
64     if (xDistance > yDistance)
65     {
66         return 14*yDistance + 10*(xDistance-yDistance);
67     }
68     else
```

```
69      {
70          return 14*xDistance + 10*(yDistance-xDistance);
71      }
72  }
73
74  // Returns 1 if the tile is valid to use for a player, and unit
75  // and 0 if not.
76  int ai_valid_tile(int unit, apoint coords)
77  {
78
79      if(coords.x < 0 || coords.x >= x_tildes)
80          return 0;
81      if(coords.y < 0 || coords.y >= y_tildes)
82          return 0;
83
84      if(gmaps[coords.x][coords.y].terrain == WALL)
85      {
86          return 0;
87      }
88
89      return 1;
90  }
91
92  // This is the main A* algorithm function. Unit parameter is not
93  // jet implemented, so it should be 0. TODO
94  int ai_shortes_path(aobj *astar, int unit, apoint source, apoint goal)
95  {
96      int                 i, a;
97      int                 closed_flag, open_flag;
98
99      avector             vector;
100     apoint              pt;
101     apoint              **solution;
102     anode               *item;
103     anode               *next;
104     anode               *old;
105     anode               *tmp_item;
106
107     printf("Our source point is: %d, %d \n", source.x, source.y);
108     printf("Our goal point is: %d, %d\n", goal.x, goal.y);
109
110
111     // Are the source and goal point valid?
112     if(!ai_valid_tile(unit, source))
113     {
114         fprintf(stderr, "Invalid source point! %d \n", ai_valid_tile(unit, \
115                     source));
116         return 0;
117     }
118     if(!ai_valid_tile(unit, goal))
119     {
120         fprintf(stderr, "Invalid goal point! %d \n", ai_valid_tile(unit, goal));
121         return 0;
122     }
123
124     i = 0;
```

```
125
126     // Creating open and closed lists
127     if (!openlist_init(astar))
128     {
129         fprintf(stderr, "Error initializing the open list\n");
130         return 0;
131     }
132     if(!closedlist_init(astar))
133     {
134         fprintf(stderr, "Error initializing the closed list\n");
135         return 0;
136     }
137
138
139     // Defining the initial node
140     printf("=================== A* ===================\n");
141
142     item = malloc(sizeof(anode));
143     if(item == NULL)
144     {
145         return 0;
146     }
147     item->deph = 0;
148     item->point = source;
149     item->x = source.x;
150     item->y = source.y;
151     item->h = hdist(item->x, item->y, goal.x, goal.y);
152     item->g = 0;
153     item->f = item->g + item->h;
154     item->parent = NULL;
155
156     // Insert the initial node to the open list
157     if(!openlist_insert(astar, item))
158     {
159         fprintf(stderr, "Coudn't add element to the open list!\n");
160         return 0;
161     }
162
163     while(!openlist_isempty(astar))
164     {
165         ///////////////////////////////////////////
166         // Remove the lowest element in open list
167         // and add it to the closed list
168         ///////////////////////////////////////////
169
170         COMMENT
171             printf("********** New Loop Cycle\n");
172             puts("Open List:\n");
173             openlist_print(astar);
174             puts("Closed List:\n");
175             closedlist_print(astar);
176         COMMENTEND
177
178         item = openlist_getmin(astar);
179         if(item == NULL)
180         {
```

```
181             fprintf(stderr, "Error deleting the priority element from open list!\
    n");
182             return 0;
183         }
184         if(!closedlist_insert(astar, item))
185         {
186             fprintf(stderr, "Error adding to hashtable!\n");
187             return 0;
188         }
189 
190 
191         ////////////////////////////////////////////
192         // Is this element the goal?
193         ////////////////////////////////////////////
194 
195         if(item->x == goal.x && item->y == goal.y)
196         {
197             solution = malloc(item->deph * sizeof(apoint *));
198 
199             if(!solution)
200             {
201                 astar->item = NULL;
202                 return 0;
203             }
204 
205             i=0;
206 
207             while(item->parent)
208             {
209                 solution[i] = &(item->point);
210                 item = item->parent;
211                 i++;
212             }
213 
214             astar->path.size = i - 1;
215             astar->path.path = solution;
216             astar->item = item;
217 
218             // Time routines for gather the needed data for
219             // our analysis.
220 
221             return 1;
222         }
223 
224         COMMENT
225             printf("This element is not the goal!.. Trying...\n");
226         COMMENTEND
227 
228         ///////////////////////////////////////////////////////
229         // For each valid move for n, also known as the neigbhors
230         // of the current node.
231         // ////////////////////////////////////////////////////
232 
233         for(a = 0; a < NUM_DIRS; a++)
234         {
235             vector = get_vector(item->point, a);
```

```
236             if(vector.x != -2 && vector.y != -2)
237             {
238                 COMMENT
239                     printf("For %d direction tile in (%d,%d) is valid?\n", \
240                             a, item->point.x, item->point.y);
241                     printf("Vector is valid...\n");
242                 COMMENTEND
243 
244                 pt.x = vector.x + item->point.x;
245                 pt.y = vector.y + item->point.y;
246                 if(ai_valid_tile(unit, pt))
247                 {
248                     //New valid element
249                     next = malloc(sizeof(anode));
250                     next->deph = item->deph + 1;
251                     next->point = pt;
252                     next->h = hdist(next->point.x, next->point.y, goal.x, \
253                             goal.y);
254                     next->x = next->point.x;
255                     next->y = next->point.y;
256                     if( a == ISO_N ||
257                         a == ISO_S ||
258                         a == ISO_W ||
259                         a == ISO_E)
260                         next->g = item->g + G;
261                     else
262                         next->g = item->g + G+4;
263                     next->f = next->g + next->h; // F = G + H
264                     next->parent = item;
265 
266                     COMMENT
267                         printf("Adding direction %d to open list!\n", a);
268                         printf("Actual H: %d G:%d F:%d Deph:%d\n", \
269                                 next->h, next->g, next->f, \
270                                 next->deph);
271                     COMMENTEND
272 
273                     open_flag = 0;
274                     closed_flag = 0;
275 
276                     ///////////////////////////////////////////////
277                     // Is this element in closed list?
278                     // If the neigbhor is in closed list and
279                     // cost less than g(neigbhor) then remove neighbor
280                     // from closed list.
281                     //////////////////////////////////////////////
282 
283                     if((old = closedlist_search(astar, next)) != NULL)
284                     {
285                         COMMENT
286                             puts("The element is in the closed list!\n");
287                         COMMENTEND
288                         if(next->f < old->f)
289                         {
290                             COMMENT
291                                 printf("Prior element from closed list is greater
```

```
 than the next value!\n");
292                             COMMENTEND
293
294                             if(!closedlist_remove(astar, old))
295                             {
296                                 printf("Error ocurred while trying to remove \
297                                         key in hashtable!\n");
298                                 COMMENT
299                                     closedlist_print(astar);
300                                     openlist_print(astar);
301                                 COMMENTEND
302                                 astar->item = NULL;
303                                 return 0;
304                             }
305                             closed_flag = 0;
306                             COMMENT
307                                 printf("Succesfully removed from closed list\n");
308                             COMMENTEND
309                         }
310                         else
311                         {
312                             closed_flag = 1;
313                         }
314                     }
315
316                     ////////////////////////////////////////////////
317                     // Is this element in open list?
318                     // If the neigbhor is in openlist and
319                     // cost less than g(neighbor) then remove
320                     // neighbor from open list.
321                     ////////////////////////////////////////////////
322
323                     if((tmp_item = openlist_search(astar,next))!=NULL)
324                     {
325                         COMMENT
326                             printf("Node exists in openlist!\n");
327                                 openlist_print(astar);
328                                 closedlist_print(astar);
329                         COMMENTEND
330                         if(tmp_item->f > next->f && closed_flag == 0)
331                         {
332                             open_flag = 0;
333                             if(!openlist_rm(astar, tmp_item))
334                             {
335                         COMMENT
336                                 printf("The node is in the open list, but we
 couldn't remove it\n");
337                                 printf("(%d,%d)\n", tmp_item->x, tmp_item->y);
338                                 openlist_print(astar);
339                                 closedlist_print(astar);
340                         COMMENTEND
341                                 return 0;
342                             }
343
344                             COMMENT
345                                 printf("The new path is better: removing and
```

```
    adding the new one!\n");
346                         openlist_print(astar);
347                         closedlist_print(astar);
348                     COMMENTEND
349                 }
350                 else
351                 {
352                     open_flag = 1;
353                 }
354             }
355             ////////////////////////////////////////////////////////////
356             // If the element is not in the open list nor in the closed
357             // we can add the new node to the open list
358             ///////////////////////////////////////////////////////////
359 
360             if(open_flag == 0 && closed_flag == 0)
361             {
362                 COMMENT
363                     printf("Node doesn't exist in closed list or in open\
364                             list!\n");
365                 COMMENTEND
366                 if(!openlist_insert(astar, next))
367                 {
368                     printf("Error ocurred while adding a new element to \
369                             open list\n");
370                     astar->item = NULL;
371                     return 0;
372                 }
373             }
374             else
375             {
376                 if(next != NULL)
377                 {
378                     free(next);
379                     next = NULL;
380                 }
381             }
382         }
383     }
384   }
385  }
386  astar->item = NULL;
387  return 0;
388 }
389 
390 void ai_free_path(apath path)
391 {
392     if(path.size)
393     {
394         FREE(path.path);
395     }
396 }
397 
398 void ai_free_aobj(aobj *astar)
399 {
400     int i;
```

```
401
402     ai_free_path(astar->path);
403     closedlist_free(astar);
404     openlist_free(astar);
405     FREE(astar->output);
406
407     for(i=0; i<x_tildes; i++)
408         FREE(astar->cache[i]);
409
410     FREE(astar->cache);
411
412 }
```

Código fuente de Fringe Search: fringe.h

```
1  /*
2   * fringe.h
3   *
4   * Description: Header file of fringe.c We define here
5   *              the fobj and fnode structs and the fringe search
6   *              public functions.
7   *
8   * Author: Manuel Mager <fongog@gmail.com> (C) 2014
9   * Copyright: GPL v3 or later
10  *
11  */
12
13
14 #ifndef FRINGE_H
15 #define FRINGE_H
16
17 #include<stdio.h>
18 #include<stdlib.h>
19
20 #include "dllist.h"
21 #include "base.h"
22
23 #define COORD(X, Y) (X * y_tildes + Y)
24 #define LNODE(X, Y) &(fringe->listnode_table[COORD(X,Y)])
25 #define GETHEAD(L) (fnode *)(L->head->data)
26
27 #define ID_STRING_LENGTH 21
28
29 typedef struct fnode{
30     int                 valid;
31     int                 visited;
32     int                 inlist;
33     char                id[ID_STRING_LENGTH];
34     int                 index;
35     int                 deph;
36     long                f, g, h;
37     int                 coord;
38     int                 count;
39     int                 x,y;
40     struct fnode        *parent;
41     dlnode              *lnode;
42 }fnode;
```

```
typedef struct fobj{
    dlnode              *listnode_table;
    dllist              *list;
    fnode               *cache;
    fnode               *result;
    int                 xgoal, ygoal;
    int                 xsource, ysource;
    int                 xsize, ysize;
    int                 size;
    int                 compleat_path;
    int                 debug;
    char                output[21];
}fobj;

fobj    *fs_create(int xsource, int ysource, int xgoal, int ygoal, \
        int xsize, int ysize, char *file);
int     fs_movtonow(fobj *fringe, int x, int y);
fnode   *fs_getnode(fobj *fringe, int x, int y);
int     fringe_search(fobj *fringe);
int     fs_free_fobj(fobj *fringe);
#endif
```

Código fuente de Fringe Search: fringe.c

```
/*
 * fringe.c
 *
 * Description: home of the fringe_search function, and the
 *              main algorithm.
 *
 * Author: Manuel Mager <fongog@gmail.com> (C) 2014
 * Copyright: GPL v3 or later
 *
 */

#include<stdio.h>
#include<stdlib.h>
#include<math.h>

#include "dllist.h"
#include "base.h"
#include "graphs.h"
#include "fringe.h"

#define INF 9999999999999

int     fs_free(void *node);
int     fs_print(void *node);
int     fs_comp(void *node1, void *node2);
int     h(fobj *fringe, fnode *node);
int     cost(avector vect);

// Functions for the Double Linked List
int fs_comp(void *node1, void *node2)
{
    if(node1 != NULL && node2 != NULL)
```

```
33      {
34          if(((anode *)node1)->f < ((anode *)node1)->f)
35              return 1;
36          else
37              return 0;
38      }
39      return 0;
40  }
41
42  int fs_print(void *node)
43  {
44      fnode *item;
45      fnode *parent;
46      if(!node)
47          return 0;
48      item = (fnode *)node;
49      parent = item->parent;
50      if(parent)
51          printf("x:%d y:%d f:%ld g:%ld Parent-> x:%d y:%d g:%ld\n", item->x, item
        ->y, \
52                  item->f, item->g, parent->x, parent->y, parent->g);
53      else
54          printf("x:%d y:%d f:%ld  \n", item->x, item->y, item->f);
55      return 1;
56  }
57
58  int fs_free(void *node)
59  {
60      if(node)
61      {
62          free(node);
63          node = NULL;
64          return 1;
65      }
66
67      return 1;
68  }
69
70  int
71  cost(avector vect)
72  {
73      if( (vect.x == 1 && vect.y == 1) || \
74          (vect.x == -1 && vect.y == 1) || \
75          (vect.x == 1 && vect.y == -1) || \
76          (vect.x == -1 && vect.y == -1))
77          return 14;
78      return 10;
79  }
80
81  // Modified Chebyshev distance to get a precise distance
82  // for NW, NE, SW and SE direccions.
83  int h(fobj *fringe, fnode *node)
84  {
85      int xDistance = abs(node->x - fringe->xgoal);
86      int yDistance = abs(node->y - fringe->ygoal);
87
```

```
88     if (xDistance > yDistance)
89     {
90         return 15*yDistance + 10*(xDistance-yDistance);
91     }
92     else
93     {
94         return 15*xDistance + 10*(yDistance-xDistance);
95     }
96 }
97
98
99 fnode *fs_getnode(fobj *fringe, int x, int y)
100 {
101     if(!fringe)
102     {
103         printf("Fringe is empty");
104         return NULL;
105     }
106     if(!fringe->cache)
107     {
108         printf("The Cache is empy\n");
109         return NULL;
110     }
111     if(fringe->size < COORD(x,y) || 0 > COORD(x,y))
112     {
113         COMMENT
114             printf("The source (%d,%d) is not in the map\n", x, y);
115         COMMENTEND
116         return NULL;
117     }
118     else
119     {
120         if(x<0 || y<0)
121         {
122             return NULL;
123         }
124     }
125     if(fringe->cache[COORD(x,y)].valid)
126     {
127         return &(fringe->cache[COORD(x,y)]);
128     }
129
130     return NULL;
131 }
132
133 int fs_movtonow(fobj *fringe, int x, int y)
134 {
135
136     if(!(fringe->cache[COORD(x,y)].visited))
137     {
138         dllist_addtonow(fringe->list, &(fringe->listnode_table[COORD(x,y)]));
139         fringe->cache[COORD(x,y)].visited = 1;
140         return 1;
141     }
142     return 0;
143 }
```

```
144
145 fobj *fs_create(int xsource, int ysource, int xgoal, int ygoal,\
146         int xsize, int ysize, char *file)
147 {
148     int i;
149     int x, y;
150     fobj *fringe;
151
152     fringe = malloc(sizeof(fobj));
153     if(fringe == NULL)
154         return NULL;
155
156     fringe->xsource = xsource;
157     fringe->ysource = ysource;
158     fringe->xgoal = xgoal;
159     fringe->ygoal = ygoal;
160     fringe->xsize = xsize;
161     fringe->ysize = ysize;
162     fringe->size = xsize * ysize;
163     fringe->list = dllist_new(&fs_comp, &fs_print, &fs_free);
164     fringe->result = NULL;
165     sprintf(fringe->output, file);
166
167     if(fringe->list == NULL)
168         return NULL;
169
170     fringe->cache = malloc(fringe->size * sizeof(fnode));
171     if(fringe->cache == NULL)
172         return NULL;
173
174     for(i = 0; i<fringe->size; i++)
175     {
176         x = i / ysize;
177         y = i % ysize;
178         fringe->cache[i].visited = 0;
179         fringe->cache[i].inlist = 0;
180         fringe->cache[i].valid = gmaps[x][y].usable;
181         fringe->cache[i].x = x;
182         fringe->cache[i].y = y;
183         fringe->cache[i].g = 0;
184
185         // This is the part of the algoritm that asign
186         // NULL to the parent pointer of all members of
187         // the cache.
188         //    C[n] <- null for n != start
189
190         fringe->cache[i].parent = NULL;
191     }
192
193     fringe->listnode_table = malloc(fringe->size * sizeof(dlnode));
194     if(!(fringe->listnode_table))
195         return NULL;
196     for(i = 0; i < fringe->size; i++)
197     {
198         fringe->listnode_table[i].next = NULL;
199         fringe->listnode_table[i].prev = NULL;
```

```
200             fringe->listnode_table[i].data = &(fringe->cache[i]);
201         }
202
203         for(i = 0; i<fringe->size; i++)
204         {
205             fringe->cache[i].lnode = &(fringe->listnode_table[i]);
206         }
207
208
209         return fringe;
210     }
211
212     int fs_free_fobj(fobj *fringe)
213     {
214         int i;
215
216         free(fringe->cache);
217         free(fringe->listnode_table);
218         fringe->cache = NULL;
219         fringe->listnode_table = NULL;
220         fringe->result = NULL;
221         free(fringe->list);
222         free(fringe);
223         fringe = NULL;
224
225
226         return 1;
227     }
228
229     int fringe_search(fobj *fringe)
230     {
231         long        flimit, fmin, g, gs, gi, f;
232         int         found, i;
233         avector     vect;
234         dlnode      *nnode, *tmp_node;
235         fnode       *start, *n, *goal, *s;
236
237
238         goal = &(fringe->cache[COORD(fringe->xgoal, fringe->ygoal)]);
239         start = &(fringe->cache[COORD(fringe->xsource, fringe->ysource)]);
240
241         n = fs_getnode(fringe, fringe->xgoal, fringe->ygoal);
242         if(!n)
243         {
244             printf("Goal is not valid.\n");
245             return 0;
246         }
247         n = fs_getnode(fringe, fringe->xsource, fringe->ysource);
248         if(!n)
249         {
250             printf("Soruce is not valid.\n");
251             return 0;
252         }
253         printf("Source: (%d,%d) Goal: (%d,%d)\n", fringe->xsource, fringe->ysource, \
254                 fringe->xgoal, fringe->ygoal);
255         n = NULL;
```

```
256
257     /////////////////
258     // Initialize: //
259     /////////////////
260
261     // Fringe<-(s)
262     fs_movtonow(fringe, fringe->xsource, fringe->ysource);
263     start->inlist = 1;
264
265     // Cache C[start]<-(0, null)
266
267     fringe->cache[COORD(fringe->xsource, fringe->ysource)].g = 0;
268     fringe->cache[COORD(fringe->xsource, fringe->ysource)].parent = NULL;
269
270     // C[n]<-null for n != start
271     // is already done in fs_create()
272
273     //flimit<-h(start)
274     flimit = h(fringe, start);
275     COMMENT
276         printf("flimit = %d\n", flimit);
277     COMMENTEND
278
279     //found<-false
280     found = 0;
281
282
283     /////////////////////////////////////////
284     // Repeat until found = true or F empty //
285     /////////////////////////////////////////
286     while(!found && !dllist_isempty(fringe->list))
287     {
288         //fmin<-infinte
289         fmin = INF;
290         nnode = fringe->list->start;
291
292         //Iterate over nodes n in F from left to right
293         do
294         {
295
296             n = (fnode *)(nnode->data);
297             if(!n)
298             {
299                 printf("n is empty\n");
300                 return 0;
301             }
302
303
304             //(g, parent)<-c[n]
305             g = n->g;
306
307             //This operation is not needed, but defined as formal
308             //in the algorithm.
309             //parent = n->parent;
310
311             // f <- g + h(n)
```

```
312             f = g + h(fringe , n);
313
314
315             COMMENT
316                 if(n->parent)
317                 {
318                     printf("%ld: (%d,%d) Parent-> (%d,%d) g:%ld\n",f, n->x, n->y,
    \
319                         n->parent->x, n->parent->y, n->parent->g);
320                 }
321                 else
322                 {
323                    printf("%ld: (%d,%d)\n",f, n->x, n->y);
324                 }
325             COMMENTEND
326             // If f>flimit
327             //   fmin<-min(f,fmin)
328             //   continue
329             COMMENT
330                 printf("%ld > %ld\n", f, flimit);
331             COMMENTEND
332             if(f>flimit)
333             {
334                 COMMENT
335                     printf("To later!\n");
336                 COMMENTEND
337                 fmin = min(f, fmin);
338                 nnode = nnode->next;
339                 continue;
340             }
341             // If n = goal
342             //   found<-true
343             //   break
344             if(n == goal)
345             {
346                 printf("Goal Found!\n");
347                 found = 1;
348                 break;
349             }
350             //Iterate over s in successors(n) from right to left:
351             for(i = 7; i >= 0; i--)
352             {
353                 vect = get_iso_vector(i);
354                 s = fs_getnode(fringe , (vect.x + n->x), (vect.y + n->y));
355                 if(s == NULL)
356                 {
357                     continue;
358                 }
359
360                 // gs<-g+cost(n,s)
361                 gs = g + cost(vect);
362
363                 // If C[s] != null
364                 //   (g',parent)<-C[s]
365                 //   If gs >= g'
366                 //       continue
```

```
367                 if(s->visited)
368                 {
369                     COMMENT
370                         printf("Node visited: (%d,%d)\n", s->x, s->y);
371                     COMMENTEND
372                     gi = s->g;
373                     if(gs >= gi)
374                     {
375                         COMMENT
376                             printf("Continue\n");
377                         COMMENTEND
378                         continue;
379                     }
380                     COMMENT
381                         printf("Go ahead\n");
382                     COMMENTEND
383                 }
384
385                 // If F contains s
386                 //   remove s from F
387                 if(s->inlist)
388                 {
389                     COMMENT
390                         printf("Node already in list: (%d,%d)\n", s->x, s->y);
391                     COMMENTEND
392                     dllist_removefromlist(fringe->list, s->lnode);
393                     s->inlist = 0;
394                 }
395
396                 // Insert s into F after n
397                 s->visited = 1;
398                 s->g = gs;
399                 s->parent = n;
400                 s->f = gs + h(fringe, s);
401                 COMMENT
402                     printf("Insert: (%d,%d)\n", s->x, s->y);
403                     printf("New g: %ld = %ld + %d h: %d f: %ld\n", gs, g, cost(
      vect), \
404                             h(fringe,s), s->f);
405                     printf("parent -> x:%d y:%d g:%ld\n", s->parent->x, s->parent
      ->y,\
406                           s->parent->g);
407                 COMMENTEND
408                 dllist_insertafter(fringe->list, s->lnode, n->lnode);
409                 s->inlist = 1;
410                 COMMENT
411                     printf("----------------\n");
412                 COMMENTEND
413             }
414
415             tmp_node = nnode->next;
416             // Remove n from F
417             COMMENT
418                 printf("Remove (%d,%d) from list\n", n->x, n->y);
419                 if(!tmp_node)
420                     printf("Next node is NULL!\n");
```

```
421             COMMENTEND
422             dllist_removefromlist(fringe->list, nnode);
423             n->inlist = 0;
424             nnode = tmp_node;
425         }while(nnode != NULL);
426
427         // flimit <- fmin
428         flimit = fmin;
429         COMMENT
430             printf("********************\n");
431         COMMENTEND
432     }
433     // If found = true
434     //    Construct path from parent nodes in cache
435     if(found)
436     {
437         fringe->result = n;
438         while(n)
439         {
440             COMMENT
441                 printf("(%d,%d)\n", n->x, n->y);
442             COMMENTEND
443             n = n->parent;
444         }
445         return 1;
446     }
447     return 0;
448 }
```

Índice alfabético

árbol, 21
- balanceado, 30
- de búsqueda, 17, 18, 34

A*, 6, 7, 9, 23, 27, 39, 51
- gráfico, 51

agente, 6, 8, 16, 26
arco, 14
- costo, 14

arreglo, 31, 32, 42

búsqueda
- informada, 16, 18
- no informada, 16, 17

búsqueda de caminos, 7
Bellman-Ford, 20, 60
BFS, 17, 39
binary heap, 10, 20, 30
Binary Heaps, 71

C, 10
camino, 15, 23
- costo, 16, 24
- mas corto, 8
- mejor, 6, 16, 17, 20, 23, 24, 49

costo, 24, 39

DFS, 17, 38, 50
dialéctica, 9
Dijkstra, 6, 7, 20, 59
distancia
- Chebyshov, 26, 34
- Euclidiana, 26
- Manhattan, 25

Estado inicial, 16
estructura de datos, 30

Fibonacci, 7
Floyd-Warshall, 60
Fringe Search, 7, 9, 38, 39, 51
función
- sucesor, 16

Google Maps, 62
gráfico, 9, 14, 27
- lattice, 8, 15
- malla, 8, 15

heurístico, 7, 19, 23, 27, 33, 34, 40

IDA*, 38, 39
inteligencia artificial, 16

Johnson, 60

lista, 18
- abierta, 10, 27, 29, 31, 33, 34, 36, 39
- cerrada, 10, 27, 29, 31, 33, 35, 39
- después, 40

doblemente ligada, 10, 41
later, 40–42
ligada, 40
now, 41, 42
ordenada, 30, 40
prioridad, 7, 20, 21, 27
saltos, 30

MapQuest, 62
mejor camino, 7

nodo, 14
destino, 23
sucesor, 21

OpenStreetMap, 62

pila, 17
procesadores, 6
prueba t, 53

regresión, 54

skiplist, 10, 30, 31, 33, 71

tabla hash, 10, 17, 31, 40, 71
test objetivo, 16

Bibliografía

Barbehenn, M. (1998). A note on the complexity of dijkstra's algorithm for graphs with weighted vertices. *IEEE Transactions on Computers*, *47*(2), 263.

Bellman, R. (1958). On a routing problem. *Quarterly of Applied Mathematics*, *16*, 87–90.

Björnsson, Y., Enzenberger, M., Holte, R. C., y Schaeffer, J. (2005, April). Fringe search: Beating a* at pathfinding on game maps. En *Proceedings of the 2005 ieee symposium on computational intelligence and games (cig05).* IEEE.

Bui-Huu, F. (2014, Diciembre). *Implementarion of a top-down threaded splay tree.* [Online], accesible: `https://github.com/fbuihuu/libtree`.

Burns, E., Lemons, S., y Ruml, W. (2010). Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research*, *39*, 689-743.

Cherkassky, B. V., y Goldberg, A. V. (1996). *Heap-on-top priority queues* (Inf. Téc.). TR 96-042, NEC RESEARCH INSTITUTE.

Claudiu, P., y Mihai, D. (2009). An optimal path algorithm for autonomous searching robots. *Annals of University of Craiova, Math. Comp. Sci. Ser.*, *36*, 37-48.

Cohen, D., y Dallas, M. (s.f.). *Implementation of parallel path finding in a shared memory architecture.* [Online], accesible: `www.kesshoryu.com/files/Parallel_Paper.pdf`. Troy, NY.

Crauser, A., Mehlhorn, K., Mayer, U., y Sanders, P. (s.f.). *An implementation of parallelizing dijkstra's algorithm.* [Online], accesible: `www.cse.buffalo.edu/.../Ye-Fall-2012-CSE633.pdf`. Saarbrücken, Alemania.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*(1), 269-271.

Drozdek, A. (2007). *Estructuras de datos y algoritmos en java* (2.ª ed.). México: Thomson.

Engels, F. (1974). *Dialéctica de la naturaleza* (1.ª ed.; E. de Lenguas Extranjeras, Ed.). Peking: Edición de Lenguas Extranjeras.

Engels, F. (1976). *La revolución de la ciencia del señor eugenio düring* (1.ª ed.; E. de Lenguas Extranjeras, Ed.). Peking: Edición de Lenguas Extranjeras.

Floyd, R. W. (1962, Junio). Algorithm 97: Shortest path. *Commun. ACM*, *5*(6), 345–. Descargado de `http://doi.acm.org/10.1145/367766.368168`

Ford Jr., L. R. (1959). *Network flow theory.* Santa Monica, California: RAND Corporation.

Fredman, M. L., Komlós, J., y Szemerédi, E. (1984, Junio). Storing a sparse table with 0(1) worst case access time. *J. ACM*, *31*(3), 538–544. Descargado de `http://doi.acm.org/10.1145/828.1884`

Fredman, M. L., y Tarjan, R. E. (1987, julio). Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, *34*(3), 596–615. Descargado de `http://doi.acm.org/10.1145/28869.28874`

Hart, P. E., Nilsson, N. J., y Raphael, B. (1968). A formal basis for the hueristic determination of minimum cost paths. *IEEE*, *SSC-4*(2), 100-107.

Heineman, T. G., Pollice, G., y Selkow, S. (2008). *Algorithms in a nutshell* (1.ª ed.). O'Reilly Media Inc.

Idris, M., Tamil, E., Razak, Z., Noor, N., y Kin, L. (2009). Smart parking system using image processing techniques in wireless sensor network environment. *Information Technology Journal*, *8*, 114-127.

Italy, L., y Shavit, N. (2000, Mayo). Skiplist-based concurrent priority queues. En *International parallel and distributed processing symposium.*

Johnson, D. B. (1977, Enero). Efficient algorithms for shortest paths in sparse networks. *J. ACM*, *24*(1), 1–13. Descargado de `http://doi.acm.org/10.1145/321992.321993`

Johnsonbaugh, R. (2005). *Matemáticas discretas* (6.ª ed.). México: Pearson Educación en México.

Keyur, R., y Mukesh, Z. (2011). A-star algorithm for energy efficient routing in wireless sensor network. *Communications in Computer and Information Science*, *197*, 232-241.

Kopin, P. V. (1966). *Lógica dialéctica* (1.ª ed.; E. J. Gijalbo, Ed.). México, D.F.: Editorial Grijalbo.

Korf, R. E. (1985, Sept.). Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, *27*(1), 97-109.

Langsam, Y., Augenstein, M. J., y Tanenbaum, A. M. (1997). *Estructuras de datos en c y c++.*

(2.ª ed.). México: Prentice-Hall.

Luis A., P. (2013). Grupo golem: Robocup@home 2013. En *Proceedings of robocup 2013.*

Mager, J. (2014, Diciembre). *Tuxhistory - educational history game.* [Online], accesible: http://anonscm.debian.org/cgit/tux4kids/tuxhistory.git/tree/.

Marx, C. (1888). Tesis sobre feuerbach. En *Ludwig feuerbach y el fin de la filosofía clásica alemana.* Editorial Progreso.

Marx, C. (1989). El método de la economía política. En *Contribución a la crítica de la economía política* (p. 150-157). Editorial Progreso.

Millington, I. (2006). *Artificial intelligence for games* (1.ª ed.). San Francisco, U.S.: Morgan Kaufmann Publishers.

Notarstefano, G., y Parlangeli, G. (s.f.). *Controllability and observability of grid graphs via reduction and symmetries.* [Online], accesible: http://arxiv.org/pdf/1203.0129.pdf. Ithaca, NY.

Oliveira Rios, L. H., y Chaminowicz, L. (s.f.). *Pnba*: A parallel bidirectional heuristic search algorithm.* [Online], accesible: homepages.dcc.ufmg.br/~chaimo/public/ENIA11.pdf. Belo Horizonte, Brasil.

OpenStreetMap. (2014, Diciembre). *Routing.* [Online], accesible: http://wiki.openstreetmap.org/wiki/Routing.

Ortiz, A. (2014, Diciembre). *Navegación para robots móviles.* [Online], accesible: http://dmi.uib.es/aortiz/mobots_navegacion.pdf.

Pacheco, P. S. (2011). *An introduction to parallel programming* (1.ª ed.). Burlinton, MA. USA: Morgan Kaufmann.

Patel, A. J. (2014, Agosto). *Amit's a* pages.* [Online], accesible: http://theory.stanford.edu/~amitp/GameProgramming/.

Pugh, W. (1990, June). Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, *33*(6), 668-676.

Ravindran, A. R. (2008). *Operations research and management science handbook* (1.ª ed.). Boca Raton, Florida: CRC Press.

Russell, S., y Norvig, P. (2004). *Inteligencia artificial, un enfoque moderno* (2.ª ed.). Madrid: Pearson Educación.

Sandy, B., y Birrada, R. (2012). Multi-core scalable and efficient pathfinding with parallel

ripple search. *Computer Animation and Virtual Worlds (SPECIAL ISSUE).*

Schrijver, A. (2010). On the history of the shortest path problem. *Documenta Mathematica*, *15*, 155-167.

Smith, J. R. (1992). *The design and analysis of parallel algorithms.* [Online], accesible: http://vorpal.math.drexel.edu/course/cuda_parallel/para.pdf.

Tanimoto, S. L. (1987). *The elements of artificial inteligence, an introduction using lisp* (1.ª ed.). Rockville, Maryland, U.S.: Computer Science Press.

Warshall, S. (1962, Enero). A theorem on boolean matrices. *J. ACM*, *9*(1), 11–12. Descargado de http://doi.acm.org/10.1145/321105.321107

Yamani, I. I., Noorzaily, M. N., Zaidi, R., y Nor, R. D. (2005, Diciembre). Parking system using chain code & a-star algorithm. En *Proceedings of the international conference on intelligent systems, icis2005.*

Yoshikazu, K., Akihiro, K., y Osamu, W. (2011, July). Evaluations of hash distributed a* in optimal sequence alignment. En *Twenty-second international joint conferences on artificial intelligence.*

Zhou, Z. (2008). *Models and algorithms for addressing travel time variability: Applications from optimal path finding and traffic equilibrium problem.* Tesis Doctoral no publicada, Utah State University.