Algoritmos de planificación de caminos

*

1st Sergio Masa Avís
Universidad Carlos III de Madrid
Madrid, España
100401052@alumnos.uc3m.es

2nd Ángel Ignacio Arnanz Paredes
Universidad Carlos III de Madrid
Toledo, España
100405096@alumnos.uc3m.es

3rd Julio Sánchez Mirón
Universidad Carlos III de Madrid
Granada, España
100394981@alumnos.uc3m.es

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

I. Breadth-First Search (BFS)

A. Descripción del Algoritmo

Se trata de una estrategia en la que se parte de un nodo raíz. Este es el primero que se expande, dando lugar a nodos sucesores. Estos se expandirán para una profundidad dada en el árbol de búsqueda antes de que ningún nodo del siguiente nivel se expanda. De modo que el nodo menos expandido es elegido para ser el próximo en expandirse.

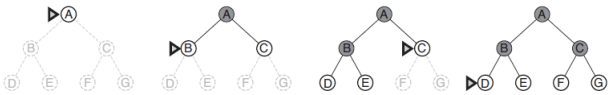


Fig. 1: BFS en un árbol binario

Esto se logra fácilmente debido a que se usan colas FIFO en la frontera. De manera que los nuevos nodos generados van a la parte de atrás de la cola mientras que los nodos antiguos, que tienen menor profundidad que los nodos nuevos, son expandidos. Este algoritmo descarta cualquier camino o conjunto ya en la frontera o conjunto explorado. Se busca siempre la ruta menos profunda para todos los nodos en la frontera ya que en cuanto el nodo es generado objetivo sabemos que todos los posibles nodos menos profundos han sido explorados.

Pseudocódigo

1) BFS (q,origen,destino,mapa)
2) Mientras existan vértices que recorrer:
3) explored → mapa con nodos visitados
4) q → siguiente nodo a evaluar de la Cola
5) Si q = destino
6) Se ha encontrado el camino
7) Si q != origen
8) Generamos 4 nodos más próximos
9) Para cada hijo se comprueba que no esté en explored y se encuentre dentro del mapa
10) Metemos al hijo en la cola q

B. Propiedades

Las propiedades del algoritmo BFS son:

- Completo.
  Se ve fácilmente que es completo, si el nodo de interés está a una profundidad finita, d, BFS encontrará el nodo objetivo (siempre que el factor bifurcación b sea finito) tras generar todos los nodos superficiales.

- Óptimo.

  El nodo objetivo más superficial no tiene porque ser el más óptimo, debido a que la distancia entre los nodos puede que no sea la misma. BFS es óptimo si la ruta de coste es una función de la profundidad del nodo no decreciente. El caso más común es que todos los caminos tengan el mismo coste.

C. Complejidad

- 1) Complejidad computacional o en tiempo.
  Suponiendo que la solución tenga una profundidad, d. En el peor caso, es el último nodo generado en ese nivel. Siendo el número de nodos generados:

$$b + b^2 + b^3 + \cdots + b^d = O(b^d)$$

Fig. 2

- 2) Complejidad en memoria o en espacio.
  Para este tipo de gráficos de búsqueda, los cuales almacenan cada nodo expandido en el conjunto explorado, la complejidad en el espacio se encuentra en el factor b de la complejidad en el tiempo. Para la búsqueda en BFS, cada nodo generado se mantiene en memoria. En el conjunto explorado hay $O(b^{d-1})$ nodos y O(b) nodos en la frontera siendo la complejidad del espacio $O(b^d)$. En una búsqueda de árbol no se ahorraría mucho espacio ni

tiempo debido a que en el espacio de estados hay muchas rutas redundantes.

II. Depth-First Search (DFS)

A. Descripción del Algoritmo

En el árbol de búsqueda siempre se expande el nodo más profundo de la frontera. La búsqueda procede inmediatamente a los niveles más profundos del árbol de búsqueda, los cuales no tienen sucesores. En caso de que no encuentre el nodo objetivo por esa rama realizará una vuelta atrás al siguiente nodo con mayor profundidad que todavía no haya sido explorado.

Se emplea la cola LIFO donde los nuevos nodos generados son los elegidos para su expansión.

Las propiedades y complejidad del DFS depende de si se emplea la versión graph-search o tree-search. En este trabajo se ha implementado la versión tree-search.

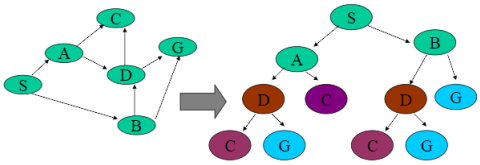


Fig. 3

Pseudocódigo

```
1)  DFS (q,origen,destino,mapa)
2)          Mientras existan nodos que recorrer:
3)          explored → mapa con nodos visitados
4)          Mientras que haya elementos en la Pila
5)          q → siguiente nodo a evaluar de la Pila
6)          Si q = destino
7)                  Se ha encontrado el camino
8)          Si q != origen
9)                  Generamos 4 nodos más próximos
10)         Para cada hijo se comprueba que no esté en
        explored y se encuentre dentro del mapa
11)         Metemos al hijo en la cola q
```

B. Propiedades

Las propiedades del algoritmo DFS son:

- Completo.
  No es completo, ya que la búsqueda del primer árbol en profundidad puede modificarse sin coste de memoria adicional para verificar los nuevos estados con los de la ruta desde el nodo raíz hasta el nodo actual. Esto evita bucles infinitos en espacios de estados finitos, pero no evita que se creen caminos redundantes.
- Óptimo.

  No es óptimo porque expande un nodo (B) y explora todo el subárbol de este nodo (B) a pesar de que el nodo vecino (C) sea el nodo objetivo. Suponiendo que el nodo J y nodo C fuera el mismo la versión DFS devolvería este camino como solución a pesar de que no fuera la mejor solución. Hasta que no evalúa todos los nodos generados a partir de este primero (nodo A) no ira a expandir los nodos hermanos de este primer nodo que se evaluó.

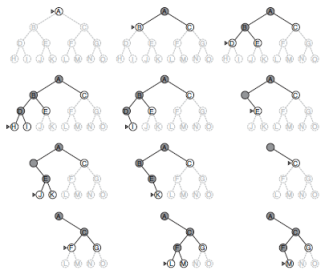


Fig. 4: DFS en un árbol binario

C. Complejidad

- 1) Complejidad computacional o en tiempo.
  Genera todos los nodos de O(b^m), donde b es el factor de ramificación y m es la profundidad máxima del nodo. Se ha de tener en cuenta que m puede ser mayor a d, que es la profundidad de las solución más superficial.
- 2) Complejidad en memoria o en espacio.
  El motivo de incluir el algoritmo DFS es la complejidad en el espacio. La ventaja obtenida es que necesita almacenar únicamente el camino del nodo raíz al nodo objetivo, junto con los nodos hermanos restantes no expandidos para cada uno de los nodos que conforman el camino. Tras ser expandido un nodo puede ser eliminado junto con sus descendientes explorados.

III. Dijkstra's algorithm

A. Idea general

Al igual que en los métodos descritos previamente, estamos ante un algoritmo ('no-informado') clásico de búsqueda de caminos entre dos nodos de un grafo. La peculiaridad reside en que este método trata de encontrar el camino mínimo, es decir, el de menor distancia desde el nodo origen (punto inicial) hasta un nodo destino (punto final).

Este algoritmo, fue diseñado originalmente por el científico holandés *Edsger Wybe Dijkstra* [wikipedia] cuando se planteó encontrar el camino más óptimo para viajar de Rotterdam a Groningen.

Fig. 5: Posibles alternativas para viajar de Rotterdam a Groningen

B. Descripción del algoritmo

La idea subyacente [http://ramonlopez.hol.es /Algoritmo20de20Dijkstra.pdf] consiste en ir explorando **todos** los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices. El algoritmo finaliza cuando se alcanza el camino mínimo del nodo origen al resto de nodos.

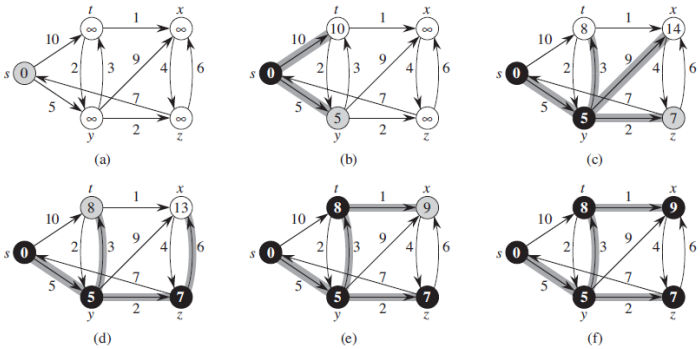


Fig. 6: Exploración llevada a cabo por Dijkstra [ref Jatin thaku http://www.codebytes.in/2014/08/dijkstras-algorithm-implementation-in.html]

Pseudocódigo

1) DIJKSTRA (origen,destino)
2) Mientras existan vértices que recorrer:
3) actual = nodoConMínimoCoste()
4) Generamos los hijos, para cada hijo:
5) Si no se había generado previamente:
6) Lo añadimos al conjunto y actualizamos coste (coste(actual) + coste(actual,hijo))
7) Si sí se había generado y el nuevo coste es menor que el que tenía:
8) Actualizamos padre y coste
9) Volvemos a 2

C. Propiedades

Las propiedades del algoritmo de Dijkstra son:

- Completo. Al recorrer todos los nodos, podemos afirmar que el algoritmo encuentra solución siempre que exista.
- Óptimo. El algoritmo de Dijkstra encuentra el camino mínimo dados dos nodos. Esta fue la principal propiedad que se trató de satisfacer, la razón de ser de Dijkstra.

D. Complejidad

Para determinar la complejidad computacional [wikipedia] se pueden contar las operaciones realizadas:

- El algoritmo consiste en n-1 iteraciones (recorrido de todos los nodos).
- En cada iteración, se identifica el nodo con menor coste y se realiza una suma y comparación para actualizar el coste de los vértices que no se hayan expandido. 2*(n-1) iteraciones

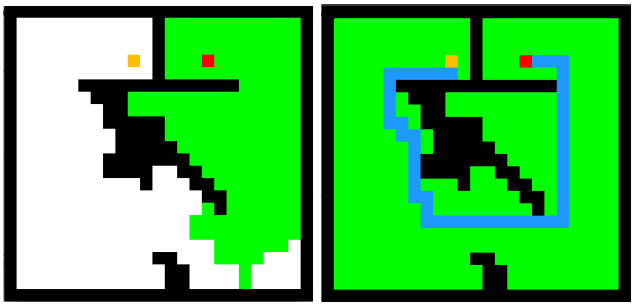
Se realizan por tanto un total de $2*(n-1)*(n-1)$. Luego la complejidad del algoritmo es de orden O(n^2). No obstante, dependiendo de la estructura de datos escogida para almacenar los nodos, las operaciones se podrían realizar más/menos rápidas y por tanto el tiempo empleado variará ligeramente.

Debido a que para cada nodo, almacenamos los caminos más cercanos al resto de los N nodos, la complejidad en espacio es del orden O($N * N$) = O(N^2)

E. Experimentación

Escogiendo aleatoriamente los mapas donde mostrar los resultados obtenemos:

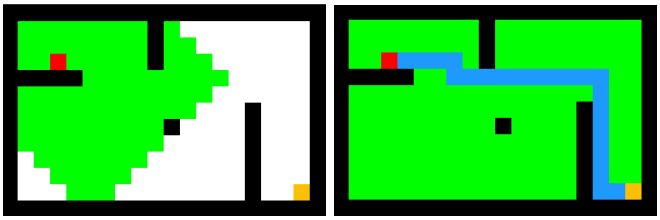
- Para el mapa *mapa5.csv* partiendo de *(4,16)-celda roja* y queriendo alcanzar *(4,10)-celda amarilla*:



(a) En fase de exploración (b) Mapa explorado-camino encontrado

Fig. 7: Simulación de Dijkstra en $mapa5.csv$

- Para el mapa *mapa8.csv* partiendo de *(3,3)-celda roja* y queriendo alcanzar *(11,18)-celda amarilla*:



(a) En fase de exploración (b) Mapa explorado-camino encontrado

Fig. 8: Simulación de Dijkstra en $mapa8.csv$

IV. Greedy First Search (GFS)

A. Descripción del algoritmo

Estamos ante el método más general de los catalogados algoritmos de búsqueda informados. GFS trata de explorar el grafo expandiendo (en primer lugar) aquellos nodos más prometedores, es decir, que se estima que están más próximos del objetivo.

¿Cómo sabemos cuál es el nodo más prometedor?

El nodo más prometedor será aquel cuya función de evaluación (coste) sea la mínima. En todos los métodos basados en esta técnica, la función de evaluación **f(n)** incluye una componente heurística, denotada **h(n)**:

h(n) = Coste estimado de la distancia entre el nodo n y el nodo objetivo.

En concreto, GFS utiliza la función de evaluación

$$f(n) = h(n)$$

Pseudocódigo

```
1) GFS (q,origen,destino)
2) Mientras que la cola q no esté vacía:
3)         actual = nodoConMínimaFunciónEvaluación(q)
4)         Si actual = destino: FIN
5)         Si No:
6)           generamos los hijos del nodo actual
7)           Volvemos a 2
```

Como vemos, la metodología en la realización de la búsqueda es parecida a la búsqueda en anchura (sección II). Las dos únicas diferencias residen:

- Línea 3. Mientras que BFS escoge el primer nodo que encuentre en la lista para expandir, GFS escoge el, a priori, más prometedor (menor f(n)).
- En búsqueda en anchura si no podemos seguir avanzando por un nodo (hijos explorados), retrocedemos. En la versión original de GFS esto no es posible, es decir, no hay backtracking o 'vuelta atrás'

B. Propiedades

Para argumentar las propiedades de este algoritmo de búsqueda, se va a utilizar como ejemplo la determinación de caminos entre ciudades del siguiente mapa de Rumanía:

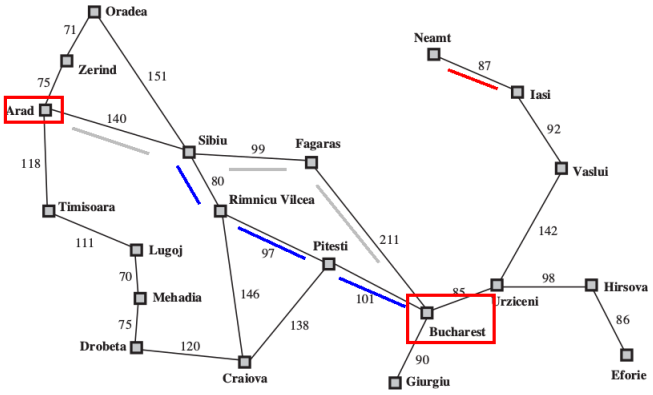


Fig. 9: Mapa de ciudades de Rumanía

Supongamos que queremos que nuestro robot vaya desde Arad a Bucharest. Para que GFS funcione, necesitamos conocer para cada ciudad C_i, **h**(c_i). Esta información se recoge en la tabla:

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Fig. 10: Heurísticas para GFS [ref libro]

Para trazar el camino más corto entre Arad-Bucharest, GFS expandirá Sibiu-Fagaras-Bucharest (camino gris) que como podemos apreciar, no es el óptimo. El camino óptimo sería Arad-Sibiu-Rimnicu Vilcea-Pitesti-Bucharest (camino azul).

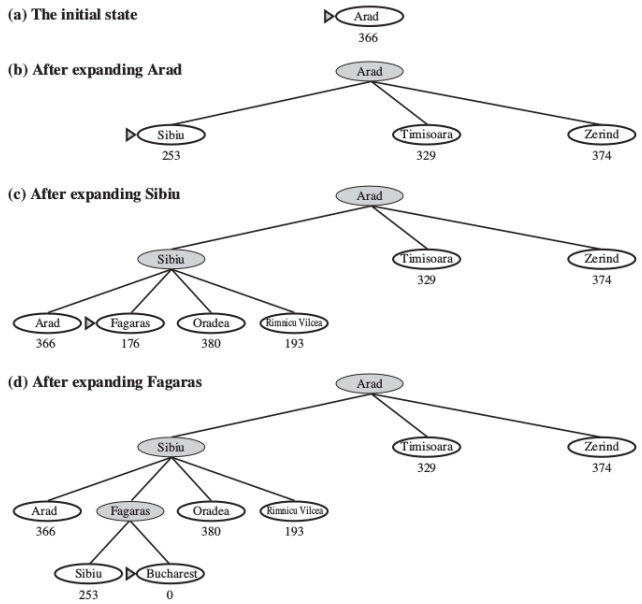


Fig. 11: Exploración de GFS [ref libro]

Además si intentáramos trazar un camino Iasi-Fagaras, el algoritmo se quedaría estancado en Neamt.

Por tanto, podemos concluir:

- GFS no es óptimo, es decir, no asegura encontrar el camino mínimo entre dos puntos.
- GFS no es completo, es decir, puede no encontrar camino alguno entre dos nodos.

C. Complejidad

Las dos propiedades negativas previamente descritas nos hacen preguntarnos si realmente es útil el uso de GFS. Es aquí donde podemos argumentar:

1) Debido a que no realiza backtracking, la complejidad en espacio requerida por el algoritmo se reduce notablemente.
2) Se reduce además la complejidad en tiempo sustancialmente si se hace uso de una buena función heurística.

D. Experimentación

En la fase de experimentación se desarrollaron dos versiones de GFS:

1) gfs_incomplete . Versión original de GFS que tal y como hemos explicado puede no encontrar el camino.
2) gfs_complete . Versión completa de GFS.

El resultado obtenido para el mapa *mapa7.csv* partiendo de (4,3)-celda roja y queriendo alcanzar (20,20)-celda amarilla con *gfs_incomplete* es:

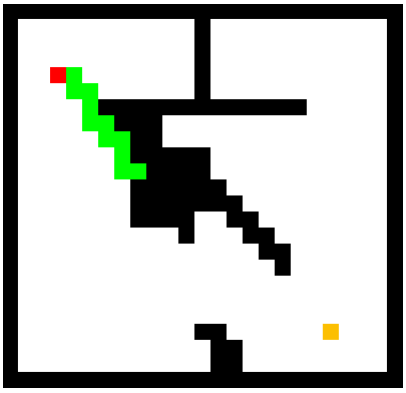
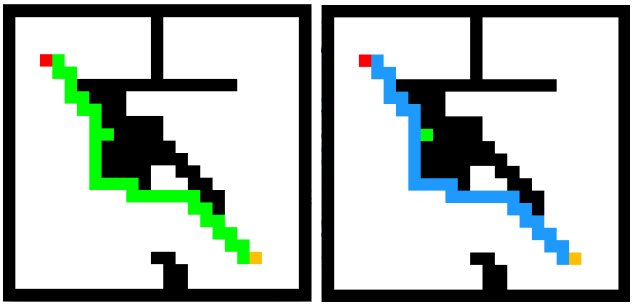


Fig. 12: Exploración *gfs_incomplete*

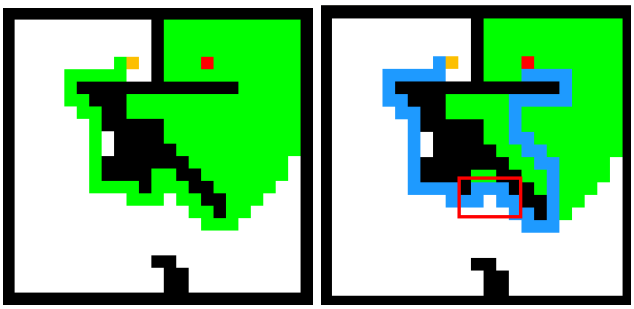
Como vemos, el algoritmo no ha sido capaz de encontrar el camino (sea o no mínimo). Es por ello que se probó con la segunda versión implementada obteniendo:



(a) Exploración (b) Camino encontrado

Fig. 13: Exploración y camino encontrado por *gfs_complete*

El resultado de *gfs_complete* para el *mapa5.csv* y los puntos: (4,16) Origen - (4,10) Destino:



(a) Exploración (b) Camino encontrado

Fig. 14: Exploración y camino encontrado por *gfs_complete*

Como vemos, *gfs_complete* encuentra el camino, pero sigue sin encontrar el óptimo. La zona rectangular roja en la figura 5.(b) lo pone de manifiesto. Una variante de algoritmos greedy que sí que consigue encontrar el camino óptimo es A*.

V. A*

A. Descripción del algoritmo

A* es un algoritmo muy popular usado en planificación de caminos. Fue publicado por primera vez en 1968 [wikipedia] por los autores *Peter Hart*,*Nils Nilsson* y *Bertram Raphael*.

El secreto del éxito de A* [ref http://theory.stanford.edu/ amitp /GameProgramming/AStarComparison.html] es que combina:

- La información que usa Dijkstra (favorecer a los nodos más cercanos al origen)
- La información heurística que GFS usa (favorecer a los nodos más cercanos al destino)

Gracias a ello, A* podría usarse para encontrar caminos óptimos (tal y como hace Dijkstra) usando heurísticas (tal y como GFS hace) y por tanto, en el caso más simple, es tán rápido como GFS.

En definitiva, A* examina cada nodo con la función de evaluación: $f(n) = g(n) + h(n)$, donde:

- $g(n)$ Es el coste exacto desde origen hasta el nodo n.
- $h(n)$ Es una estimación heurística del coste del camino desde n hasta el destino.

Pseudocódigo

1) A* (origen,destino)
2) Definimos CERRADOS = nodos evaluados
3) Definimos ABIERTOS = nodos no evaluados
4) Añadir(ABIERTOS,origen)
5) Mientras que ABIERTOS no esté vacía:
6) nodoM = Nodo con menor f(n) = g(n) + h(n)
7) Si nodoM = destino : FIN
8) Eliminamos nodoM de ABIERTOS
9) Añadimos nodoM a CERRADOS
10) Generamos los hijos del nodoM
11) Para cada hijo:
12) Si el hijo está en CERRADOS : Volvemos a 5
13) d = Distancia de origen al hijo
14) Si hijo no está en ABIERTOS:
15) Añadir(ABIERTOS,hijo)
16) Si d < Coste Anterior que tenía guardado hijo:
17) Actualizamos el hijo : Ponemos su mejor padre nodoN
18) Actualizamos f(n) del hijo y volvemos a 5

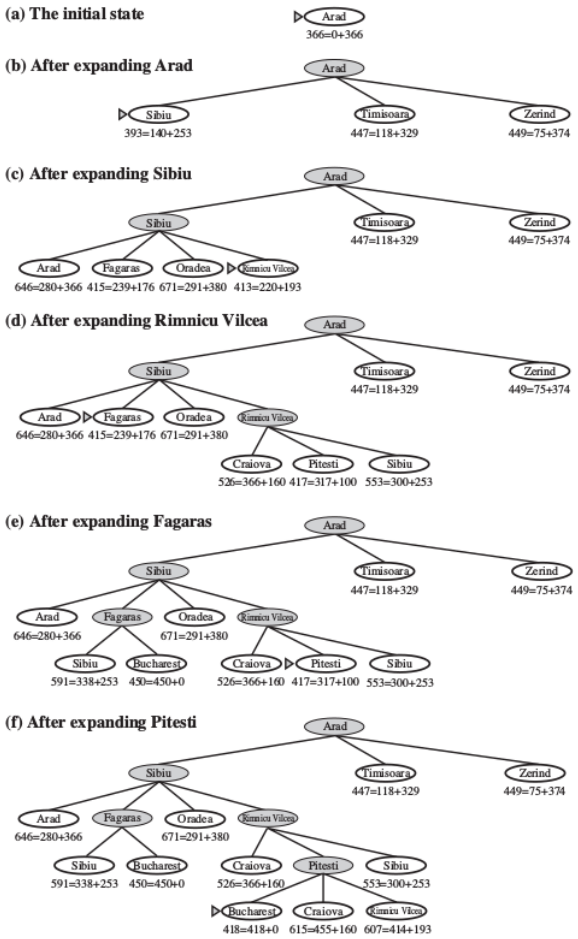


Fig. 15: Exploración de A* [ref libro]

Vemos como en la figura anterior (Fig.11) cada nodo se etiqueta con f(n) = g(n) + h(n).

B. Propiedades

Las propiedades [wikipedia][https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/astar.html] de A* son:

- El algoritmo es completo. Gracias a ir almacenando los sucesores del mejor nodo en ABIERTOS, si el camino no puede seguir por un nodo, realiza un backtracking al siguiente nodo con mejor f(n).
- El algoritmo garantiza encontrar la solución óptima sii la heurística de la que hace uso es *admisible* (Una heurística *h* es admisible si la estimación h(n) nunca sobreestima la distancia real del nodo n al nodo destino).

$$\forall n \; h(n) <= CosteReal(n, destino)$$

C. Complejidad

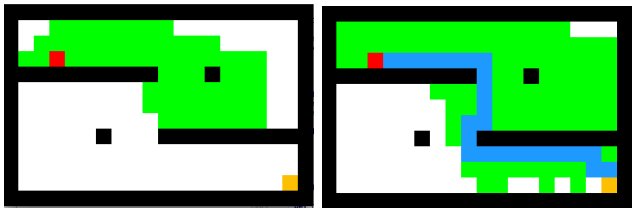
En lo que respecta a la complejidad [wikipedia] de A*:

- La complejidad *computacional* está íntimamente relacionada con la calidad de la heurística que se utilice.
  - En el peor de los casos exponencial : O(b^d).
  - En el mejor de los casos polinomial.
- La complejidad en *memoria* es su mayor problema ya que es exponencial respecto al tamaño del problema.

D. Experimentación

Escogiendo aleatoriamente los mapas donde mostrar los resultados obtenemos:

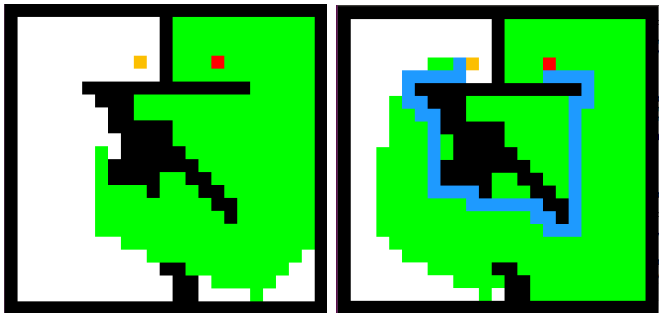
- Para el mapa *mapa6.csv* partiendo de *(3,3)-celda roja* y queriendo alcanzar *(11,18)-celda amarilla*:



(a) En fase de exploración (b) Mapa explorado-camino encontrado

Fig. 16: Simulación de A* en $mapa6.csv$

- Para el mapa *mapa5.csv* partiendo de *(4,16)-celda roja* y queriendo alcanzar *(4,10)-celda amarilla*:



(a) En fase de exploración (b) Mapa explorado-camino encontrado

Fig. 17: Simulación de A* en $mapa5.csv$

VI. Rapidly exploring random tree (RRT)

En esta sección se presenta un enfoque de algoritmos de planificación basado en muestreo.

La idea fudamental de este tipo de planificadores es la de aproximar la conectividad del espacio de configuraciones mediante una estructura de grafos. El espacio a explorar es muestreado, generando estados que formarán los vertices de dicho grafo. Las líneas que unen los vértices del grafo denontan segmentos de camino válidos.

El objetivo es capturar la conectividad del espacio de configuración del agente o robot en movimiento a través de un muestreo aleatorio. Los árboles de exploración aleatoria rápida (Rapidly Exploring Random Tree o RRT) fueron desarrollados por Steven M.LaValle y James J.Kuffner Jr. Pueden ser vistos como una técnica para generar trayectorias de bucle abierto con el fin de buscar, de manera eficiente, los espacios de alta dimensión no convexos mediante la construcción aleatoria de un árbol de relleno de espacio.

A. Descripción del algoritmo

La idea de un RRT es hacer crecer un arbol de búsqueda, que mejore gradualmente, arraigado en la configuración inicial utilizando muestras aleatorias del espacio de búsqueda. A medida que se toma cada muestra, se intenta realizar una conexión entre ella y el estado más cercano en el árbol. Si la conexión es factible (pasa completamente a través del espacio libre y obedece cualquier restricción), se produce la adición del nuevo estado al árbol.

La longitud de la conexión entre el árbol y un nuevo estado está frecuentemente limitada por un factor de crecimiento. Si la muestra aleatoria se encuentra a una distancia mayor de su estado más cercano en el árbol de lo que permite este límite,

se genera un nuevo estado a la distancia máxima del árbol a lo largo de la línea hasta la muestra aleatoria en lugar de la muestra aleatoria en sí. Las muestras aleatorias se pueden ver como control de la dirección del crecimiento del árbol, mientras que el factor de crecimiento determina su tasa. Esto mantiene el sesgo de relleno de espacio del RRT al tiempo que limita el tamaño del crecimiento incremental.

El crecimiento de RRT puede estar sesgado al aumentar la probabilidad de muestrear estados de un área específica. La mayoría de las implementaciones prácticas de las RRT hacen uso de esto para guiar la búsqueda hacia los objetivos de planificación de problemas. Esto se logra introduciendo una pequeña probabilidad de muestrear la meta al procedimiento de muestreo estatal. Cuanto mayor es esta probabilidad, más veloz crece el árbol hacia la meta.

El empleo de este tipo de algoritmos en planificadores, presenta varias etapas, las cuáles se encuentran representadas en la Fig 14.

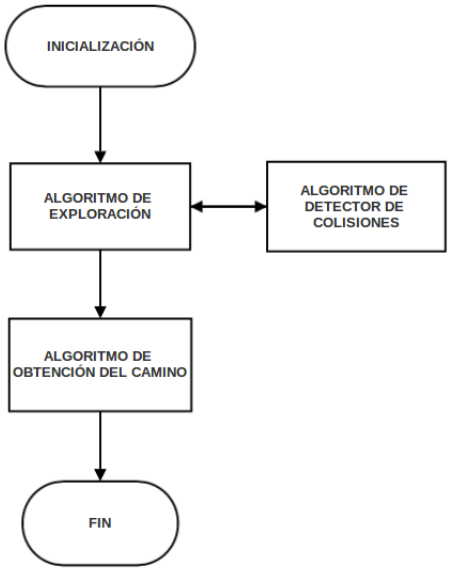


Fig. 18: Etapas de RRT [ref libro]

En el estado de inicialización se realizan las configuraciones necesarias del algoritmo, tales como la lectura del espacio de estados, la determinación de punto origen del robot, del destino etc.

ALGORITMO DE EXPLORACIÓN

El segundo estado que presenta el diagrama de flujo, trata el algoritmo de exploración utilizando un árbol aleatorio de búsqueda.

El pseudocódigo que permite la construcción de un árbol de exploración básico en un espacio de configuraciones sin obstáculos (donde todo el espacio es completamente libre).

1) SIMPLE_RDT (q0)
2) Iniciamos el grafo G con q0.
3) Desde i=1 hasta NumeroIteraciones:
4) Añadimos a G un nuevo vértice aleatorio generado : $\alpha(i)$
5) $q_n \Leftarrow MasCercano(S(G), \alpha(i))$
6) Añadir línea que une $q_n\alpha(i)$

En resumidas cuentas, el algoritmo requiere la disponibilidad de una secuencia densa, α, y se conecta iterativamente desde $\alpha(i)$ al punto más cercano entre todos los alcanzados por el arbol, G.

En la Fig.15 se muestra en (a) un arbol que ha sido construido estableciendo un punto inicial q0. En (b) un nuevo eje aleatoriamente es añadido conectándose desde $\alpha(i)$ al punto más cercano en S, vértice q_n.

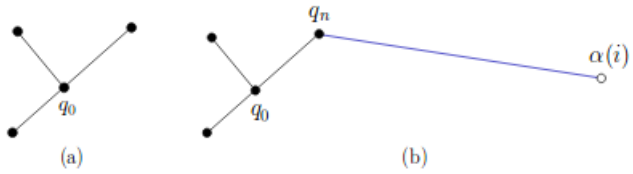


Fig. 19: Crecimiento del árbol (1)

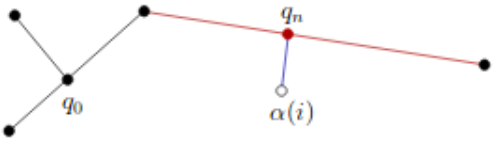
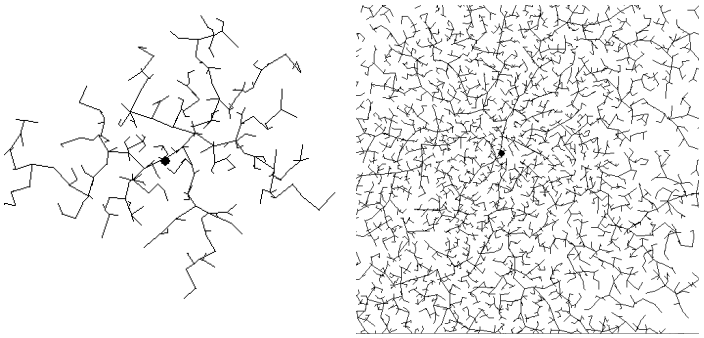


Fig. 20: Crecimiento del árbol (2)

En la Fig.17 se muestra la visualización en un espacio libre de obstáculos de la misma simulación una vez transcurridas 3500 y 6500 iteraciones.



(a) Tras 3500 iteraciones (b) Tras 6500 iteraciones

Fig. 21: Simulación en mapa libre de obstáculos

Una vez se ha muestreado todo el espacio libre del espacio de configuraciones, se obtiene el camino mediante la obtención de los padres de cada nodo.

B. Propiedades

Las propiedades del RRT son:

- Probabilísticamente completos. A medida que crece el número de iteraciones, la probabilidad de encontrar una ruta de solución tiende a la unidad si al menos existe una solución, de lo contrario el algortimo se repite para siempre.
- Algoritmo no óptimo. Debido a la naturaleza aleatoria de los árboles de búsqueda, este tipo de planificadores no obtienen la ruta más óptima ya que en cada simulación, los nodos serán diferentes y por tanto el camino también lo será.

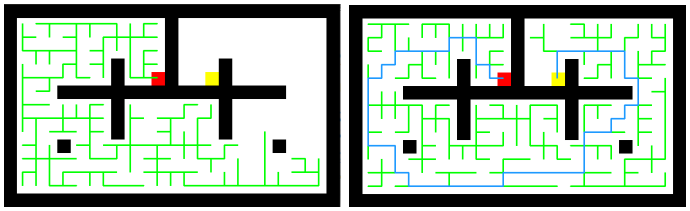
C. Complejidad

Para determinar la complejidad del tiempo (time complexity) será de O(M(N+K), donde M es el número de iteraciones, N, el tiempo empleado en encontrar el nodo más cercado, qn , y K el tiempo empleado en conectar ambos nodos.

D. Experimentación

Escogiendo aleatoriamente los mapas donde mostrar los resultados, obtenemos:

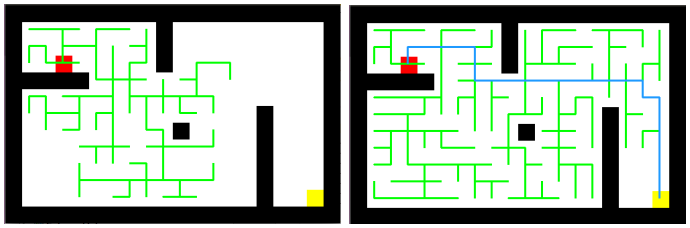
- Para el mapa *mapa3.csv* partiendo de *(5,11)-celda roja* y queriendo alcanzar *(5,15)-celda amarilla*:



(a) En fase de exploración (b) Mapa explorado-camino encontrado

Fig. 22: Simulación de RRT en $mapa5.csv$

- Para el mapa *mapa8.csv* partiendo de *(3,3)-celda roja* y queriendo alcanzar *(11,18)-celda amarilla*:



(a) En fase de exploración (b) Mapa explorado-camino encontrado

Fig. 23: Simulación de RRT en $mapa8.csv$

References

Please number citations consecutively within brackets [1]. The sentence punctuation follows the bracket [2]. Refer simply to the reference number, as in [3]—do not use “Ref. [3]” or “reference [3]” except at the beginning of a sentence: “Reference [3] was the first . . .”

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the abstract or reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors’ names; do not use “et al.”. Papers that have not been published, even if they have been submitted for publication, should be cited as “unpublished” [4]. Papers that have been accepted for publication should be cited as “in press” [5]. Capitalize only the first word in a paper title, except for proper nouns and element symbols.

For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [6].

References

[1] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.
[3] I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.
[4] K. Elissa, “Title of paper if known,” unpublished.
[5] R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.
[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].
[7] M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.