# Capítulo 1. Algoritmos que vamos a estudiar

En este capítulo de presentaran los diferentes algoritmos que vamos a usar en el análisis de rendimiento en el campo del pathfinding.

Si bien A\* es un algoritmo de búsqueda que siempre da la solución óptima y es muy eficiente en tiempo, no lo es en memoria, como sí lo son los algoritmos de backtraking, por eso, en muchas ocasiones no podemos usar A\* ya que los costes de memoria serían enormes, es por eso que en este trabajo también se trataran otros dos algoritmos que son más adecuados para el problema que estamos tratando, que es el pathfinding en mallas cuadradas, a costa de sacrificar otros factores como es el encontrar la solución óptima, usando como referencia por sus propiedades A\*.

## A\*

Los algoritmos voraces, se guían solamente por la función heurística, lo que hace que no siempre se indique el camino con coste más bajo, derivando en que, en caso de encontrar la solución, no sea la óptima, es por eso por lo que A\* no solo se basa en una función heurística, sino que también tiene en cuenta el coste real del recorrido, obteniendo así la función de evaluación del algoritmo:

Donde g(n) es el coste real del camino hasta el nodo n, y h(n) es la función heurística del nodo n, en este caso usaremos la distancia octil como función heurística ya que se adapta perfectamente a nuestro problema.

Este algoritmo, si existe encontrará la solución óptima, pero para ello, se necesita que se cumplan algunas propiedades:

* El heurístico debe ser admisible, ósea, que no sobrestime el coste real de alcanzar el nodo (si no cumple esto, el algoritmo será A, y no asegura encontrar la solución óptima)
* Si un heurístico satisface la restricción monótona, A encontrara un camino óptimo para todo nodo seleccionado que cumple que g(n)=g\*(n)

Sabiendo esto, solo queda mostrar la forma en que trabaja A\*, y es que de una forma simplificada, trabaja con una lista de nodos abiertos y cerrados, la lista de abiertos se ordenan de menor a mayor mediante la función f(n) vista anteriormente y va expandiendo por el primer nodo de la lista, que pasa a estar cerrado, todo nodo apunta a su “padre” para después saber cuál es el camino solución y si nos encontramos con que llegamos a un nodo por dos o más caminos distintos, su padre será con el que menor f(n) tenga el nodo. Cuando encontremos en nodo objetivo y toque su expansión habremos acabado de ejecutar el algoritmo y habremos encontrado la solución óptima.

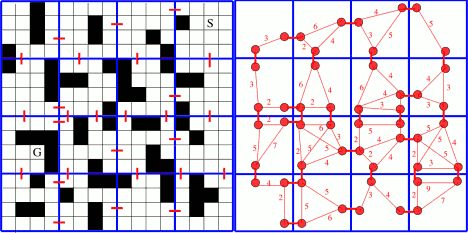
A continuación, se muestra el pseudocódigo del algoritmo A\*:

1. Crear un árbol de búsqueda A con raíz en s, y una lista de nodos ABIERTOS con s.
2. Crear una lista de nodos CERRADOS vacía.
3. Si ABIERTOS está vacía, entonces devolver ‘FRACASO’.
4. Seleccionar n <- primero (ABIERTOS). Borrar n de ABIERTOS y añadirlo a CERRADOS.
5. Si n es objetivo, entonces devolver el camino de s hasta n en A.
6. Expandir n, M <- sucesores (n, G) – antecesores (n, A).
7. Para cada n2 en M,
   1. Su n2 es nuevo (n2 no está ABIERTO ni CERRADO),
      1. Poner un puntero de n2 -> n.
      2. Añadir n2 a ABIERTOS.
   2. Si n2 no es nuevo, y el valor de g(n2) es menor a través del nuevo camino,
      1. Redirigir su puntero hacia n.
      2. Si n2 está en CERRADOS, entonces pasarlo a ABIERTOS.
8. Ordenar ABIERTOS por orden creciente en el valor de f(n) = g(n)+h(n).
9. Volver a 3.

## HPA\*

Como hemos dicho antes si bien A\* es un algoritmo rápido, no ocupa poca memoria, entonces HPA\* se encarga de eso, poder pasar por ejemplo en una malla de 512x512, es decir 262144 nodos a cerca de 10000 nodos, lo que supondría una rebaja bastante grande a la hora de la ocupación de memoria.

HPA\* no es más que un añadido al algoritmo anterior, si bien al final del todo aplicaremos A\*, la malla donde se moverá nuestro personaje se abstrae dividiéndose en clusters de tamaño predefinido y solo formaran parte de nuestro grafo las uniones de estos clusters, nuestros puntos de inicio y de final. También estos clusters formarán grupos que a su vez serán clusters más grande, por lo que podemos escalar el problema sin aumentar mucho el número de nodos de nuestra malla. La consecuencia de esto es que obtenemos una solución en menor tiempo que el algoritmo A\* a cambio de que esa solución puede que no sea la óptima.



<https://gamedev.stackexchange.com/questions/135967/hpa-pathfinding-building-the-hierarchical-graph-is-too-slow>

Si lo pensamos este problema se puede extrapolar a viajar entre ciudades: Si queremos ir desde la Universidad aquí en Málaga hacia la Universidad de Almería, por ejemplo, nosotros podemos aplicar A\* a un mapa con todas las carreteras de Andalucía, pero eso sería costoso computacionalmente, entonces lo que nosotros hacíamos es primero salir de Málaga y entrar en una autovía (salir de nuestro pequeño cluster), luego seguir esa autovía hasta llegar a Almería (viajar de un cluster a otro) y por ultimo ya estando en Almería buscar la universidad en la ciudad (movernos dentro de otro cluster).

Por último, tras aplicar A\*, podemos hacer un “suavizado” al camino obtenido consiguiendo así recorrer aún menos distancia y un movimiento más realista del personaje (nosotros por ejemplo no nos movemos en una malla cuadrada, por lo que, aunque nuestro escenario si lo sea el movimiento puede ser en línea recta desde el inicio al final).

Por ejemplo, en la siguiente figura se muestra en negro el camino obtenido por A\* y en azul el obtenido tras suavizar el camino.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

Aunque no sea mucha la ganancia en este ejemplo se aprecia que en los primeros movimientos el camino de la flecha azul es más directa y realista que el de la flecha negra.

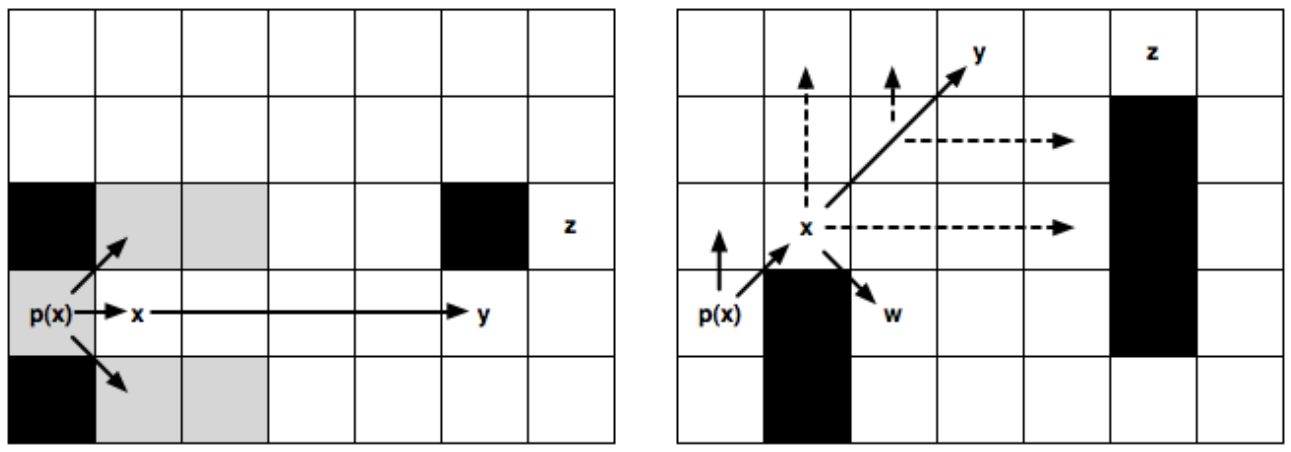
Entonces nuestro algoritmo HPA\* se basa en 3 pasos:

1. Preprocesar la.
   1. Dividir la malla en clusters.
   2. Establecer que nodos pertenecen a cada cluster.
   3. Establecer la conexión interna de cada cluster y la unión entre estos.
2. Aplicar el algoritmo A\* al grafo obtenido en el paso 1.
3. Si A\* devuelve FRACASO acaba el algoritmo y devuelve FRACASO.
4. Aplicar un “suavizado” (si es posible) al camino obtenido por A\*.
5. Devolver el camino obtenido.

## Jump Point Search

A diferencia del algoritmo HPA\*, que es un añadido por así decirlo al algoritmo base A\*, JPS o jump point search es una modificación del propio A\* pensada para mallas cuadradas.

A grandes rasgos lo que hace este algoritmo es aplicar A\*, pero mediante podas de los vecinos de algunos nodos, lo que reduce considerablemente el número de nodos explorados y saltos a nodos en la misma línea o diagonal.



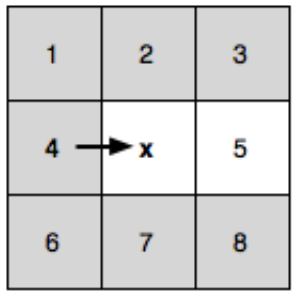
### Poda de vecinos

La poda de vecinos sigue dos reglas, con el fin de identificar y podar todos los nodos n vecinos de otro nodo x que no necesitan ser evaluados para alcanzar el objetivo de forma óptima. Esto se consigue comparando la longitud de dos caminos A, que comienza con el nodo p(x) (padre de x), visita x y acaba en n y B que empieza en p(x) y acaba en n pero no pasa por el nodo x.

Se da por hecho que todos los nodos visitados por A o B pertenecen a los vecinos de x.

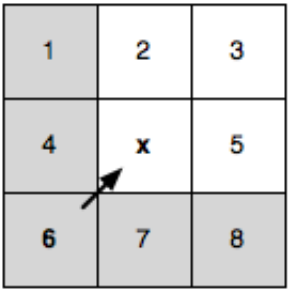
Si el nodo inicial es el nodo x, entonces p(x) es null y no se poda nada.

Las dos reglas que sigue la poda dependen de si el movimiento de p(x) a x es diagonal u horizontal/vertical:

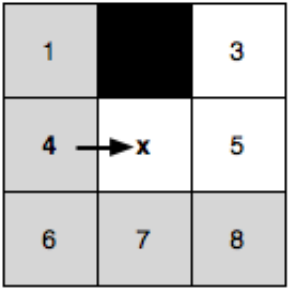
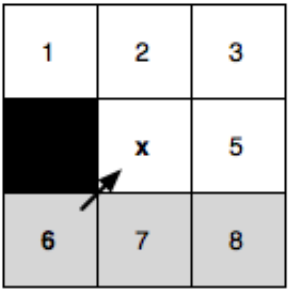
* Movimientos horizontales/verticales: Podamos cualquier nodo n € vecinos(x) que cumpla la siguiente condición:

Por ejemplo, en la siguiente imagen se ve como el único nodo que no se poda es el 5.

* Movimientos diagonales: Es similar a la regla anterior solo que en este caso la longitud debe ser estrictamente menor:

Esto se puede apreciar en la siguiente imagen, como los nodos que no se podan son el 2, 3 y 5.

Asumiendo que no tenemos obstáculos, a los nodos restantes tras la poda los llamamos vecinos naturales de x. En caso de que sí que tengamos obstáculos como pasa en la mayoría de los escenarios, no podremos podar todos los vecinos no naturales de x. Si esto ocurre diremos que la evaluación de cada vecino es forzada como ocurre en las imágenes siguientes.

**Definicion 1**

Un nodo n€vecinos(x) es forzado si:

1. N no es vecino natural de x.

### Saltos

Para empezar, describiremos de una forma más precisa el concepto de punto de salto.

**Definición 2.**

Un nodo y es un punto de salto desde el nodo x con dirección d, si y minimiza el valor k tal que y=x+k\*d y se cumple alguna de estas condiciones:

* El nodo y es el nodo objetivo.
* El nodo y tiene al menos un vecino cuya evaluación sea forzada acorde a la Definición 1.
* d es un movimiento diagonal y existe un nodo z=y+ki\*di que se encuentra a ki € N pasos en dirección d € [d1, d2] tal que z es un punto de salto desde y por la condición 1 o 2

Para identificar los sucesores usaremos el siguiente código:

Siendo s el nodo de partida, g el nodo objetivo y x el nodo actual:

1. Vaciamos la lista de sucesores de x.
2. Vecinos(x) <- podar (x, vecinos(x))
3. For all n € vecinos(x) do

n <- jump (x, direction(x, n), s, g)

añadir n a sucesores(x)

1. Devolver sucesores(x)

La función salto definida en la Definición 2, podemos verla en el siguiente código:

Siendo s el nodo de partida, g el nodo objetivo, x el nodo actual y d la dirección:

1. n <-avanzar (x, d)
2. Si n es un obstáculo o esta fuera de la malla entonces devuelve null.
3. Si n == g entonces devuelve n.
4. Si existe un n’ € vecinos(n) y n’ es forzado entonces devuelve n.
5. Si d es diagonal entonces

For all i €[1,2] do

If jump(n, di, s, g) is not null then return n

Return jump(n, d, s, g)