Inteligencia Artificial para Videojuegos – Capítulo 3

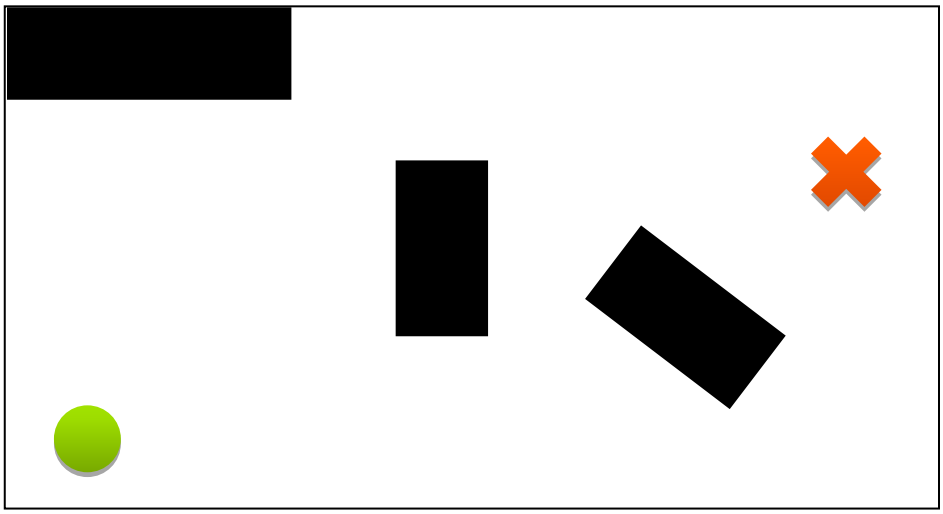
Introducción

En los videojuegos los personajes se mueven por un entorno dinámico y lleno de obstáculos. Muchas veces el jugador controla directamente el movimiento del personaje como por ejemplo en un FPS, pero muchas otras el jugador simplemente indica mediante un clic dónde debe ir el personaje, el camino que elige y cómo lo resuelve depende enteramente de nosotros los desarrolladores. Con los enemigos esta situación es aún peor ya que los mismos deben no sólo moverse en un entorno cambiante sin ningún tipo de ayuda sino que además deben tomar decisiones basadas en sus objetivos. Este capítulo trata justamente de cómo resolver estos problemas, es decir, cómo lograr que nuestros personajes se muevan "inteligentemente" por los escenarios. En la primer parte del apunte veremos un poco de teoría de grafos que es un pilar fundamental para la búsqueda de caminos y en la segunda mitad veremos algoritmos de búsqueda y su aplicación en videojuegos.

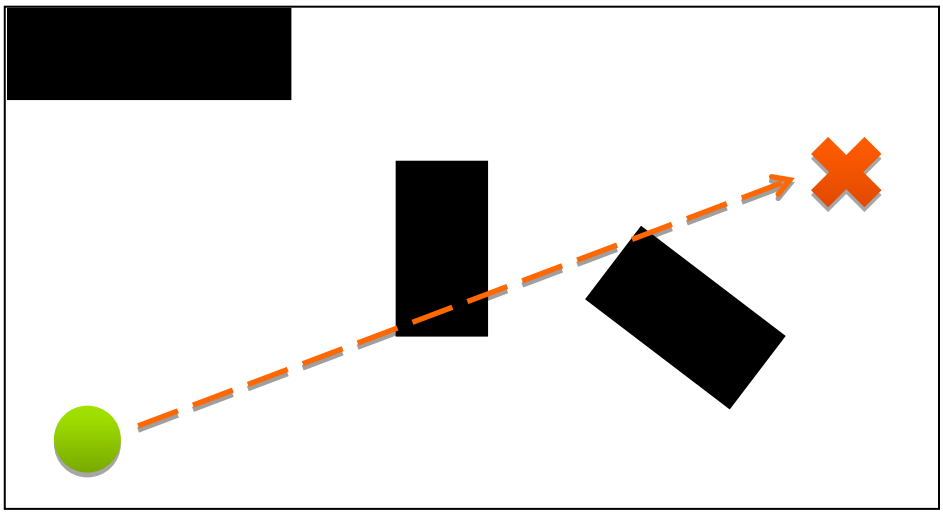
Grafos

¿Te gustan los juegos de estrategia?

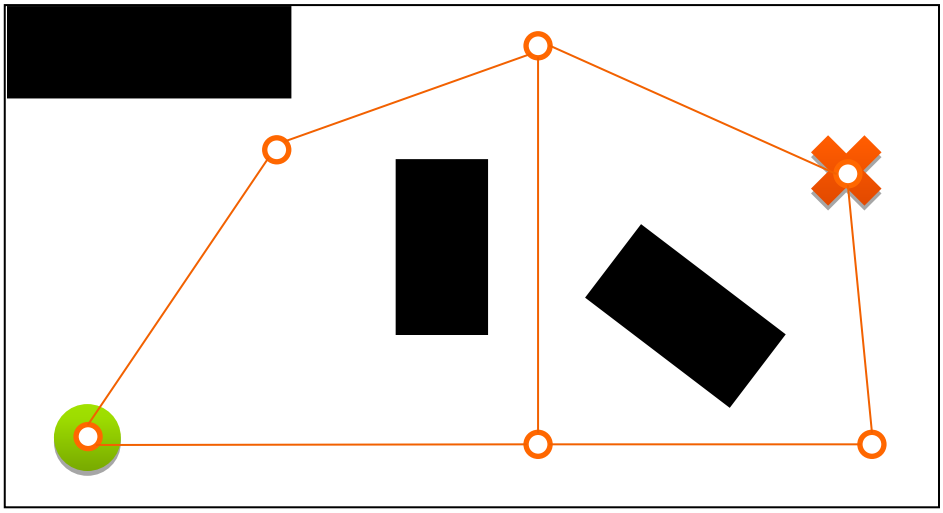
Imaginemos un juego de estrategia/rol dónde uno controla una unidad con el mouse:



En el gráfico el círculo verde representa nuestra unidad, la "x" roja el punto a dónde deseamos que vaya y los objetos negros son áreas por las que nuestra unidad no puede pasar. Si la unidad no posee información sobre el mundo, el camino lógico que debería elegir es ir al punto en línea recta:



Sin embargo dicho camino es imposible de realizar ya que se chocará contra un obstáculo. Para solucionar esto debemos proveer un poco más de información sobre el mundo a nuestra unidad. Supongamos que ahora a nuestra unidad le damos una serie de puntos en los cuales puede estar y unas líneas que los unen que describen a que puntos puede viajar sin problemas de obstáculos:

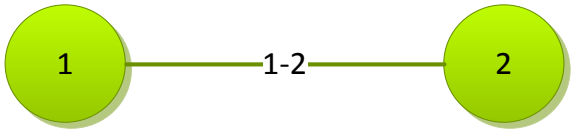


En este caso nuestra unidad se encuentra en el primer punto y sabe que moviéndose por las líneas naranjas a los puntos que están conectados no corre riesgo de chocar con nada. Esta es la idea fundamental detrás de todos los algoritmos de búsqueda de caminos. Y esta red de puntos y rectas que hemos dibujado se llama grafo.

Un grafo es una representación simbólica de una red, los puntos se llaman nodos y las rectas que los conectan aristas. En nuestro caso esta red representaba información espacial pero esto no es siempre así. Para ser un poco más formales definamos un grafo:

Un grafo es un conjunto de nodos N y aristas E y se escribe normalmente G={N,E}

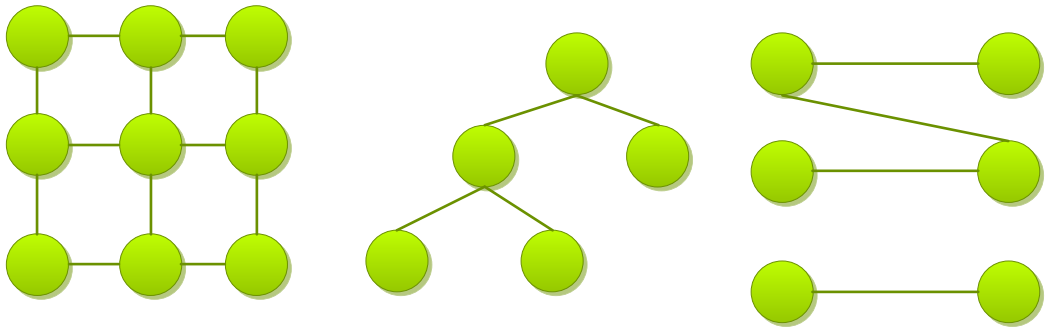
Los nodos se los suele identificar mediante un número y las aristas mediante los números de nodos que conectan. Por ejemplo dado el siguiente grafo:



Tenemos los siguientes conjuntos:

$$N = \{1,2\}$$
$$E = \{1 - 2\}$$

Es decir dos nodos y una arista identificada por los nodos que une. Los grafos pueden tener muchas formas, por ejemplo:



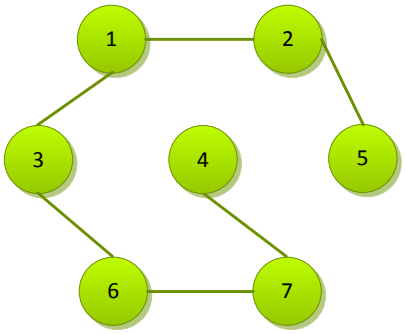
Se dice que un grafo es conectado cuando es posible trazar un camino entre cualquier par de nodos. En caso contrario el grafo es no conectado.

En la figura de arriba los dos primeros grafos son conectados mientras que el tercero es no conectado.

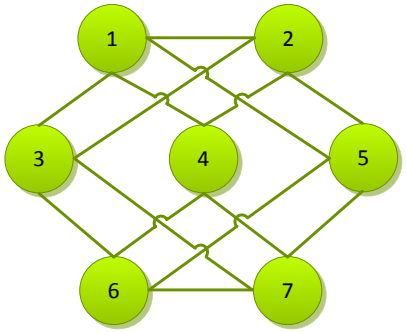
Densidad de un grafo

Los grafos pueden ser densos(dense) o ralos (sparse). Esto esta determinado por la razón entre nodos y aristas. Los grafos ralos poseen pocas conexiones por nodos, mientras que los densos poseen muchas. Es útil conocer de antemano como serán los grafos con que trabajaremos ya que

una implementación que sea eficiente con grafos densos probablemente no lo sea con grafos ralos. Por supuesto que siempre que sea posible es conveniente trabajar con grafos ralos, especialmente en videojuegos dónde debemos recorrerlo constantemente.



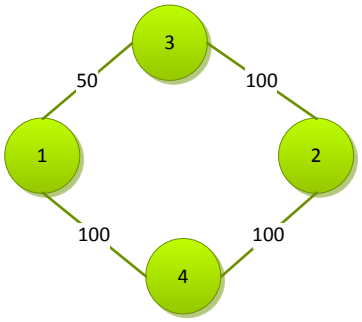
Ralo



Denso

Costos

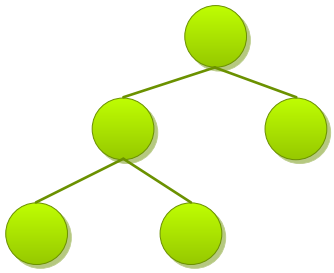
Normalmente a las aristas se les suele asociar un costo. El costo suele representar cuánto hay que pagar para poder recorrerla. En la búsqueda de caminos este costo suele ser la distancia. Pero también podría ser la nafta que ocuparemos, o la peligrosidad de la ruta, etc.



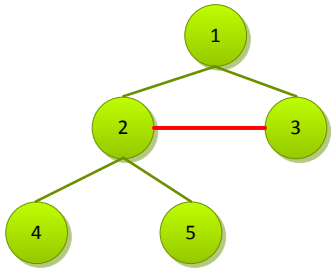
En el grafo de arriba podemos ver que para ir del nodo 1 al nodo 2 tenemos 2 caminos: Ir por las aristas {1-4,4-2} dónde ambas tienen un costo de 100, por lo que ir por ese camino nos cuesta la suma de cada arista, es decir 200. En cambio si tomamos el camino {1-3,3-2} podemos ver que el costo es 150, esto quiere decir que éste último camino es mejor “en algún sentido” al otro.

Árboles

Los árboles son una estructura de datos muy utilizada en programación. Un árbol es un tipo especial de grafo. Para que un grafo sea un árbol el mismo debe ser acíclico, es decir no debe contener caminos circulares. Esto quiere decir un árbol es aquél grafo dónde no podemos volver a un nodo A sin pasar dos veces por una misma arista. Veamos ahora un ejemplo:



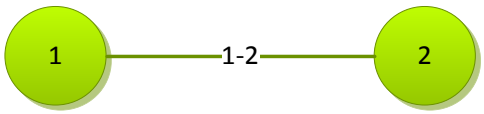
Esta figura es un árbol ya que no contiene caminos circulares, ahora si agregamos por ejemplo una arista nueva:



El grafo deja de ser un árbol ya que acabamos de crear un camino circular entre los nodos 1,2 y 3.

Digrafos

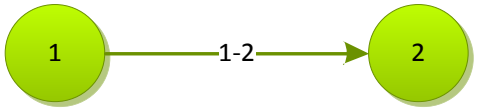
Hasta el momento cuando hablábamos de aristas suponíamos que podíamos recorrerla en cualquier sentido. Es decir dado el siguiente grafo:



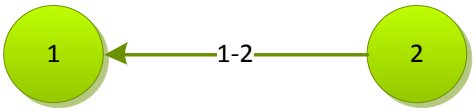
Podíamos usar la arista 1-2 para ir desde el nodo 1 hacia el 2, o en sentido inverso. Sin embargo muchas veces es útil y necesario que las aristas sólo puedan ser recorridas en un sentido. Para ver esto imaginemos un juego dónde el protagonista en una zona determinada pueda caminar hacia un lado y no hacia el otro. ¿Se te ocurre algún ejemplo? Por ejemplo si el héroe cruza una puerta mágica que sólo le permite entrar a la habitación pero no salir por la misma. Esto se representa con grafos indicando cuál es el sentido de la arista que puede recorrer el héroe.

También puede suceder que se pueda recorrer en ambos sentidos pero que el costo no sea el mismo. Por ejemplo si la arista representa una zona del terreno empinada, bajarla tal vez tenga un costo diferente a subirla. Para representar estos casos usamos un tipo especial de grafos denominados dígrafos (*digraphs*). En los mismos las aristas tiene una dirección o sentido. Y la

dirección de una arista se especifica por el orden en que se nombra la misma. En el ejemplo de arriba si la llamamos {1-2} quiere decir:



Mientras que {2-1} significa

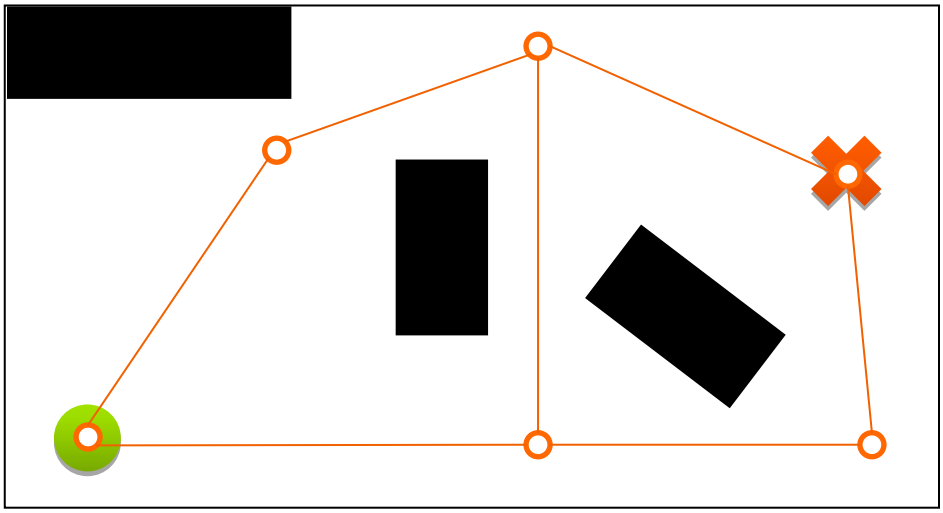


Incluso es posible representar grafos no orientados usando esta convención ya que si la arista se puede recorrer en ambos sentidos la ponemos dos veces en la lista {1-2, 2-1}.

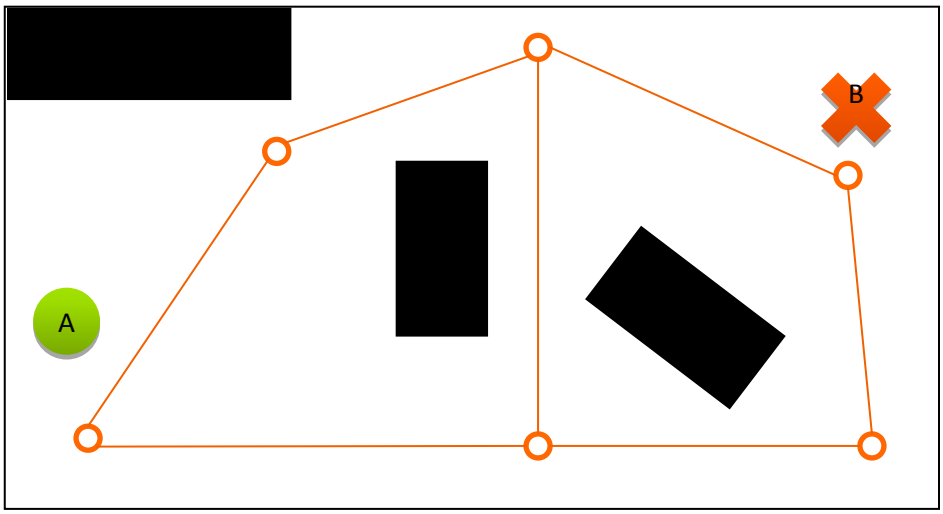
Usos de grafos en videojuegos

Navegación

El uso más común de un grafo en un videojuegos es para realizar la búsqueda espacial de caminos, es decir para la navegación del mapa. En estos casos los grafos se denominan grafos de navegación o *navgraphs*. Como vimos al principio cada nodo representa un área clave del mapa y las aristas las conexiones espaciales entre dichos puntos. Si volvemos al ejemplo del principio



podemos observar un navgraph del pequeño mapa que propusimos. Esto no quiere decir que el personaje solo se puede mover por las líneas del grafo como si fuera un tren sobre rieles. Generalmente para ir de un punto “A” a un punto “B” fuera del grafo como muestra la siguiente figura:

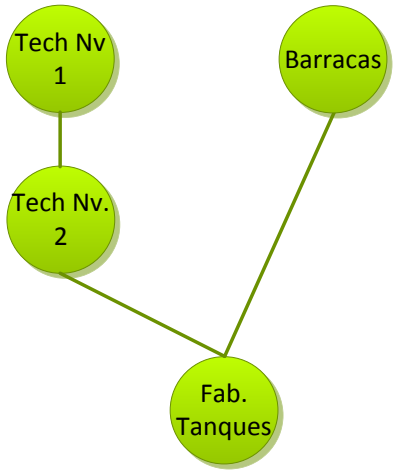


El personaje busca los nodos más cercanos a los puntos A y B y calcula el camino entre ellos. Esta claro que esta técnica supone que los nodos del grafo se encuentran bien ubicados para que el personaje pueda llegar siempre al nodo más cercano sin obstáculos. Armar los navgraphs suele ser una tarea bastante trabajosa y requiere mucha experiencia para lograr que el movimiento en el nivel sea fluido y sin problemas.

Otra representación espacial utilizada para la navegación especialmente en juegos de estrategia es dividir el escenario en cuadraditos formando una grilla regular. En esta grilla cada centro de celda es un nodo del árbol. Por ejemplo cada celda de la grilla es un tipo de terreno (agua, tierra, césped, montaña, etc) y las aristas que unen las celdas indican el costo debido al tipo de terreno.

Grafos de dependencia

En los juegos dónde debemos manejar recursos se utilizan grafos para indicar las dependencias entre los distintos edificios que podemos construir, los materiales, tecnología, etc. Esto se ve principalmente en los juegos de estrategia, dónde por ejemplo para poder construir una fábrica de tanques primero necesitamos barracas y tecnología de nivel 2. Pero para tener Tecnología de nivel 2 necesitamos primero el nivel 1.

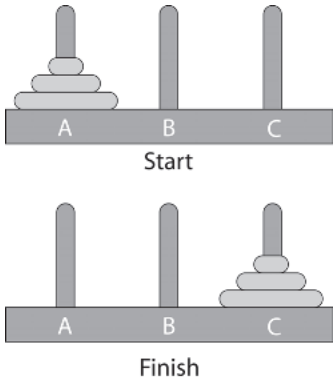


De esta forma es sencillo determinar las dependencias para la IA. Por ejemplo si la IA quiere construir Tanques debe examinar el grafo y determinar los prerrequisitos. De esta forma puede organizar un plan de construcción que la lleve a poder fabricar tanques.

También es útil este tipo de grafos para inferir que construcciones tiene el enemigo a partir de haber visto una unidad. Por ejemplo si la IA observa que se mete en su territorio un tanque del jugador, puede ir al grafo y "descubrir" que si el jugador tiene un tanque entonces quiere decir que ya construyó barracas y tiene tecnología del nivel 2.

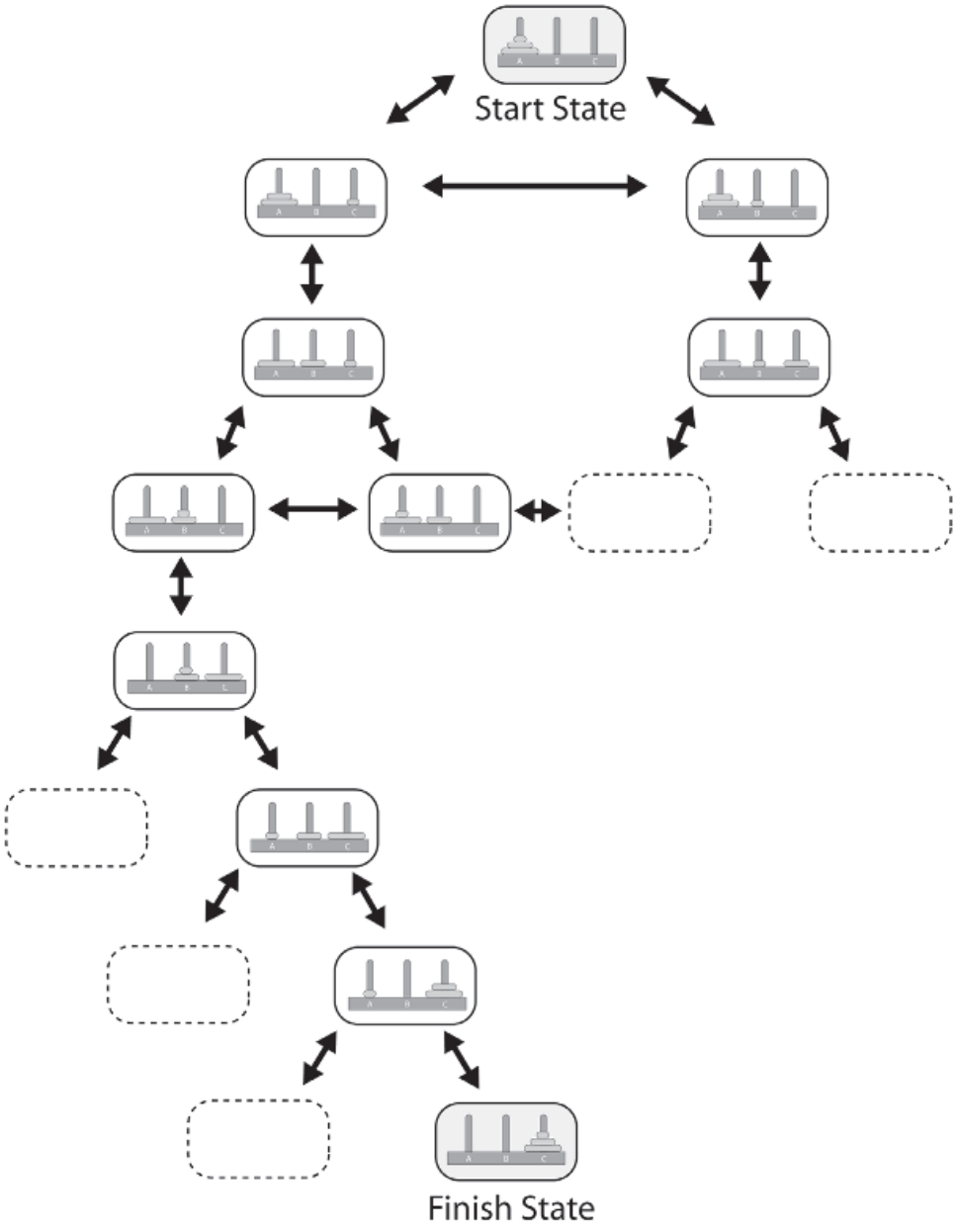
Grafos de estado

Los grafos se pueden utilizar para representar todos los posibles estados en los cual puede estar un sistema. Por ejemplo si utilizamos el puzzle conocido como la torre de Hanoi:



El cual consiste en tres torres en las cuales podemos colocar tres anillos de distinto tamaño. El objetivo es partir de la configuración dónde todos los anillos se encuentran en la primera torre y llegar a la configuración dónde todos los anillos se encuentran en la última como se muestra en la figura anterior. Sólo se puede mover de a un anillo por vez y un anillo sólo puede ser colocado en una torre vacía o sobre otro anillo más grande.

Es posible representar este puzzle como un grafo dónde cada nodo representa uno de los posibles estados que puede atravesar el juego. Y estos nodos están unidos si es posible pasar del estado de uno al estado de otro con sólo un movimiento. Para construir este grafo se comienza colocando el primer nodo o nodo raíz el cual contiene todos los anillos en la primer torre. Luego se "expanden" como nodos hijos todos los estados que se pueden alcanzar con solo un movimiento a partir del estado padre. Este proceso se repite hasta que no queden estados posibles. A continuación se muestra un ejemplo de dicho grafo dónde sólo se han expandido algunos nodos para simplificar el dibujo. Los nodos punteados no han sido expandidos.



Representar el problema como un grafo permite convertir el problema de buscar una solución al puzzle en una simple búsqueda en un grafo, hasta encontrar el nodo que posee el estado deseado. Para poder hacer esto expandimos todos los posibles nodos. Si el nodo tiene un promedio de hijos llamado *branching factor* bajo entonces es posible hacerlo, pero si cada nodo posee muchos hijos se torna computacionalmente imposible expandirlo entero, por lo que se necesita buscar algoritmos que no expandan todo el árbol.

Algoritmos de búsqueda

La teoría de grafos ha sido un área popular de estudio por parte de los matemáticos debido a la gran cantidad de problemas que se pueden resolver mediante búsquedas. Es por ello que existen numerosos algoritmos. En el curso veremos los más populares y especialmente aquél conocido como A*(se lee A Star) que nos brinda soluciones óptimas bajo ciertas condiciones.

En general los algoritmos de búsqueda nos permiten realizar las siguientes tareas:

- Visitar todos los nodos de un grafo
- Encontrar algún camino entre dos nodos. Estos es útil si sólo nos importa llegar al nodo objetivo y no es un problema el costo asociado a hacerlo. En el caso de la torre de Hanoi, con encontrar una solución nos alcanza, no necesitamos buscar el camino más corto.
- Encontrar el **mejor** camino entre dos nodos. El mejor camino dependerá del tipo de problema. Si estamos trabajando con un navgraph entonces el mejor camino podría ser
    - El más corto entre los dos nodos.
    - El que nos lleva menos tiempo recorrer. Por ejemplo si hay zonas del terreno en la cual nuestra unidad demora en pasar (Pantanos, montañas, etc)
    - El que nos permita evadir la línea de visión de los enemigos
    - El que nos lleve a hacer menos ruido, etc.

Los algoritmos los podemos clasificar en dos clases: algoritmos ciegos (del inglés *blind*) o desinformados (*uninformed)* y algoritmos de relajación de aristas (del inglés edge relaxation).

Búsquedas Desinformadas

Este tipo de algoritmos buscan en un grafo sin costos asociados a las aristas. Pueden distinguir entre nodos y aristas individuales, permitiéndoles distinguir si llego al objetivo o si ya paso por un nodo dado.

Depth First Search (DFS)

Este algoritmo como lo dice su nombre (*Busqueda primero en profundida*d) siempre explora primero el nodo más profundo encontrado hasta el momento hasta llegar a un nodo que no tenga más aristas salientes. Cuando esto sucede retrocede y elije el próximo nodo más profundo que tenga aristas salientes. Esto lo repite hasta que no queden más nodos sin explorar.

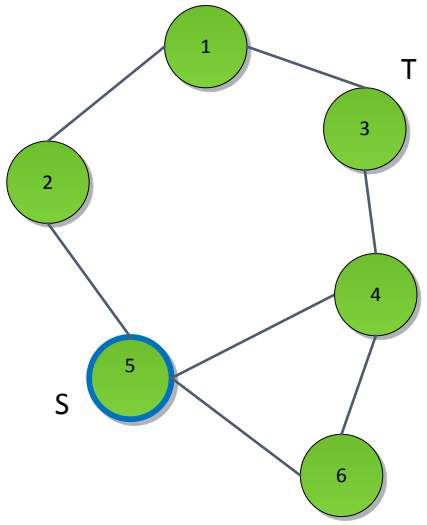
Veamos ahora en más detalle como funciona el algoritmo. En primer lugar vamos a definir un conjunto de nodos que llamaremos *frontera de búsqueda*. Esta frontera contendrá aquellos nodos que son candidatos a ser explorados en la próxima iteración. Luego definamos otro conjunto al cual llamaremos *nodos explorados*, el cual contiene todos los nodos que ya han sido visitados por el algoritmo. Además sea *S* el nodo de inicio y *T* el nodo buscado. Al principio *S* es el único nodo que se encuentra en la frontera de búsqueda. El pseudocódigo del algoritmo es de la siguiente forma:

1. Sacamos el nodo **n** de ***mayor profundidad*** de la frontera de búsqueda
    a. Si el nodo **n** es el buscado termina el algoritmo.

2. Por cada arista saliente de **n**:
    a. Recorrer arista hasta el nodo destino **nd**
    b. Agregar el nodo **nd** a la frontera de búsqueda si es que no estaba ya allí y si este no estaba entre los nodos expandidos.

3. Metemos el nodo **n** a la lista de nodos explorados
4. Volvemos al paso 1

Este algoritmo nos garantiza que todos los nodos serán explorados si el grafo es conectado. Veamos ahora un ejemplo de aplicación del algoritmo. Supongamos que tenemos el siguiente grafo a buscar:

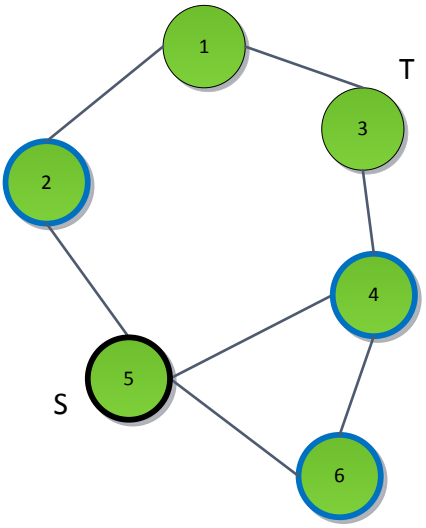


Dónde debemos comenzar por el nodo 5. Tenemos como destino el nodo 3. Los nodos sin borde resaltado son nodos no explorados. Los nodos con un borde azul son nodos pertenecientes a la frontera de búsqueda, los nodos negros son nodos ya visitados.

Primer paso

1. Sacamos el nodo **n** de mayor profundidad de la frontera de búsqueda: El único nodo que tenemos en la frontera es S (5). Y es distinto a T.
2. Recorremos las aristas salientes de 5 y agregamos los nodos a los que se puede llegar (2,4,6).
3. Metemos el nodo 5 a la lista de nodos explorados.

Finalmente el estado de la búsqueda en el primer paso es:

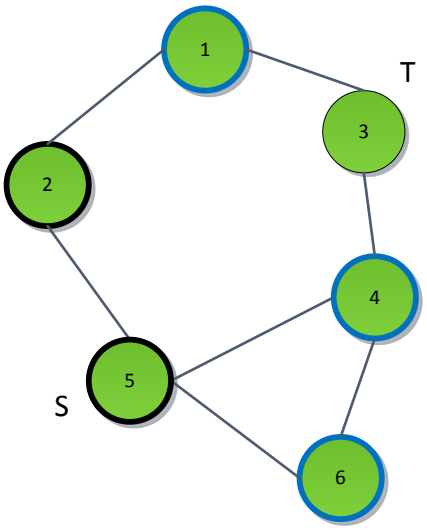


Aquí se puede observar como la frontera de búsqueda está conformada por los nodos a los cuales se podía llegar partiendo de 5. Además se observa que 5 se marcó como visitado.

Segundo paso

1. Sacamos el nodo **n** de mayor profundidad de la frontera de búsqueda. La profundidad se toma como si 5 es la raíz. En este caso los nodos 2,4 y 5 se encuentran a la misma profundidad. Por lo que se utiliza algún criterio arbitrario para elegir, (puede ser el mas izquierdo en caso de arboles, etc). Elegimos el 2.
2. Recorremos las aristas salientes de 2 y agregamos los nodos a los que se puede llegar (1) a la frontera de búsqueda.
3. Metemos el nodo 2 a la lista de nodos explorados.

Finalmente el grafo ahora queda:

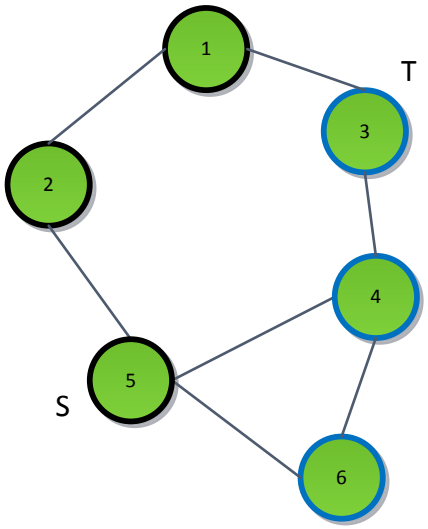


En este caso la frontera de búsqueda está compuesta por los nodos 1,4 y 6. Y el nodo 2 paso al grupo de los explorados.

Tercer paso

1. Sacamos el nodo **n** de mayor profundidad de la frontera de búsqueda. En este caso el nodo 1 tiene una profundidad mayor (2) que los nodos 6 y 4. Por lo que elegimos el nodo 1 para examinar. Este tampoco es el nodo buscado.
2. Recorremos las aristas salientes de 1 y agregamos los nodos a los que se puede llegar (3) a la frontera de búsqueda. Este es el nodo destino, pero todavía no podemos parar, ya que según el algoritmo debemos parar en el paso 1, es decir cuando lo elegimos para examinar.
3. Metemos el nodo 1 a la lista de nodos explorados.

Ahora se puede observar como en la frontera de búsqueda tenemos los nodos 3, 4 y 6. Y en particular 3 es el nodo al cual queremos llegar!.



Cuarto paso

1. Sacamos el nodo **n** de mayor profundidad de la frontera de búsqueda. En este caso el de mayor profundidad es el nodo 3. Esto es cierto porque llegamos a 3 por el camino de aristas {5-2,2-1,1-3}. Si hubiéramos ido por otro lado la profundidad hubiera sido distinta. Como **n**==T entonces termina el algoritmo habiendo encontrado el objetivo.

Este algoritmo es uno de los más sencillos y tiene algunos inconvenientes. Si tenemos grafos muy profundos entonces el algoritmo DFS puede perder mucho tiempo bajando por un camino incorrecto. Incluso en algunos casos puede ser que el problema sea tan grande que una mala elección en las primeras iteraciones lleve a que el problema sea imposible de resolver, debido al costo inmenso de almacenamiento.

También es necesario destacar que DFS nos encuentra una ruta al objetivo, pero no nos encuentra la ruta más corta. A continuación veremos otro algoritmo muy parecido pero que sí devuelve la ruta más corta.

Breadth First Search (BFS)

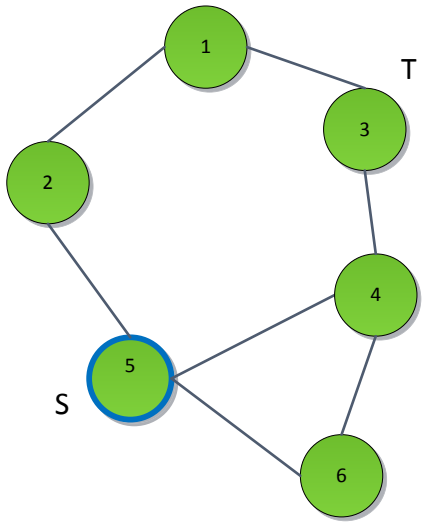
Este algoritmos es muy parecido a DFS, lo que lo diferencia es cómo elige el próximo nodo de la frontera a explorar. En este caso no se elige el nodo más profundo, sino que se toma el nodo menos profundo. Esto provoca que el algoritmo primero analice todos los nodos de profundidad 1, luego los de profundidad 2 y así sucesivamente hasta encontrar el objetivo. Debido a esto una vez que encuentra el nodo, se garantiza que contiene la menor cantidad de aristas posibles (pueden haber otros con la misma cantidad, pero no con menos). A continuación mostraremos el algoritmo:

1. Sacamos el nodo **n** de ***menor profundidad*** de la frontera de búsqueda.
    a. Si el nodo **n** es el buscado termina el algoritmo.
2. Por cada arista saliente de **n**:
    a. Recorrer arista hasta el nodo destino **nd**
    b. Agregar el nodo **nd** a la frontera de búsqueda si es que no estaba ya allí y si este no estaba entre los nodos expandidos.

3. Metemos el nodo **n** a la lista de nodos explorados
4. Volvemos al paso 1

Como se puede observar el algoritmo es idéntico, como se dijo anteriormente. A continuación veamos cómo se comporta utilizando el mismo ejemplo anterior.

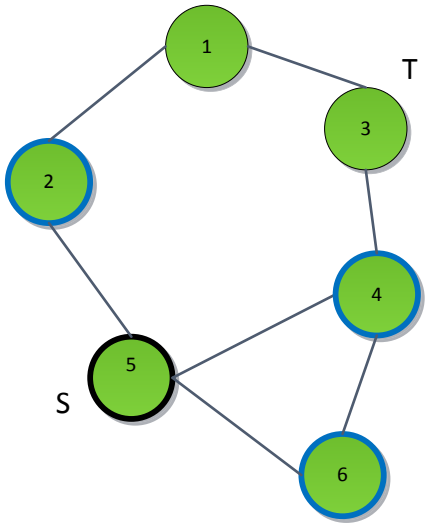
Al comienzo sólo el nodo 5 forma parte de la frontera de búsqueda.



Primer paso

1. Sacamos el nodo **n** de menor de la frontera de búsqueda: El único nodo que tenemos en la frontera es S (5). Y es distinto a T.
2. Recorremos las aristas salientes de 5 y agregamos los nodos a los que se puede llegar (2,4,6).
3. Metemos el nodo 5 a la lista de nodos explorados.
4. Volvemos al paso 1.

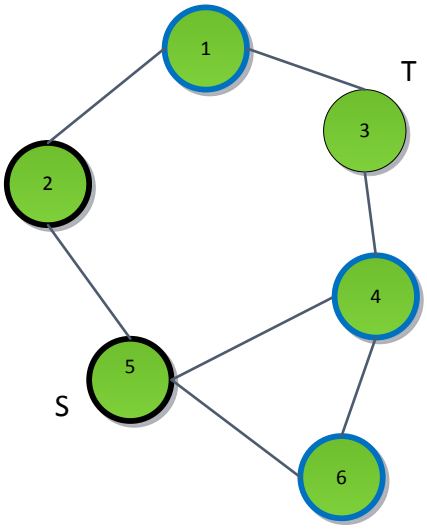
Hasta aquí el algoritmo se comporta igual que DFS y nos da los conjuntos:



Segundo paso

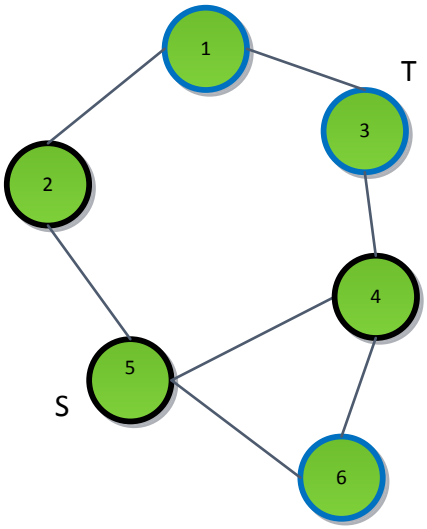
1. Al igual que en DFS tenemos varios nodos que cumplen el criterio, por lo que elegimos arbitrariamente el 2 al igual que anteriormente. Este nodo es distinto a T.
2. Agregamos 1 a la frontera de búsqueda.
3. Metemos 2 a la lista de explorados.

En este paso ahora tenemos la nueva frontera la cual es idéntica a DFS:



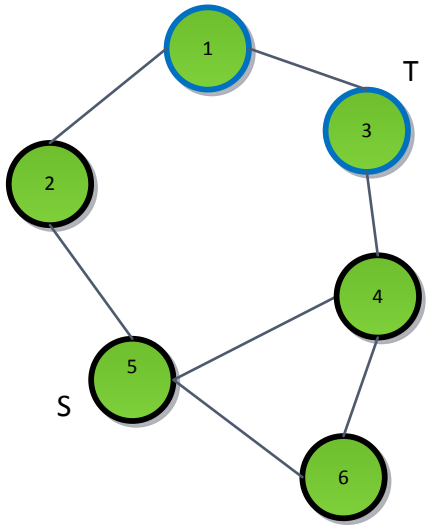
Tercer paso

1. En este paso BFS comienza a diferir de DFS. Los nodos más cercanos en este caso son 4 y 6. Elegimos 4, el cual no es el nodo buscado
2. Agregamos el nodo 3 a la frontera.
3. Marcamos 4 como visitado.



Cuarto paso

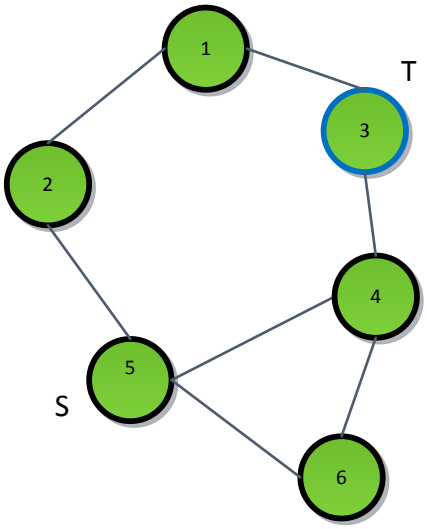
1. El nodo menos profundo es el 6, el cual tampoco es el de búsqueda.
2. No tiene aristas hacia nodos no visitados, por lo que no se agrega ningún nodo.
3. Se mete 6 a la lista de visitados.



Es importante notar que 4 no se mete ya que lo habíamos encontrado antes por un camino más corto.

Quinto paso

1. El nodo 1 y 3 están a la misma distancia. Así que tomamos el 1 que fue el primero que agregamos. El nodo 1 no es el nodo buscado.
2. El nodo 3 ya está en la frontera, así que no lo agregamos (lo había metido 4)
3. Marcamos 1 como visitado.



Sexto paso

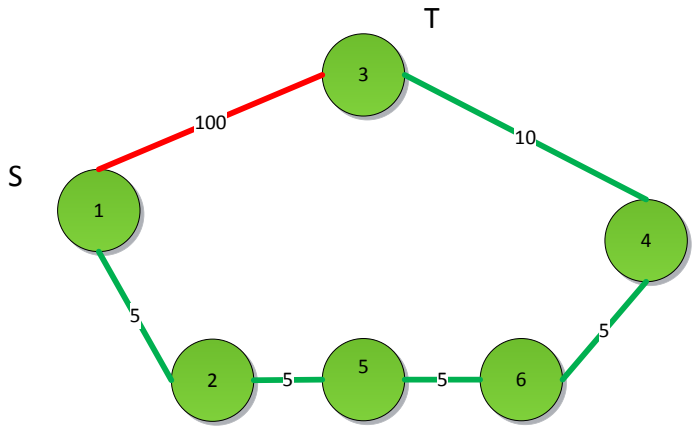
1. Sacamos de la frontera a 3 y vemos que es el nodo buscado. El algoritmo termina.

El camino encontrado es el {5-4,4-3} que a diferencia de DFS si es óptimo. En este caso el nodo 3 quiso ser agregado por dos caminos distintos. Cuando esto sucede sólo debemos agregarlo la primera vez. Cada nodo sabe cuál es el camino óptimo anterior hasta llegar a él. Este es un dato

que se debe ir almacenando en el nodo cada vez que lo metemos en la frontera. De esta forma una vez llegado al objetivo podemos reconstruir dicho camino. Si hubiésemos agregado el 3 nodo reescribiéndolo (pero esta vez con un camino más largo que el que tenía) el algoritmo nos hubiera dado una solución no óptima. Hay que tener cuidado con esto.

Búsquedas basadas en costos

Como se vio anteriormente, es común que en muchos problemas las aristas tengan un costo asociado. A diferencia de los algoritmos vistos hasta ahora (DFS y BFS) no basta con contar la cantidad de saltos sino que ahora el algoritmo debe buscar minimizar los costos de las aristas que recorre. Es común que a veces el camino con menos aristas sea el más costoso. Por ejemplo si tenemos el siguiente problema:



En este caso la ruta más barata entre 1 y 3 no es {1-3} recorriendo sólo una arista sino que es {1-2,2-5,5-6,6-4,4-3} recorriendo varias aristas, pero cuyo costo sumado no alcanza al costo de recorrer la arista {1-3}.

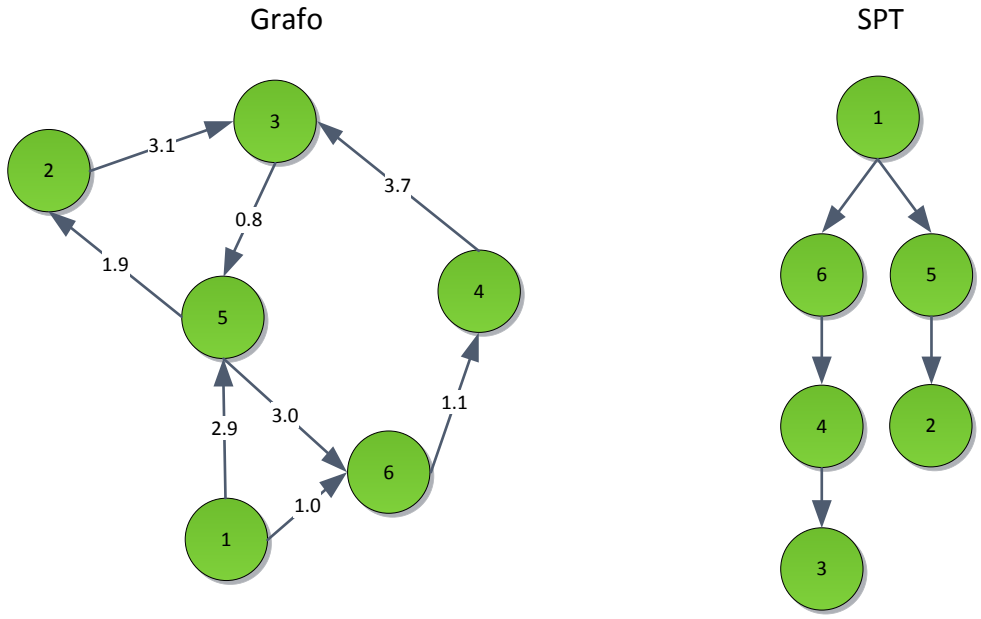
Relajación de aristas (Edge relaxation)

Los algoritmos que veremos a continuación se basan en una técnica conocida como *edge relaxation*. A medida que el algoritmo avanza el mismo recopila información sobre el camino más corto del nodo raíz a todos los otros nodos camino al objetivo. Esta información se va actualizando a medida que el algoritmo avanza. Esto quiere decir que si llegamos a un nodo ya visitado anteriormente pero el camino actual es mejor entonces actualizamos la información de dicho nodo con el nuevo camino. En este algoritmo cada nodo almacena el mejor costo encontrado hasta el momento y cuál fue el camino tomado hasta él para obtenerlo.

Árboles de camino mínimo (Shortest Path Tree -SPT-)

Dado un grafo G, un SPT es el subárbol de G que representa el camino más corto de cualquier nodo hacia la raíz.

Por ejemplo:



En la figura se puede observar el grafo a la izquierda y el árbol extraído a la derecha que se construyó con los caminos mínimos tomando al nodo 1 como raíz. Si desean puede verificar que efectivamente el árbol indica el camino más corto entre la raíz y cualquier otro nodo. El problema está en cómo construir dicho árbol de manera eficiente. A continuación veremos unos algoritmos que nos darán respuesta a esto.

Algoritmo de Dijsktra

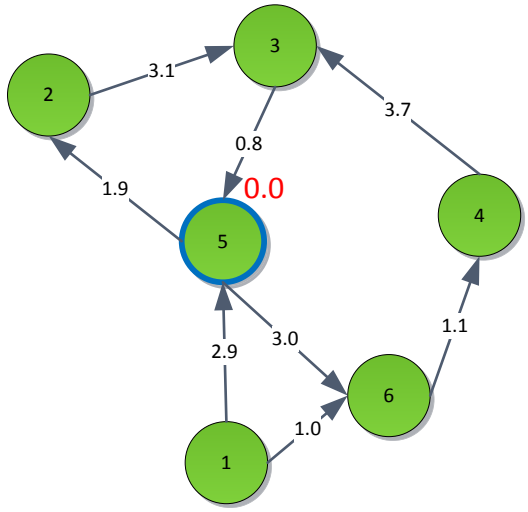
El algoritmo de Dijkstra construye un SPT examinando de a una arista a la vez. El mismo comienza agregando el nodo de origen como raíz del árbol y luego va agregando los nodos que se encuentren más cerca de la raíz y que no hayan sido agregados al árbol aún.

Si se da un nodo objetivo entonces el algoritmo termina cuando lo encuentra. Si se lo deja seguir hasta examinar el último nodo entonces el algoritmo de Dijkstra calcula el STP completo del grafo.

El algoritmo es el siguiente:

1. Se elije el nodo **n** de la frontera de búsqueda cuya distancia acumulada a la raíz sea menor.
2. Se recorren las aristas salientes de **n** y por cada nodo de destino **nd**:
    a. Se calcula el costo hasta **nd** como C(n)+C(n->d) es decir el costo que había hasta el nodo **n** más el costo de la arista de **n** a **nd**.
    b. Si **nd** no tiene costo asociado se agrega a la frontera de búsqueda junto con su costo.
    c. Si **nd** ya fue visitado pero el costo nuevo es menor que el anterior entonces se modifica el costo hasta **nd** que se encuentra en la frontera de búsqueda.
    d. Si **nd** ya fue agregado al SPT se ignora.
3. Se agrega el nodo **n** al SPT.
4. Volver al paso 1

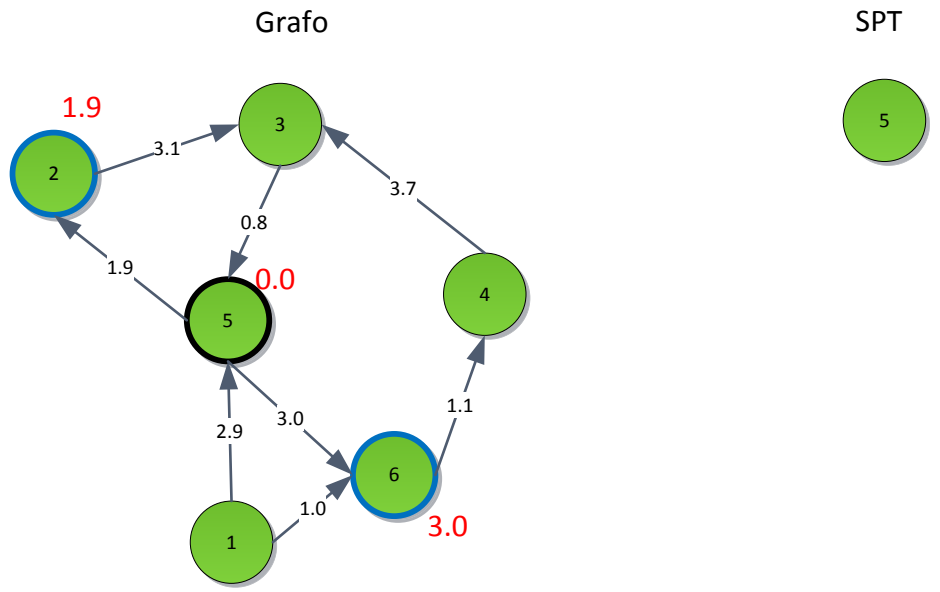
Como se puede observar el algoritmo es muy parecido a DFS y BFS solo cambia el criterio de selección del próximo nodo en el paso 1. En este caso se tiene en cuenta el costo total acumulado hasta dicho nodo. Veamos ahora un ejemplo de cómo funciona el algoritmo. Utilizando el mismo grafo que vimos como ejemplo:



Supongamos que comenzamos del nodo 5, que tiene un costo inicial de 0. En rojo los números indican los costos hasta los nodos.

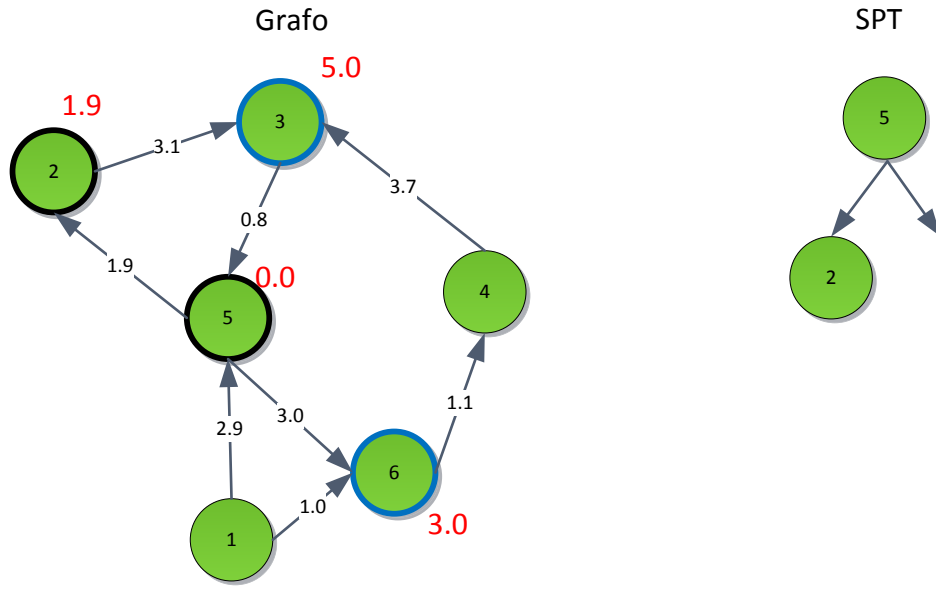
Primer paso

1. La frontera solo contiene al nodo 5. Por lo cual lo elegimos.
2. Se agregan los nodos a los cuales apunta 5, es decir 2 y 6. El costo de 2 es el costo de 5 más el costo de la arista 5-2, es decir C(2)=C(5)+C(5->2)=1.9. Por otro lado C(6)=C(5)+C(5->6)=3.0
3. Se agrega el nodo 5 al SPT.



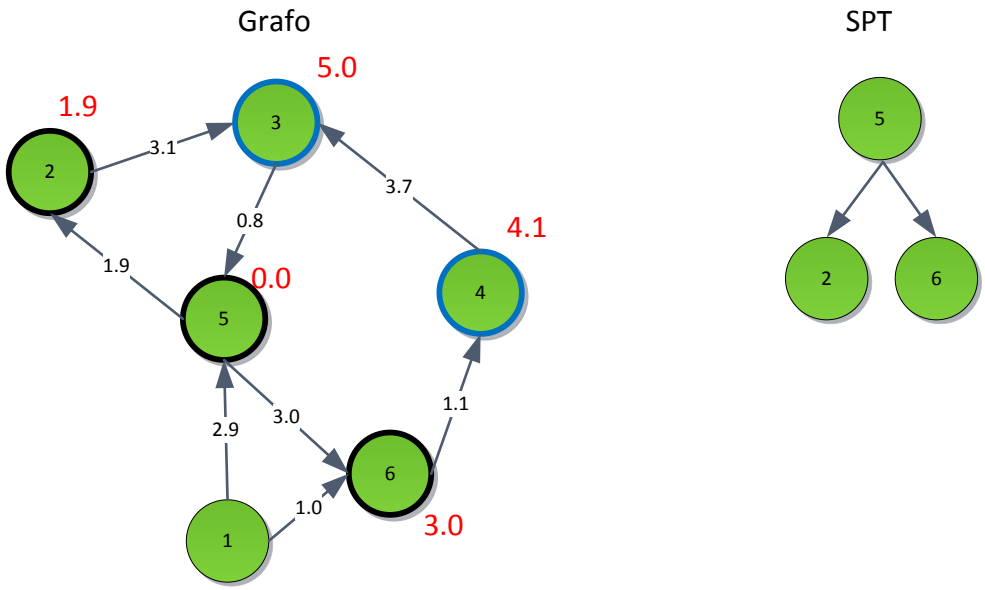
Segundo paso

1. Elegimos de la frontera de búsqueda el nodo más “cercano”. Es 2 ya que tiene un costo de 1.9 contra 3.0 del nodo 6.
2. Se agrega el nodo 3 con C(3)=C(2)+C(2->3)=1.9+3.1=5.0
3. Se agrega el nodo 2 al SPT.



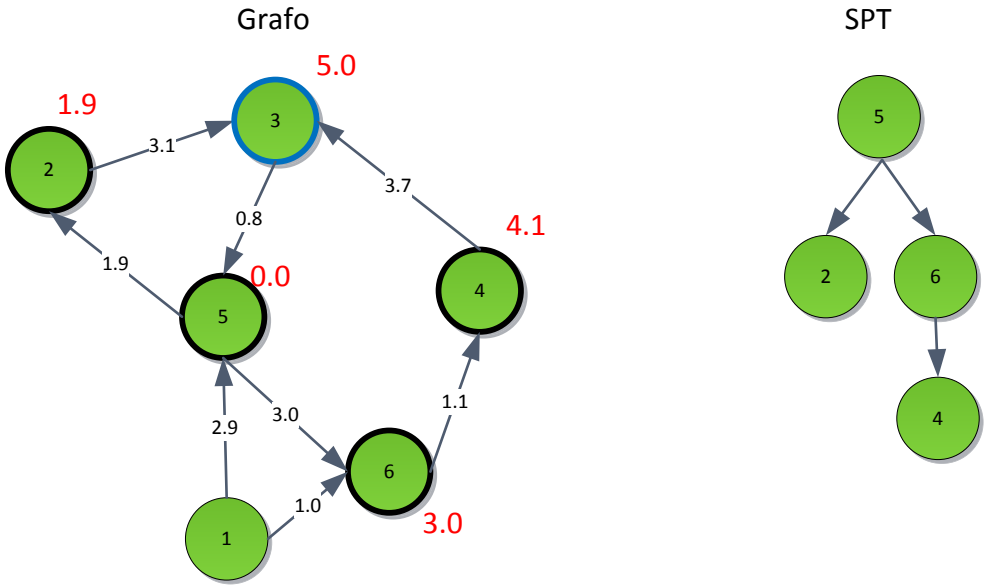
Tercer paso

1. En la frontera de búsqueda tenemos los nodos 3 con C(3)=5.0 y 6 con C(6)=3.0. Elegimos el 6 que es quien tiene asociado el costo menor.
2. El 6 sólo se conecta con el 4, por lo que se calcula C(4)=C(6)+C(6->4)=3.0+1.1=4.1 y se agrega el nodo a la frontera de búsqueda
3. Se agrega el nodo 6 al SPT.

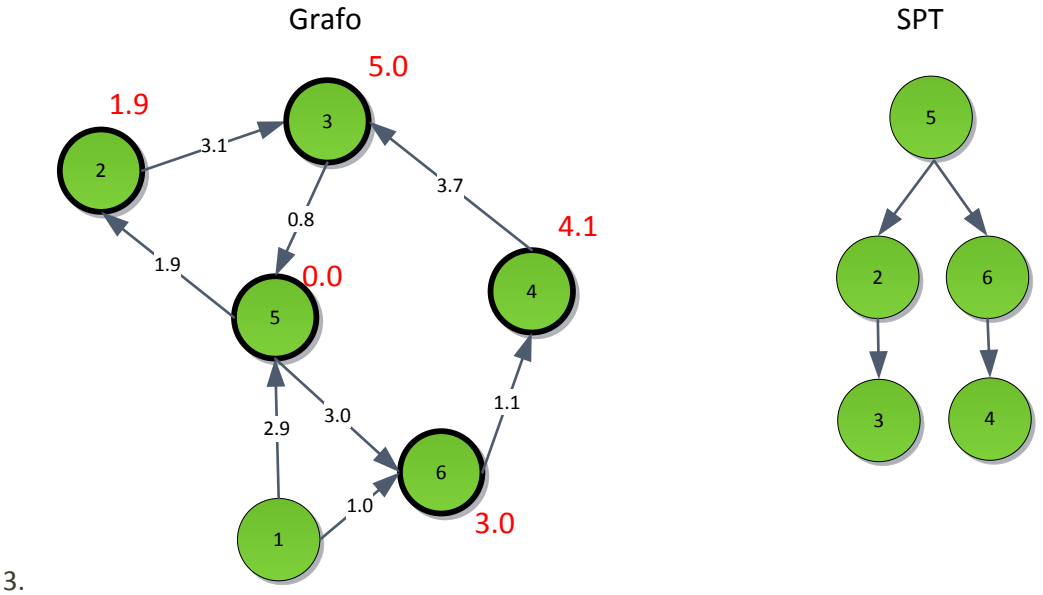


Tercer paso

1. El nodo con menor costo es el 4. Por lo que lo elegimos
2. El 4 conecta con el 3 que ya fue agregado a la frontera. Pero debemos ver si el camino actual no es mejor que el que tenía. Para ello calculamos el costo C'(3)=C(4)+C(4->3)=4.1+3.7=7.8. Como se puede observar este costo es mayor que el que habíamos encontrado anteriormente por lo que no modificamos el costo del nodo 3. Si hubiera sido menor, entonces si debíamos modificarlo.
3. Se agrega el nodo 4 al SPT.



Cuarto paso

1. El único nodo en la frontera es el nodo 3.
2. El 3 sólo conecta con el nodo 5, y como el nodo 5 ya fue marcado como visitado, no se agrega a la frontera nuevamente.
3. 

El algoritmo termino debido a que no quedan más nodos en la frontera para examinar. En este ejemplo dejamos que el algoritmo construya el SPT entero. Si sólo deseábamos buscar un nodo entonces en el paso 1 del algoritmo deberíamos preguntar si es el nodo que buscamos, y en caso afirmativo cortar el proceso.

Este algoritmo garantiza encontrar el camino más corto pero debe examinar una gran cantidad de nodos en el camino. Existe una alternativa mejor que veremos a continuación.

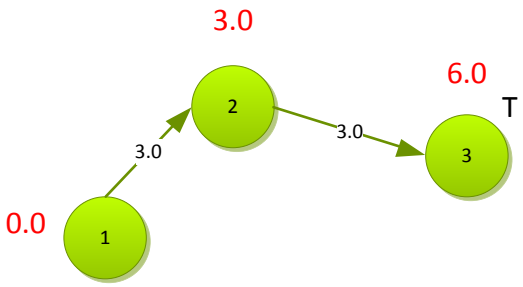
A: Una vuelta de tuerca a Dijsktra*

El algoritmo de Dijkstra como vimos anteriormente trata de buscar el mejor camino minimizando el costo de las aristas que atraviesa. Ahora vamos a ver una mejor alternativa conocida como A*(A-Star) . Este algoritmo es muy parecido al algoritmo de Dijsktra, la única diferencia radica en cómo estima el costo de un nodo antes de ponerlo en la frontera de búsqueda. Anteriormente habíamos visto que el costo de un nodo, era la suma de los costos de las aristas para llegar, y a esta función la llamábamos C. Ahora vamos a definir una nueva función F. Esta nueva función es el costo que usaremos en A* y se define como la suma de 2 funciones:

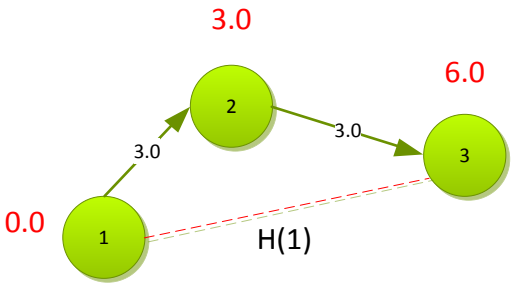
$$F(n) = C(n) + H(n)$$

Al costo tradicional C(n) ahora le vamos a sumar una función H. Está función se suele llamar heurística y nos da una estimación de la distancia del nodo n hasta el objetivo. Está claro que no conocemos el costo real de cada nodo al objetivo, es por ello que decimos que H es una estimación del costo real.

La función H puede tomar distintas formas, sólo debe cumplir que sea una subestimación del costo real, es decir estime costos menores que los reales. Es decir que H es una estimación optimista del costo. Cualquier función que satisfaga esto, es apta para ser usada. Por ejemplo si tenemos el siguiente grafo dónde el costo de las aristas es la distancia entre los nodos y deseamos ir del nodo 1 al 3.



En rojo están marcados los valores de la función C. Ahora veamos cómo definir H. Tomemos primero el nodo 1. El costo de dicho nodo viene dado por $C(1) = 0.0 + H(1)$. En este caso la distancia real del nodo 1 al objetivo sería 6 (verifíquelo), sin embargo como nosotros estamos buscando una función para subestimar dicha distancia, entonces podría elegir como H la distancia euclídea[1] entre el nodo 1 y el nodo objetivo:



Como es bien sabido $H(1)$ será menor que tomar el camino a través del nodo 2. Luego la función de costo en el nodo 1 será:

$$C(1) = 0.0 + H(1) = 0.0 + distancia(1,3)$$

Con este costo meteremos el nodo en la frontera de búsqueda. El efecto de agregar la función H es hacer que la búsqueda sea dirigida hacia el lado del objetivo, evitando explorar nodos que van hacia otro lado, o se alejan del objetivo. Si hacemos H=0 entonces estamos recuperando el algoritmo de Dijkstra.

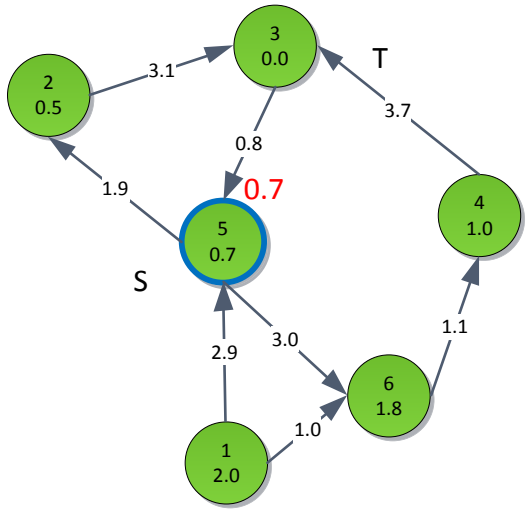
---

[1] $Distancia\ euclídea(\overline{v1}, \overline{v2}) = \sqrt{(v1_x - v2_x)^2 + \left(v1_y - v2_y\right)^2}$ Es decir que es la distancia que siempre usamos entre dos vectores.

El pseudocódigo será el siguiente:

1. Se elije el nodo **n** de la frontera de búsqueda cuyo costo F sea menor. Si **n** es el nodo destino terminamos la búsqueda.
2. Se recorren las aristas salientes de **n** y por cada nodo de destino **nd**:
    a. Se calcula el costo hasta **nd** como C(n)+C(n->d)+H(**nd**) es decir el costo que había hasta el nodo **n** más el costo de la arista de **n** a **nd** más la distancia estimada H de **nd** al objetivo.
    b. Si **nd** nunca fue visitado se agrega a la frontera de búsqueda junto con su costo.
    c. Si **nd** ya fue visitado pero el costo nuevo es menor que el anterior entonces se modifica el costo hasta **nd** que se encuentra en la frontera de búsqueda.
    d. Si **nd** ya fue agregado al SPT se ignora.
3. Se agrega el nodo **n** al SPT.
4. Volver al paso 1

Veamos ahora un ejemplo del algoritmo en acción. Para ello tomemos el mismo grafo del ejemplo anterior. En el mismo los costos de las aristas supongamos que representan el combustible necesario para ir de un nodo a otro. Asimismo en cada nodo se agregó el valor de H, en este caso consideramos que será el combustible consumido en ir entre el nodo y el destino de forma recta, es decir consumiendo la mínima cantidad de combustible. (Son números a modo de ejemplo).



Se desea ir del nodo marcado como inicio (S) al nodo marcado como objetivo (T). Nótese que el costo del nodo 1 será:

$$F(5) = C(5) + H(5) = 0.0 + 0.7 = 0.7$$

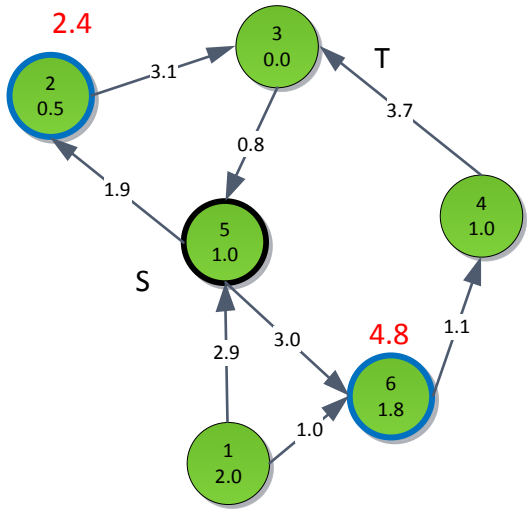
Primer paso

1. Tomamos el único nodo de la frontera (5), el mismo no es el nodo deseado aún.
2. El 5 se conecta con el 2 y el 6. Se agregan dichos nodos a la frontera con los costos:

a. $C(2) = C(5) + C(5 \rightarrow 2) + H(2) = 0.0 + 1.9 + 0.5 = 2.4$
b. $C(6) = C(5) + C(5 \rightarrow 6) + H(6) = 0.0 + 3.0 + 1.8 = 4.8$

3. Agregamos el nodo 5 al SPT, en particular como estamos buscando un nodo y no construir el árbol simplemente lo marcamos como ya agregado.

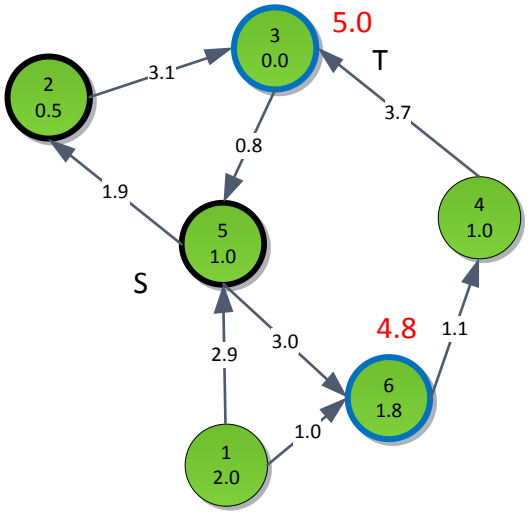
Finalmente se muestran en rojo los valores de F (Atención de F no de C):



Segundo paso

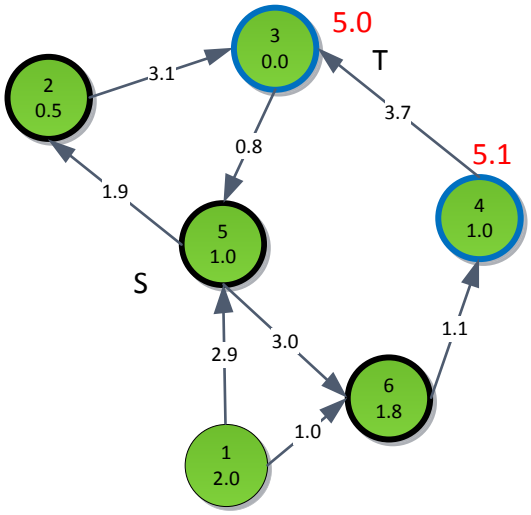
1. El nodo de la frontera con menor valor de F es el 2.
2. El 2 sólo se conecta con el 3, por lo que es agregado a la frontera con el costo:
   a. $C(3) = C(2) + C(2 \rightarrow 3) + H(3) = 1.9 + 3.1 + 0.0 = 5.0$. Es importante notar en esta ecuación dos cosas. En primer lugar el costo de 2 no es 2.4, sino que es el costo de las aristas atravesadas para llegar al mismo desde el origen sin tener en cuenta la heurística, es decir 1.9. Por otro lado como 3 es le nodo destino, la distancia del 3 a si mismo será 0.
3. Marcamos a 2 como ya visitado.

El grafo ahora nos queda:



Tercer paso

1. El próximo nodo que sacaremos de la frontera será el 6, ya que su F es la menor.
2. El 6 sólo se conecta con el nodo 4, por lo que lo agregamos a la frontera con el costo:
   a. $C(4) = C(6) + C(6 \rightarrow 4) + H(4) = 3.0 + 1.1 + 1.0 = 5.1$
3. Se marca el nodo 6 como visitado.



Tercer paso

1. Tomamos el nodo 3 que es aquél cuya función de costo es menor. Como es el nodo buscado entonces corta el algoritmo.

Por supuesto en este caso al igual que anteriormente en cada nodo almacenamos cuál es el mejor camino al mismo, para poder reconstruir el camino una vez hayamos llegado al objetivo.

Implementación

Las implementaciones de estos algoritmos de manera eficiente a veces pueden ser un poco tediosas. Es por este motivo que no vamos a detenernos sobre las mismas, si alguien lo desea puede usar el material de referencia donde se encuentran explicados en mayor detalle.

En la plataforma se encuentra subido un ejemplo el cuál usaremos para la práctica. En el ejemplo se carga un escenario y permite al usuario modificar los nodos del grafo de navegación y el punto de inicio y fin del personaje. Una vez seteado permite ejecutar el algoritmo y ver como se mueve el personaje por el escenario.

Para representar un grafo utilizaremos la clase SparseGraph. Su definición es la siguiente:

```
template <class node_type, class edge_type>
class SparseGraph
{
public:
	//Permitir acceso rapido a los tipos de edges y nodos
	typedef edge_type EdgeType;
	typedef node_type NodeType;
	//typedefs por comodidad
	typedef std::vector<node_type> NodeVector;
	typedef std::list<edge_type> EdgeList;
	typedef std::vector<EdgeList> EdgeListVector;
private:
	//la lista de nodos
	NodeVector m_Nodes;
	//un vector donde cada posicion es
	//una lista de aristas adyacentes
	EdgeListVector m_Edges;
	//indica si es un grafo dirigido
	bool m_bDigraph;

	//El indice del proximo nodo a agregar
	int m_iNextNodeIndex;
public:
	//ctor
	SparseGraph(bool digraph): m_iNextNodeIndex(0), m_bDigraph(digraph){}
	//Devuelve un nodo dado
	const NodeType& GetNode(int idx)const;
	//Devuelve la arista que une from con to
	const EdgeType& GetEdge(int from, int to)const;
	//devuelve el indice del proximo nodo libre
	int GetNextFreeNodeIndex()const;
	//agrega un nodo al grafo
	int AddNode(NodeType node);
	//remueve un nodo
	void RemoveNode(int node);
	//Agrega una arista
	void AddEdge(EdgeType edge);
	//remueve una arista
	void RemoveEdge(int from, int to);
	//devuelve la cantidad de nodos
	int NumNodes()const;
```

```
      //devuelve la cantidad de aristas
      int NumEdges()const;
      //indica si es un grafo dirigido
      bool isDigraph()const;
      //devuelve verdadero si el grafo esta vacio
      bool isEmpty()const;
      //devuelve verdadero si existe el nodo nd
      bool isPresent(int nd)const;
      //Limpia el grafo
      void Clear();
      //Iteradores para poder recorrerlo
      class ConstEdgeIterator;
      class EdgeIterator;
      class NodeIterator;
      class ConstNodeIterator;
};
```

Como se puede observar la clase está templatizada por el tipo de nodo y el tipo de aristas. Esto es para poder tener grafos generales. Luego se puede diseñar el tipo de nodo y la información de la arista y esta clase funcionara perfectamente. También se puede observar que contiene métodos para agregar nodos, agregar aristas, quitar aristas, etc.

Por otro lado tenemos una definición genérica de nodos para ser usada con la clase grafo:

```
class GraphNode
{
protected:

  //Todo nodo tiene un indice
  int        m_iIndex;

public:

  GraphNode():m_iIndex(invalid_node_index){}
  GraphNode(int idx):m_iIndex(idx){}
  virtual ~GraphNode(){}

  int  Index()const{return m_iIndex;}
  void SetIndex(int NewIndex){m_iIndex = NewIndex;}

 …

};
```

Como se puede observar esta clase solo contiene información de índice, pero como vimos anteriormente si queremos crear un navgraph, entonces necesitaremos que los nodos además contengan información espacial, para ello creamos un nuevo tipo de nodo que hereda del anterior:

```
template <class extra_info = void*>
class NavGraphNode : public GraphNode
```

```
{
protected:

  //La posicion del nodo
  Vector2D     m_vPosition;

public:

  Vector2D   Pos()const
  void       SetPos(Vector2D NewPosition)
…
};
```

Este nodo posee la misma información que el genérico y además un vector en dos dimensiones para almacenar la posición.

También vamos a necesitar una clase genérica para aristas:

```
class GraphEdge
{
protected:

  //indices de los nodos que conecta
  int     m_iFrom;
  int     m_iTo;

  //el costo de la arista
  double  m_dCost;

}
```

En la misma almacenaremos el costo y los nodos que conecta. Si vamos a trabajar con un navgraph y deseamos que las aristas tengan información por ejemplo sobre el tipo de terreno que atraviesan entonces podríamos crear una nueva arista:

```
class NavGraphEdge : public GraphEdge
{
public:

  //ejemplos de terrenos
  enum
  {
    normal            = 0,
    swim              = 1 << 0,
    crawl             = 1 << 1,
    creep             = 1 << 3,
    jump              = 1 << 3,
    fly               = 1 << 4,
    grapple           = 1 << 5,
    goes_through_door = 1 << 6
  };
…

}
```

La cual contiene un enum que indica el tipo de terreno.

En el ejemplo de la plataforma pueden ver como se utilizan estas estructuras y jugar un poco con las mismas. Finalmente vamos a ver cómo se implementan los algoritmos que hemos visto. En particular veremos la estructura de DFS, ya que son todos muy parecidos entre sí. Como los grafos pueden variar su tipo, entonces vamos a declarar una clase templatizada que lleve adelante el algoritmo. La misma recibirá el grafo como argumento, realizará la búsqueda y devolverá un camino:

```
template<class graph_type>
class Graph_SearchDFS
{
private:

  enum {visited, unvisited, no_parent_assigned};


private:

  //posee una referencia al grafo que utilizaremos
  const graph_type& m_Graph;

  //guarda los indices de los visitados
  std::vector<int>  m_Visited;

  //Aqui guardamos la ruta encontrada
  std::vector<int>  m_Route;

  //los nodos de origen y destino
  int               m_iSource,
                    m_iTarget;

  //verdadero si se encontro un camino
  bool              m_bFound;


  //este metodo realiza la busqueda
  bool Search();

  //devuelve un vector con el camino encontrado
  std::list<int> GetPathToTarget()const;
};
```

Como se puede observar contiene métodos para calcular el camino y devolverlo. Los algoritmos son un poco complejos por lo que no es obligación que los miren, pero si sería interesante que intenten hacerlo.

El único algoritmo que difiere un poco es A* al cual debemos pasarle además cuál será la función heurística (H). Por ello la clase de A* está templatizada de la siguiente manera:

```
template <class graph_type, class heuristic>
class Graph_SearchAStar
{
…

}
```

Es decir, para crear un algoritmo de búsqueda A*, debemos pasarle el tipo del grafo y además el tipo de la función heurística que deberá utilizar. Por ejemplo podemos definir la función distancia euclídea que usamos en los ejemplos de la siguiente manera:

```
class Heuristic_Euclid
{
public:

  Heuristic_Euclid(){}

  //Calcula la distancia en linea recta entre dos nodos
  template <class graph_type>
  static double Calculate(const graph_type& G, int nd1, int nd2)
  {
    return Vec2DDistance(G.GetNode(nd1).Pos(), G.GetNode(nd2).Pos());
  }
};
```

Luego para crear un algoritmo de tipo A* deberíamos hacer:

```
Graph_SearchAStar<graph_type,Heuristic_Euclid> nombre(grafo,nodoOrigen,
nodoDestino);
```

Dónde como primer argumento del template le pasamos le tipo del grafo y como segundo argumento el tipo de la función heurística. Seguramente han notado que tanto los algoritmos como las funciones heurísticas son clases.