Algoritmos de Búsqueda

Investigación Operativa, Universidad de San Andrés

Si encuentran algún error en el documento o hay alguna duda, mandenmé un mail a rodriguezf@udesa.edu.ar y lo revisamos.

1. Introducción

Los algoritmos de búsqueda son herramientas fundamentales en la resolución de problemas de optimización y toma de decisiones. En esta clase vamos a estudiar cómo se aplican estos algoritmos a problemas clásicos de grafos y optimización, como caminos mínimos, decisiones secuenciales y recorridos eficientes. Hay dos grandes tipos, unos son los algoritmos de árboles y grafos, y los otros son los algoritmos de listas.

Los algoritmos que vamos a ver para árboles y grafos son:

- **Búsqueda Ciega**: Para encontrar el camino más corto en árboles o grafos. Dentro de esta categoría entran BFS (Breadth-First Search) y DFS (Depth-First Search).
- **Búsqueda Informada**: Para encontrar el camino mínimo en grafos con pesos positivos. Dentro de esta categoría entran A* y Greedy Best-First Search. Informada significa que tiene una idea de a dónde es que tiene que llegar
- **Dijkstra**: No es ciego como BFS/DFS pero tampoco es heurístico. Se lo suele agrupar dentro de búsquedas no informadas pero es un algoritmo de búsqueda de costo uniforme.

Por otra parte, estos son los algoritmos para búsquedas en listas que vamos a ver:

- **Búsqueda Secuencial**: Para búsquedas en listas o secuencias de datos.
- **Búsqueda Binaria**: Para búsquedas eficientes en listas ordenadas.

2. Conceptos Básicos de Grafos

Antes de introducir los algoritmos, recordemos algunos conceptos fundamentales:

- **Nodo (Vértice)**: Elemento fundamental del grafo.

- **Arista**: Conexión entre dos nodos.
- **Grafo Dirigido**: Las aristas tienen dirección.
- **Grafo No Dirigido**: Las aristas no tienen dirección.
- **Grafo Ponderado**: Las aristas tienen pesos asociados.
- **Camino**: Secuencia de nodos conectados por aristas.

3. Notación Big O

La notación Big O es una forma de describir la eficiencia de un algoritmo en términos de tiempo de ejecución y uso de memoria. Nos permite comparar algoritmos independientemente del hardware o lenguaje de programación.

Complejidades más comunes:

- $O(1)$ **- Constante**: El tiempo no depende del tamaño de entrada.
- $O(\log n)$ **- Logarítmica**: Muy eficiente, como búsqueda binaria.
- $O(n)$ **- Lineal**: El tiempo crece proporcionalmente al tamaño.
- $O(n \log n)$ **- Linealítmica**: Como algoritmos de ordenamiento eficientes.
- $O(n^2)$ **- Cuadrática**: Como algoritmos de ordenamiento simples.
- $O(2^n)$ **- Exponencial**: Muy ineficiente para entradas grandes

El ejemplo más facil de entender es si tenemos un ciclo (como en Python cuando teníamos un ciclo for). En ese caso la complejidad ahí es $O(n)$. Como si tuvieramos que revisar una bolsa llena de figuritas del mundial para encontrar la de Messi, en el peor de los casos vamos a tardar $O(n)$ (osea, repetiremos la tarea de buscar la figurita de Messi una y otra vez hasta que la encontremos). Si metemos un ciclo adentro de otro ciclo entonces la complejidad ahí es $O(n^2)$. Como si tuvieramos que revisar la bolsa, agarrar cada figurita, pegarla y volver a repetir (son dos tareas por figurita). Si tenemos otro ciclo más adentro de ese ciclo, como además dar un salto mortal cada vez que sacamos una figurita entonces ya es $O(n^3)$.

4. Búsquedas No Informadas (o Ciegas)

Las búsquedas no informadas, también llamadas ciegas, son aquellas que no tienen información adicional sobre el problema ni tampoco tienen una función de evaluación para saber que tan lejos están de resolver el problema. BFS expande primero todos los nodos de un nivel antes de pasar al siguiente, mientras que DFS se mete lo más profundo posible en una rama antes de retroceder. Es importante aclarar que si bien los siguientes ejemplos son de árboles, los algoritmos se pueden aplicar a grafos.

4.1. Breadth-First Search (BFS)

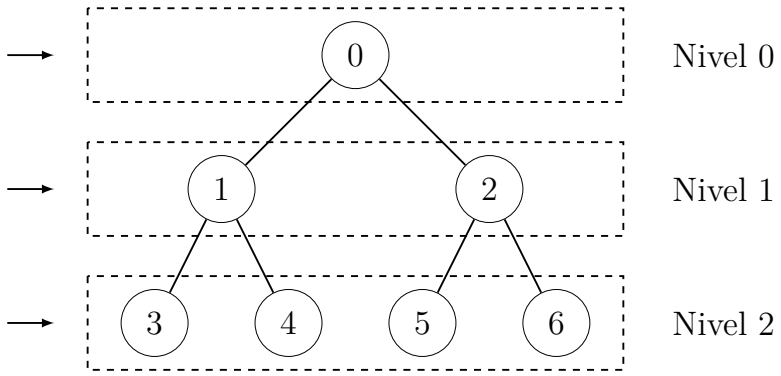
El algoritmo BFS explora un grafo nivel por nivel. Es especialmente útil para encontrar el camino más corto en grafos no ponderados.

4.1.1. Características Principales

- Explora todos los vecinos del nodo actual antes de pasar al siguiente nivel
- Cola FIFO (First In First Out) para mantener el orden en el que explora.
- Garantiza encontrar el camino más corto en términos de número de aristas
- Complejidad temporal: $O(V + E)$ donde V es el número de vértices y E el número de aristas

4.1.2. Visualización del Algoritmo BFS

El siguiente diagrama muestra el orden de exploración de BFS, donde los números indican el orden en que se visitan los nodos:



Orden de exploración:

- **0**: Nodo inicial (0)
- **1, 2**: Vecinos directos de 0 (1, 2)
- **3, 4**: Vecinos de 1 a visitar
- **5, 6**: Vecinos de 2 a visitar

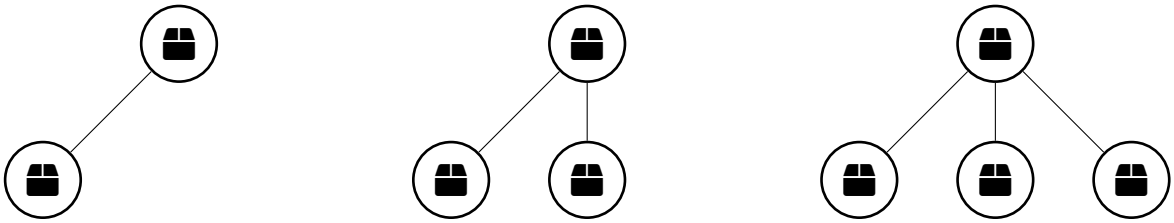
BFS explora nivel por nivel, garantizando que encontremos el camino más corto desde el nodo inicial a cualquier otro nodo. El problema es que si el grafo es muy grande, BFS puede ser muy ineficiente.

4.1.3. Ejemplo: Buscando algo en cajas cerradas

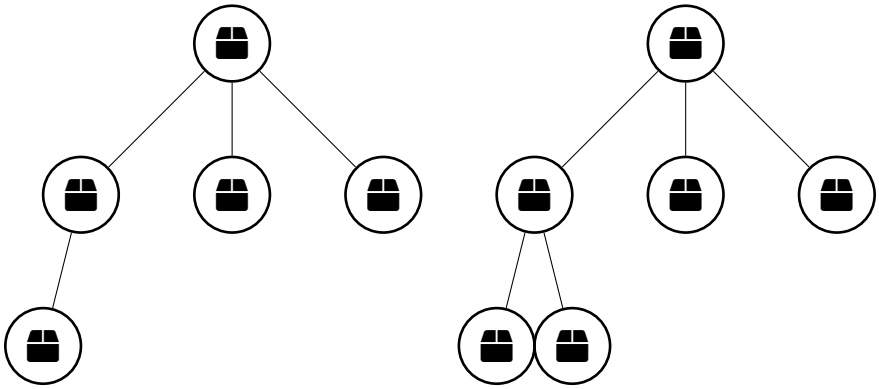
Imaginemos que tenemos una caja con una cantidad de cajas adentro que no sabemos ni cuantas hay, ni que tantas cajas y subcajas tenemos adentro. En algún momento hay un regalo, pero no sabemos cuando vamos a llegar a él.

Para nosotros, la caja al principio se ve así:

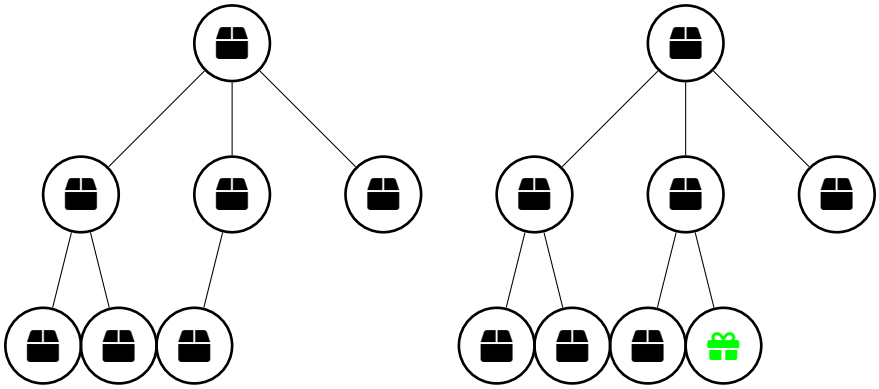
Una vez que abrimos la caja, nos encontramos con una caja, revisamos al costado y vemos otra caja. Seguimos revisando al costado y vemos también otra caja. Todavía no está el regalo.



Vamos entonces a abrir la primera de las cajas que encontramos, y encontramos también una caja. Revisamos al costado de esa nueva caja y vemos... también una caja. Nada más adentro de esa caja, así que tenemos que pasar a la siguiente caja.



Revisamos entonces la nueva caja, y vemos primero... otra caja. Buscamos al costado y... ¡encontramos el regalo!



Una vez que encontramos el regalo, podemos frenar dado que no tiene sentido seguir abriendo cajas.

Este algoritmo funciona bien en los casos en donde lo que buscamos se encuentra en alguno de los primeros niveles, dado que si se encuentra en uno de los niveles más profundos entonces perderemos mucho tiempo buscando en cada uno de ellos.

4.2. Depth-First Search (DFS)

El algoritmo DFS explora un grafo siguiendo una rama hasta el final antes de retroceder. Es útil para explorar completamente un grafo o detectar ciclos.

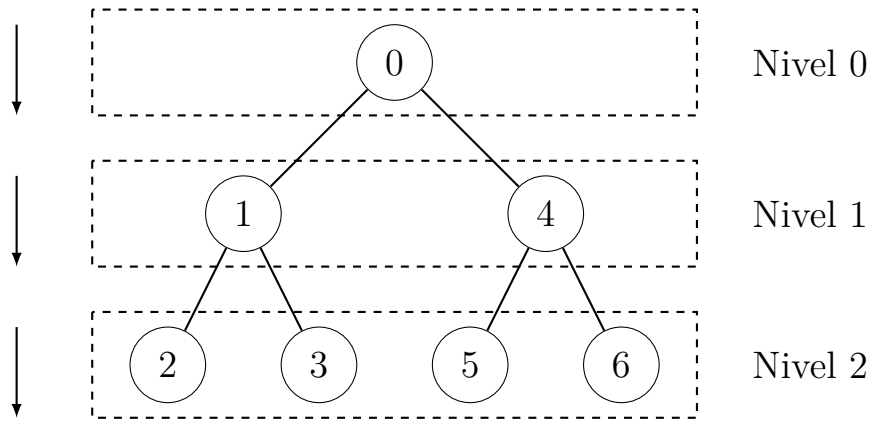
4.2.1. Características Principales

- Explora una rama completa antes de retroceder

- Utiliza una pila (LIFO) implícita a través de recursión
- Puede encontrar caminos largos pero no necesariamente el más corto
- Complejidad temporal: $O(V + E)$
- Útil para detectar ciclos y componentes conexas

4.2.2. Visualización del Algoritmo DFS

El siguiente diagrama muestra el orden de exploración de DFS, donde los números indican el orden en que se visitan los nodos:



Orden de exploración:

- **0**: Nodo inicial (0)
- **1**: El primer vecino de 0
- **2, 3**: El primer vecino de 1, y de no haber nada abajo de uno sigue con el segundo vecino de 1, y así.
- **4**: Dado que se termino toda la exploracion de los hijos y subhijos de 1, vamos con el segundo vecino de 0.
- **5, 6**: El primer vecino de 2, y de no haber nada abajo de uno sigue con el segundo vecino de 2, y así.

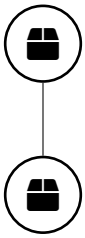
DFS explora una rama completa antes de retroceder, por lo que puede encontrar caminos largos pero no necesariamente el más corto. El problema es que si el grafo es muy grande, DFS puede ser muy ineficiente.

4.2.3. Ejemplo: Buscando nuevamente algo en cajas cerradas

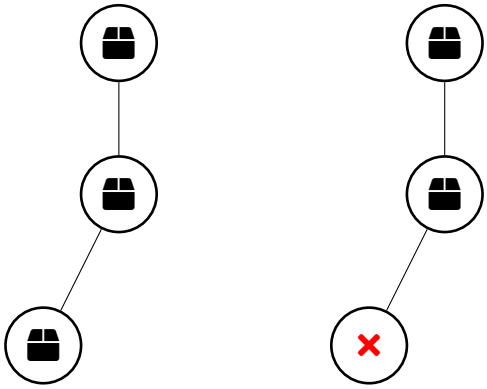
Imaginemos de vuelta que tenemos una caja con una cantidad de cajas adentro que no sabemos ni cuantas hay, ni que tantas cajas y subcajas tenemos adentro. En algún momento hay un regalo, pero no sabemos cuando vamos a llegar a él.

Para nosotros, la caja al principio se ve así:

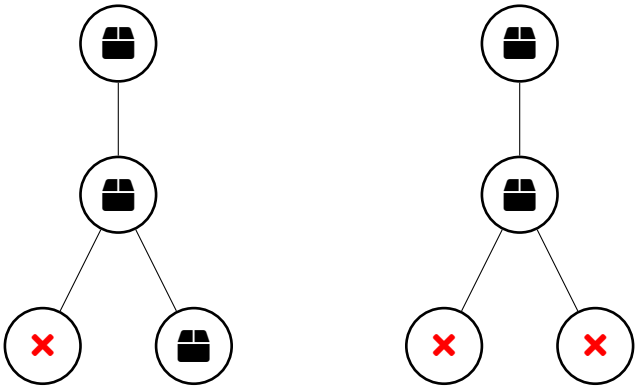
Una vez que abrimos la caja, nos encontramos con que tiene al menos una caja adentro (por ahora no nos importan si hay más), y ahora nuestro gráfico pasa a verse así.



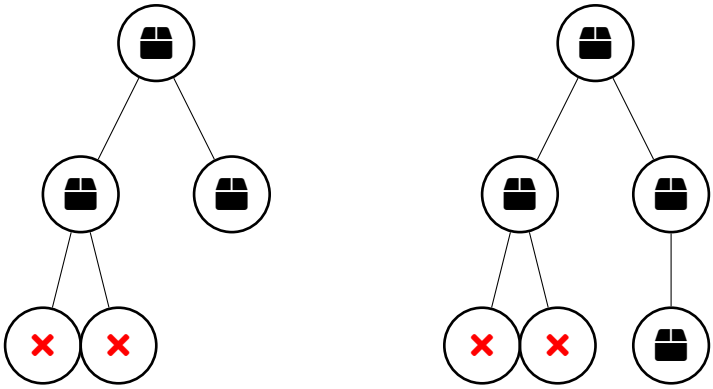
Vamos a abrir esa caja y ver que había adentro. Nos encontramos con una caja asi que la exploramos pero solo para ver que hay una cruz roja gigante.



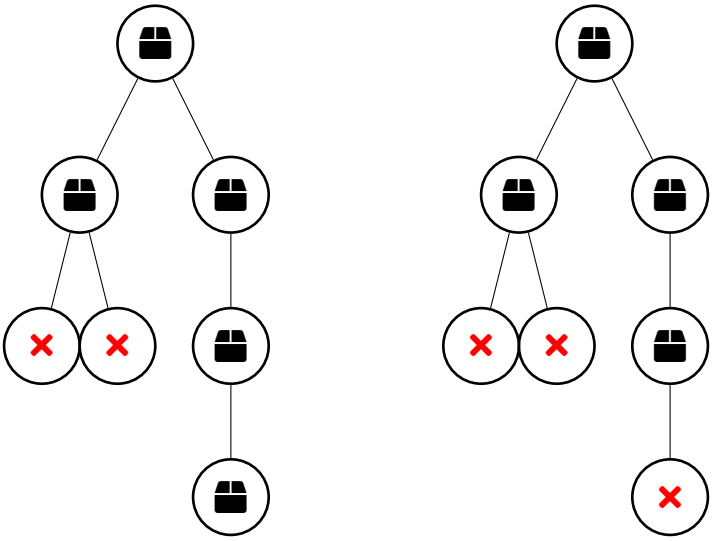
Pasamos entonces a ver si esta caja que abrimos y no tenía nada tiene alguna otra caja hermana. Encontramos que sí y entonces la abrimos, pero solo para ver que hay otra cruz roja gigante.



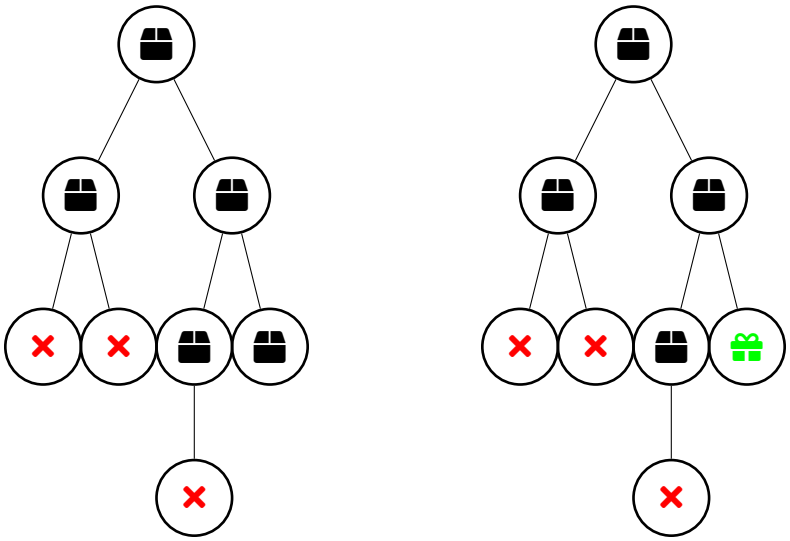
Tenemos que volver un paso para arriba y ver si la caja tiene hermanas. Vemos que tiene y la abrimos, pero para encontrar que hay otra caja adentro.



Tenemos que seguir explorando entonces para abajo, asi que entramos a la caja y encontramos otra caja. La abrimos y encontramos otra cruz roja.



Vamos a volver entonces para arriba y hacer el mismo proceso. Vemos que hay otra caja y entonces la abrimos y... ¡encontramos el regalo!

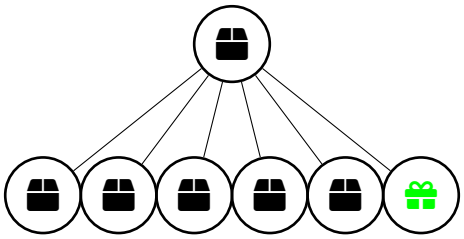


Frenamos porque ya está, no necesitamos nada más.

Este algoritmo funciona mejor si el regalo está en un nivel más profundo, pero si está en un nivel más cercano al inicio y no es de los primeros vecinos podemos perder mucho tiempo.

4.3. Conclusiones de Árboles de Búsqueda

Consideremos que un árbol fuese así en el primer nivel, y que cada caja tiene 100 niveles más abajo.



En este caso funcionaría mejor usar BFS, pero si el árbol hubiese tenido cuatro niveles, y justo el regalo estuviera en el cuarto nivel hijo de todos los primeros de cada nivel, DFS hubiese sido mejor.

Lo importante es que **nunca sabemos cuál es el mejor algoritmo** para un problema dado, y que debemos elegir el que mejor se adapte a las características del problema. Podemos tener alguna intuición sobre cuál usar, pero no podemos estar seguros. ¿Qué hacemos? Tan simple como probar con los dos.

5. Algoritmo de Dijkstra

El algoritmo de Dijkstra encuentra el camino más corto desde un nodo origen a todos los demás nodos en un grafo ponderado con pesos no negativos. Se llama así por su creador Edsger Dijkstra, quien lo publicó en 1959. Osea, tiene casi que 70 años de antigüedad y sigue siendo muy vigente.

5.1. Propósito

El propósito es encontrar el camino más corto desde un nodo inicial a **todos** los demás nodos en un grafo ponderado (con pesos en las aristas no negativos). Si queremos encontrar el camino más corto a un único nodo destino podemos detener el algoritmo en el momento en que sacamos ese destino de la cola de prioridad (porque en ese instante ya tenemos su costo mínimo garantizado). Al principio tiene todos los demás nodos con distancia infinita y vamos a ir eligiendo el nodo con distancia mínima que no haya sido visitado, actualizando las distancias de los vecinos como:

$$\text{nueva distancia} = \text{distancia actual} + \text{peso de la arista}$$

y marcamos el nodo como visitado. Luego, repetimos para todos los nodos hasta que todos queden visitados o hayamos encontrado el nodo destino. Esto **garantiza** encontrar el camino más corto desde el nodo inicial a todos los demás nodos. Es la base absoluta y total de algoritmos más avanzados de búsquedas en espacios como A* que veremos a continuación.

5.2. Ejemplo: Buscando la salida del laberinto

Estamos en un laberinto y no sabemos donde está la salida, pero tenemos una hoja y una lapicera en donde podemos ir dibujando el mapa. ¿Qué hacemos para comportarnos como el algoritmo de Dijkstra? Vamos a ir explorando todos los caminos posibles, paso a paso, pero primero revisando los que sean "más baratos" en cuanto a la distancia recorrida. No tenemos idea donde esta la salida pero sabemos cuales son los caminos que nos costaron menos, así que vamos a empezar o seguir por esos. Con tiempo y explorando todos los caminos ordenadamente vamos a llegar a la salida y por haber anotado y tomando nota sabemos que el camino que tomamos es el más corto/barato posible, aunque nos haya tomado mucho tiempo llegar a la solución. Recordemos, lo que importa no es el tiempo que nos tarda llegar a la solución sino que el camino que publicamos después como la solución sea el más corto posible.

Ahora bien... ¿qué es el costo ese del que me hablas? Claro, no aclaramos eso. Dijimos que estamos en un grafo ponderado, si cada arista (o metro recorrido) cuesta $1 entonces el costo acumulado son la cantidad de metros que recorrimos. Bien podría pasar que algunos metros cuesten distinto, podría existir que en alguno de los pasillos haya un fisura tirado que nos ataque y nos robe plata, pero como dijimos "no negativos" no existe ningún caso en donde nosotros seamos el fisura que le roba plata a alguien y nos haga más barato el trayecto.

6. Búsquedas Informadas

Las búsquedas informadas son aquellas que tienen información adicional sobre el problema, y que se basan en una función de evaluación para elegir el siguiente nodo a explorar. Básicamente cuentan con una función de evaluación que nos dice qué tan bueno es un nodo para llegar a la solución o al nodo de destino.

6.1. A* (A-Star)

El algoritmo A* es una búsqueda informada que utiliza una función de evaluación para elegir el siguiente nodo a explorar. La función de evaluación es una combinación de la distancia desde el nodo inicial y la distancia estimada desde el nodo actual hasta el nodo de destino. Es prácticamente una mejora de Dijkstra sumándole que incorpora la heurística de la estimación de cuanto me falta para llegar, explorando menos nodos.

6.1.1. Ejemplo: Buscando el camino más corto para salir del laberinto

Imaginemos que estamos nuevamente en un laberinto (o un shopping enorme para variar), sabemos donde estamos parados y sabemos **aproximadamente** donde está la salida. Capaz puede ser porque alguien nos dijo que la salida esta en la esquina noreste, sabemos la latitud y longitud de la salida y tenemos un pequeño GPS que nos acompaña. Vamos a tener que decidir por donde nos movemos para llegar lo más rapido posible a la salida, lo cual podría incluir volver para atrás sobre nuestros pasos.

¿Cómo funciona? A* combina dos cosas:

- La distancia que ya recorriste, osea el costo que ya pagaste. Por ejemplo cuantos pasos diste hasta ahora.
- La distancia que te falta por recorrer, lo cual es una **estimación** como por ejemplo la distancia en línea recta. Nunca podemos saber a ciencia cierta cuánto nos falta por recorrer.

Para cada paso A* va a entonces elegir moverse hacia el nodo que parecería tener el camino total mas corto:

$$\text{Costo total estimado} = \text{Costo hasta ahora} + \text{Heurística hacia el premio}$$

¡Importante!: Todo va a depender de la heurística que usemos. Si se sobreestima o subestima el camino real, el algoritmo puede fallar.

6.1.2. Recursos Extra

Algunos recursos extra que pueden ser útiles para entender mejor el algoritmo A* son los siguientes links a YouTube:

- A* (A Star) Search and Heuristics Intuition in 2 minutes

- A* (A Star) Search Algorithm - Computerphile
- The hidden beauty of the A* algorithm

6.2. Greedy Best-First Search

El algoritmo Greedy Best-First Search es una búsqueda informada que utiliza una función de evaluación para elegir el siguiente nodo a explorar. La función de evaluación es una estimación de la distancia desde el nodo actual hasta el nodo de destino. Básicamente el algoritmo solo contempla cuanto falta, y no cuanto caminé. Si yendo para donde está el premio se choca con un callejón sin salida puede retrodecer pero no planifica "a largo plazo".

6.3. Conclusiones de Búsquedas Informadas

En definitiva, A* en la que vé que ese camino le va a tomar más tiempo pega la vuelta, mientras que GBFS va a seguir intentándolo hasta fallar. Si desde la entrada hasta la salida faltan diez pasos, GBFS va a recorrer esos diez pasos sin pegar la vuelta, sin importar si ya caminaste 20. Esto va a retornar un camino de 30 pasos en total, mientras que A* considera cuanto ya caminaste y cuanto falta para minimizar el costo de llegar a la salida. Podría pasar que GBFS sea más rapido en tiempo de ejecución pero A* va a ser más confiable y más preciso en encontrar el camino más corto hacia la salida. Lo que queremos es optimizar justamente la distancia que recorremos de un lado a otro.

Va un ejemplo: El tiempo de ejecución son milésimas de segundo y ni nos vamos a dar cuenta pero, ¿nos importa que Google Maps nos muestre el camino lo más rapido posible? ¿o nos importa que nos diga cómo llegar de la Universidad a nuestra casa de la manera más rápida?

7. Búsquedas en Listas

Las búsquedas en listas son aquellas que se basan en la búsqueda secuencial o binaria para encontrar un elemento en una lista o secuencia de datos.

7.1. Búsqueda Secuencial

La búsqueda secuencial es fácil y no requiere más de un párrafo de explicación. El algoritmo no necesita que la lista este ordenada, da lo mismo si está desordenada.

Lo único que tiene que haber es un orden en los datos: este voy a chequear primero, este segundo y así. Lo que queremos buscar es un elemento y lo vamos a buscar entre todos los elementos que tenga la bolsa. La complejidad entonces es **O(n)** en el peor de los casos (sacando último lo que estoy buscando).

7.2. Búsqueda Binaria

La búsqueda binaria es un algoritmo eficiente para encontrar elementos en listas ordenadas, es decir, con un ordenamiento ascendente o descendente. El algoritmo es muy simple, se divide la lista en dos partes iguales, y se va a chequear si el elemento que buscamos está en la primera mitad o en la segunda mitad. Si está en la primera mitad, se vuelve a dividir la primera mitad en dos partes iguales y se va a chequear si el elemento que buscamos está en la primera mitad de la primera mitad o en la segunda mitad de la primera mitad. Si está en la segunda mitad, se vuelve a dividir la segunda mitad en dos partes iguales y se va a chequear si el elemento que buscamos está en la primera mitad de la segunda mitad o en la segunda mitad de la segunda mitad. Y así sucesivamente hasta que se encuentre el elemento o se determine que no está en la lista. Este es mucho más eficiente que la búsqueda secuencial, y la complejidad es **O(log n)** en el peor de los casos.

Vamos con un ejemplo: ¿Cómo buscarían el teléfono de una persona en una guía de telefonos? Abrimos una página y es la J, ¿vamos a la página siguiente? ¿O saltamos bastante para pasarnos y poder volver para atrás? Eso se aproximaría a una búsqueda binaria, aunque no exacta.

8. Aplicaciones Prácticas

8.1. Breadth-First Search y Depth-First Search

Estamos en París y queremos comer una baguette. Salimos del hotel y nos encontramos con una calle que se llama Rue de Falopini. Buscamos en esa cuadra y no hay nada, giramos a la derecha y buscamos y seguimos girando y buscando... Si camináramos toda la Rue de Falopini hasta el final capaz no encontraríamos nada, o capaz sí. Caminar la Rue de Falopini hasta el fondo sería DFS, mientras que girar y buscar en la de al lado seria BFS.

Hablando ahora de alguna aplicación de negocios... Ponele que estamos buscando a qué segmento pegarle en una campaña. Podemos empezar segmentando por un nicho (por ejemplo fitness y después ir a fitness vegano y después ir a fitness vegano de

embarazadas y después ir a fitness vegano de embarazadas en el primer trimestre) e ir metiendonos cada vez mas deep en ese nicho para o bien clavarla en el ángulo o bien errarle estrepitósamente. Si en vez de ir cada vez mas profundo en el nicho exploraramos distintas cosas (como por ejemplo fitness, después ropa y después computadoras) podríamos encontrar un mercado que tenga tracción sin ser tan específico. En resumen, DFS es una estrategia de enfoque profundo mientras que BFS es una estrategia de exploración amplia.

8.2. Dijkstra, A* y Greedy Best-First Search

Rutas de entrega o logística para optimizar costos de transporte entre almacenes y clientes. Calcular rutas de transporte de datos entre torres para optimizar costos en telecomunicaciones. En definitiva, cualquier problema de optimización de rutas.

¿Y cuándo usar cada uno? Bueno, si no tenemos idea donde está la meta Dijkstra, si tenemos forma de estimarla A*. GBFS podría usarse para sugerir algo rápido sin importar que sea lo más óptimo, como un algoritmo que va priorizando leads en marketing según algún scoring dinámico para mostrar o dejar de mostrar anuncios.

8.3. Búsqueda Secuencial y Binaria

La secuencial la vamos a usar cuando no tengamos las cosas ordenadas, como por ejemplo si estamos buscando un elemento en una bolsa. La binaria la podemos usar por ejemplo para encontrar un momento en el tiempo en el que se produjo un evento. Si tenemos un video de seguridad y queremos encontrar cuando es que se robaron una bici, aún si el video fuese desde el principio del cosmos hasta hoy no nos tardaría mas de unos (posta) muy pocos minutos en encontrar el momento exacto en el que alguien se roba la bici.

Vamos con la matemática:

$$\text{pasos} = \left[\log_2\left(\frac{\text{rango de tiempo}}{\text{precision}}\right)\right]$$

$$\text{rango de tiempo} \approx 13.8 \text{ mil millones de años}$$

Si quiero ubicar el segundo en el que se produjo el robo solamente tengo que hacer 59 folds. Si quiero ubicar al milisegundo (porque why not) solo necesito 69 folds. ¿Y cuánto me puede tomar cada fold? Si cada fold me toma 3 segundos (y capaz estamos exagerando) entonces el tiempo total es de tan solo (!) 207 segundos, o 3 minutos y 27 segundos. God bless binary search.