Universidad Andrés Bello
Conectar · Innovar · Liderar

Introducción a la inteligencia artificial

🌐 Introducción a la Búsqueda Anytime

¿Qué es?

- La **búsqueda *anytime*** (o algoritmos de búsqueda *anytime*) se plantea como respuesta a un problema muy común en Inteligencia Artificial y ciencias de la computación:
- 👉 **¿Qué pasa cuando no tenemos tiempo suficiente para ejecutar un algoritmo de búsqueda hasta el final?**

Problema

Imagina que eres el conductor de una ambulancia en una ciudad grande.
Recibes una llamada urgente: debes trasladarte a un hospital lo más rápido posible.

- **Opción A:** Podrías sentarte con calma, revisar un mapa completo de la ciudad, calcular la ruta óptima considerando todos los semáforos, calles y posibles atascos. Seguramente encontrarías el camino más corto… pero cuando termines de calcular, ¡ya habrá pasado demasiado tiempo!

- **Opción B:** En cambio, decides tomar rápidamente la primera ruta que conoces y que sabes que funciona, aunque quizás no sea la mejor. Mientras avanzas, tu GPS va recalculando constantemente y mejorando la ruta si detecta un atajo o menos tráfico.

Ese segundo escenario es exactamente lo que busca resolver un **algoritmo *anytime***:

- Entregar una **primera solución rápidamente** (la ruta conocida).

- Seguir refinando mientras queda tiempo o recursos (el GPS optimiza la ruta mientras conduces).

- Garantizar que, si el tiempo se acaba de golpe, al menos **ya tienes una solución en mano y no te quedas paralizado**.

Formalidad

Definición

Un algoritmo de búsqueda *anytime* es aquel que:

- Produce **una solución válida en cualquier momento** después de iniciarse.
  Mejora progresivamente la calidad de la solución cuanto más tiempo se le concede.
- Puede ser interrumpido en cualquier instante, entregando la **mejor solución encontrada hasta ese momento**.

Propiedades clave

- **Interrumpibilidad**: el algoritmo puede detenerse en cualquier momento y devuelve la mejor solución hallada.
- **Mejora gradual**: la calidad de las soluciones mejora con el tiempo.
- **Monotonía**: cada nueva solución encontrada es de igual o mejor calidad que la anterior.
- **Medida de calidad**: debe existir una función objetiva que evalúe la calidad de la solución.

¿Qué significa un ciclo de búsqueda con *W*?

- En AW-A*, no se usa un único valor de *w*, sino que se **inicia con un *w* grande** (ej: 2.5 o 3.0) para obtener una solución inicial rápido.
- Luego se **disminuye progresivamente *W*** (ej: 2.0 → 1.5 → 1.0) y se repite la búsqueda en ciclos.
- Cada ciclo con un nuevo *w* intenta **mejorar la ruta encontrada previamente**.

Formalidad

¿Qué significa "cota superior"?

- En algoritmos de búsqueda hablamos de **cotas** para acotar el espacio de exploración.
- La **cota superior** es **el costo de la mejor solución encontrada hasta ahora**.
    - Ejemplo: si ya encontramos un camino de longitud 25, eso significa que sabemos que **ninguna ruta de longitud ≥ 25 es mejor**.

Metáfora :

“Es como cuando ya encontraste un restaurante que cuesta $25 por persona. Si alguien te propone otro que sabes que saldrá $30 o más, ni siquiera lo consideramos: ¿para qué, si ya tienes uno mejor y más barato? Eso es la poda usando una cota superior.”

🔷 ¿Cómo se usa esa cota? (la poda)

Cuando el algoritmo está expandiendo nodos, cada nodo tiene un costo estimado:

$$f(n) = g(n) + w \cdot h(n)$$

- **Si** $g(n) + h(n) \geq \text{cota superior}$ → ese nodo **no puede llevar a una solución mejor que la actual**, así que se descarta ("poda").
- Esto evita explorar rutas inútiles y acelera la búsqueda.

🔷 Ejemplo concreto

1. El algoritmo encuentra una ruta del inicio a la meta con costo **25**.
   → Guardamos esa ruta como la **cota superior actual = 25**.
2. En otro ciclo, aparece un nodo n con:
   - $g(n) = 20$ (ya gastó 20 pasos hasta ahí).
   - $h(n) = 10$ (mínimo que le falta según heurística).
     → $g + h = 30$.
3. Como **30 ≥ 25**, ese nodo no puede generar una ruta mejor que la ya encontrada.
   → El algoritmo lo descarta sin explorarlo más (poda).

Conexión con la búsqueda en IA

La búsqueda *anytime* se usa cuando:

- No sabemos cuánto tiempo de cómputo tendremos.
- El ambiente es cambiante y necesitamos soluciones que se ajusten sobre la marcha.
- Una solución subóptima inmediata es mejor que nada.

Es como tener un **“GPS inteligente”** en nuestros algoritmos: empieza con algo básico y va mejorando en la medida que el tiempo lo permite.

Ejemplos

Ejemplos de algoritmos *anytime*

- **Anytime Repairing A*** (ARA*): comienza con una heurística inflada (subóptima) y reduce progresivamente el factor de inflación para acercarse a la solución óptima.

- **Anytime Weighted A***: usa una heurística con peso que prioriza encontrar rápido una solución factible, y luego va disminuyendo el peso.

- **Monte Carlo Tree Search (MCTS)**: en juegos, mejora las decisiones conforme más tiempo se le da para simular jugadas.

Diferencia con búsqueda tradicional

- Un A* tradicional no devuelve nada hasta que encuentra la solución óptima.

- Un *anytime A*** devuelve primero una solución rápida (quizás no óptima), pero sigue trabajando hasta mejorarla o alcanzar la óptima.

Caso práctico: Robot en un laberinto

📌 **Contexto**:
Un robot debe salir de un laberinto hacia la meta.

- Con **A*** clásico, el robot se quedaría “pensando” hasta encontrar el mejor camino.
- Con **Anytime A***, el robot encuentra rápidamente un camino válido (aunque no el más corto), comienza a moverse, y mientras avanza, el algoritmo sigue buscando atajos que mejoran la ruta.

Idea general - Generador de mapas

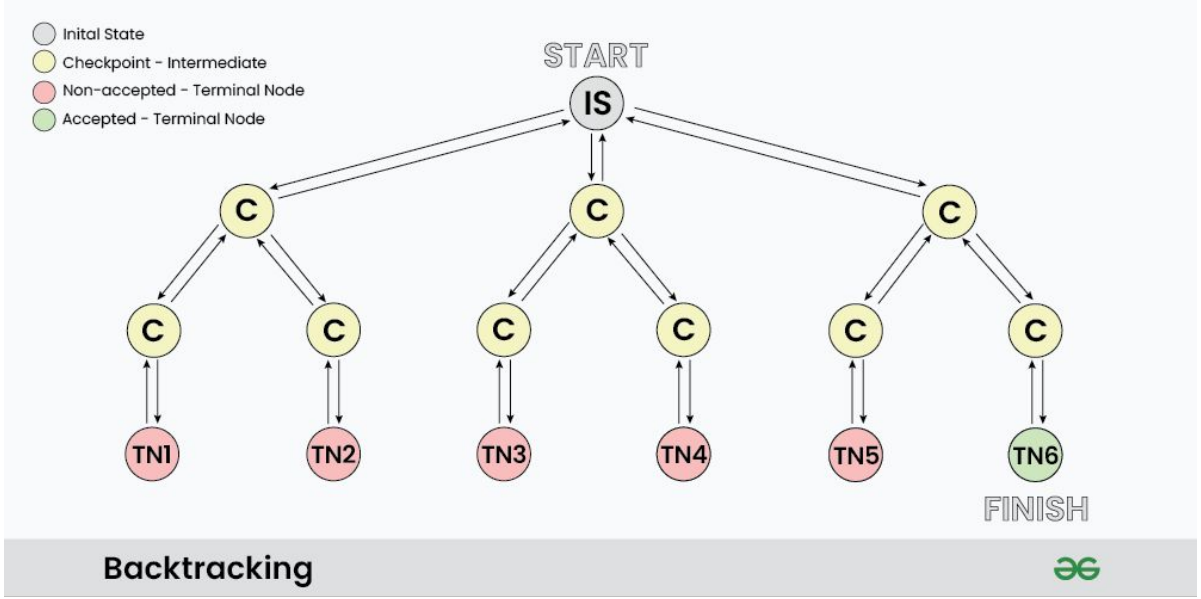
Un laberinto puede representarse como una **matriz** donde:

- `1 = muro`
- `0 = espacio libre`.

Lo que hace el algoritmo es **construir un laberinto perfecto** (conectado, sin ciclos triviales) recorriendo celdas de manera aleatoria y abriendo pasillos.

Algoritmo Backtracking DFS (explicado simple)

1. Comenzamos en una celda inicial (ej: `(1,1)`).
2. Marcamos esa celda como visitada (pasillo libre = `0`).
3. Elegimos al azar un **vecino a dos pasos** (para dejar un muro entre medio).
4. Si ese vecino no ha sido visitado:
    - “Tiramos abajo” el muro intermedio (lo ponemos en `0`).
    - Avanzamos al vecino y repetimos el proceso (recursión o pila).
5. Si no quedan vecinos disponibles, **retrocedemos** (backtracking) hasta encontrar una celda con vecinos sin visitar.



Backtracking