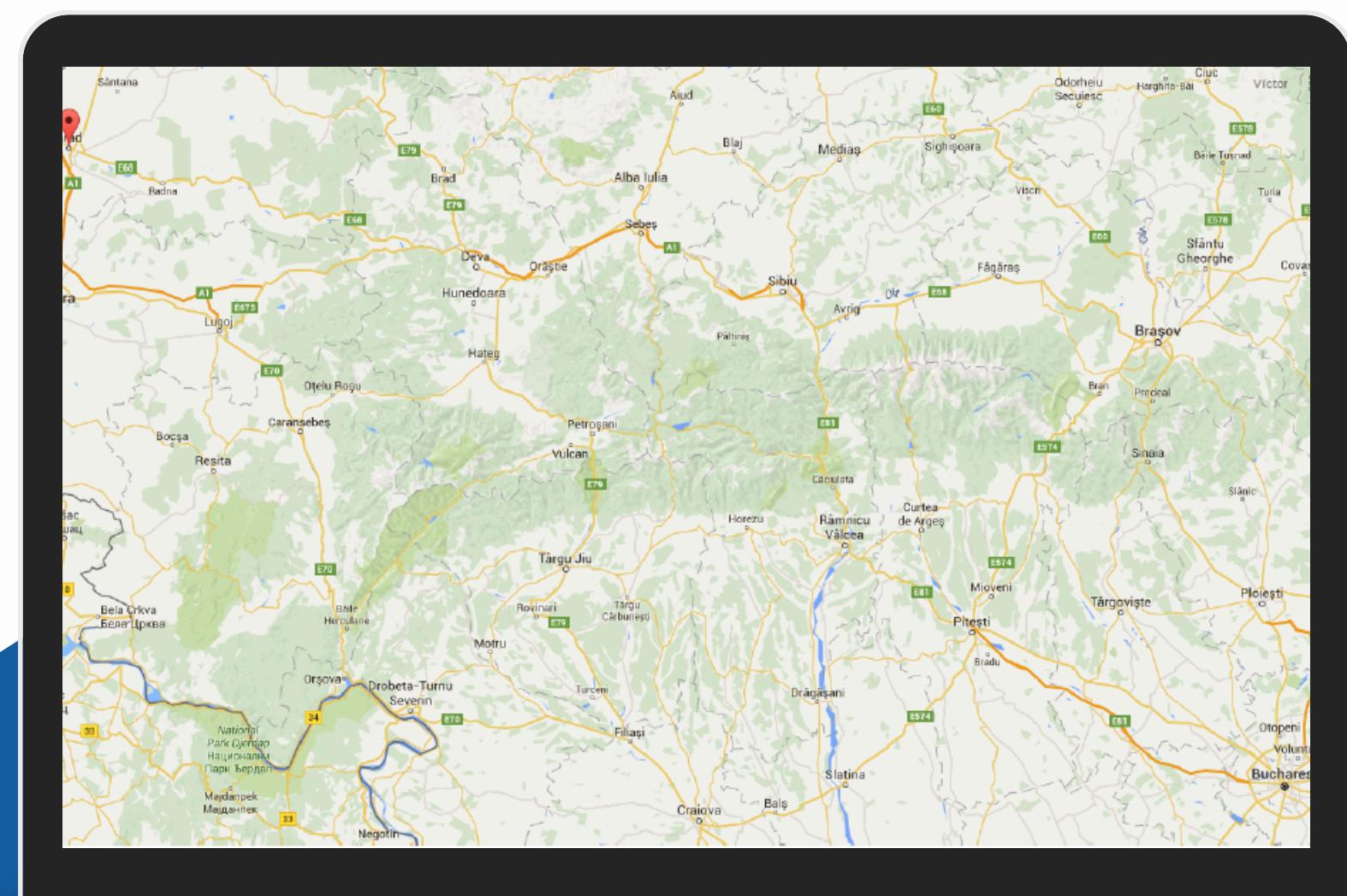




Universidad Rafael Landívar

Problemas de Busqueda



Agenda

- Problemas
- ¿Qué es un problema de búsqueda en IA?
- Tree Search vs Graph Search
- Algoritmos de Búsqueda Desinformada
- Algoritmos de Búsqueda Informada
- Comparación de Algoritmos
- Aplicaciones Prácticas

PROBLEMAS

Actualmente, ¿como resolvemos problemas?

- Prueba y error
- Tercerización
- Planificación
- Una serie de pasos para lograr una meta



PROBLEMAS

Actualmente, ¿como resolvemos problemas?

- Para problemas simples → prueba y error puede ser suficiente.
- Si el problema requiere conocimientos especializados → tercerización es una opción.
- Si el problema es complejo pero predecible → planificación ayuda a optimizar.
- Si el problema es repetitivo y estructurado → seguir un algoritmo es lo mejor.



Enfocar esfuerzos para diseñar agentes inteligentes capaces de encontrar soluciones eficientes a problemas complejos.

2025: El futuro de la inteligencia artificial son los agentes
! Y están a punto de cambiarlo todo...



Santiago Bilinkis



Agentes de inteligencia
artificial.



**Agentes
de IA**

Agente Inteligente

En IA es un sistema que observa su entorno, toma decisiones inteligentes y se adapta para lograr sus objetivos de la mejor manera posible.

La inteligencia de estos agentes proviene de su capacidad para tomar decisiones basadas en conocimientos, razonamiento, planificación y, en muchos casos, aprendizaje

01

Percibe su entorno a través de sensores

02

Razonamiento: Procesa la información del entorno para tomar decisiones racionales.

03

Actúa en el entorno mediante actuadores para alcanzar objetivos específicos.

04

Aprende y mejora su desempeño a lo largo del tiempo.



Reflex Agents

- "Si ocurre esto, haz aquello"
- Resuelven sus problemas en base a su percepción inmediata
- NO consideran las consecuencias de sus acciones
- Actuan en base a como el mundo ES en el momento que deben decidir

Planning Agents

- "¿Que pasaria si?"
- Decisiones basadas en hipótesis
- Necesitan un modelo de como el mundo reacciona a sus acciones
- Deben tener un objetivo (y probarlo)
- Consideran como el mundo debe ser



Agentes Inteligentes - Inteligencia Artificial



Copy link

Los agentes de IA pueden ser desde sistemas sencillos que siguen **reglas predefinidas** hasta entidades complejas y autónomas que aprenden y se **adaptan en función de su experiencia**. Se utilizan en diversos campos, como la robótica, los juegos, los asistentes virtuales y los vehículos autónomos, entre otros. Estos agentes pueden ser **reactivos** (responden directamente a estímulos), **deliberativos** (planifican y toman decisiones) o incluso tener **capacidad de aprendizaje** (adaptan su comportamiento en función de datos y experiencias).



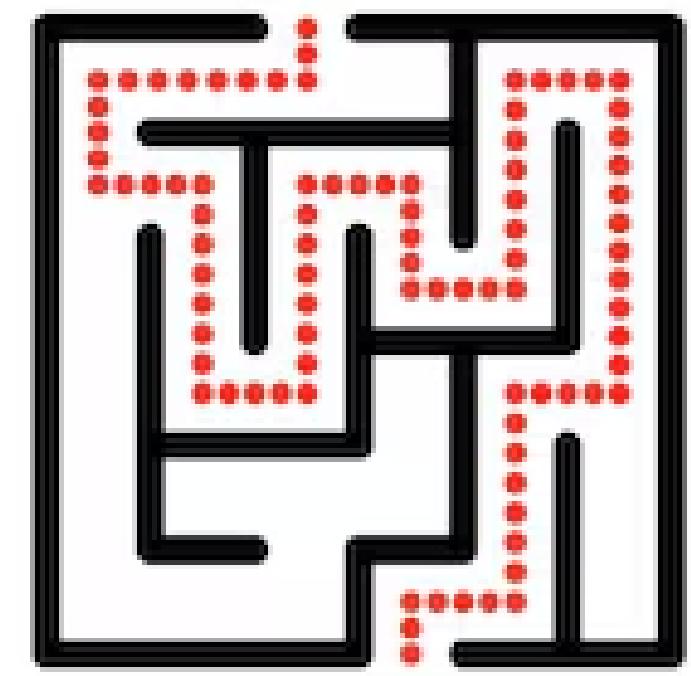
 synthesia

Watch on  YouTube

¿Qué es un problema de búsqueda en IA?

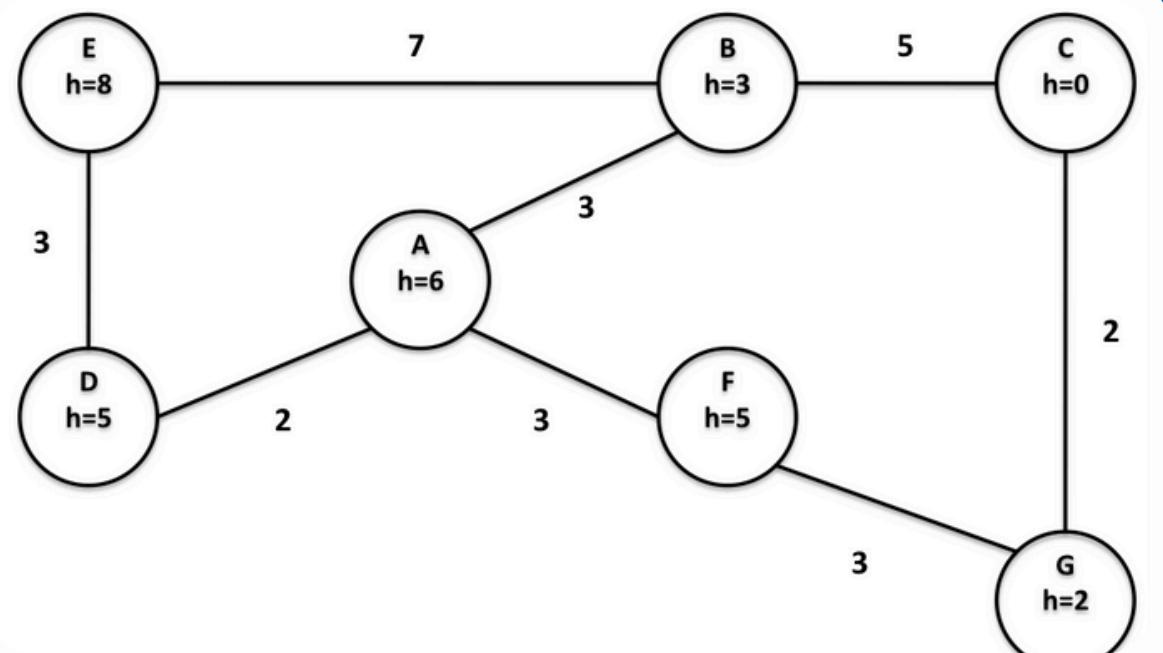
Se refiere a encontrar una secuencia de acciones que lleve desde un estado inicial a un estado meta, cumpliendo ciertos objetivos.

Estos problemas son comunes en áreas como juegos, planificación, optimización, y navegación de robots.



DECISION

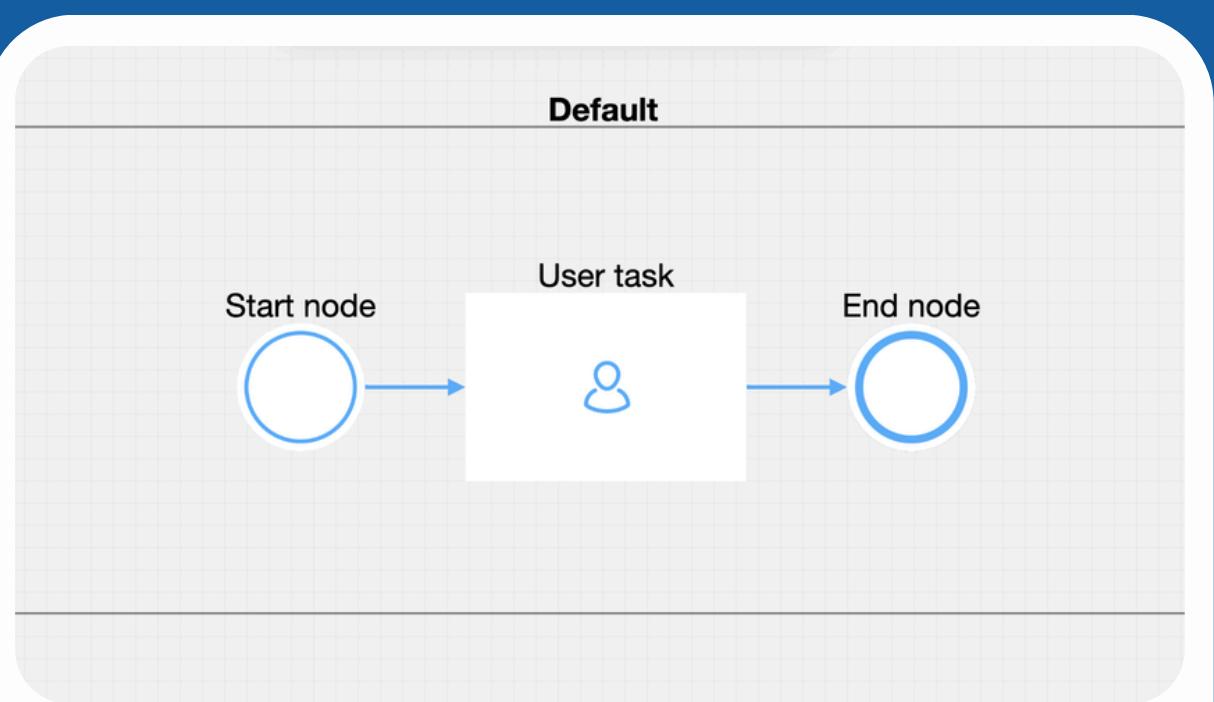
Elementos de un problema de búsqueda



Espacio de estados

Es el conjunto de todos los posibles estados en los que puede estar el agente durante la búsqueda.

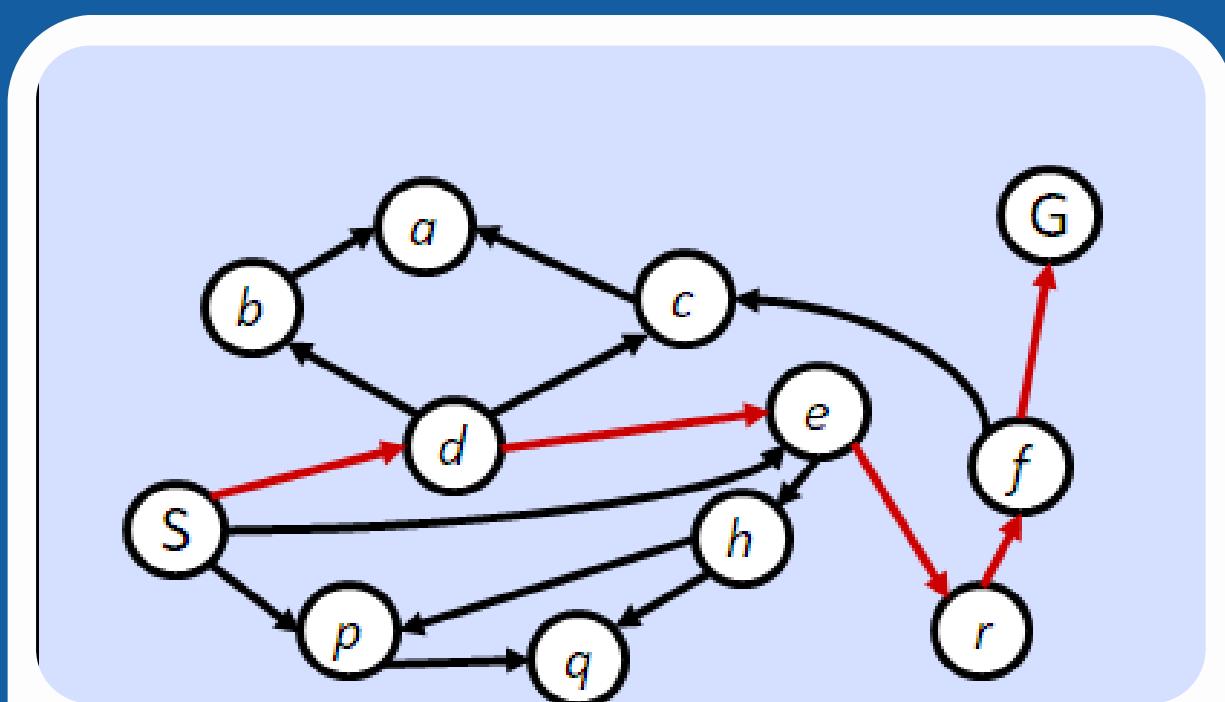
Cada estado representa una configuración única del problema.



Nodo inicial y nodos meta

Nodo inicial: Es el estado en el que comienza el agente.

Nodos meta: Son los estados objetivo que el agente debe alcanzar



Operadores de transición

Son las acciones disponibles para moverse de un estado a otro.

Función de costo (opcional)

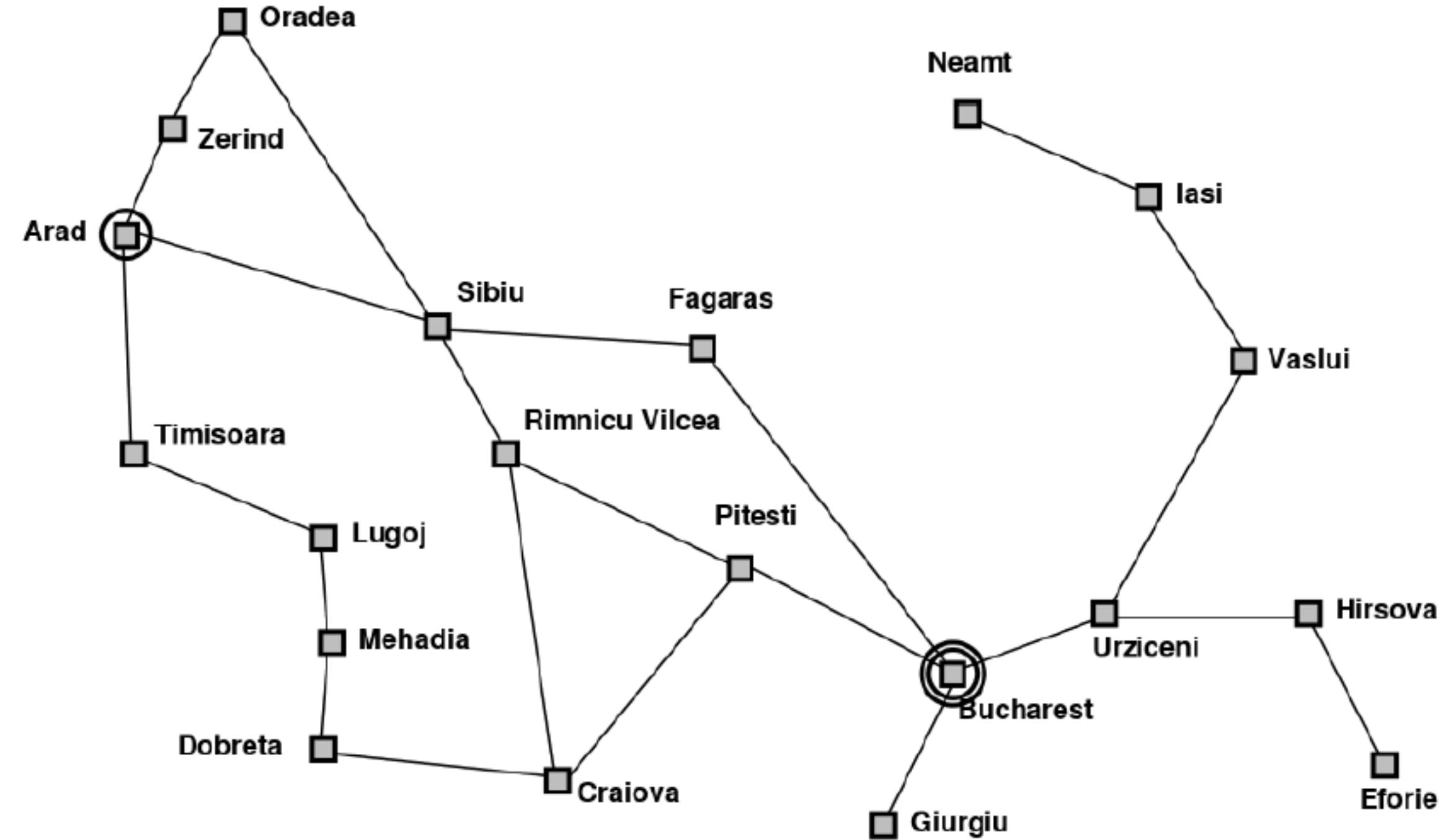
Evalúa el costo asociado con una acción o secuencia de acciones.

Ejemplo

Encuentre la ruta más corta entre Arad y Bucharest

Elementos del Problema

- Conjunto de estados
- Acciones posibles en un estado en particular
- Posibles resultados al aplicar una acción en un estado
- Validar si el estado actual es la meta
- Calcular el costo de aplicar la acción al cambiar de estado
- Calcular el costo del camino completo



Enfoques de Búsqueda: Tree Search vs Graph Search

Tree Search

Es un enfoque de búsqueda en el que no se lleva un registro de los estados visitados.

Ideal para espacios de búsqueda pequeños o sin ciclos.

En problemas con ciclos, puede quedar atrapado en un bucle.

Graph Search

Registra los estados visitados para evitar ciclos y redundancias.

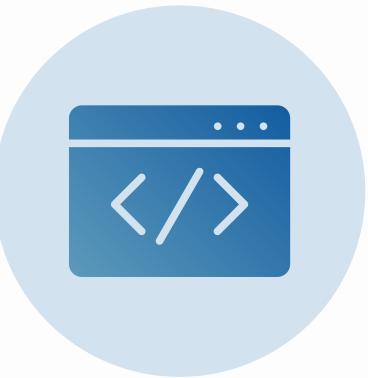
Mantiene un registro de los nodos explorados en estructura de datos.

No explora estados repetidos, ahorrando tiempo y esfuerzo computacional.

Es más complicado de implementar en comparación con Tree Search.



Cuándo usar cada enfoque

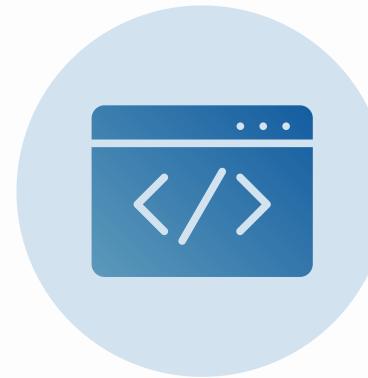


Tree Search

Útil para problemas pequeños, simples, o cuando los estados no tienen redundancias ni ciclos.

Ejemplo

Resolver un laberinto sin caminos que regresen al inicio.



Graph Search

Ideal para problemas complejos con grandes espacios de búsqueda, posibles ciclos, o redundancias.

Ejemplo

Encontrar la ruta más corta entre dos puntos en un grafo de ciudades.

Comparación

Característica	Tree Search	Graph Search
Registro de estados	No registra estados visitados	Registra estados visitados
Redundancia	Puede explorar estados repetidos	Evita explorar estados repetidos
Ciclos infinitos	Puede caer en ciclos infinitos	Evita ciclos infinitos
Uso de memoria	Menor uso de memoria	Mayor uso de memoria
Complejidad	Más simple de implementar	Más complejo de implementar
Eficiencia	Menos eficiente en espacios con ciclos	Más eficiente en espacios con ciclos

Tree Search

Se refiere a un enfoque sistemático para explorar un espacio de estados en busca de una solución.

En este caso, el agente utiliza una estructura de árbol para representar los posibles caminos o planes que puede seguir para alcanzar un objetivo.

Expandir nuestros planes potenciales (leaf-nodes)

01

Cada camino desde el nodo raíz hasta un nodo hoja representa un plan potencial que el agente podría seguir para resolver el problema

Mantener una frontera de consideración

02

Es un conjunto de nodos que están disponibles para ser expandidos. Los nodos más allá de ella no se consideran por el momento

Expandir pocos nodos

03

En lugar de expandir todos los nodos posibles (lo que sería ineficiente y consumiría muchos recursos)

Expandir solo aquellos nodos que tienen más probabilidades de conducir a una solución óptima

Algoritmo Tree-Search



Function TreeSearch(problema):

1. Inicializar la frontera con el estado inicial del problema.

frontera = [nodo_raíz] # nodo_raíz = (estado_inicial, camino_vacio, costo_acumulado=0)

2. Mientras la frontera no esté vacía:

a. Seleccionar un nodo de la frontera (depende de la estrategia: BFS, DFS, A*, etc.).

nodo_actual = frontera.eliminar()

b. Si el estado del nodo_actual es un estado objetivo:

Devolver la solución (camino desde el nodo raíz hasta el nodo_actual).

c. Expandir el nodo_actual:

Para cada acción posible en el estado actual:

i. Generar un nuevo estado aplicando la acción.

ii. Crear un nuevo nodo con el nuevo estado, el camino actualizado y el costo acumulado.

iii. Añadir el nuevo nodo a la frontera.

3. Si la frontera está vacía y no se encontró una solución:

Devolver "Fallo" (no hay solución).

Graph Search

Es una extensión de Tree Search que tiene en cuenta los estados ya visitados para evitar redundancias y ciclos.

01

Expandir nuestros planes potenciales (leaf-nodes)

No se expanden nodos cuyos estados ya han sido visitados, lo que reduce la cantidad de planes redundantes

02

Mantener una frontera de consideración

Antes de añadir un nuevo nodo a la frontera, se verifica si su estado ya ha sido visitado. Si es así, el nodo no se añade

03

Expandir pocos nodos

Graph Search es más eficiente que Tree Search en este aspecto, ya que evita la expansión de nodos redundantes

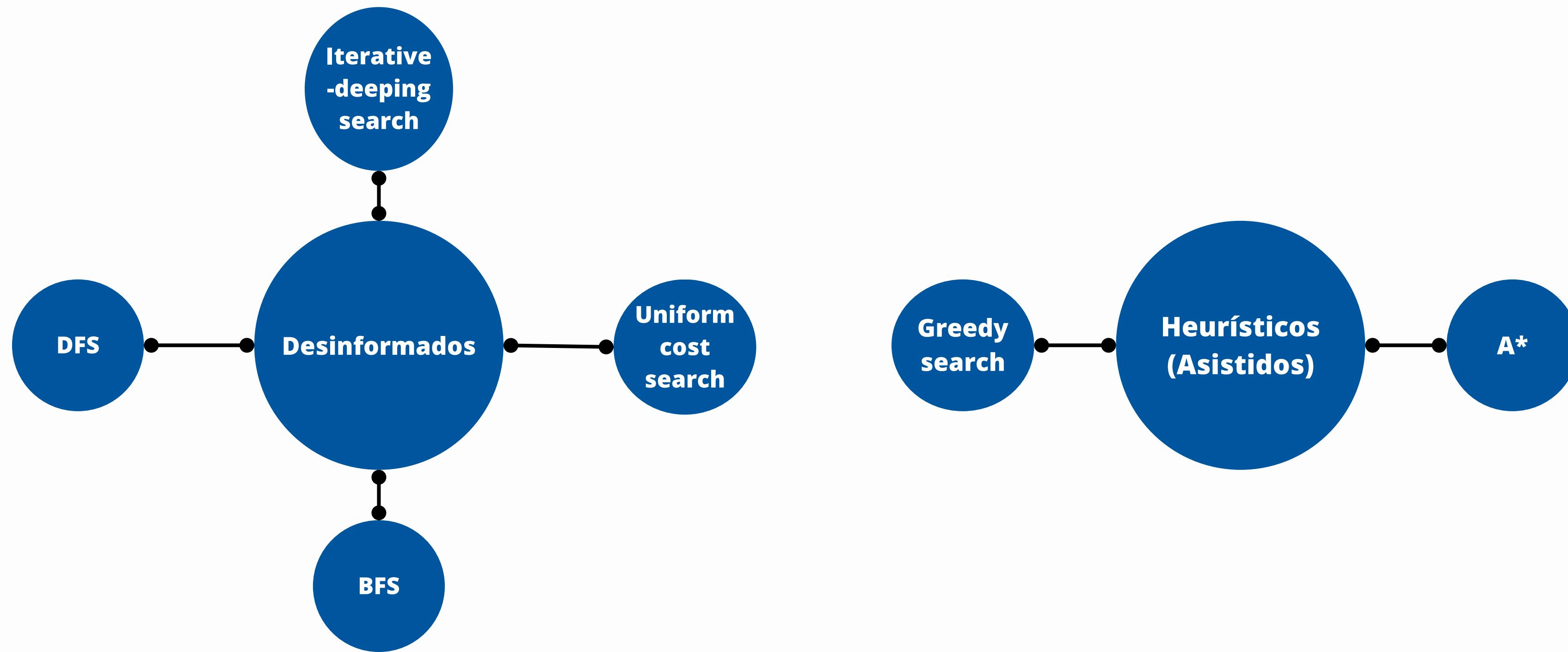
Algoritmo Graph-Search



```
Function GraphSearch(problema):
```

1. Inicializar la frontera con el estado inicial del problema.
frontera = [nodo_raíz] # nodo_raíz = (estado_inicial, camino_vacio, costo_acumulado=0)
2. Inicializar un conjunto para registrar los estados visitados.
visitados = Conjunto()
3. Mientras la frontera no esté vacía:
 - a. Seleccionar un nodo de la frontera (depende de la estrategia: BFS, DFS, A*, etc.).
nodo_actual = frontera.eliminar()
 - b. Si el estado del nodo_actual es un estado objetivo:
Devolver la solución (camino desde el nodo raíz hasta el nodo_actual).
 - c. Si el estado del nodo_actual no ha sido visitado:
 - i. Marcar el estado como visitado.
visitados.añadir(estado_actual)
 - ii. Expandir el nodo_actual:
Para cada acción posible en el estado actual:
 - Generar un nuevo estado aplicando la acción.
 - Crear un nuevo nodo con el nuevo estado, el camino actualizado y el costo acumulado.
 - Añadir el nuevo nodo a la frontera.
4. Si la frontera está vacía y no se encontró una solución:
Devolver "Fallo" (no hay solución).

Algoritmos de búsqueda



Propiedades de los algoritmos de búsqueda

01

Completo: Es capaz de encontrar una solución

02

Óptimo: Se garantiza encontrar la mejor solución

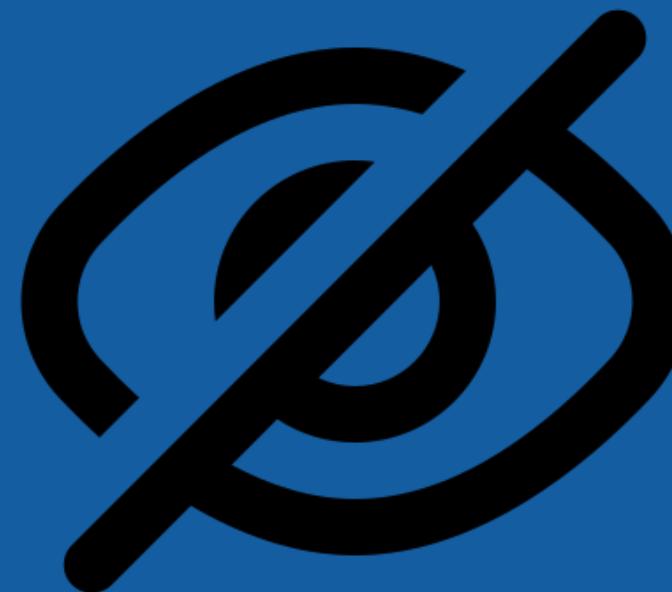
03

Complejidad tiempo: Cuantos nodos se expanden

04

Complejidad nodos: Que tan grande se vuelve el arbol

Algoritmos de Búsqueda No Informada



01

BFS: Búsqueda en anchura

02

DFS: Búsqueda en profundidad

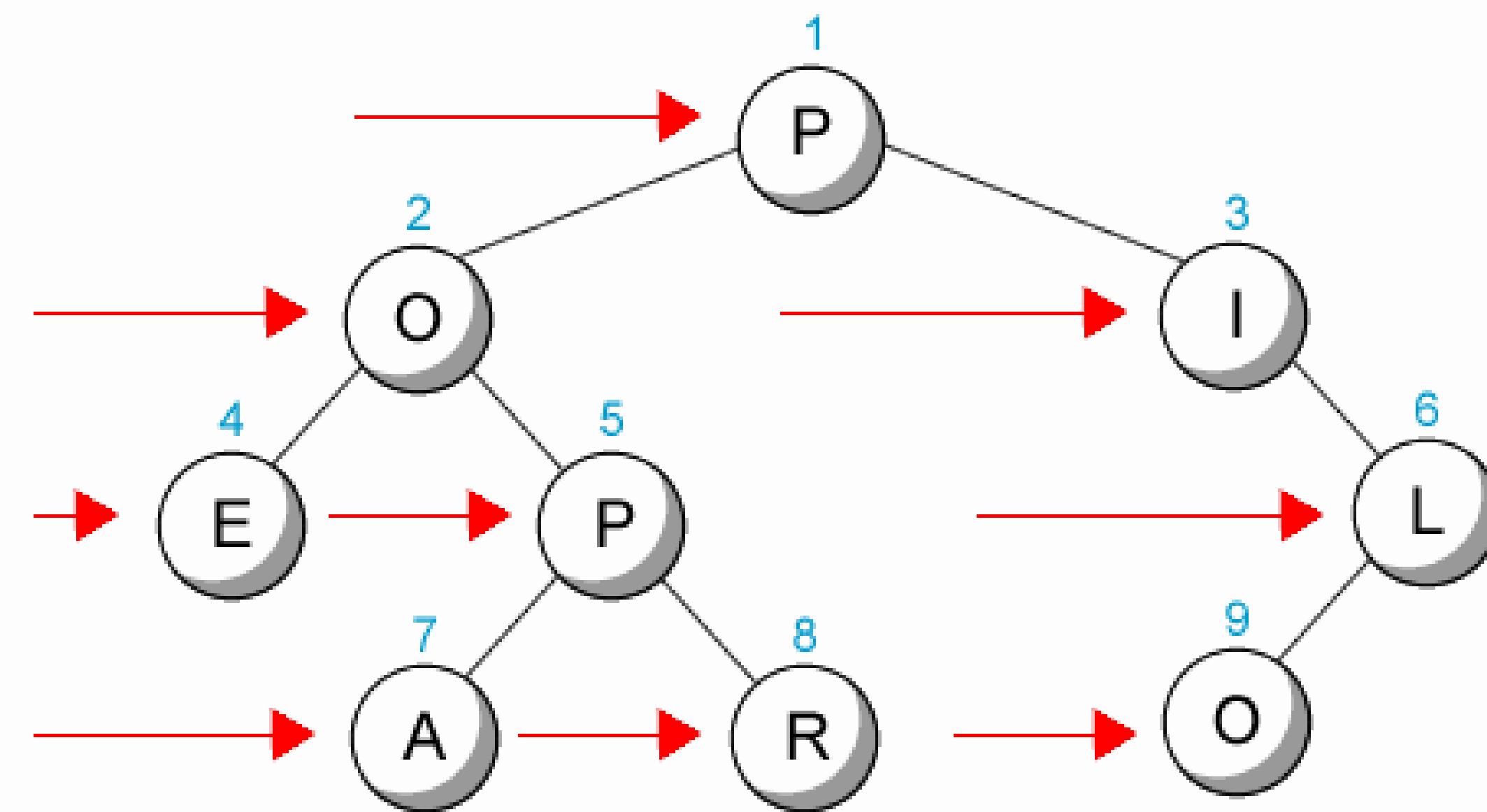
03

IDS: Iterative-Deepening Search

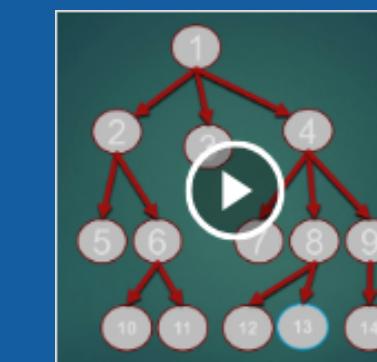
04

UCS: Búsqueda de Costo Uniforme

Búsqueda en Anchura (BFS)



Breadth-First Search es una estrategia de búsqueda no informada que explora un grafo o árbol nivel por nivel, expandiendo primero todos los nodos de un nivel antes de pasar al siguiente. Utiliza una cola (FIFO) para mantener la frontera de consideración, asegurando que los nodos más cercanos al inicio se expandan primero.

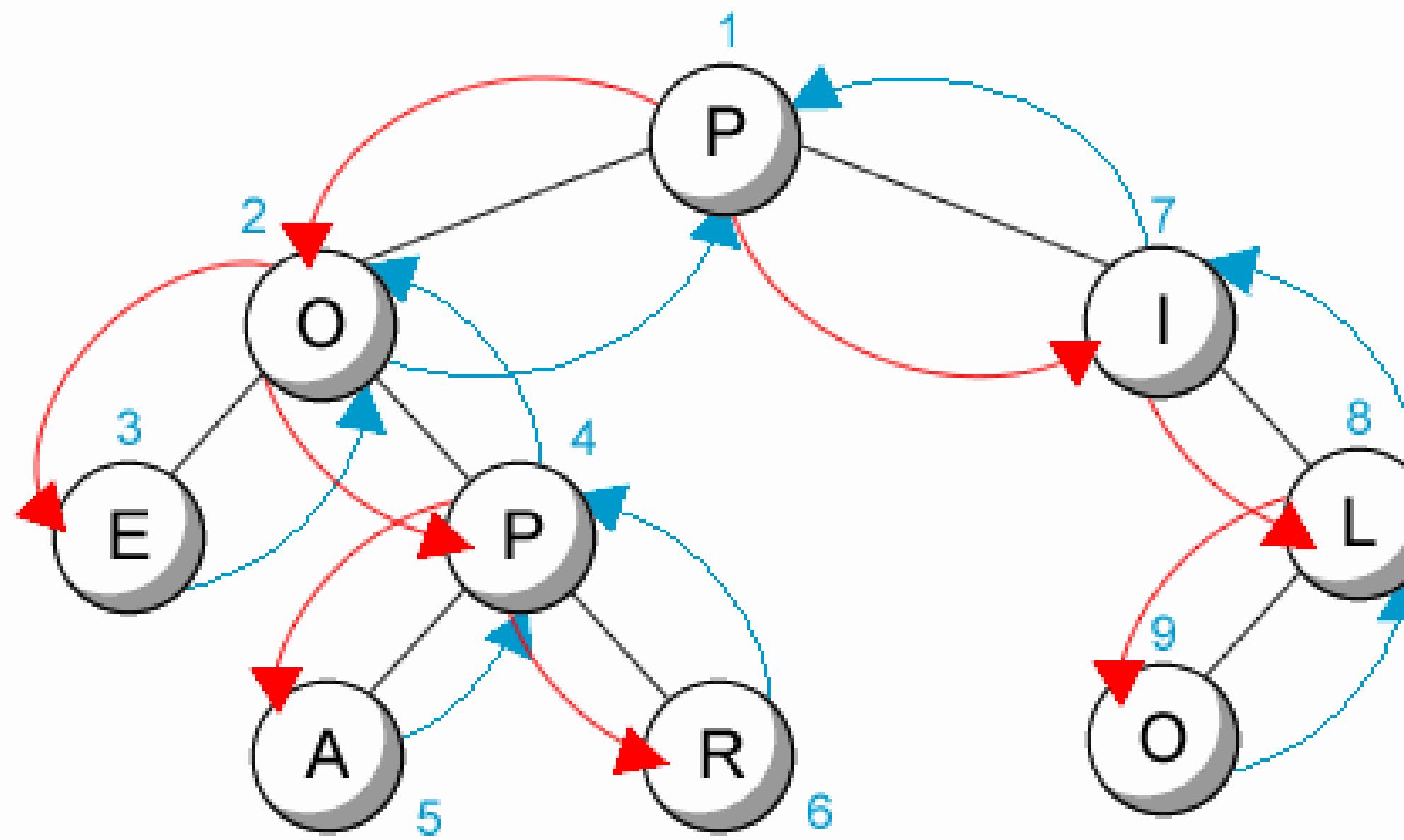


File:BFS Tree.gif

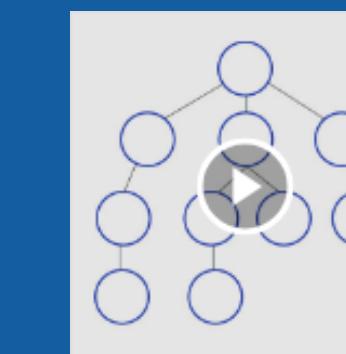
Click on a date/time to view the file as it appeared at that time.

Wikipedia

Búsqueda en Profundidad (DFS)



Depth-First Search es una estrategia de búsqueda que explora un grafo o árbol siguiendo un camino hasta su máxima profundidad antes de retroceder y explorar otros caminos. Utiliza una pila (LIFO) para gestionar la frontera de consideración, lo que permite priorizar la exploración de ramas profundas antes que las superficiales.



File:Depth-First-Search.gif

Click on a date/time to view the file as it appeared at that time.

Wikipedia

DFS



Puede ser implementado como tree search o graph search

No garantiza encontrar la ruta más corta, pero es eficiente en términos de uso de memoria y puede ser útil en problemas donde la solución está en una rama profunda del árbol

BFS



Puede ser implementado como tree search o graph search

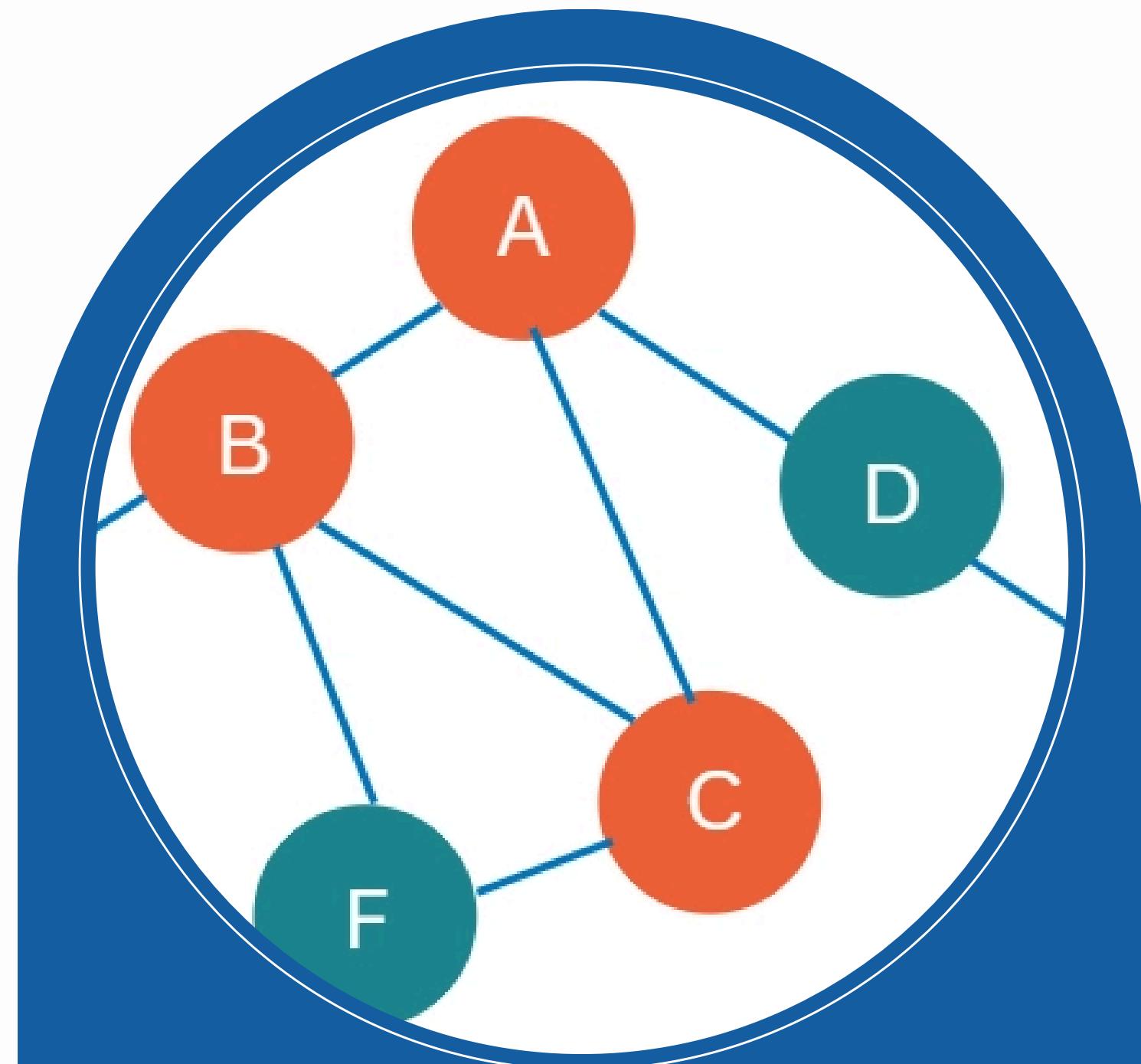
Garantiza que, si existe una solución, encontrará la ruta más corta desde el estado inicial hasta el estado objetivo. Esto lo hace ideal para problemas donde el costo de cada acción es uniforme.

Iterative-Deepening Search (IDS)

Es una combinación de las ventajas de BFS (Breadth-First Search) y DFS (Depth-First Search).

IDS funciona de la siguiente manera:

- Realiza una búsqueda en profundidad (DFS) con un límite de profundidad **L**.
- Si no encuentra la solución, incrementa el límite de profundidad **L** y repite la búsqueda.
- Continúa este proceso hasta encontrar la solución.



Uniform-Cost Search (UCS)

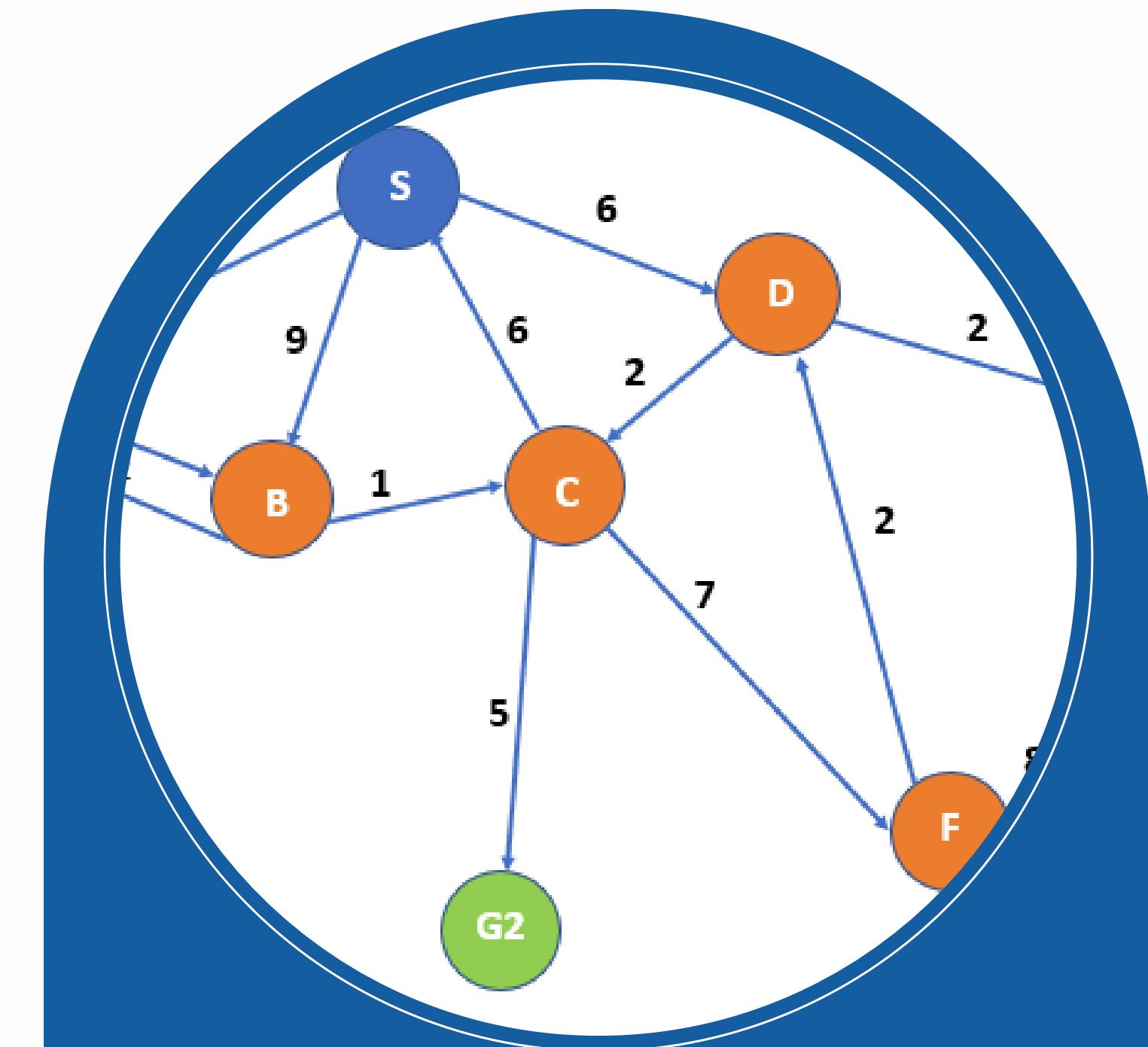
Es un algoritmo de búsqueda utilizado para encontrar la solución óptima en problemas donde cada acción tiene un costo asociado.

Es una variante del algoritmo de Dijkstra para grafos con costos en las aristas.

UCS explora un grafo o árbol de la siguiente manera:

1. Comienza desde el nodo raíz (o estado inicial).
2. Expande el nodo con el menor costo acumulado hasta el momento.
3. Repite este proceso hasta encontrar el estado objetivo o agotar todos los nodos.

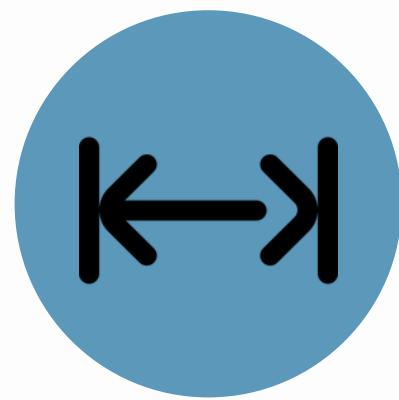
Mantiene una cola de prioridad donde los nodos se ordenan por su costo acumulado



Resumen

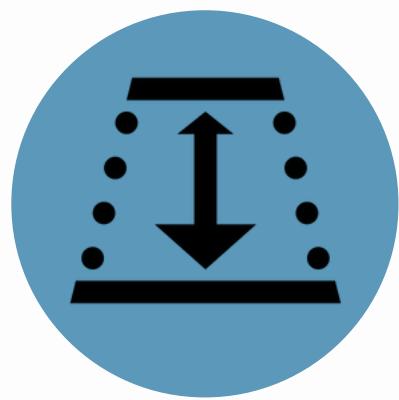
Característica	Búsqueda en Anchura (BFS)	Búsqueda en Profundidad (DFS)	Iterative Deepening Search (IDS)	Búsqueda de Costo Uniforme (UCS)
Estrategia	Explora nivel por nivel	Explora en profundidad antes de retroceder	Usa DFS con un límite de profundidad creciente	Expande el nodo con menor costo acumulado.
Estructura de Datos	Cola	Pila	Cola con control de profundidad	Cola de prioridad
Eficiencia Espacial	Alto uso de memoria.	Bajo uso de memoria.	Usa menos memoria que BFS.	Alto uso de memoria.
Eficiencia Temporal	Puede ser lento en problemas grandes.	Puede ser rápido, pero no garantiza mejor solución.	Mejor balance entre BFS y DFS.	Puede ser lento en problemas con costos variables.

¿Cuándo usar?



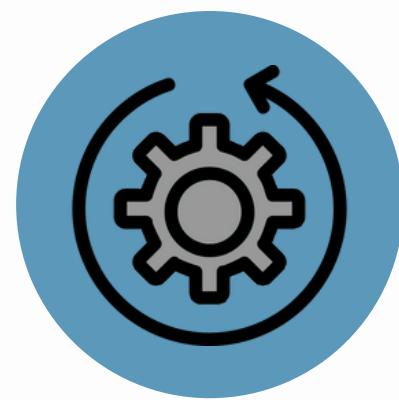
Búsqueda en Anchura (BFS)

Cuando necesitas la solución más corta y el espacio no es un problema.



Búsqueda en Profundidad (DFS)

Cuando quieres explorar rápidamente o el espacio es un problema.



Iterative Deepening Search (IDS)

Cuando necesitas la solución más corta y el espacio no es un problema.



Búsqueda de Costo Uniforme (ucs)

Cuando quieres explorar rápidamente o el espacio es un problema.

Algoritmos de Búsqueda Heurística

01

Greedy

02

A*

La heurística es una función que estima el costo o la distancia desde un estado actual hasta el objetivo en un problema de búsqueda



Heurística

La heurística no es un cálculo exacto, sino una aproximación que guía al algoritmo hacia la solución de manera más eficiente.

La calidad de la heurística influye directamente en el rendimiento del algoritmo: una buena heurística reduce el tiempo de búsqueda, mientras que una mala puede llevar a soluciones subóptimas o ineficientes.

- En mapas, se usa la distancia Euclidiana o Manhattan.
- En puzzles, se usan métricas como piezas mal colocadas o distancias.
- En problemas de optimización, se estiman costos restantes.

Búsqueda Voraz (Greedy Best-First Search)

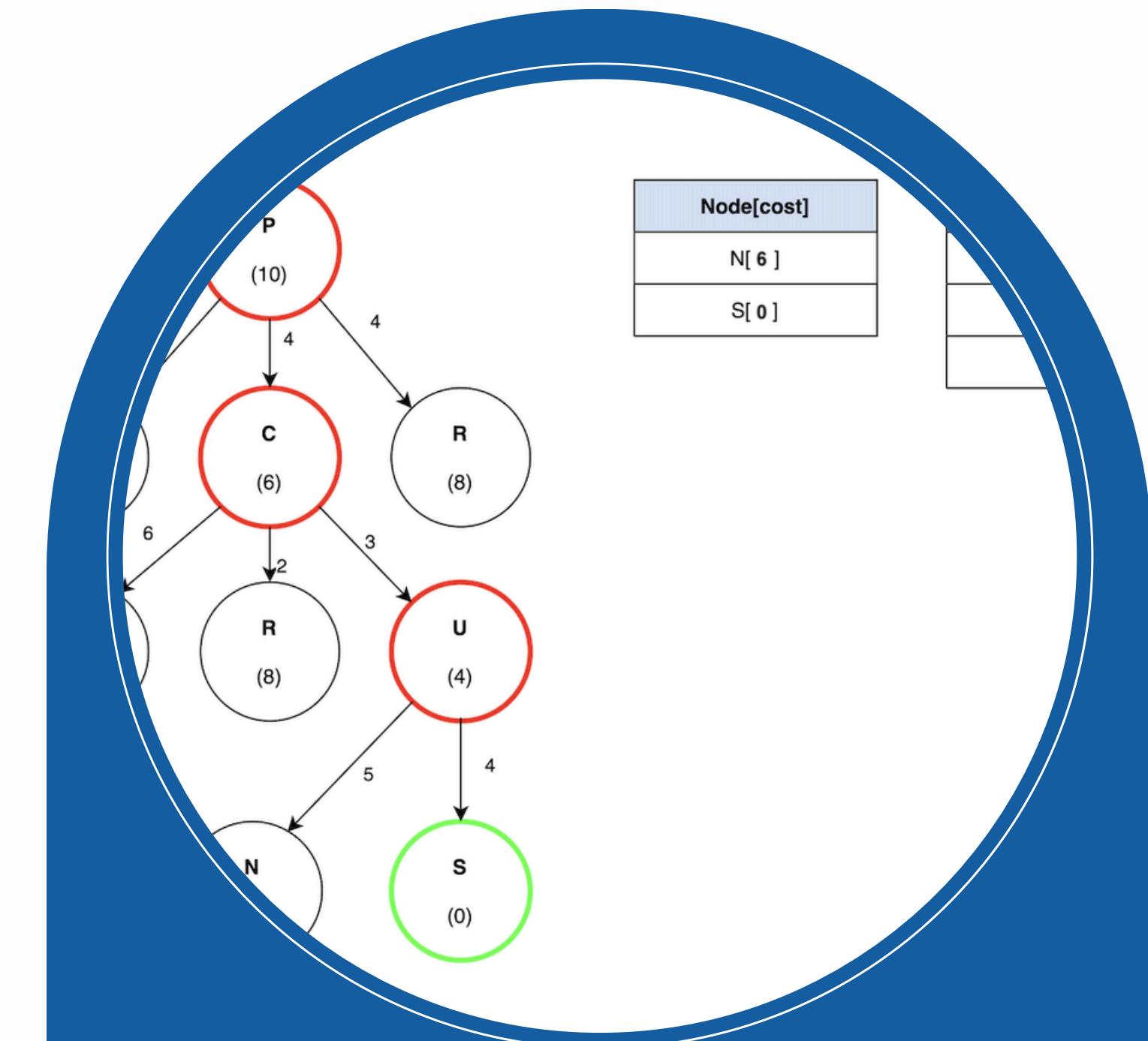
Es un algoritmo de búsqueda informada y no óptima que expande los nodos en función de una heurística que estima cuán cerca está un nodo del objetivo.

Se basa únicamente en la función de evaluación:

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

Donde:

- $h(n)$ es una heurística que estima el costo desde el nodo actual hasta el objetivo.



Es rápido y eficiente en ciertos casos, pero puede quedar atrapado en caminos sin salida o seleccionar caminos más largos que la solución óptima.

Búsqueda A* (A-Star Search)

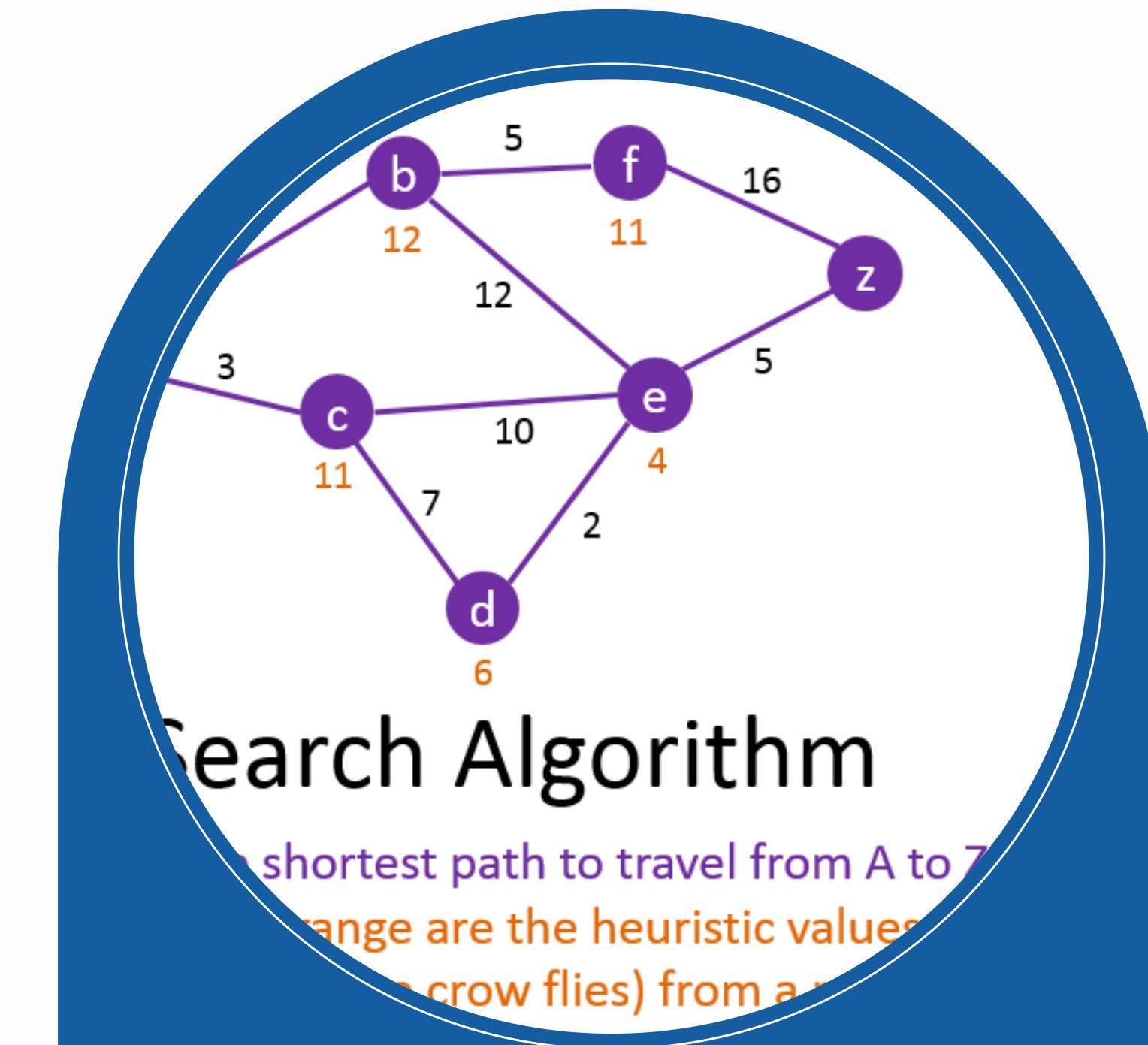
El algoritmo A* es una estrategia de búsqueda informada y óptima que combina Búsqueda de Costo Uniforme (UCS) y Búsqueda Greedy para encontrar el camino más corto hacia un objetivo.

A* evalúa cada nodo con la función de costo:

$$f(n) = g(n) + h(n)$$

Donde:

- $g(n)$ es el costo acumulado desde el nodo inicial hasta el nodo actual.
- $h(n)$ es una estimación heurística del costo desde el nodo actual hasta el objetivo.
- $f(n)$ es el costo total estimado del camino pasando por ese nodo.



A* search algorithm

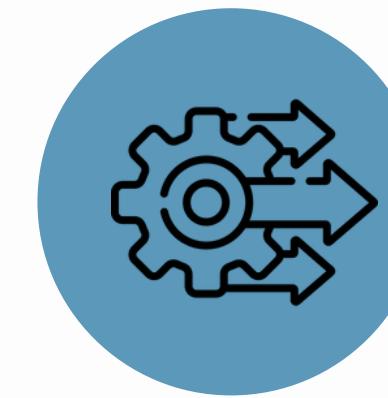
A* (pronounced "A-star") is a graph traversal and pathfinding algorithm that is used in many fields of computer science due to its completeness,...

w Wikipedia

Resumen

Característica	Greedy	A*
Estrategia	Usa solo una heurística, elige el nodo más prometedor sin considerar el costo acumulado.	Sí, siempre encuentra la mejor solución si la heurística es admisible.
¿Óptimo?	No garantiza la mejor solución.	Sí, siempre encuentra la mejor solución si la heurística es admisible.
Velocidad	Rápido en problemas simples.	Un poco más lento que Greedy pero más preciso.
Consumo de Memoria	Bajo (solo almacena nodos inmediatos).	Alto (almacena más información de costos y caminos).

¿Cuándo usar?



Greedy

Si buscas rapidez y no te importa que sea lo óptimo.



A*

Si necesitas una ruta óptima con eficiencia .

Comparaciones entre algoritmos de búsqueda

01

Algoritmos de Búsqueda Ciega

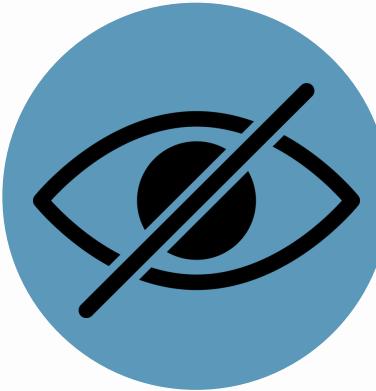
02

**Algoritmos de Búsqueda
Heurística**

Comparación

Característica	Búsqueda No Informada	Búsqueda Informada
Definición	Expande nodos en función de un criterio general como profundidad o costo.	Expande nodos basándose en una heurística que estima la distancia al objetivo.
Estrategia	Expande nodos en función de un criterio general como profundidad o costo.	Expande nodos basándose en una heurística que estima la distancia al objetivo.
¿Garantiza solución óptima?	<ul style="list-style-type: none">✓ UCS lo garantiza si los costos son positivos.✗ DFS y BFS no siempre.	<ul style="list-style-type: none">✓ A* lo garantiza si la heurística es admisible.✗ Greedy no siempre encuentra la solución óptima.
Eficiencia	Puede ser muy costosa en memoria y tiempo (ej. BFS en grafos grandes).	Generalmente más eficiente, ya que usa heurísticas para reducir la exploración innecesaria.

Cuándo usar cada Tipo de Búsqueda

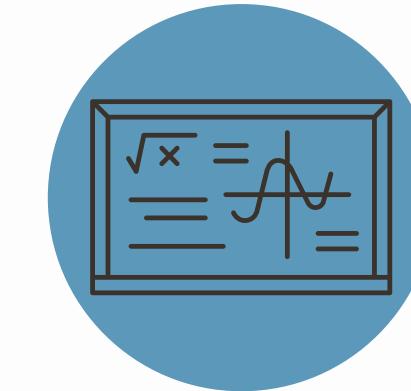


Búsqueda No Informada

Si no tienes información sobre la meta, usa búsqueda no informada.

Aplicaciones

Problemas donde no hay heurísticas disponibles, como exploración de estados desconocidos.



Búsqueda Informada

Si puedes estimar qué caminos son mejores, usa búsqueda informada para mayor eficiencia.

Aplicaciones

Problemas donde hay información previa sobre la solución, como juegos y navegación en mapas.

Aplicaciones Prácticas

01

Redes Sociales y Sistemas de Recomendación

02

Análisis de Código y Generación de Mapas

03

Rutas Óptimas en GPS y Redes de Transporte

04

Problemas en Juegos y Planeación de Movimientos

Búsqueda en Anchura

Redes sociales (Facebook, LinkedIn, Twitter, etc.)

Encontrar el número mínimo de conexiones entre dos personas en LinkedIn.

¿Cómo funciona?

- Se empieza con el usuario inicial.
- Se exploran todos sus amigos (nivel 1).
- Luego, los amigos de sus amigos (nivel 2), y así sucesivamente.
- La primera conexión encontrada es la más corta.

Búsqueda en Profundidad

Compiladores y detección de dependencias en código fuente.

Un compilador verifica si hay dependencias cíclicas en un código.

¿Cómo funciona?

- DFS explora todos los módulos hasta el final antes de retroceder.
- Si detecta un nodo ya visitado en el camino actual, hay un ciclo.
- Si no, sigue explorando otros módulos.

Búsqueda de Costo Uniforme

Solución de Problemas en Juegos y Planeación de Movimientos

Encontrar la mejor jugada en una partida de ajedrez.

¿Cómo funciona?

- Primero analiza todos los movimientos posibles en un nivel de profundidad.
- Luego aumenta el límite de profundidad y analiza más opciones.
- Se sigue repitiendo hasta que se encuentra la mejor jugada dentro del tiempo disponible.

Búsqueda de Costo Uniforme

Rutas Óptimas en GPS y Redes de Transporte

Encontrar la ruta con menor costo en tiempo o dinero.

¿Cómo funciona?

- Se comienza en la ubicación actual.
- Se evalúan todas las rutas posibles.
- Se selecciona la que tiene el menor costo acumulado (combustible, peajes, distancia).
- Se repite hasta llegar al destino.

Greedy

Sistemas GPS y videojuegos de estrategia

Un GPS que siempre elige la carretera más corta hacia el destino.

¿Cómo funciona?

- Evalúa solo la distancia actual hasta la meta.
- Selecciona el siguiente punto que parezca más cercano.
- No considera si el camino total es óptimo.

A*

Optimización de Rutas y Robótica

Un robot de entrega autónomo que encuentra la ruta más rápida evitando obstáculos.

¿Cómo funciona?

- Evalúa la distancia recorrida hasta el momento (costo real).
- También estima la distancia restante hasta la meta (heurística).
- Usa la suma de ambos valores para decidir la mejor ruta.