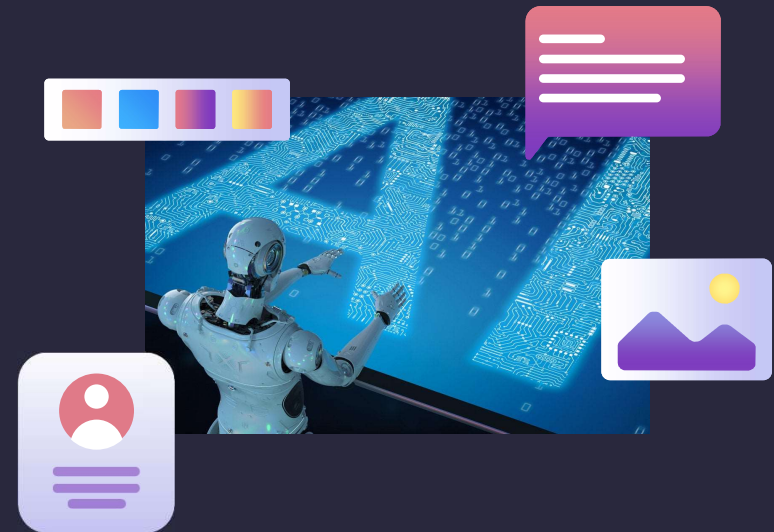




TEMA 3 – Algoritmos y ED para IA

space state search





/CONTENIDOS



- /01** /CONCEPTOS
- /02** /ALGORITMOS PARA LA EXPLORACIÓN
- /03** /EL BACKTRACKING
- /04** /EJEMPLOS DE BACKTRACKING
- /05** /BÚSQUEDAS CON HEURISTICAS. EL ALGORITMO A*
- /06** /TEORÍA DE JUEGOS



/01 / Conceptos

Exploración en espacios de estados

Proceso usado en la **inteligencia artificial**, en el que se consideran estados sucesivos de un problema con la intención de encontrar una solución con una determinada propiedad

$$\text{espacio} = \langle S, A, \text{Accion}(s), \text{Resultado}(a, s), \text{Coste}(a, s) \rangle$$

S = Conjunto de estados

A = Conjunto de acciones

$\text{Accion}(s)$ = ¿Qué acción se puede ejecutar sobre qué estado?

$\text{Resultado}(a, s)$ = Retorna el estado alcanzado al efectuar la acción a sobre el estado s

$\text{Coste}(a, s)$ = ¿Cuánto cuesta realizar la acción a sobre el estado s ?

USOS:

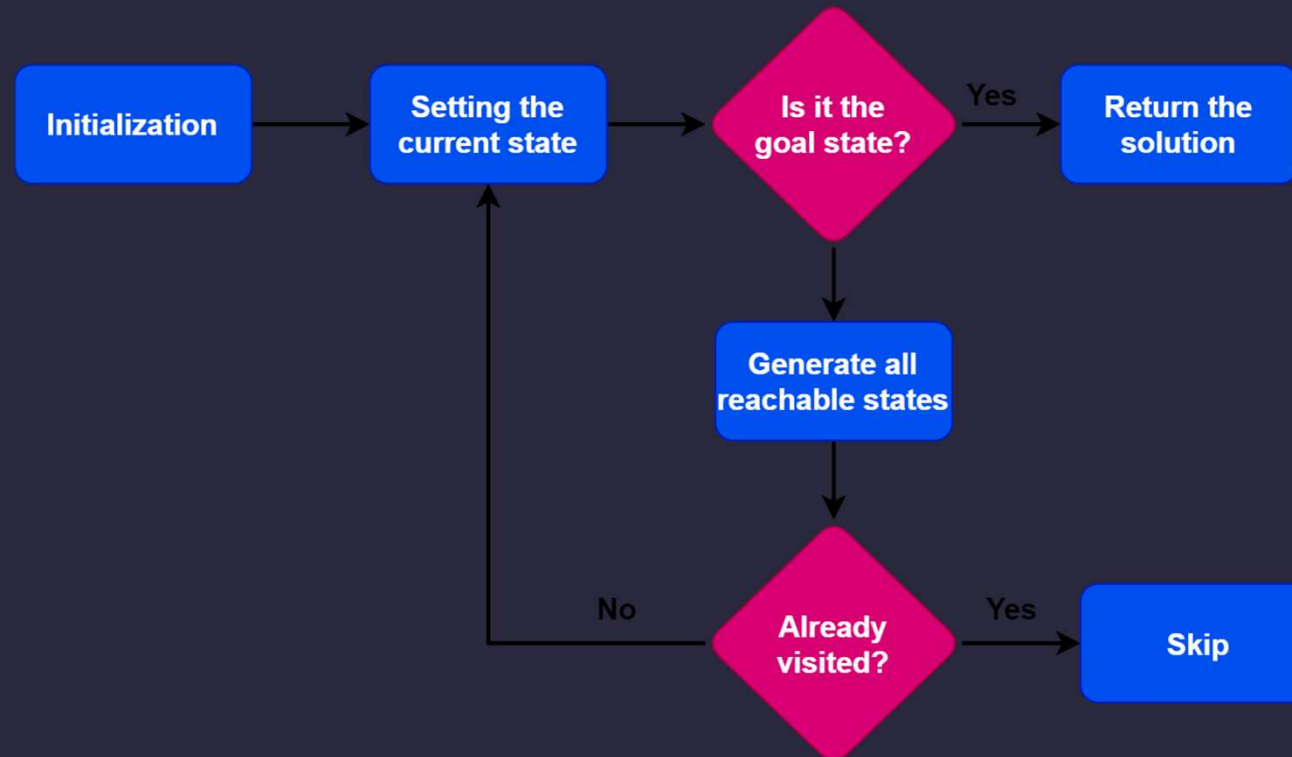
Teoría de juegos

Profundización iterativa

Estructura de datos y procedimientos de búsqueda genéricas

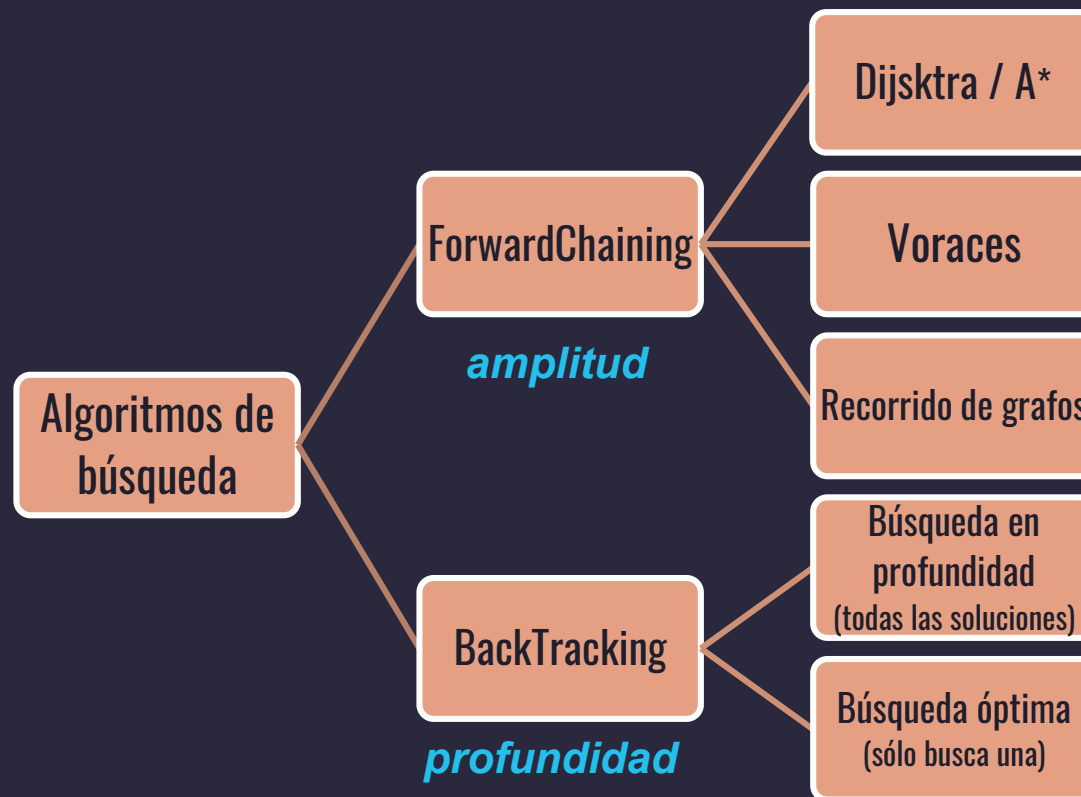
Resolución de problemas de combinatoria

/02 / ALGORITMOS PARA LA EXPLORACIÓN



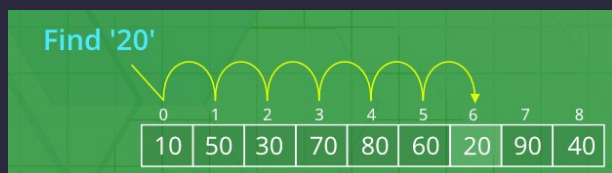
Existen diferentes métodos para **visitar** todos los estados y **heurísticas** para evitar tener que visitarlos todos

/02 /ALGORITMOS PARA LA EXPLORACIÓN



Algoritmos de búsqueda

ALGORITMICA CLÁSICA

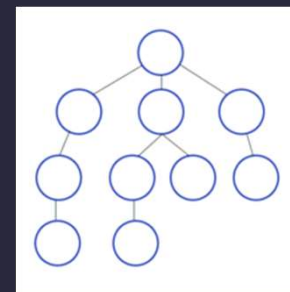


Búsqueda lineal

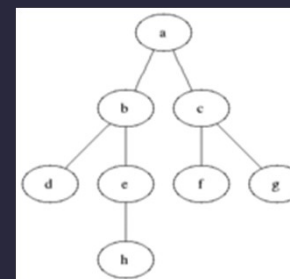
	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91
23 > 16 take 2 nd half	L=0	1	2	3	M=4	5	6	7	8	H=9
	2	5	8	12	16	23	38	56	72	91
23 > 56 take 1 st half	0	1	2	3	4	L=5	6	M=7	8	H=9
	2	5	8	12	16	23	38	56	72	91
Found 23, Return 5	0	1	2	3	4	L=5, M=5	H=6	7	8	9
	2	5	8	12	16	23	38	56	72	91

Búsqueda binaria

INTELIGENCIA ARTIFICIAL



Búsqueda en profundidad



Búsqueda en amplitud

Las estructuras de datos más importantes en inteligencia artificial

Tensores

en cualquiera de sus formas: vectores, matrices, obj.multidim.
y en cualquiera de sus implementaciones: listas, pilas, colas...

Dataframes

Árboles

Grafos



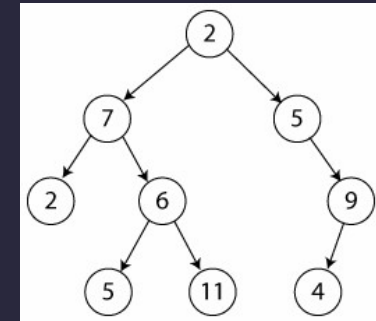
Definición de árbol

a : nulo

$\text{valor_nodo} + \text{lista}[a]$

Algunas definiciones:

- ❑ Hijo vs Padre
- ❑ Nodo hoja vs nodo no terminal
- ❑ Profundidad vs Amplitud
- ❑ Descendiente vs Ancestro



Ejercicio:
*Con la clase Tree, programa
un recorrido en profundidad y
otro en amplitud*

```
class Tree:  
    def __init__(self, data):  
        self.children = []  
        self.data = data
```

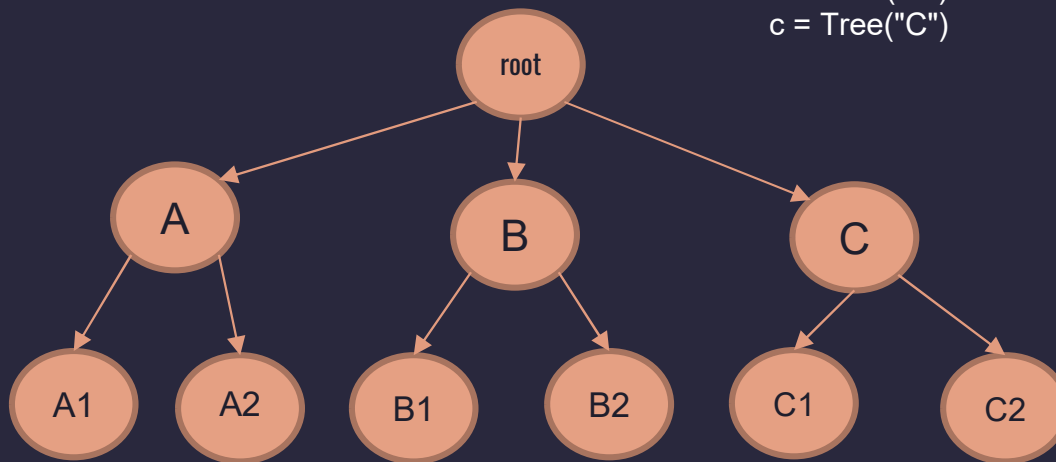
```
a = Tree("A")  
b = Tree("B")  
c = Tree("C")
```

```
root = Tree("root")  
root.children.append(a)  
root.children.append(b)  
root.children.append(c)
```

```
a1=Tree("A1")  
a2=Tree("A2")  
root.children[0].children.append(a1)  
root.children[0].children.append(a2)
```

```
b1=Tree("B1")  
b2=Tree("B2")  
root.children[1].children.append(b1)  
root.children[1].children.append(b2)
```

```
c1=Tree("C1")  
c2=Tree("C2")  
root.children[2].children.append(c1)  
root.children[2].children.append(c2)
```



/03 /EL BACKTRACKING

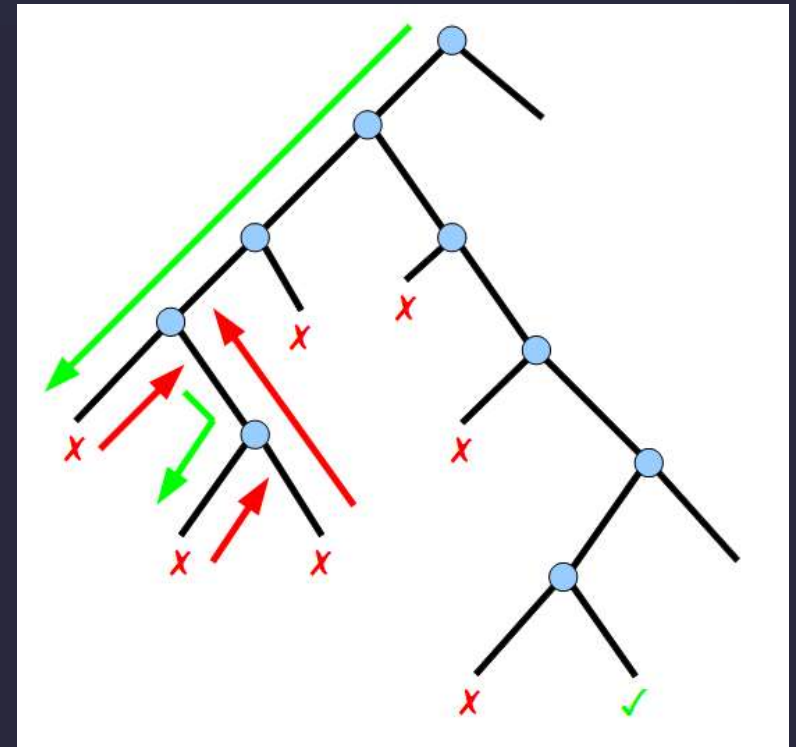
La técnica de **backtracking** nos permite explotar combinatoriamente todas las posibilidades que tenemos para buscar una o varias soluciones a un problema.

Los problemas pueden tener, o no tener, una solución viable

Si tienen varias soluciones, podemos usar una función de coste para evaluar cuál es la mejor solución

Aprovecha la pila de la recursividad para generar **árboles de exploración**. **Si no hay éxito buscando una solución, se vuelve al paso anterior para probar otra combinación.**

Sin el uso de heurísticas (poda) puede resultar un ejercicio de **FUERZA BRUTA**



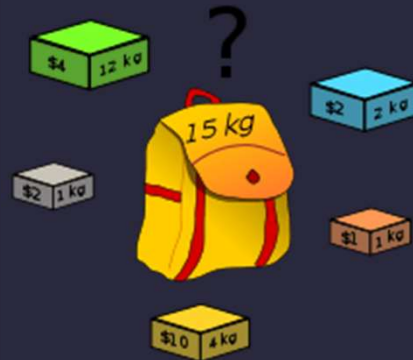
/03 /EL BACKTRACKING

Problemas de
combinatoria



El problema de las
nueve reinas

Problemas de
optimización



El problema de las
mochila

Problemas de
fuerza bruta

5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Resolución de
problemas
matemáticos

Problemas de
fuerza bruta
con heurística



Búsqueda de rutas
con vuelta atrás



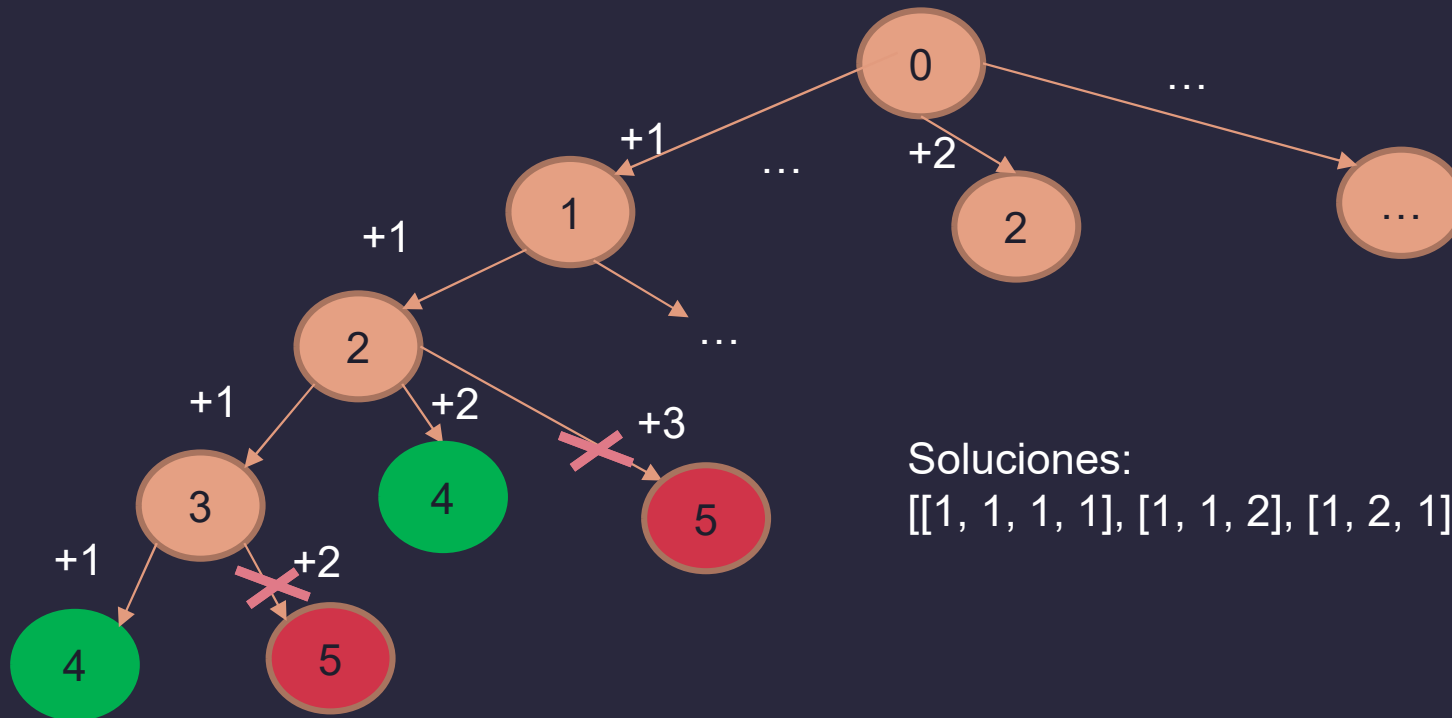
/03 /EL BACKTRACKING ESQUEMA SIMPLE

```
backtracking(problema,opciones,paso):  
    exito=false  
    opciones=opciones_aceptables(problema)  
    Mientras len(opciones)>0 y no exito  
        opcion=extraer_siguiete(opciones)  
        si esOpcionAceptable(problema,paso,opcion) entonces  
            anotar(problema,opcion)  
            si solucionCompleta(problema,paso,opcion) entonces  
                exito=true  
            sino  
                exito=backtracking(problema,opciones,paso+1)  
                si not exito entonces  
                    desanota(problema,opcion)  
            fin-si  
        fin-si  
    fin-si  
fin-mientras  
retorna exito
```



/04 / EJEMPLOS DE BACKTRACKING

¿Cuántas posibilidades tenemos de sumar un número entero positivo en concreto?
Por ejemplo... el 4. Completa el árbol de posibilidades...



Soluciones:

[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1,3], [2, 1, 1], [2, 2], [3,1]]



SOLUCIÓN

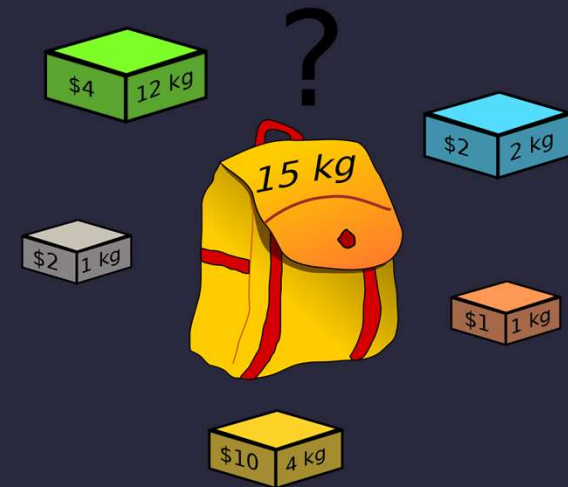
```
def combinaciones_suma(target, solucion_parcial, primera_opcion, soluciones):  
    exito=False  
    opciones=list(range(primer_opcion,target))  
    i=0  
    while i<len(opciones):  
        total=opciones[i]+sum(solucion_parcial)  
        if total <= target: #opcion aceptable?  
            solucion_parcial.append(opciones[i])  
            if total==target: #es solución?  
                soluciones.append(solucion_parcial.copy())  
                exito=True  
            else:  
                exito=combinaciones_suma(target,solucion_parcial,primera_opcion,soluciones)  
                solucion_parcial.pop() #desanotar  
        else:  
            exito=False  
            break; #heurística: nos pasamos de cantidad, no hace falta seguir comprobando el resto de opciones  
        i+=1  
  
    return exito
```



/04 / EJEMPLOS DE BACKTRACKING

El problema de la mochila:

¿Dada una mochila con un peso máximo (peso_max) qué combinación de elementos (precio, peso) maximiza el valor total de la mochila?





PRÁCTICA 3

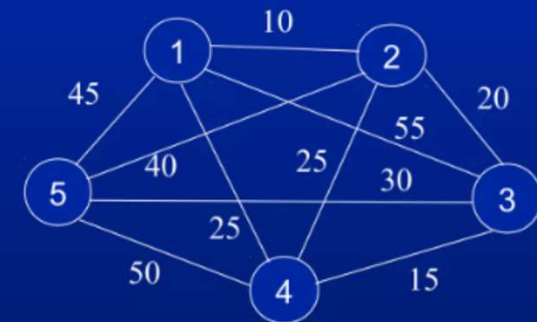
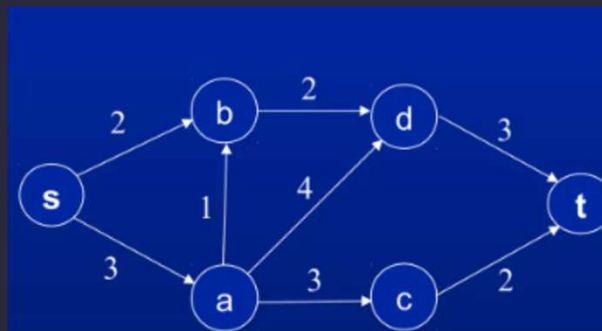
RESOLUCIÓN DE SUDOKUS MEDIANTE BACKTRACKING




/05 /BÚSQUEDAS CON HEURISTICAS. EL ALGORITMO A*

□ Concepto de grafo
 $G(\text{Vertices}, \text{Aristas})$

□ Tipos de grafos: Ponderados, dirigidos...



/grafos: clase Node

 graph.py

- ❑ Examina cómo se construye un nodo
- ❑ Examina los siguientes atributos de la clase **Node**.

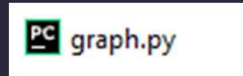
```
def __init__(self, value, coordinates, neighbors=None):
```

```
    value : str  
    x : int  
    y : int  
    heuristic_value : int  
    distance_from_start  
    neighbors : list  
    parent : Node
```

- ❑ Examina los siguientes métodos de la clase **Node**.

```
has_neighbors(self) -> Boolean  
number_of_neighbors(self) -> int  
add_neighbor(self, neighbor) -> None  
extend_node(self) -> list  
__eq__(self, other) -> Boolean  
__str__(self) -> str
```

/grafos: clase Graph



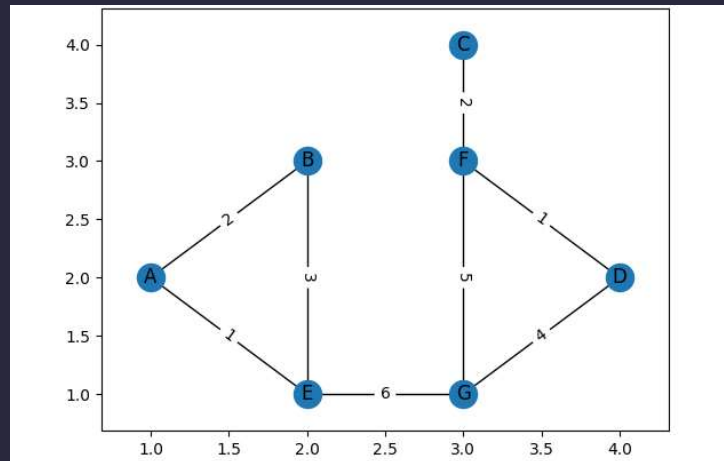
- ❑ Examina cómo se construye un grafo
`def __init__(self, nodes=None):`
- ❑ Examina el atributo *nodes* de la clase **Graph**.
- ❑ Examina los siguientes métodos de la clase **Graph**.

```
add_node(self, node) -> None  
find_node(self, value) -> Node  
add_edge(self, value1, value2, weight=1) -> None  
number_of_nodes(self) -> int  
are_connected(self, node_one, node_two) -> Boolean  
__str__(self) -> str
```

/grafos: Ejercicio con la clase Graph

 graph.py

- Con la clase `graph.py`, crea un programa que represente este grafo:



- Realiza un programa para recorrer el grafo y mostrarlo en un gráfico de `matplotlib`



GRAFOS CON NetworkX

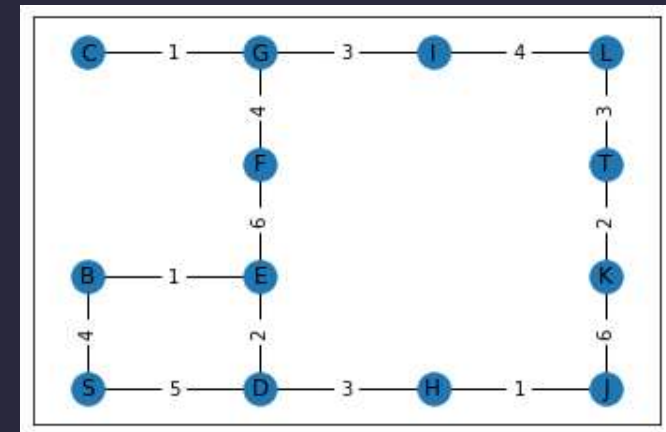
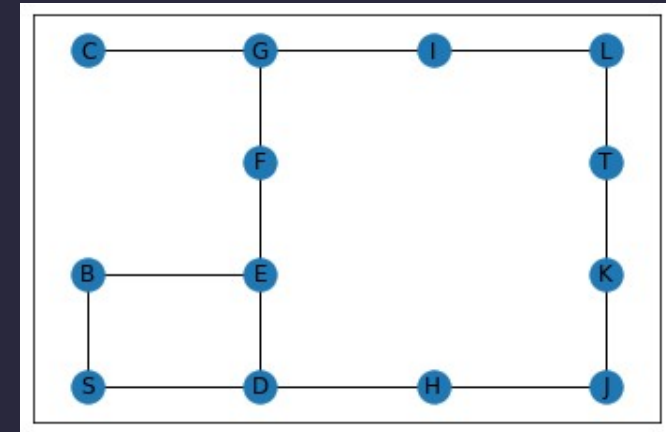


```
import networkx as nx
g = nx.Graph()

# Add vertices
g.add_node('S', pos=(1,1))
g.add_node('B', pos=(1,2))
....
g.add_edge('S', 'B', lenght=4)
g.add_edge('S', 'D', lenght=5)
g.add_edge('B', 'E', lenght=1)
...

labels = nx.get_edge_attributes(g,'lenght')
pos=nx.get_node_attributes(g,'pos')

print("etiquetas",labels)
print("Posiciones",pos)
nx.draw_networkx(g, pos,with_labels=True)
nx.draw_networkx_edge_labels(g, pos,edge_labels=labels)
```



/05 /BÚSQUEDAS CON HEURISTICAS. EL ALGORITMO A*

- ❑ Dado un grafo con un *mapa* busca el camino de un origen a un destino.
- ❑ Es un algoritmo de Forward Chaining, explora en amplitud pero no da “vuelta atrás”
- ❑ Utiliza una función heurística para estimar el coste de cada posible origen al objetivo
- ❑ El camino será óptimo si el valor de la función heurística siempre es menor que el valor real del coste al objetivo. Es decir, la heurística siempre *subestima*: *Ejemplo: Tardo 1h en llegar a Albacete y son 100 kilómetros de distancia.*
- ❑ Utiliza dos listas:
 - ABIERTA: Contiene los nodos candidatos a formar parte de la ruta
 - CERRADA: Contiene los nodos que han sido seleccionados

/05 /EI ALGORITMO A*

❑ Inicialización:

Se calcula la heurística del nodo origen y se añade a la lista ABIERTA

❑ Búsqueda:

Mientras no tengamos éxito

Si la lista ABIERTA está vacía

break #<no hay solución.>

nodo_seleccionado=nodo con menor heurística de la lista ABIERTA

Si nodo_seleccionado==destino

retornar ruta

añadir nodo_seleccionado a lista CERRADA

por cada nodo_hijo en la lista de posibles destinos desde el nodo seleccionado

calcular la nueva heurística y actualizar padre y distancia si es mejor que la anterior

si el nodo_hijo no está ni en la lista ABIERTA ni en la CERRADA

nodo_hijo.padre=nodo_seleccionado

añadir nodo_hijo a la lista ABIERTA

EJEMPLO DE EJECUCIÓN DEL A*

	0	1	2	3
0	S			
1				
2				
3			T	


EJEMPLO DE EJECUCIÓN DEL A*

S=start=inicio

T=target=objetivo

Heurística=distancia de manhattan de un punto (x1,y1) a otro (x2,y2)

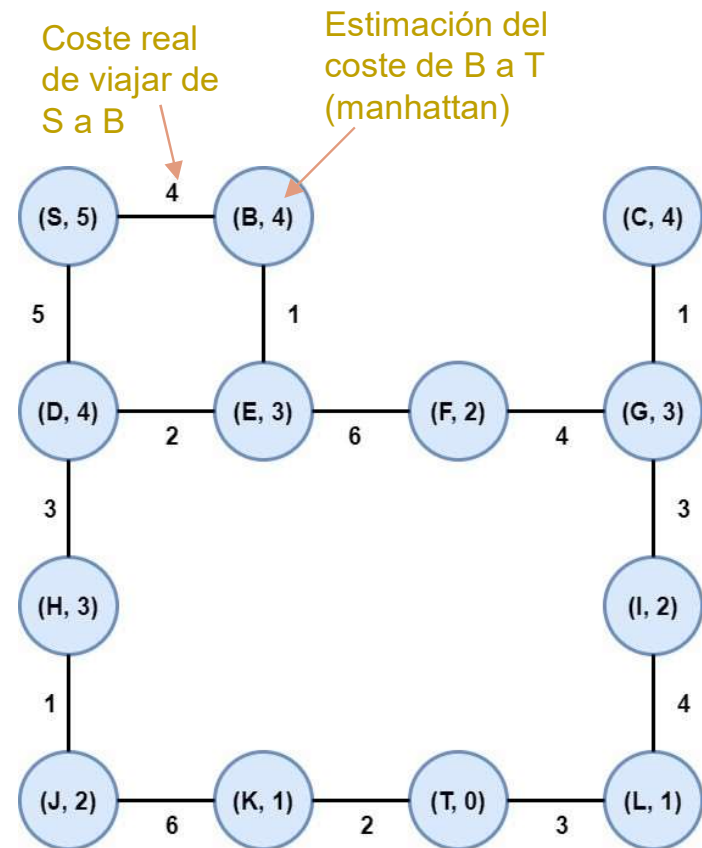
$$\text{manhattan}((x1, y1), (x2, y2)) = |x1 - x2| + |y1 - y2|$$



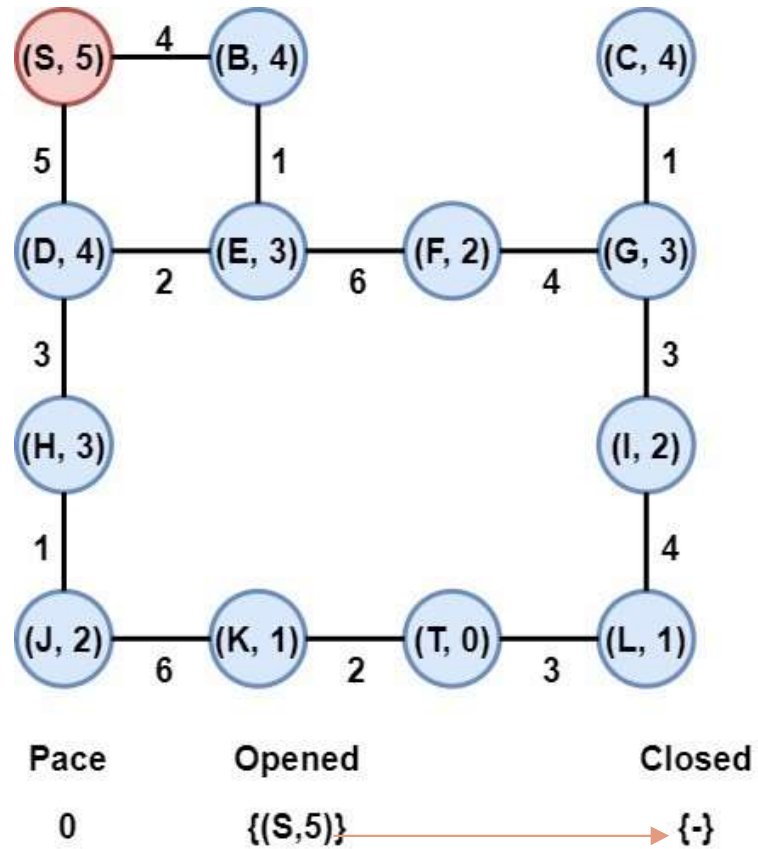
	0	1	2	3
0	5	4		4
1	4	3	2	3
2	3			2
3	2	1	T	1

CONVERTIMOS EL MAPA EN UN GRAFO:

	0	1	2	3
0	5	4		4
1	4	3	2	3
2	3			2
3	2	1	T	1

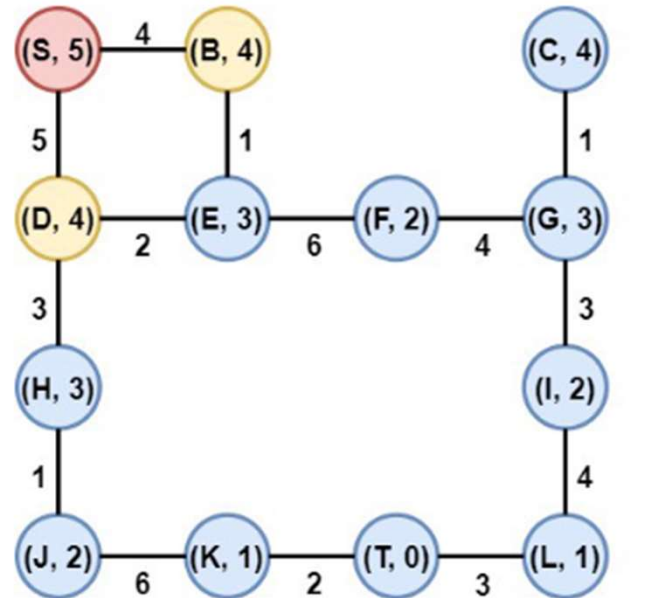


INICIALIZACIÓN:

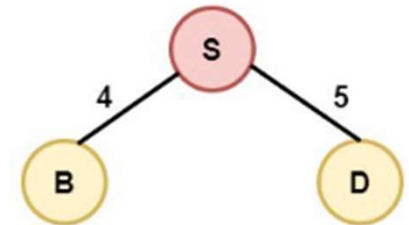


ITERACIÓN 1:

Se calcula la heurística de los hijos de S [B y D] y se añaden a la lista abierta

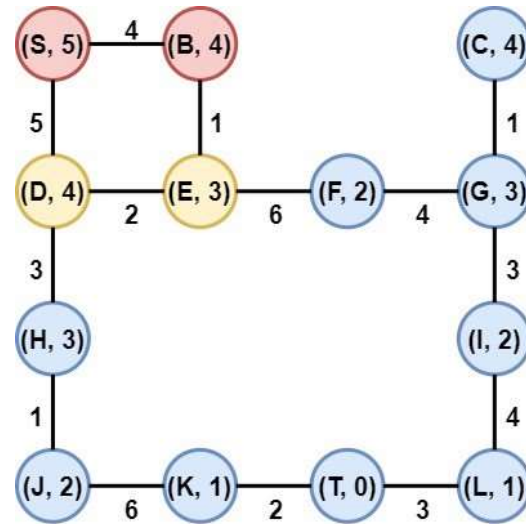


Pase	Opened	Closed
0	{(S,5)}	{-}
1	{(B,8),(D,9)}	{(S,5)}

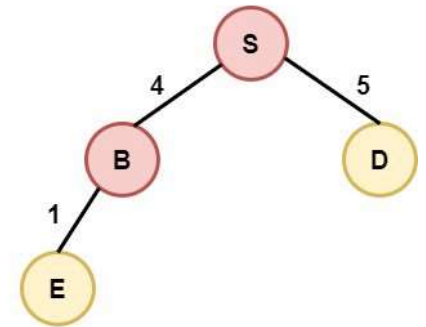


ITERACIÓN 2:

Se selecciona B con
heurística 8, que se
mete la lista CERRADA.
Se expande E y se
calcula su heurística



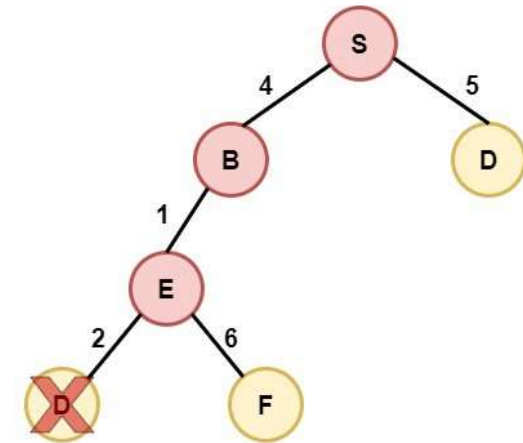
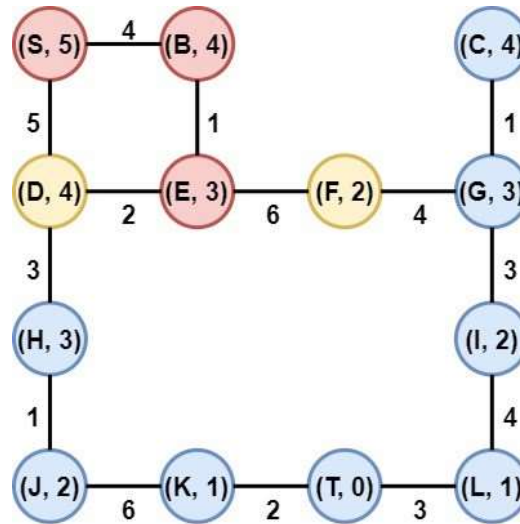
Pase	Opened	Closed
0	{{(S,5)}}	{-}
1	{{(B,8),(D,9)}}	{{(S,5)}}
2	{{(E,8),(D,9)}}	{{(S,5),(B,8)}}



ITERACIÓN 3:

Se selecciona *E* con heurística 8, que se mete la lista CERRADA. Se expanden sus hijos *D* y *F* con heurísticas 11 y 13.

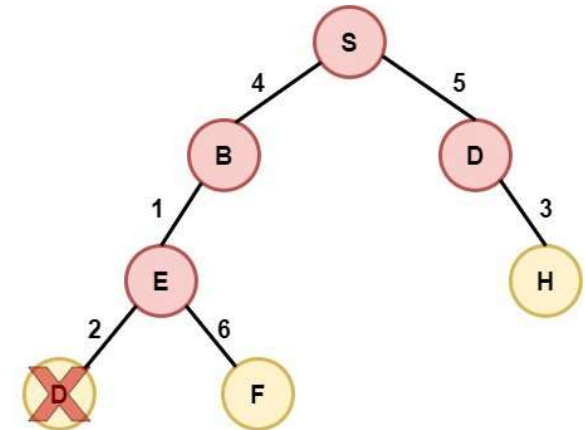
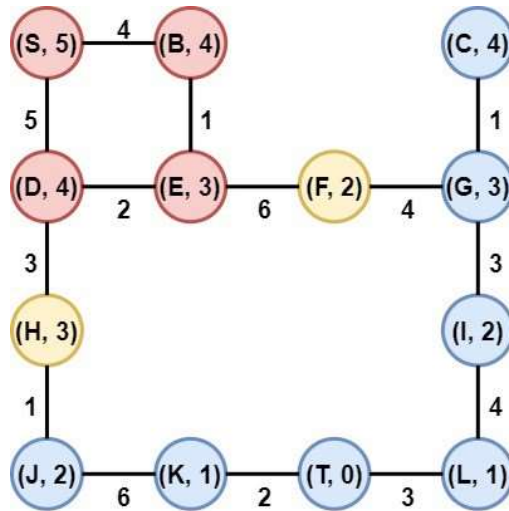
D ya está en la lista abierta con heurística 9 (menor que 11), se descarta como ruta



Pase	Opened	Closed
0	{{(S,5)}}	{-}
1	{{(B,8),(D,9)}}	{{(S,5)}}
2	{{(E,8),(D,9)}}	{{(S,5),(B,8)}}
3	{{(F,13),(D,9)}}	{{(S,5),(B,8),(E,8)}}

ITERACIÓN 4:

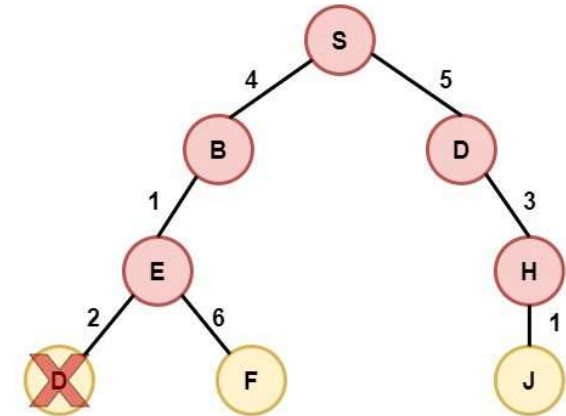
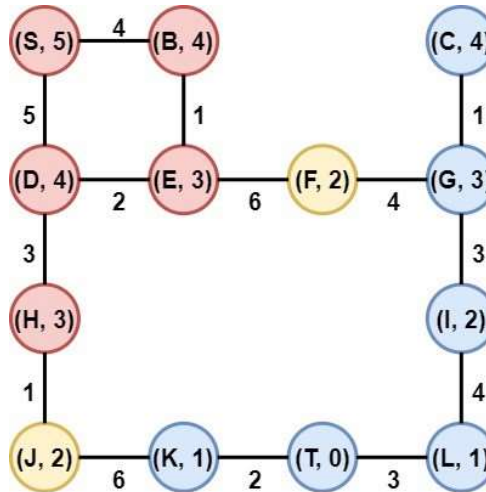
Se selecciona D con heurística 9, que se mete la lista CERRADA. Se expanden sus hijos H (heurística 11) y E, *que ya está en la lista cerrada, se descarta*



Pase	Opened	Closed
0	{(S,5)}	{-}
1	{(B,8),(D,9)}	{(S,5)}
2	{(E,8),(D,9)}	{(S,5),(B,8)}
3	{(F,13),(D,9)}	{(S,5),(B,8),(E,8)}
4	{(F,13),(H,11)}	{(S,5),(B,8),(E,8),(D,9)}

ITERACIÓN 5:

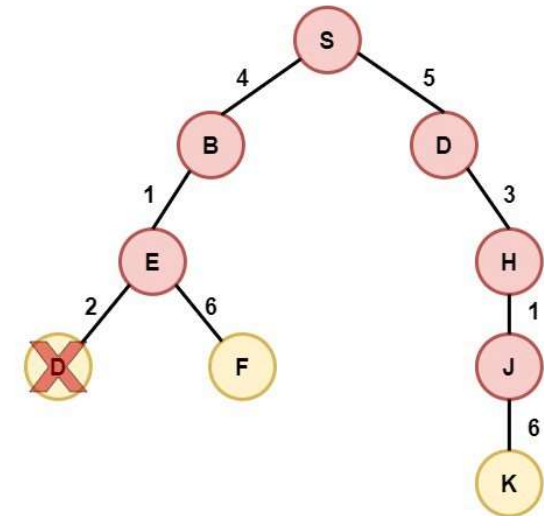
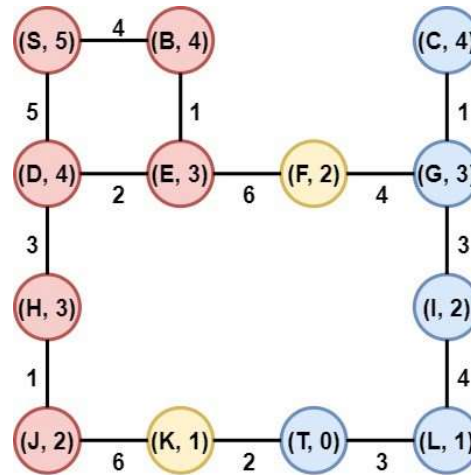
Se selecciona H con heurística 11, que se mete la lista CERRADA. Se expande su hijo J (heurística 11)



Pase	Opened	Closed
0	{(S,5)}	{-}
1	{(B,8),(D,9)}	{(S,5)}
2	{(E,8),(D,9)}	{(S,5),(B,8)}
3	{(F,13),(D,9)}	{(S,5),(B,8),(E,8)}
4	{(F,13),(H,11)}	{(S,5),(B,8),(E,8),(D,9)}
5	{(F,13),(J,11)}	{(S,5),(B,8),(E,8),(D,9),(H,11)}

ITERACIÓN 6:

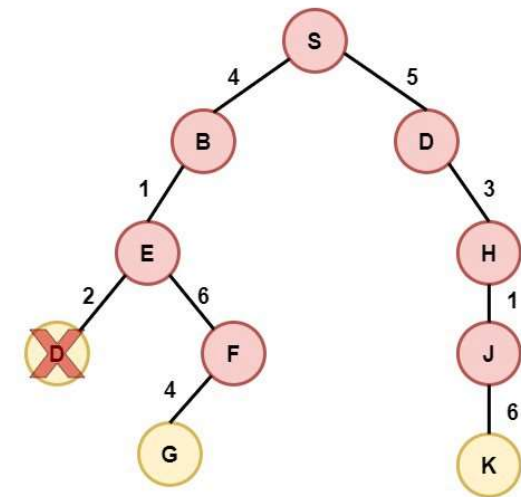
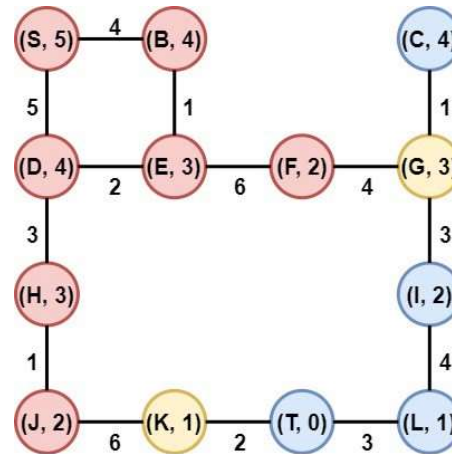
Se selecciona J con
heurística 11, que se mete
la lista CERRADA. Se
expande su hijo K
(heurística 16)



Pase	Opened	Closed
0	{(S,5)}	{-}
1	{(B,8),(D,9)}	{(S,5)}
2	{(E,8),(D,9)}	{(S,5),(B,8)}
3	{(F,13),(D,9)}	{(S,5),(B,8),(E,8)}
4	{(F,13),(H,11)}	{(S,5),(B,8),(E,8),(D,9)}
5	{(F,13),(J,11)}	{(S,5),(B,8),(E,8),(D,9),(H,11)}
6	{(F,13),(K,16)}	{(S,5),(B,8),(E,8),(D,9),(H,11),(J,11)}

ITERACIÓN 7:

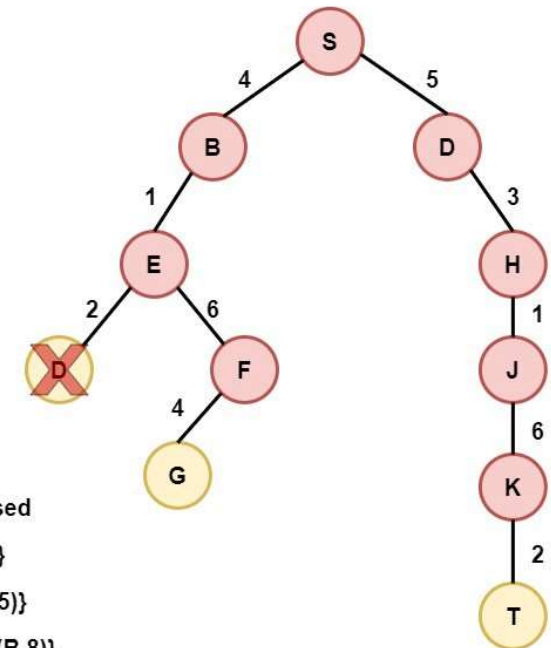
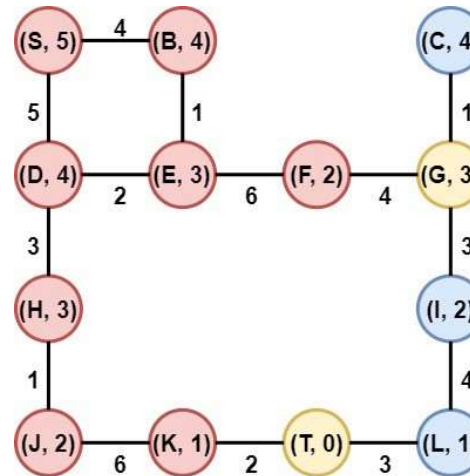
Se selecciona *F* con heurística 13, que se mete la lista CERRADA. Se expande su hijo *G*, (heurística 18)



Pase	Opened	Closed
0	{(S,5)}	{-}
1	{(B,8),(D,9)}	{(S,5)}
2	{(E,8),(D,9)}	{(S,5),(B,8)}
3	{(F,13),(D,9)}	{(S,5),(B,8),(E,8)}
4	{(F,13),(H,11)}	{(S,5),(B,8),(E,8),(D,9)}
5	{(F,13),(J,11)}	{(S,5),(B,8),(E,8),(D,9),(H,11)}
6	{(F,13),(K,16)}	{(S,5),(B,8),(E,8),(D,9),(H,11),(J,11)}
7	{(K,16),(G,18)}	{(S,5),(B,8),(E,8),(D,9),(H,11),(J,11),(F,13)}

ITERACIÓN 8:

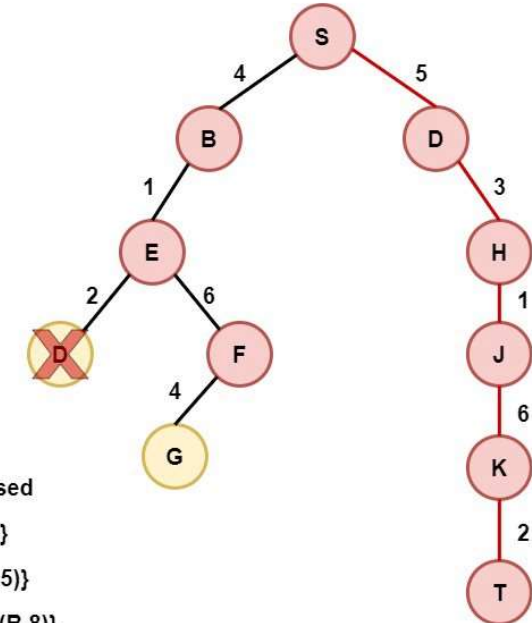
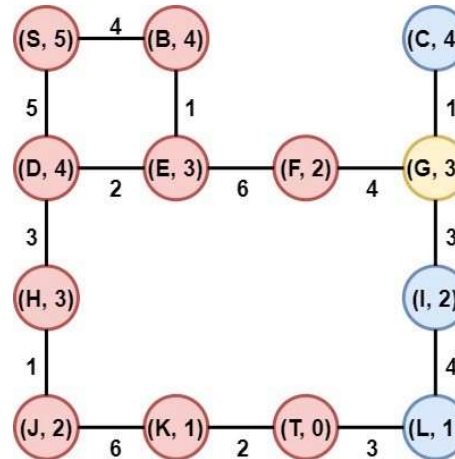
Se selecciona K con heurística 16, que se mete la lista CERRADA. Se expande su hijo T (objetivo), (heurística 17)



Pase	Opened	Closed
0	{{(S,5)}}	{-}
1	{{(B,8),(D,9)}}	{{(S,5)}}
2	{{(E,8),(D,9)}}	{{(S,5),(B,8)}}
3	{{(F,13),(D,9)}}	{{(S,5),(B,8),(E,8)}}
4	{{(F,13),(H,11)}}	{{(S,5),(B,8),(E,8),(D,9)}}
5	{{(F,13),(J,11)}}	{{(S,5),(B,8),(E,8),(D,9),(H,11)}}
6	{{(F,13),(K,16)}}	{{(S,5),(B,8),(E,8),(D,9),(H,11),(J,11)}}
7	{{(K,16),(G,18)}}	{{(S,5),(B,8),(E,8),(D,9),(H,11),(J,11),(F,13)}}
8	{{(T,17),(G,18)}}	{{(S,5),(B,8),(E,8),(D,9),(H,11),(J,11),(F,13),(K,16)}}

ITERACIÓN 9:

Se selecciona T con heurística 17, que es **OBJETIVO**. Se retorna la ruta de S a T junto con su coste total.



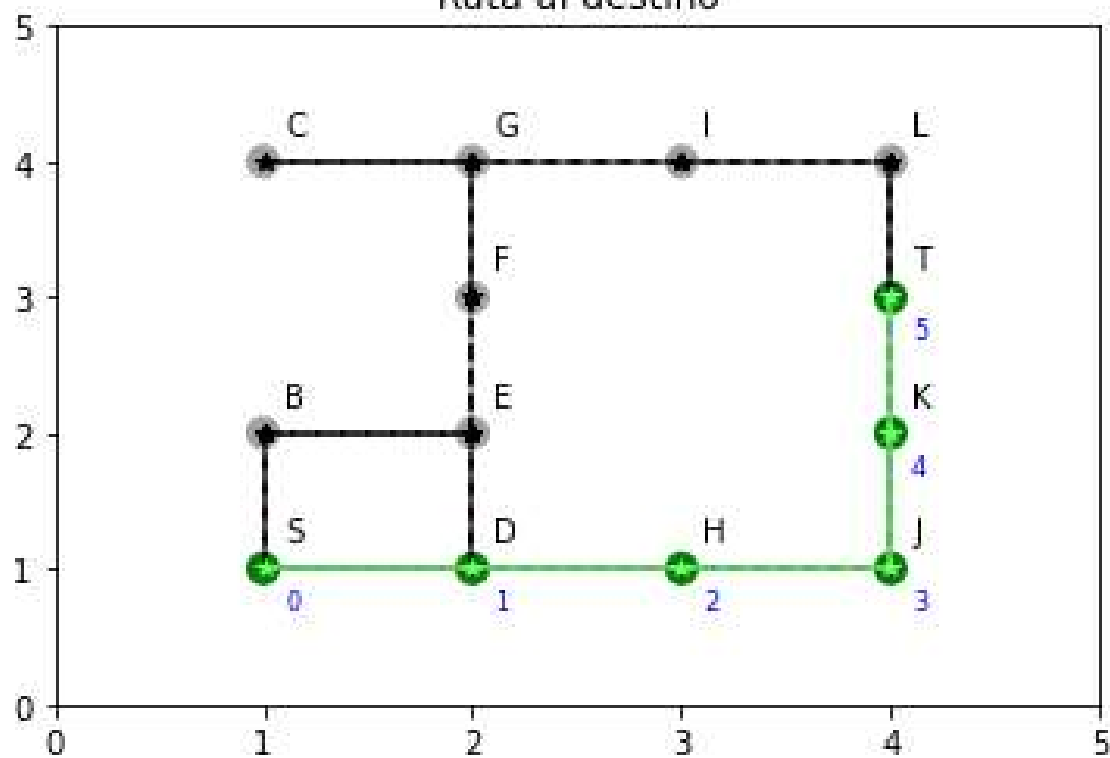
Pase	Opened	Closed
0	{(S,5)}	{-}
1	{(B,8),(D,9)}	{(S,5)}
2	{(E,8),(D,9)}	{(S,5),(B,8)}
3	{(F,13),(D,9)}	{(S,5),(B,8),(E,8)}
4	{(F,13),(H,11)}	{(S,5),(B,8),(E,8),(D,9)}
5	{(F,13),(J,11)}	{(S,5),(B,8),(E,8),(D,9),(H,11)}
6	{(F,13),(K,16)}	{(S,5),(B,8),(E,8),(D,9),(H,11),(J,11)}
7	{(K,16),(G,18)}	{(S,5),(B,8),(E,8),(D,9),(H,11),(J,11),(F,13)}
8	{(T,17),(G,18)}	{(S,5),(B,8),(E,8),(D,9),(H,11),(J,11),(F,13),(K,16)}
9	{(G,18)}	{(S,5),(B,8),(E,8),(D,9),(H,11),(J,11),(F,13),(K,16),(T,17)}

Path : S - D - H - J - K - T

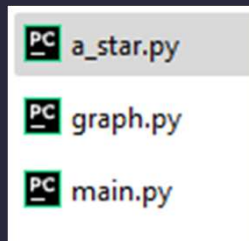
Total Cost: 17



Ruta al destino



Ejemplo de ejecución
Código en Python:

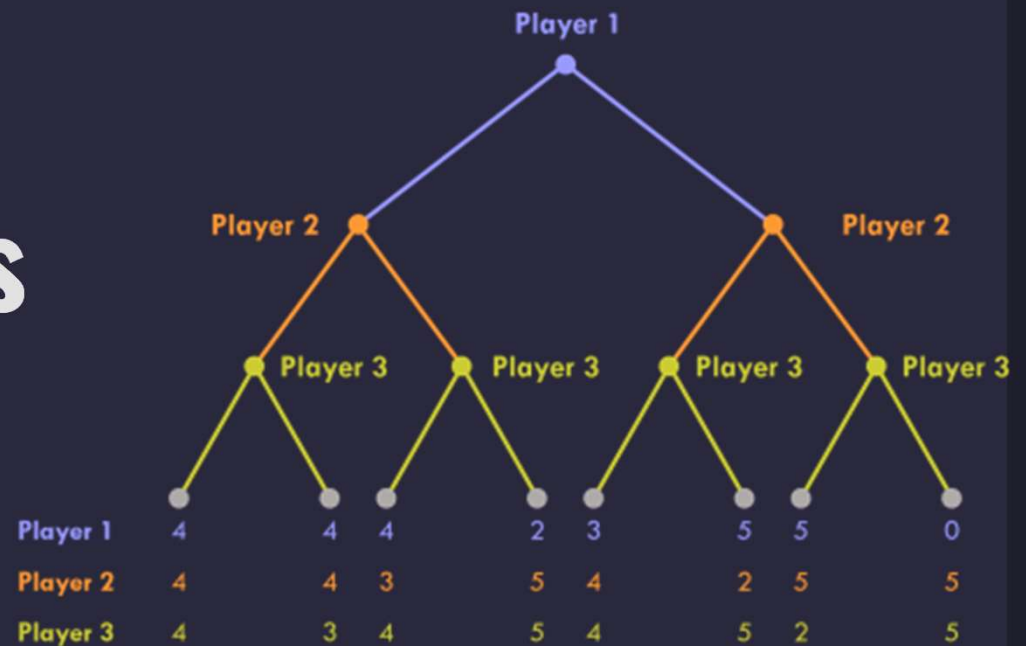


```
alg = AStar(graph, "S", "T")  
path, path_length = alg.search()  
print(" -> ".join(path))  
print(f"Length of the path: {path_length}")
```

/06

Teoría de juegos

... o el estudio de modelos matemáticos para interacciones estratégicas con agentes racionales



Dinámicas de juegos con I.A.

- ❑ **Simetría.** ¿Los jugadores tienen el mismo objetivo? Ajedrez vs Wolves-Rabbit
- ❑ **Información perfecta o imperfecta.** ¿Se conoce toda la información del juego o sólo parte? Ajedrez vs Black Jack
- ❑ **Simultanea o secuencial.** ¿Cada jugador sabe las acciones del otro? Piedra-Papel-Tijera vs Ajedrez
- ❑ **Cooperativa.** ¿Los jugadores pueden formar alianzas? Mus vs Poker
- ❑ **Zero vs Non-Zero sum.** ¿Las pérdidas de uno son las ganancias de otro? Ajedrez vs Mercado de valores



Un caso de éxito: Deep Blue

Árbol de Juego

Evaluación de Posiciones

Minimax y Poda Alfa-Beta

Preparación de Aperturas y Finales



MINIMAX: Algoritmo de inteligencia artificial para la toma de decisiones en juegos a través de backtracking

En teoría de juegos se utiliza para encontrar el movimiento óptimo para un jugador, asumiendo que el rival también ejecutará su movimiento óptimo

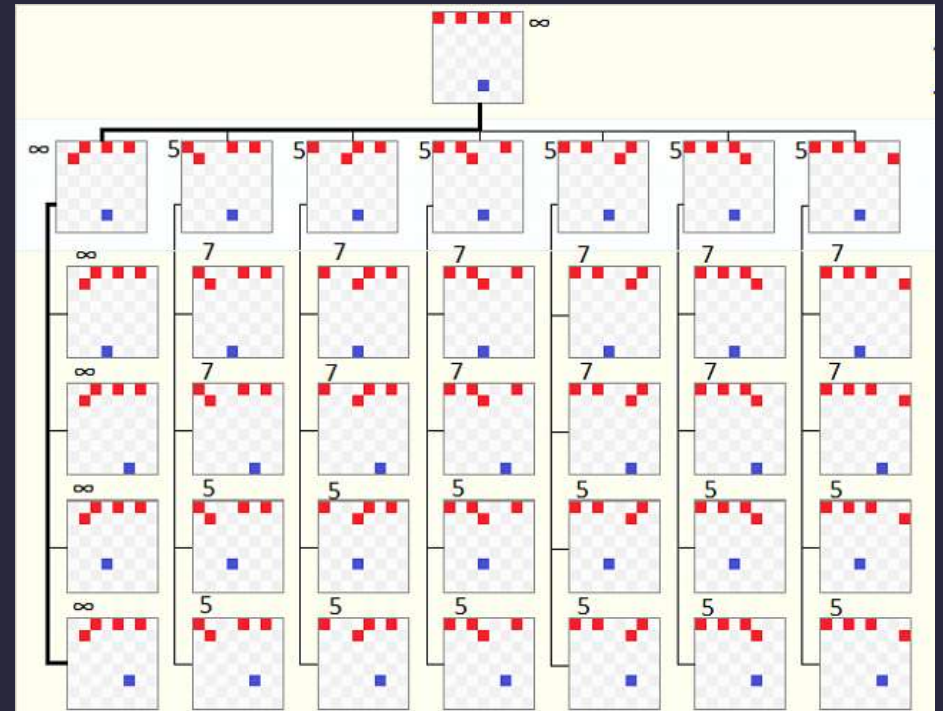
Ideal para juegos de tipo **zero-sum**, **sequential** para dos jugadores como el ajedrez, backgammon, Tic-Tac-Toe, etc.

Con un poco de pericia, se puede extrapolar a cualquier juego multijugador con número de jugadores > 2 (yo contra el resto) y a cualquier juego de tipo **Non-zero-sum**

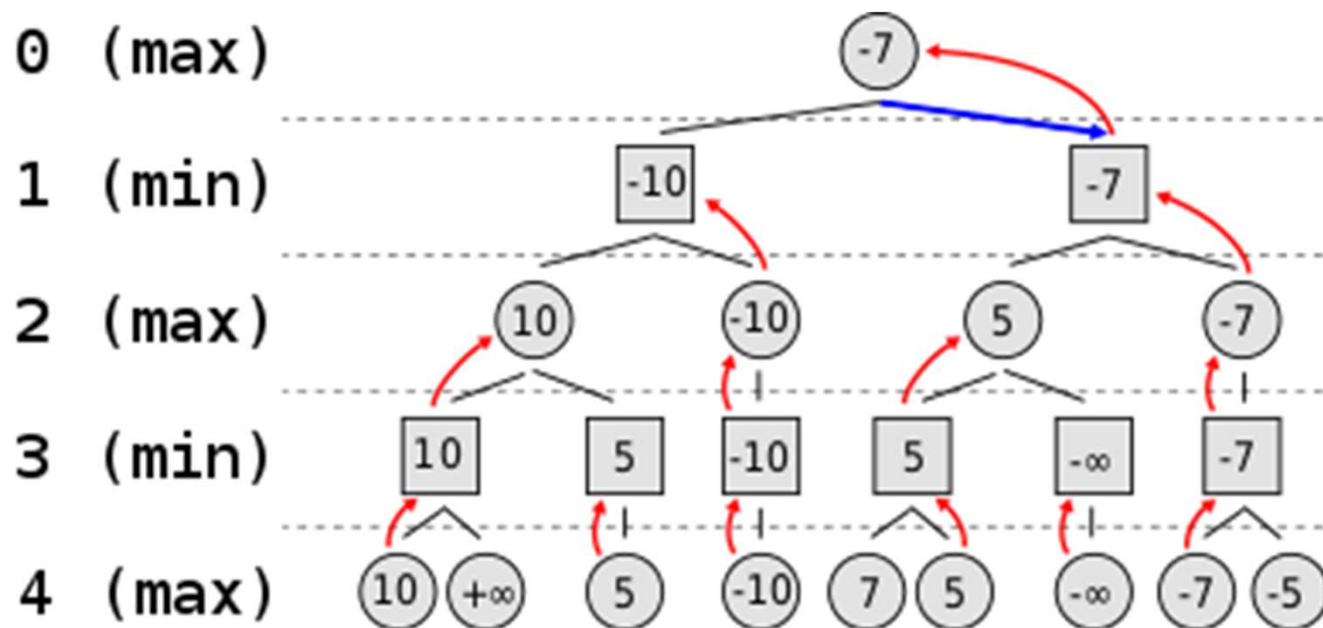
LA IDEA DE MINIMAX:

- ❑ La IA (CPU) es el maximizador -> intentará obtener el máximo número de puntos en una jugada
- ❑ El humano (Player1) es el minimizador -> intentará realizar la jugada con menor puntuación para la máquina
- ❑ El cálculo de la puntuación lo realiza una función heurística $h(x)$ que mide cómo de cerca está la IA de la victoria
- ❑ Cuando es el turno de la CPU: Para cada jugada posible, se evalúa todos los movimientos a través de un **backtracking** con un límite en la **profundidad** y calcula, en cada nivel, la heurística de cada movimiento. La CPU se quedará con el movimiento que obtenga la máxima puntuación

- ❑ El backtracking consiste precisamente en “realizar movimientos virtuales (**pensar**)” para calcular su impacto en la puntuación y volver atrás a la situación de juego actual “**volver a la realidad**”.
- ❑ Esto implica que cada movimiento virtual hay que “**desanotarlo**” para generar otra combinación. Hay que generar “**copias**” del tablero de juego en cada movimiento.

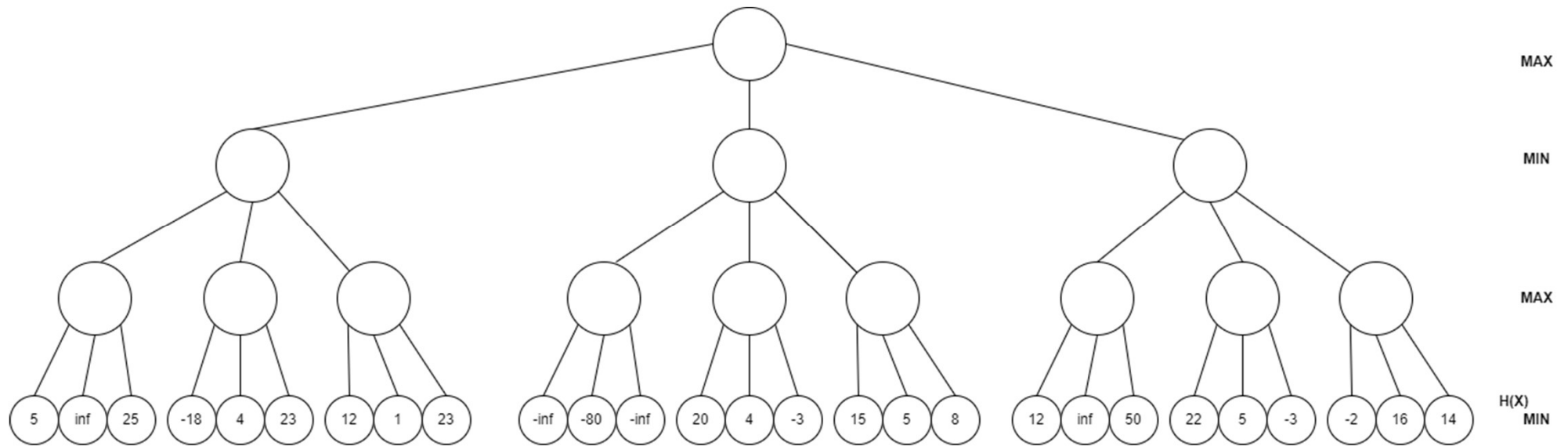


EJEMPLO: Para una posible jugada, se calcula el árbol minimax para explorar qué acción tomaría la CPU (max) y el humano (min)





EJERCICIO: Dada la siguiente situación donde se han calculado ya las heurísticas en nivel 4 ¿Qué camino tomará la IA en la siguiente jugada?





Esquema del MINIMAX

```
# llamada inicial
tablero = crearTablero()
jugada, valor_minimax =
    minimax(0, tablero, True, 5)
```

```
def minimax(profundidad, tablero, maximizador, profundidadMaxima):
    # caso base : profundidad máxima alcanzada
    if (profundidad == profundidadMaxima):
        return puntuacion(tablero)

    if (maximizador):
        maximo = -math.inf
        for jugada in jugadasPosibles(tablero, maximizador):
            aplicar(jugada, tablero)
            nuevo_score = minimax(profundidad + 1, tablero, False, profundidadMaxima)[1]
            if nuevo_score > maximo:
                maximo = nuevo_score
                jugada_seleccionada = jugada
        return jugada_seleccionada, maximo

    else:
        minimo = math.inf
        for jugada in jugadasPosibles(tablero, maximizador):
            aplicar(jugada, tablero)
            nuevo_score = minimax(profundidad + 1, tablero, True, profundidadMaxima)[1]
            if nuevo_score < minimo:
                minimo = nuevo_score
                jugada_seleccionada = jugada
        return jugada_seleccionada, minimo
```



MINIMAX CON PODA α - β

- ❑ Calcular todo el árbol de posibilidades es muy costoso computacionalmente
- ❑ Se puede limitar el número de caminos que se exploran introduciendo dos parámetros α - β , podando los caminos que no sean productivos
- ❑ α es el mejor valor que el maximizador puede asegurar en el nivel actual o superior
- ❑ β es el mejor valor que el minimizador puede asegurar en el nivel actual o superior



MINIMAX CON PODA α - β

- ❑ Se añaden dos parámetros a la llamada a minimax

```
def minimax(profundidad, tablero, maximizador,
            profundidadMaxima,
            alpha, beta):
```

- ❑ Valores iniciales
 $\alpha = -\infty$
 $\beta = \infty$

a la vuelta de la recursividad:

- ❑ Si es nivel maximizador
 actualiza α

```
alpha = max(alpha, nuevo_score)
if alpha >= beta:
    break #fin de la búsqueda
```

- ❑ Si es nivel minimizador
 actualiza β

```
beta = min(beta, nuevo_score)
if alpha >= beta:
    break #fin de la búsqueda
```




Esquema del MINIMAX con poda α - β

```
# llamada inicial
tablero = crearTablero()
jugada, valor_minimax = minimax(0, tablero,
    True, 5, -math.inf, math.inf)
```

```
def minimax(profundidad, tablero, maximizador, profundidadMaxima, alpha, beta):
    # caso base : profundidad máxima alcanzada
    if (profundidad == profundidadMaxima):
        return puntuacion(tablero)

    if (maximizador):
        maximo = -math.inf
        for jugada in jugadasPosibles(tablero, maximizador):
            aplicar(jugada, tablero)
            nuevo_score = minimax(profundidad + 1, tablero, False, profundidadMaxima, alpha, beta)[1]
            if nuevo_score > maximo:
                maximo = nuevo_score
            jugada_seleccionada = jugada

            alpha = max(alpha, nuevo_score)
            if alpha >= beta:
                break

        return jugada_seleccionada, maximo

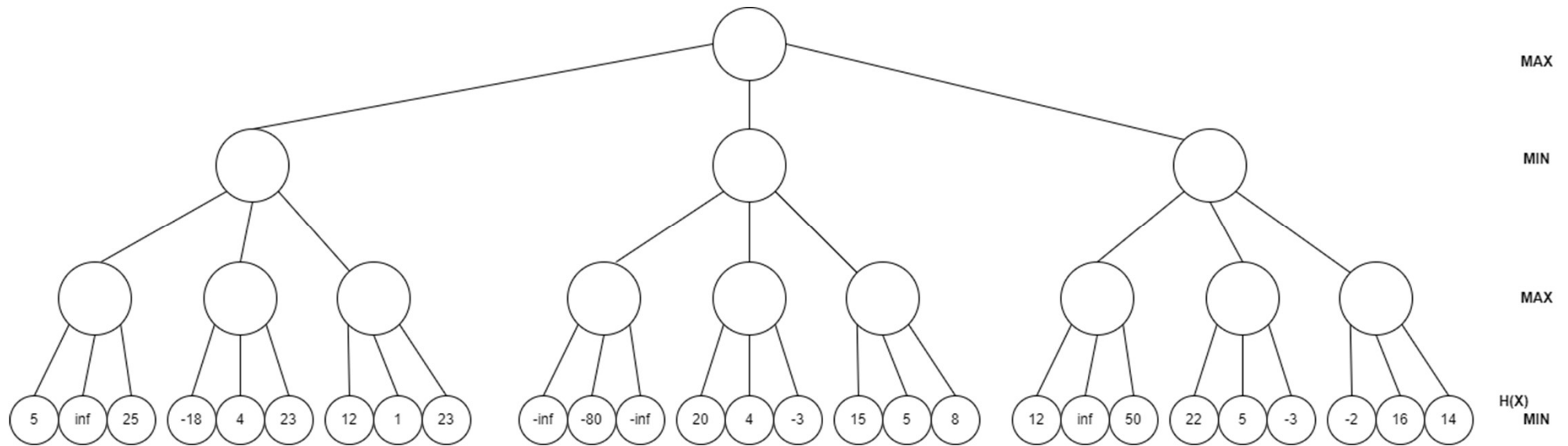
    else:
        minimo = math.inf
        for jugada in jugadasPosibles(tablero, maximizador):
            aplicar(jugada, tablero)
            nuevo_score = minimax(profundidad + 1, tablero, True, profundidadMaxima, alpha, beta)[1]
            if nuevo_score < minimo:
                minimo = nuevo_score
            jugada_seleccionada = jugada

            beta = min(beta, nuevo_score)
            if alpha >= beta:
                break

        return jugada_seleccionada, minimo
```

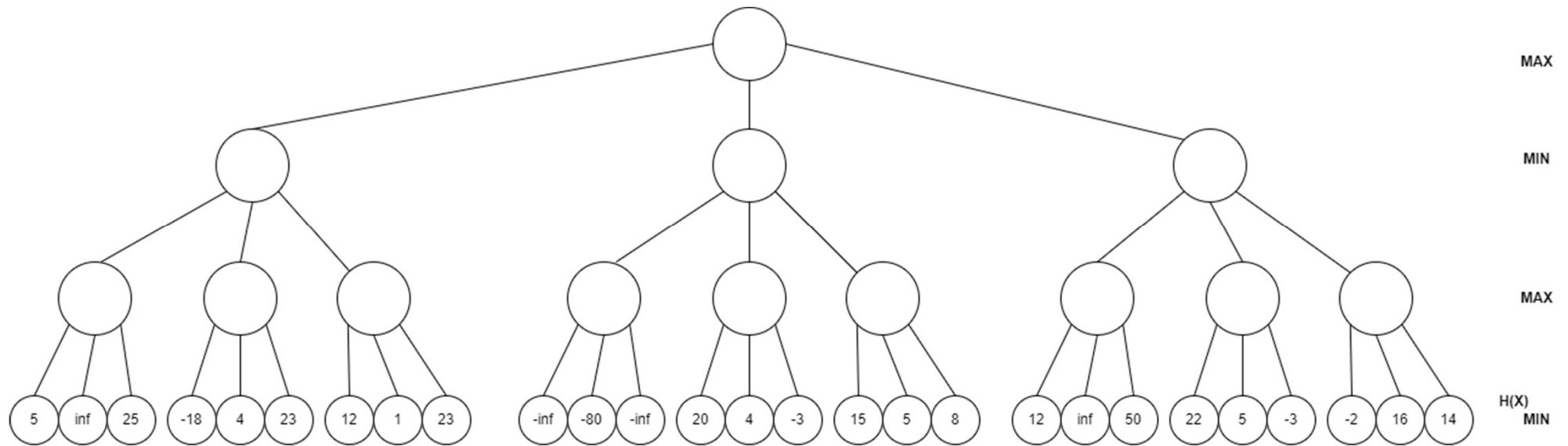


EJERCICIO de minimax con poda α - β : Dada la siguiente situación donde se han calculado ya las heurísticas en nivel 4 ¿Qué camino tomará la IA en la siguiente jugada? Escribe los parámetros Alpha-beta para cada nivel





Desde la solución de minimax a la solución con poda alpha-beta?
¿Cuántas nodos nos hemos ahorrado visitar?



EJEMPLO DE MINIMAX CON PODA α - β EL CONECTA4:

Heurística (sistema de recompensa-penalización):

Las fichas en la columna central valen 3 puntos

ventana = líneas verticales, horizontales o diagonales de tamaño 4

Si en la ventana hay cuatro fichas de la IA = 100 puntos (Victoria)

Si en la ventana hay tres fichas de la IA y 1 vacía = 5 puntos (gran oportunidad)

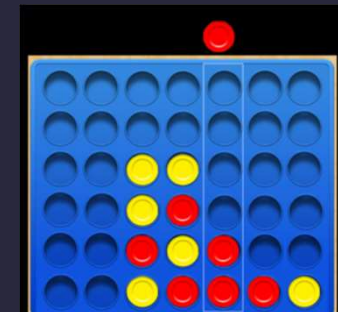
Si en la ventana hay dos fichas de la IA y 2 vacías = 2 puntos (oportunidad)

Si en la ventana hay tres fichas del HUMANO y 1 vacía = -4 puntos (OJO que pierdo)

Si encuentro un movimiento perdedor puntuá $-\infty$

Si encuentro un movimiento ganador puntuá $+\infty$

Demo: versión texto
y versión GUI



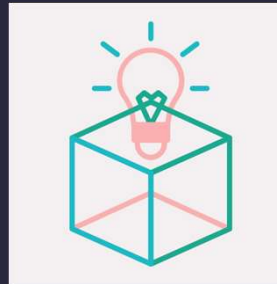


AHORA TÚ:

Proyecto (En grupos de 4)

PROGRAMA UN JUEGO DE SUMA CERO UTILIZANDO MINIMAX Y/O PODA ALPHA BETA:

TEMÁTICA LIBRE



Ejemplos: Damas, Ajedrez con peones y otra pieza, Othello, Hare and Hounds....

