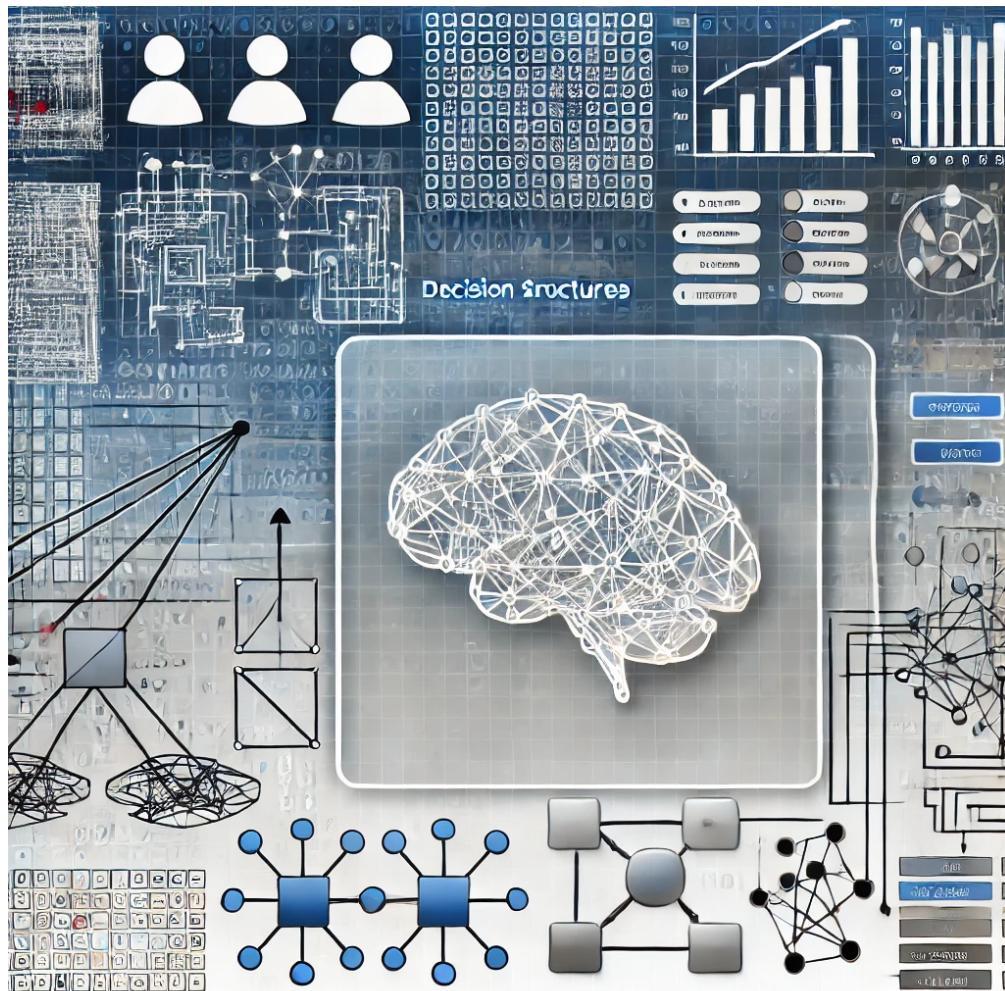


UNIDAD 3

ALGORITMOS Y ESTRUCTURAS DE DATOS PARA IA

Programación de Inteligencia Artificial

Curso de Especialización en Inteligencia Artificial y Big Data



CHATGPT prompt: Crea una imagen relacionada con el tema “Algoritmos y estructuras de datos para Inteligencia Artificial”

Carlos M. Abrisqueta Valcárcel
IES Ingeniero de la Cierva 2024/25

ÍNDICE

1.	EVOLUCIÓN DE LA ALGORÍTMICA	3
2.	BÚSQUEDA EN ESPACIOS DE ESTADOS	4
2.1.	DEFINICIONES	4
3.	ESTRUCTURAS DE DATOS PARA INTELIGENCIA ARTIFICIAL.....	6
3.1.	TENSORES	6
3.2.	DATAFRAMES.....	7
3.3.	ÁRBOLES.....	8
3.4.	GRAFOS	9
3.4.1.	<i>Tipos de grafos y sus componentes</i>	9
3.4.2.	<i>Matriz de Adyacencia</i>	10
4.	ALGORITMOS PARA EXPLORACIÓN EN ESPACIOS DE ESTADO	11
4.1.	ALGORITMOS DE BÚSQUEDA TRADICIONALES	12
4.1.1.	<i>La Búsqueda Lineal.....</i>	12
4.1.2.	<i>Algoritmo de Búsqueda Binaria</i>	12
4.2.	DEFINICIÓN Y RECORRIDO DE ÁRBOLES EN PYTHON	13
4.2.1.	<i>Recorrido de árboles</i>	14
5.	BÚSQUEDA CON VUELTA ATRÁS: BACKTRACKING.....	15
5.1.	ESQUEMA DE RESOLUCIÓN CON BACKTRACKING.....	15
5.2.	EJEMPLOS DE PROBLEMAS RESUELTOS CON BACKTRACKING EN PYTHON	16
5.2.1.	<i>Problema de las N-Reinas</i>	16
5.2.2.	<i>Problema de la Suma de Subconjuntos</i>	18
6.	BÚSQUEDAS CON HEURÍSTICAS. LOS GRAFOS Y EL ALGORITMO A*	20
6.1.	DEFINICIÓN DE HEURÍSTICA.....	20
6.1.1.	<i>Representación de Grafos</i>	20
6.1.2.	<i>Una clase en Python para representar grafos</i>	21
6.1.3.	<i>Representación de Grafos con NetworkX</i>	28
6.1.4.	<i>Creación y Manipulación de Grafos</i>	28
6.2.	ALGORITMO A*	30
7.	TEORÍA DE JUEGOS.....	31
7.1.	DINÁMICAS DE JUEGOS CON INTELIGENCIA ARTIFICIAL.....	31
7.2.	UN EJEMPLO DE ÉXITO REAL: DEEP BLUE.....	32
7.3.	EL ALGORITMO MINIMAX	33
7.4.	CÁLCULO DE VALORES EN UN ÁRBOL MINIMAX.....	34
7.5.	PODA ALFA-BETA EN EL ALGORITMO MINIMAX	35

1. EVOLUCIÓN DE LA ALGORÍTMICA

La algorítmica es una disciplina de la informática y las matemáticas que se ocupa del estudio, diseño, análisis, implementación y optimización de los algoritmos. Un algoritmo puede ser definido como un conjunto finito de instrucciones o pasos que se siguen para resolver un problema o realizar una tarea.

Niklaus Wirth, un renombrado científico de la computación, define un algoritmo como un procedimiento compuesto por un conjunto de reglas definidas y no ambiguas que especifican una secuencia de operaciones para resolver un problema en un número finito de pasos, partiendo de un estado inicial y unas entradas dadas. Según Wirth, los algoritmos deben poseer las siguientes características:

- **Entrada:** Deben tener cero o más entradas, especificadas desde el inicio.
- **Salida:** Deben producir al menos una salida o resultado.
- **Definición:** Cada paso del algoritmo debe estar claramente definido.
- **Finitud:** Deben terminar después de un número finito de pasos.
- **Efectividad:** Cada operación debe ser suficientemente básica para poderse realizar, en principio, exactamente y en un tiempo finito.

Niklaus Wirth:

Algoritmos + Estructuras de Datos = Programas

La algorítmica juega un papel crucial en la informática y la Inteligencia Artificial, ya que proporciona las bases para el desarrollo de algoritmos eficientes que pueden realizar tareas complejas, aprender de los datos y mejorar su rendimiento con el tiempo.

La algorítmica ha experimentado una transformación significativa a lo largo de los años, evolucionando desde algoritmos tradicionales hasta técnicas avanzadas empleadas en la Inteligencia Artificial.

Los algoritmos tradicionales se centran en la solución de problemas específicos mediante procedimientos bien definidos y secuenciales. Estos algoritmos se caracterizan por tener una estructura fija y un conjunto de instrucciones preestablecido para realizar tareas como ordenación, búsqueda, y operaciones matemáticas. Ejemplos de estos algoritmos incluyen el algoritmo de ordenación de burbuja, algoritmo de búsqueda binaria, y el algoritmo de Euclides para el cálculo del máximo común divisor.

Con el tiempo, la necesidad de resolver problemas más complejos y optimizar procesos llevó al desarrollo de algoritmos heurísticos y metaheurísticos. Estos algoritmos no garantizan la obtención de la solución óptima, pero proporcionan soluciones aproximadas en un tiempo razonable. Algoritmos como el recocido simulado, algoritmos genéticos, y la búsqueda tabú son ejemplos de este tipo de técnicas.

La era de la Inteligencia Artificial ha introducido una nueva dimensión en la algorítmica. En lugar de seguir pasos secuenciales fijos, los algoritmos en la IA aprenden de los datos y mejoran su rendimiento con el tiempo.

En el aprendizaje supervisado, los modelos se entrena n utilizando un conjunto de datos etiquetado para predecir etiquetas o valores en nuevos datos. Algoritmos como las redes neuronales, máquinas de soporte vectorial y árboles de decisión son ampliamente utilizados en esta categoría.

El aprendizaje no supervisado implica trabajar con conjuntos de datos no etiquetados, buscando patrones o estructuras en los datos. Técnicas como la agrupación (clustering), reducción de dimensionalidad, y reglas de asociación son ejemplos de algoritmos en este dominio.

En el aprendizaje por refuerzo, los modelos aprenden tomando decisiones y recibiendo retroalimentación en forma de recompensas o penalizaciones. Algoritmos como Q-learning y Deep Q Networks (DQN) son parte de esta categoría.

2. BÚSQUEDA EN ESPACIOS DE ESTADOS

La búsqueda en espacios de estados o *Space State Search* (SSS) es una técnica fundamental en Inteligencia Artificial y en la resolución de problemas computacionales, que consiste en explorar de manera sistemática un conjunto de posibles configuraciones o estados para encontrar una solución a un problema específico. Se utiliza para resolver problemas de optimización, planificación, y toma de decisiones. Permite a los sistemas inteligentes explorar de manera sistemática las posibles soluciones y elegir la mejor opción basada en criterios específicos. Se utiliza en los siguientes campos:

- Teoría de juegos
- Profundización iterativa
- Estructura de datos y procedimientos de búsqueda genéricas
- Resolución de problemas de combinatoria

2.1. DEFINICIONES

Búsqueda 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.

Estado: Representa una configuración específica o situación en la que se puede encontrar un sistema o problema. Cada estado contiene toda la información necesaria para describir completamente la situación en un momento dado.

Espacio de Estados: Conjunto de todos los posibles estados en los que puede encontrarse un problema o sistema. La definición del espacio de estados depende del problema específico que se esté resolviendo.

Nodo: Representa un punto en el espacio de estados, asociado a un estado específico. Cada nodo puede tener uno o varios nodos sucesores, que representan los estados que se pueden alcanzar desde él.

Wikipedia

La búsqueda en el espacio de estados difiere de los métodos de búsqueda tradicionales porque el espacio de estados está implícito: El grafo del espacio de estados típico es demasiado grande para generarla y guardarla en memoria. En su lugar, los nodos se generan en el momento en que se exploran y generalmente son descartados después. Una solución puede consistir solamente en un estado objetivo, o en un camino desde un estado inicial hasta el estado final.

En la exploración en espacios de estados, un estado se representa formalmente como una tupla S :

$$S : \langle S, A, Acción(s), Resultado(s,a), Coste(s,a) \rangle$$

en la cual:

- S , es el conjunto de todos los posibles estados
- A , es el conjunto de todas las posibles acciones, no referida a un estado particular, sino a todo el espacio de estados
- $Action(s)$, es la función que establece qué acción se puede realizar a un estado en particular
- $Resultado(s,a)$, es la función que retorna el estado alcanzado al realizar la acción en el estado s
- $Coste(s,a)$, es el coste de realizar la acción a en el estado s

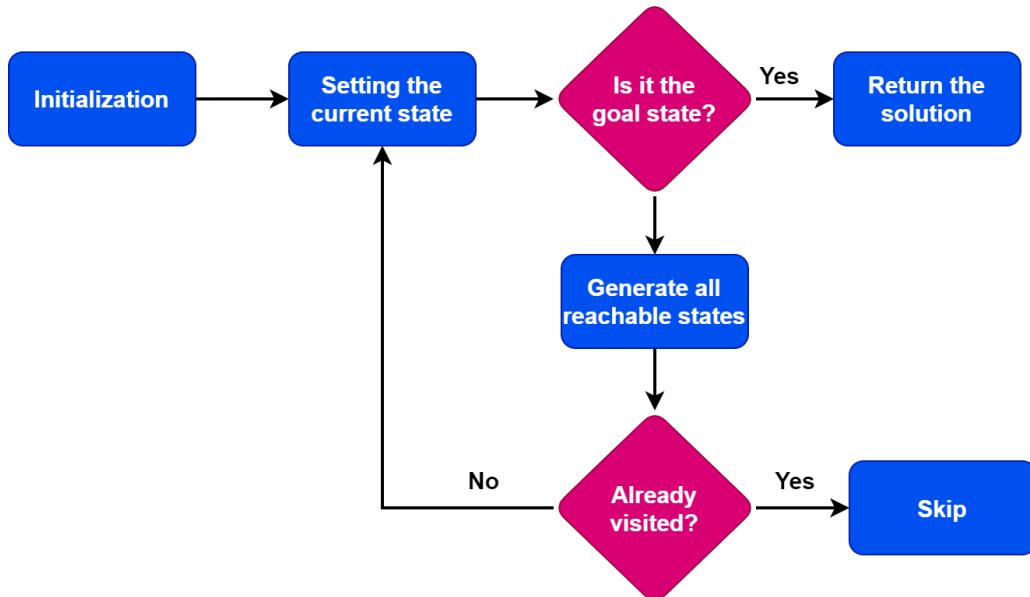


Figura 1: La exploración en espacios de estados



Figura 2: Los algoritmos para S.S.S.

3. ESTRUCTURAS DE DATOS PARA INTELIGENCIA ARTIFICIAL

Las estructuras de datos son fundamentales en el campo de la inteligencia artificial (IA), ya que permiten organizar y almacenar la información de manera eficiente para su posterior análisis y procesamiento. A continuación, se describen algunas de las estructuras de datos más relevantes en Inteligencia Artificial.

3.1. TENSORES

Los tensores son una generalización de vectores y matrices a dimensiones superiores, y son ampliamente utilizados en IA, especialmente en aprendizaje profundo. Pueden ser relacionados con:

- **Vectores (Arrays Unidimensionales)**: Un tensor de una dimensión. Ejemplo: [1, 2, 3].
- **Matrices (Arrays Bidimensionales)**: Un tensor de dos dimensiones. Ejemplo: [[1, 2, 3], [4, 5, 6]].
- **Arrays Multidimensionales**: Tensores de tres o más dimensiones. Ejemplo: [[[1, 2], [3, 4]], [[5, 6], [7, 8]]].

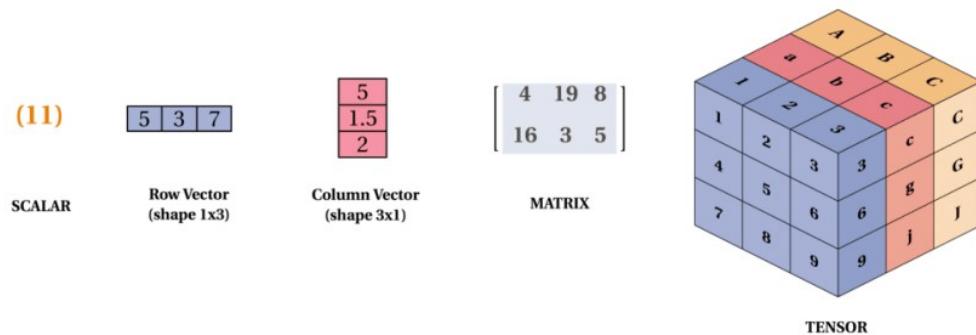


Figura 3: Tensores

Los tensores son utilizados para representar datos en formatos variados como imágenes, series temporales y datos tabulares, y son esenciales para las operaciones en bibliotecas de aprendizaje profundo como **TensorFlow** y **PyTorch**.

Los tensores son estructuras de datos fundamentales en aprendizaje profundo y otras áreas de la inteligencia artificial. Pueden ser categorizados según su rango, el cual indica la cantidad de dimensiones que tienen. Aquí se presentan ejemplos de tensores de rango 0, 1, 2 y 3 junto con su implementación en Python usando la biblioteca **NumPy**:

- **Tensor de Rango 0 (Escalar)**: Un solo número

```
import numpy as np
tensor_rango_0 = np.array(5)
print(tensor_rango_0)
```

- **Tensor de Rango 1 (Vector)**: Un array unidimensional de números

```
tensor_rango_1 = np.array([1, 2, 3])
print(tensor_rango_1)
```

- **Tensor de Rango 2 (Matriz)**: Un array bidimensional de números

```
tensor_rango_2 = np.array([[1, 2, 3], [4, 5, 6]])
print(tensor_rango_2)
```

- **Tensor de Rango 3**: Un array tridimensional de números

```
tensor_rango_3 = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(tensor_rango_3)
```

Cada uno de estos tensores tiene diferentes aplicaciones dependiendo de la naturaleza de los datos y el problema que se esté resolviendo.

3.2. DATAFRAMES

Los *dataframes* son una estructura de datos bidimensional, similar a una tabla de una base de datos, una hoja de cálculo de Excel o una tabla en **R**. Pueden contener datos de diferentes tipos y son especialmente útiles para manipular y analizar conjuntos de datos. En Python, la biblioteca **Pandas** proporciona funcionalidades extensas para trabajar con *dataframes*.

A continuación se presenta un ejemplo de cómo crear un *dataframe*, cómo realizar filtrados y cómo recorrerlo:

```
import pandas as pd

data = {
    'Nombre': ['Ana', 'Luis', 'Marta', 'Carlos', 'Sofía'],
    'Edad': [23, 34, 45, 22, 25],
    'Ciudad': ['Madrid', 'Barcelona', 'Valencia', 'Sevilla', 'Bilbao']
}
```

```
df = pd.DataFrame(data)  
print(df)
```

Este código creará un *dataframe* con tres columnas: ‘Nombre’, ‘Edad’ y ‘Ciudad’, y cinco filas de datos.

Filtrado de Datos

Se puede filtrar los datos de un *dataframe* en función de condiciones. Por ejemplo, para seleccionar a todas las personas mayores de 25 años

```
mayores_25 = df[df['Edad'] > 25]  
print(mayores_25)
```

Recorrido de un Dataframe

Se puede recorrer las filas de un *dataframe* utilizando el método `iterrows()`:

```
for index, row in df.iterrows():  
    print('Nombre:', row['Nombre'], '- Edad:', row['Edad'])
```

Este script imprimirá el nombre y la edad de cada persona en el *dataframe*.

Los *dataframes* son una herramienta poderosa y flexible para el análisis y manipulación de datos, y son ampliamente utilizados en ciencia de datos e inteligencia artificial para manejar y preprocesar conjuntos de datos.

3.3. ÁRBOLES

Los árboles son estructuras de datos jerárquicas que consisten en nodos conectados por bordes. Se utilizan en IA para representar decisiones y jerarquías, y son la base de algoritmos como los árboles de decisión, utilizados en aprendizaje supervisado.

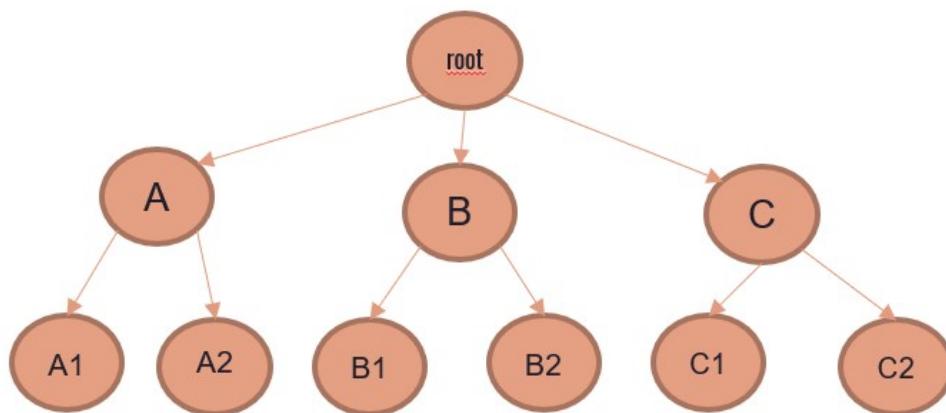


Figura 4: Ejemplo de un árbol binario, balanceado

Los árboles simulan una estructura arbórea con un conjunto de nodos conectados por bordes. Un árbol se compone de los siguientes componentes:

- **Nodo:** Un punto individual en el árbol que puede contener un valor o dato

- **Raíz:** El nodo superior del árbol desde el cual se originan todos los demás nodos
- **Hojas:** Nodos que no tienen hijos
- **Borde o arista:** Conexión entre dos nodos
- **Altura de un Nodo:** La longitud del camino más largo desde ese nodo hasta una hoja
- **Altura del Árbol:** La altura de la raíz

Existen diversos tipos de árboles, como los árboles binarios, los árboles B y los árboles AVL, cada uno con sus propias características y aplicaciones.

3.4. GRAFOS

Los grafos son estructuras que constan de nodos (vértices) y aristas (bordes) que conectan pares de nodos. En IA, los grafos se utilizan para representar redes de relaciones, y son cruciales en algoritmos de búsqueda, optimización de rutas y en el modelado de redes sociales, entre otros.

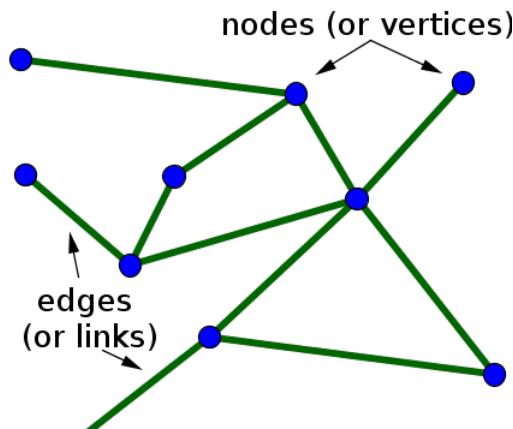


Figura 5: Ejemplo de un grafo no dirigido

3.4.1. Tipos de grafos y sus componentes

Un grafo es una estructura de datos que consta de un conjunto de nodos (vértices) y un conjunto de aristas (bordes) que conectan pares de nodos. Los grafos se utilizan para modelar relaciones entre pares de objetos, siendo ampliamente utilizados en diversos campos como la informática, la biología, el transporte, las redes sociales, entre otros.

Componentes de un Grafo

- **Vértices (o Nodos):** Son los puntos que forman el grafo.
- **Aristas (o Bordes):** Son las líneas que conectan los vértices. Pueden tener una dirección (grafo dirigido) o no tenerla (grafo no dirigido).
- **Peso:** Algunos grafos asignan un valor o peso a sus aristas, que puede representar una distancia, un costo, etc.

Tipos de Grafos

- **Grafo Dirigido:** Las aristas tienen una dirección definida.
- **Grafo No Dirigido:** Las aristas no tienen dirección.
- **Grafo Ponderado:** Las aristas tienen pesos asignados.
- **Grafo No Ponderado:** Las aristas no tienen pesos asignados.

- **Grafo Cíclico:** Existe al menos un ciclo en el grafo.
- **Grafo Acíclico:** No existen ciclos en el grafo.
- **Grafo Conexo:** Existe al menos un camino entre cada par de vértices.
- **Grafo Disconexo:** No existe un camino entre cada par de vértices.
- **Grafo Completo:** Existe exactamente una arista conectando cada par de vértices.

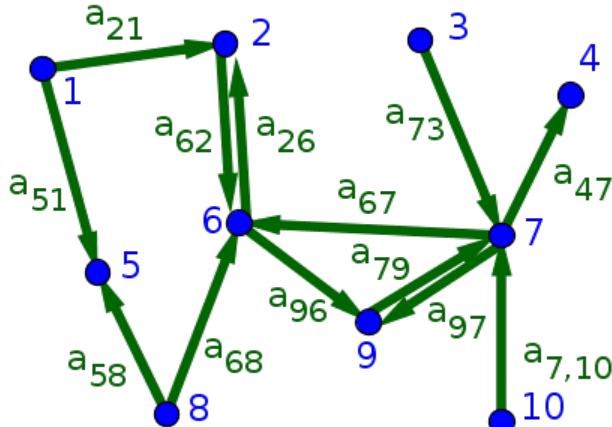


Figura 6: Grafo dirigido y ponderado

Cada tipo de grafo tiene sus propias propiedades y casos de uso, y la elección del tipo de grafo a utilizar depende del problema específico que se esté tratando de resolver. Detallaremos cómo definir grafos y recorrerlos en la sección de búsquedas con heurística y el algoritmo A*.

3.4.2. Matriz de Adyacencia

La matriz de adyacencia es una de las formas más comunes de representar un grafo en una estructura de datos. Consiste en una matriz cuadrada A de tamaño $n \times n$, donde n es el número de vértices en el grafo. Cada elemento $A[i][j]$ de la matriz representa la relación entre el vértice i y el vértice j .

Definición: Para un grafo no dirigido, la matriz de adyacencia se define de la siguiente manera

$$A[i][j] = \begin{cases} 1 & \text{si existe una arista entre } i \text{ y } j \\ 0 & \text{en caso contrario} \end{cases}$$

Para un grafo dirigido, se sigue la misma definición, pero la presencia de una arista de i a j no implica la presencia de una arista de j a i .

Grafos Ponderados

En el caso de los grafos ponderados, la matriz de adyacencia puede almacenar el peso de las aristas en lugar de simplemente 1 o 0. Si no existe arista entre i y j , el elemento correspondiente de la matriz suele ser 0 o un valor especial que indica la ausencia de conexión (como ∞ para representar un costo infinito en algoritmos de rutas más cortas).

Propiedades

- Para grafos no dirigidos, la matriz de adyacencia es simétrica.
- El grado de un vértice puede ser calculado sumando los valores en la fila o columna correspondiente.

- La matriz permite una rápida verificación de la existencia de una arista entre dos vértices.

Ejemplo

Considera un grafo no dirigido con tres vértices A , B , y C , y aristas entre A y B , A y C . Su matriz de adyacencia sería:

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

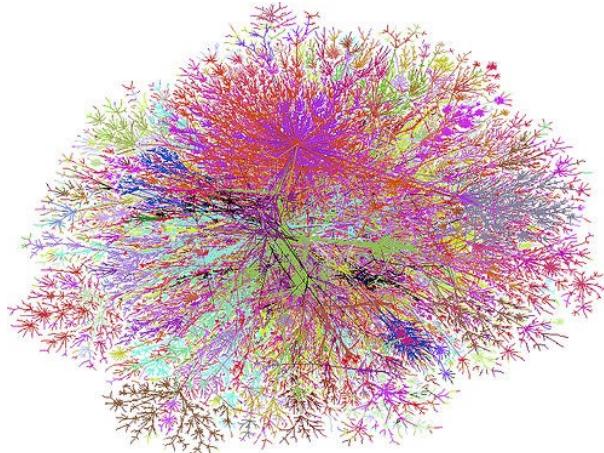


Figura 7: Grafo complejo de conexiones entre usuarios de internet

4. ALGORITMOS PARA EXPLORACIÓN EN ESPACIOS DE ESTADO

Se pueden dividir los algoritmos en dos grandes grupos: La búsqueda no informada, también conocida como búsqueda ciega, explora el espacio de estados sin utilizar información adicional sobre la distancia o costo para alcanzar el estado objetivo. Ejemplos de este tipo de búsqueda incluyen la búsqueda en amplitud y la búsqueda en profundidad o búsqueda de coste uniforme.

La búsqueda informada, por otro lado, utiliza información adicional sobre el problema para guiar la exploración del espacio de estados de manera más eficiente. Un ejemplo común es el algoritmo A*, que utiliza una función heurística para estimar el costo de alcanzar el estado objetivo desde un nodo dado.



Figura 8: Los algoritmos para S.S.S.

4.1. ALGORITMOS DE BÚSQUEDA TRADICIONALES

En esta sección se explican dos algoritmos de búsqueda muy sencillos, según la algorítmica tradicional.

4.1.1. La Búsqueda Lineal

La búsqueda lineal es un método sencillo para encontrar un elemento específico dentro de un vector. El algoritmo recorre cada elemento del vector hasta encontrar el elemento que se desea buscar. A continuación, se presenta el pseudocódigo para el algoritmo de búsqueda lineal, así como una explicación detallada de su funcionamiento.

```
funcion busquedaLineal(vector , elementoABuscar):
    para i de 0 hasta longitud(vector) - 1 hacer
        si vector[i] == elementoABuscar entonces
            retornar i    # Elemento encontrado, retornar índice
        fin si
    fin para
    retornar -1    # Elemento no encontrado, retornar -1
fin función
```

Explicación del Algoritmo

- **Entrada:** El algoritmo toma como entrada un *vector* y un *elementoABuscar*.
- **Proceso:**
 - Se itera sobre cada elemento del vector desde el índice *0* hasta el índice *longitud(vector) - 1*.
 - En cada iteración, se compara el elemento actual del vector con *elementoABuscar*.
 - Si se encuentra una coincidencia, se retorna el índice del elemento encontrado.
- **Salida:**
 - Si se encuentra el *elementoABuscar*, se retorna su índice en el vector.
 - Si el *elementoABuscar* no está presente en el vector, se retorna *-1*.

4.1.2. Algoritmo de Búsqueda Binaria

La búsqueda binaria, también conocida como búsqueda dicotómica, es un método eficiente para encontrar un elemento en un vector ordenado. El algoritmo divide repetidamente a la mitad la porción del vector que podría contener al elemento, hasta reducir las posibles ubicaciones a solo una. A continuación se presenta el pseudocódigo para el algoritmo de búsqueda binaria, así como una explicación detallada de su funcionamiento.

```
funcion busquedaBinaria(vector , elementoABuscar):
    bajo := 0
    alto := longitud(vector) - 1
    mientras bajo <= alto hacer
        medio := (bajo + alto) / 2
        si vector[medio] == elementoABuscar entonces
            retornar medio    # Elemento encontrado, retornar índice
        sino si vector[medio] < elementoABuscar entonces
```

```

        bajo := medio + 1      # Ajustar rango de búsqueda a la mitad
                                superior
    sino
        alto := medio - 1      # Ajustar rango de búsqueda a la mitad
                                inferior
    fin si
fin mientras
retornar -1      # Elemento no encontrado, retornar -1
fin función
    
```

Explicación del Algoritmo:

- **Entrada:** El algoritmo toma como entrada un vector ordenado y un elementoABuscar.
- **Proceso:**
 - Se definen dos índices, *bajo* y *alto*, que representan el rango del vector donde se buscará el elemento.
 - Mientras *bajo* sea menor o igual a *alto*, se calcula el índice *medio* del rango actual.
 - Si el *elementoABuscar* es igual al elemento en la posición *medio*, se ha encontrado el elemento y se retorna su índice.
 - Si el *elementoABuscar* es mayor, se ajusta el rango de búsqueda a la mitad superior.
 - Si el *elementoABuscar* es menor, se ajusta el rango de búsqueda a la mitad inferior.
- **Salida:**
 - Si se encuentra el *elementoABuscar*, se retorna su índice en el vector.
 - Si el *elementoABuscar* no está presente en el vector, se retorna -1.

4.2. DEFINICIÓN Y RECORRIDO DE ÁRBOLES EN PYTHON

Se puede definir un árbol como una estructura de datos recursiva. La definición de un árbol en Python es muy sencilla.

```

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)
```

4.2.1. Recorrido de árboles

A continuación, se muestran los pseudocódigos para recorrer árboles en profundidad y en amplitud.

Recorrido en Profundidad

El recorrido en profundidad es una técnica que visita todos los nodos de un árbol, yendo tan profundo como sea posible antes de retroceder.

```
function RecorridoEnProfundidad(nodo)
    if nodo is null then return
    print(nodo.valor)
    RecorridoEnProfundidad(nodo.izquierdo)
    RecorridoEnProfundidad(nodo.derecho)
end function
```

Recorrido en Amplitud

El algoritmo de Recorrido en Amplitud es una técnica utilizada para explorar todos los nodos de un árbol o grafo, navegando a través de ellos nivel por nivel. A continuación, se presenta y explica el pseudocódigo para realizar un recorrido en amplitud en un árbol.

```
function RecorridoEnAmplitud(raiz)
    if raiz is null then return
    cola := new Cola ()
    cola.enqueue(raiz)
    while not cola.isEmpty() do
        nodo := cola.dequeue()
        print(nodo.valor)
        if nodo.izquierdo is not null then
            cola.enqueue(nodo.izquierdo)
        end if
        if nodo.derecho is not null then
            cola.enqueue(nodo.derecho)
        end if
```

```
    end while
end function
```

Descripción del Algoritmo

El algoritmo comienza verificando si el nodo raíz es nulo. En caso de serlo, termina la ejecución, ya que un árbol vacío no tiene nodos para recorrer.

Si la raíz no es nula, se inicializa una cola y se encola el nodo raíz. La cola se utiliza para mantener un registro de los nodos a visitar.

El bucle principal del algoritmo se ejecuta mientras la cola no esté vacía. En cada iteración, se desencola un nodo de la cola y se imprime su valor. Luego, si el nodo tiene un hijo izquierdo, este se encola. Lo mismo ocurre con el hijo derecho.

Este proceso se repite hasta que la cola esté vacía, lo que significa que se han visitado todos los nodos del árbol. Al visitar los nodos nivel por nivel, el algoritmo realiza un recorrido en amplitud del árbol.

Complejidad del Algoritmo

La complejidad temporal del algoritmo es $O(n)$, donde n es el número de nodos en el árbol, ya que cada nodo se visita exactamente una vez. La complejidad espacial también es $O(n)$, ya que, en el peor de los casos, la cola podría almacenar todos los nodos de un nivel del árbol.

5. BÚSQUEDA CON VUELTA ATRÁS: BACKTRACKING

El *backtracking* es una técnica de programación para resolver problemas de búsqueda y optimización. Consiste en explorar sistemáticamente todas las posibles configuraciones de la solución de un problema para encontrar todas las soluciones posibles o la mejor solución. Si en algún punto se determina que una configuración parcial no puede llevar a una solución válida, el algoritmo retrocede (*backtracks*) para probar la siguiente opción.

5.1. ESQUEMA DE RESOLUCIÓN CON BACKTRACKING

El siguiente es un esquema general en pseudocódigo para resolver un problema usando *backtracking*.

```
backtracking(problema ,opciones ,paso):
    exito=false
    opciones=opciones_aceptables(problema)
    Mientras len(opciones)>0 y no exito
        opcion=extraer_siguiente(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
```

Este esquema muestra cómo el *backtracking* explora las opciones disponibles, añadiendo opciones al candidato actual si son válidas, y retrocediendo cuando se llega a un punto muerto.

5.2. EJEMPLOS DE PROBLEMAS RESUELTOS CON BACKTRACKING EN PYTHON

5.2.1. Problema de las N-Reinas

El problema de las N-Reinas consiste en colocar N reinas en un tablero de ajedrez de $N \times N$ de tal manera que ninguna reina amenace a otra.

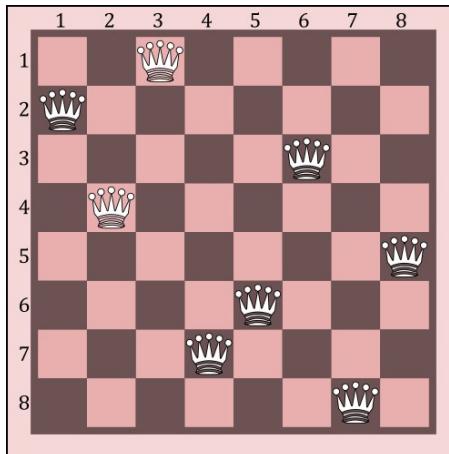


Figura 9: El problema de las 8 reinas

```
def es_seguro(tablero, fila, col, n):
    # Revisar esta fila en la columna izquierda
    for i in range(col):
        if tablero[fila][i] == 1:
            return False

    # Revisar diagonal superior izquierda
    for i, j in zip(range(fila, -1, -1), range(col, -1, -1)):
        if tablero[i][j] == 1:
            return False

    # Revisar diagonal inferior izquierda
    for i, j in zip(range(fila, n, 1), range(col, -1, -1)):
        if tablero[i][j] == 1:
            return False
    return True

def resolver_n_reinas(tablero, col, n):
    if col >= n:
```

```

        return True

    for i in range(n):
        if es_seguro(tablero, i, col, n):
            tablero[i][col] = 1
            if resolver_n_reinas(tablero , col + 1, n):
                return True
            tablero[i][col] = 0

    return False

def imprimir_solucion(tablero , n):
    for i in range(n):
        for j in range(n):
            print(tablero[i][j], end=" ")
        print ()

def n_reinas(n):
    tablero = [[0 for _ in range(n)] for _ in range(n)]
    if not resolver_n_reinas(tablero , 0, n):
        print("No existe solución")
        return False
    imprimir_solucion(tablero , n)
    return True

n_reinas(4)

```

A continuación se explica el algoritmo utilizado para resolver este problema.

Función `es_seguro`: Esta función determina si es seguro colocar una reina en la posición dada del tablero.

- **Entradas:** `tablero` (la configuración actual del tablero), `fila` y `col` (las coordenadas de la posición que se está comprobando) y `n` (el tamaño del tablero).
- **Proceso:** Se comprueba si hay alguna reina en la misma fila a la izquierda, en la diagonal superior izquierda y en la diagonal inferior izquierda. Si hay una reina en alguna de estas posiciones, no es seguro colocar otra reina en la posición dada.
- **Salida:** Devuelve `True` si es seguro colocar una reina y `False` en caso contrario.

Función `resolver_n_reinas`: Esta función intenta resolver el problema de las N-Reinas utilizando backtracking.

- **Entradas:** `tablero` (la configuración actual del tablero), `col` (la columna actual que se está comprobando) y `n` (el tamaño del tablero).
- **Proceso:** Si `col` es igual a `n`, todas las reinas están colocadas y la función devuelve `True`. De lo contrario, intenta colocar una reina en todas las filas de la columna actual. Si encuentra una fila segura, coloca una reina y hace una llamada recursiva para colocar las reinas en las siguientes columnas. Si colocar una reina en la fila actual y hacer una llamada

recursiva no conduce a una solución, se quita la reina (*backtrack*) y se prueba con la siguiente fila en la columna actual.

- **Salida:** Devuelve *True* si todas las reinas están colocadas correctamente, *False* en caso contrario.

Función *imprimir_solución*: Esta función imprime la configuración actual del tablero.

- **Entradas:** *tablero* (la configuración actual del tablero) y *n* (el tamaño del tablero).
- **Proceso:** Recorre el tablero e imprime la posición de las reinas.
- **Salida:** No devuelve nada, pero imprime el tablero en la consola.

Función *n_reinas*: Esta es la función principal que resuelve el problema de las N-Reinas.

- **Entrada:** *n* (el tamaño del tablero).
- **Proceso:** Inicializa el tablero, intenta resolver el problema y, si se encuentra una solución, imprime el tablero.
- **Salida:** Devuelve *True* si se encontró una solución, *False* en caso contrario.

Ejemplo de Ejecución

```
n_reinas(4)
```

La función *n_reinas* se llama con *n* = 4, intentando resolver el problema para un tablero de 4×4 . Si se encuentra una solución, el tablero se imprimirá en la consola.

5.2.2. Problema de la Suma de Subconjuntos

El problema de la suma de subconjuntos consiste en determinar si existe un subconjunto de un conjunto dado que sume una cantidad específica.

```
def es_solucion(conjunto, solucion, suma_objetivo):
    return sum(solucion) == suma_objetivo

def es_valido(conjunto, solucion, suma_objetivo, i):
    return sum(solucion) + conjunto[i] <= suma_objetivo

def suma_subconjuntos(conjunto, suma_objetivo):
    solucion = []
    if _suma_subconjuntos(conjunto, solucion, suma_objetivo, 0):
        print("Subconjunto encontrado:", solucion)
    else:
        print("No se encontró subconjunto")

def _suma_subconjuntos(conjunto, solucion, suma_objetivo, i):
    if es_solucion(conjunto, solucion, suma_objetivo):
        return True
    if i >= len(conjunto):
        return False
    if es_valido(conjunto, solucion, suma_objetivo, i):
```

```

solucion.append(conjunto[i])
if _suma_subconjuntos(conjunto, solucion, suma_objetivo, i + 1):
    return True
solucion.pop()
return _suma_subconjuntos(conjunto, solucion, suma_objetivo, i + 1)
conjunto = [3, 34, 4, 12, 5, 2]
suma_objetivo = 9
suma_subconjuntos(conjunto, suma_objetivo)
    
```

Este algoritmo busca si existe un subconjunto de un conjunto dado de números que sume un valor objetivo específico.

Funciones y su descripción:

- *es_solución(conjunto, solución, suma_objetivo)*: Esta función comprueba si la suma de los elementos en la lista *solución* es igual a *suma_objetivo*. Devuelve *True* si lo es, y *False* en caso contrario.
- *es_valido(conjunto, solución, suma_objetivo, i)*: Esta función verifica si es posible agregar el elemento actual *conjunto[i]* a la solución sin exceder *suma_objetivo*. Devuelve *True* si es posible, y *False* en caso contrario.
- *suma_subconjuntos(conjunto, suma_objetivo)*: Esta es la función principal que inicia el proceso de *backtracking*. Imprime el subconjunto encontrado si existe, o un mensaje indicando que no se encontró ninguno.
- *_suma_subconjuntos(conjunto, solución, suma_objetivo, i)*: Esta es la función recursiva de *backtracking* que intenta encontrar un subconjunto que sume *suma_objetivo*. *i* representa la posición actual en *conjunto* que estamos considerando para incluir en *solución*.

Proceso del algoritmo:

El algoritmo empieza con un conjunto dado *conjunto*, una lista vacía *solución*, y un valor *suma_objetivo*. La idea es explorar todas las posibles combinaciones de números en *conjunto* para ver si alguna de ellas suma *suma_objetivo*.

Si en algún momento la suma de los números en *solución* es igual a *suma_objetivo*, hemos encontrado una solución y la función devuelve *True*. Si la suma excede *suma_objetivo* o si hemos considerado todos los números en *conjunto* sin encontrar una solución, la función devuelve *False*.

Cuando agregamos un número a *solución*, hacemos una llamada recursiva para considerar el siguiente número en *conjunto*. Si esta llamada recursiva no lleva a una solución, quitamos el último número añadido a *solución* (*backtracking*) y procedemos con la siguiente llamada recursiva para considerar el siguiente número.

Ejemplo de uso:

```

conjunto = [3, 34, 4, 12, 5, 2]
suma_objetivo = 9
suma_subconjuntos(conjunto, suma_objetivo)
    
```

Este ejemplo busca un subconjunto de [3, 34, 4, 12, 5, 2] que sume 9. El algoritmo encuentra que [3, 4, 2] es un subconjunto que satisface esta condición e imprime "Subconjunto encontrado: [3, 4, 2]".

6. BÚSQUEDAS CON HEURÍSTICAS. LOS GRAFOS Y EL ALGORITMO A*

6.1. DEFINICIÓN DE HEURÍSTICA

Una heurística es un enfoque práctico para la resolución de problemas que, aunque no garantiza una solución óptima, es suficientemente bueno para ciertas situaciones. Las heurísticas son especialmente útiles en problemas complejos donde una solución exacta es impracticable debido a restricciones de tiempo o recursos computacionales.

Por ejemplo, una heurística comúnmente utilizada en problemas de búsqueda de rutas, como en el algoritmo A*, es la distancia en línea recta (también conocida como distancia euclídea). Esta heurística estima el costo mínimo desde un punto en el espacio hasta el objetivo, asumiendo que no hay obstáculos en el camino.

La distancia en línea recta entre dos puntos, $A(x_1, y_1)$ y $B(x_2, y_2)$, en un espacio bidimensional se calcula utilizando la fórmula:

$$d(A, B) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Esta heurística es admisible, lo que significa que nunca sobreestima el costo real para llegar al objetivo, una propiedad importante para garantizar que el algoritmo A* encuentre una solución óptima.

6.1.1. Representación de Grafos

Los grafos pueden representarse de diversas maneras en programación. Dos de las más comunes son mediante listas de adyacencia y matrices de adyacencia.

Lista de Adyacencia

Una lista de adyacencia representa cada vértice con una lista de los vértices a los que está conectado.

```
# Representación de un grafo con lista de adyacencia
grafo_lista = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# Función para agregar una arista
```

```
def agregar_arista(grafo, u, v):
    grafo[u].append(v)
    # Si el grafo es no dirigido, descomentar la siguiente línea
    # grafo[v].append(u)

# Ejemplo de uso
agregar_arista(grafo_lista , 'A', 'G')
```

Matriz de Adyacencia

Una matriz de adyacencia utiliza una matriz bidimensional para representar las conexiones entre los vértices, donde cada fila y columna representan un vértice.

```
# Representación de un grafo con matriz de adyacencia
grafo_matriz = [
    [0, 1, 1, 0, 0, 0],
    [1, 0, 0, 1, 1, 0],
    [1, 0, 0, 0, 0, 1],
    [0, 1, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 1],
    [0, 0, 1, 0, 1, 0]
]

# Función para agregar una arista
def agregar_arista_matriz(grafo, u, v):
    grafo[u][v] = 1
    # Si el grafo es no dirigido, descomentar la siguiente línea
    # grafo[v][u] = 1

# Ejemplo de uso
agregar_arista_matriz(grafo_matriz , 0, 3)
```

6.1.2. Una clase en Python para representar grafos

A continuación se presenta una implementación de una clase ***Graph*** para que puedas utilizarla en tus representaciones de grafos. Este módulo *graph.py* incluye dos clases, la clase nodo *Node* y la clase grafo *Graph*.

La clase *Node* está diseñada para representar cada vértice en un grafo, ofreciendo funcionalidades y atributos esenciales para la manipulación de grafos.

Atributos

- *value(str)*: Representa el valor del nodo.
- *x(int)*: Representa la coordenada *x* del nodo.
- *y(int)*: Representa la coordenada *y* del nodo.
- *heuristic_value(int)*: Corresponde a la distancia de Manhattan más la distancia desde el nodo inicial al nodo actual. El valor predeterminado es *-1*.

- *distance_from_start(float)*: Representa la distancia del nodo desde el nodo inicial. El valor predeterminado es *infinito*.
- *neighbors(list)*: Lista de nodos a los que está conectado el nodo actual.
- *parent(Node)*: Representa el nodo padre del nodo actual. El valor predeterminado es *None*.

Métodos

- *has_neighbors(self) -> Boolean*: Verifica si el nodo actual está conectado con otros nodos.
- *number_of_neighbors(self) -> int*: Calcula y devuelve el número de vecinos del nodo.
- *add_neighboor(self, neighbor) -> None*: Añade un nuevo vecino a la lista de vecinos.
- *extend_node(self) -> list*: Devuelve una lista de nodos con los que el nodo actual está conectado.
- *eq(self, other) -> Boolean*: Determina si dos nodos son iguales, comparando sus valores.
- *str(self) -> str*: Imprime los datos del nodo.

```
from math import inf
class Node:
    """
        This class used to represent each Vertex in the graph ...
        Attributes
        -----
        value : str
            Represent the value of the node
        x : int
            Represent the x-coordinate of the node
        y : int
            Represent the y-coordinate of the node
        heuristic_value : int
            Corresponds to the Manhattan distance plus the distance from the
            initial node to the current node. Default value is -1
        distance_from_start
            Corresponds to the distance of the node from the initial node. Default
            value is -1
        neighbors : list
            A list with the nodes the current node is connected
        parent : Node
            Represents the parent-node of the current node. Default value is None
        ...
        Methods
        -----
        has_neighbors(self) -> Boolean
            Check if the current node is connected with other nodes (return True).
            Otherwise return False
    """
```

```

number_of_neighbors(self) -> int
    Calculate and return the number the of the neighbors
add_neighboor(self, neightbor) -> None
    Add a new neighbor in the list of neighbors
extend_node(self) -> list
    return a list of nodes with which the current node is connected
__eq__(self, other) -> Boolean
    Determines if two nodes are equal or not, checking their values
__str__(self) -> str Prints the node data
"""

def __init__(self, value, coordinates, neighbors=None):
    self.value = value
    self.x = coordinates[0]
    self.y = coordinates[1]
    self.heuristic_value = -1
    self.distance_from_start = inf
    if neighbors is None:
        self.neighbors = []
    else:
        self.neighbors = neighbors
    self.parent = None

def has_neighbors(self):
    """
        Return True if the current node is connected with at least another
node.
        Otherwise return false
    """
    if len(self.neighbors) == 0:
        return False
    return True

def number_of_neighbors(self):
    """
        Return the number of nodes with which the current node is connected
    """
    return len(self.neighbors)

def add_neighboor(self, neightbor):
    """
        Add a new node to the neighbor list. In other words create a new
connection between the current node and the neighbor

        Parameters
        -----
        neightbor : tuple with the nodo and the weight
            Represent the node with which a new connection is created
    """

```

```
    self.neighbors.append(neighboor)

def extend_node(self):
    """
        Extends the current node, creating and returning a list with all
connected nodes
        Returns
        -----
        List
    """
    children = []
    for child in self.neighbors:
        children.append(child[0])
    return children

def __gt__(self, other):
    """
        Define which node, between current node and other node , has the
greater value.
        First examine the heuristic value. If this value is the same for both
nodes the function checks the lexicographic series
        Parameters
        -----
        other: Node
            Represent the other node with which the current node is
compared
        Returns
        -----
        Boolean
    """
    if isinstance(other , Node):
        if self.heuristic_value > other.heuristic_value:
            return True
        if self.heuristic_value < other.heuristic_value:
            return False
        return self.value > other.value

def __eq__(self, other):
    """
        Define if current node and other node are equal, checking their values.
        Parameters
        -----
        other: Node:
            Represent the other node with which the current node is
compared
        Returns
        -----
        Boolean
    """


```

```

        if isinstance(other, Node):
            return self.value == other.value
        return self.value == other

    def __str__(self):
        """
            Define that a node is printed with its value.
            Returns
            -----
            str
        """
        return self.value
    
```

La clase *Graph* está diseñada para representar la estructura de datos de un grafo. Ofrece funcionalidades para la gestión de nodos y aristas, facilitando la creación y manipulación de grafos.

Atributos

- *nodes (list)*: Lista que contiene todos los nodos del grafo.

Métodos

- *add_node (self, node) ->None*: Añade un nuevo nodo al grafo.
- *find_node (self, value) ->Node*: Busca y devuelve el nodo del grafo con el valor dado.
- *add_edge (self, value1, value2, weight=1) ->None*: Añade una nueva arista al grafo.
- *number_of_nodes (self) ->int*: Calcula y devuelve el número de nodos del grafo.
- *are_connected (self, node_one, node_two) ->Boolean*: Verifica si dos nodos dados están conectados entre sí.
- *str (self) ->str*: Imprime los nodos del grafo.

```

class Graph:
    """
        This class used to represent the graph data structure.

        ...
        Attributes
        -----
        nodes : list
            List with all the nodes of the graph
        ...
        Methods
        -----
        add_node(self, node) -> None
            Add a new node in the list of nodes
        find_node(self, value) -> Node
            Find and return the node of the graph with the given value.
        add_edge(self, value1, value2, weight=1) -> None
            Add a new edge in the graph
    
```

```

number_of_nodes(self) -> int
    Calculate and return the number of nodes of the graph
are_connected(self, node_one, node_two) -> Boolean
    Check if the two given nodes are connected each other
__str__(self) -> str
    Prints the nodes of the graph
"""

def __init__(self, nodes=None):
    if nodes is None:
        self.nodes = []
    else:
        self.nodes = nodes

def add_node(self, node):
    """
        Add a new node (vertex) in the grpah
        Parameters
        -----
        node: Node
            Represent the nserted node in the graph
    """
    self.nodes.append(node)

def find_node(self, value):
    """
        Return True if the node with the given value exist in the graph.
        Otherwise it return False
        Parameters
        -----
        value: str
            Is the value of the node we want to find
        ...
        Return
        -----
        Node
    """
    for node in self.nodes:
        if node.value == value:
            return node
    return None

def add_edge(self, value1, value2, weight=1):
    """
        Add a new edge between the two given nodes
        Parameters
        -----
        value1: str
            The value of the first node
    """

```

```

        value2: str
            The value of the second node
        weight:
            The weight of the edge. Default value 1
        ...
    Return
    -----
        Node
"""

node1 = self.find_node(value1)
node2 = self.find_node(value2)

if (node1 is not None) and (node2 is not None):
    node1.add_neighboor((node2, weight))
    node2.add_neighboor((node1, weight))
else:
    print("Error: One or more nodes were not found")

def number_of_nodes(self):
    """
        Return the number of nodes of the graph
        ...
    Return
    -----
        int
"""
    return f"The graph has {len(self.nodes)} nodes"

def are_connected(self, node_one, node_two):
    """
        Return True if the given nodes are connected. Otherwise return False
        ...
    Parameters
    -----
        node_one: str
            The value of the first node
        node_two: str
            The value of the second node
    Return
    -----
        Boolean
"""
    node_one = self.find_node(node_one)
    node_two = self.find_node(node_two)

    for neighbor in node_one.neighbors:
        if neighbor[0].value == node_two.value:
            return True

```

```
return False

def __str__(self):
    """
        Define the way the nodes of graph will be printed.
        Return
    -----
        str
    """
    graph = ""
    for node in self.nodes:
        graph += f"{node.__str__()}\n"
    return graph
```

6.1.3. Representación de Grafos con NetworkX

NetworkX es un paquete de Python diseñado para la creación, manipulación y análisis de la estructura y dinámicas de redes complejas. Permite la creación de grafos dirigidos y no dirigidos, con una amplia variedad de funciones para agregar nodos, aristas y otras funcionalidades.

6.1.4. Creación y Manipulación de Grafos

Para comenzar a trabajar con *NetworkX*, primero debemos importar el paquete. A continuación, se muestra cómo crear y manipular grafos. Algunos de sus métodos clave son:

- **Constructor Graph.** El constructor `Graph()` crea un nuevo grafo. Este grafo es no dirigido por defecto, permitiendo agregar nodos y aristas de manera flexible.
- **Método add_node.** El método `add_node()` añade un nodo al grafo. Se puede especificar el nodo por su identificador, que puede ser un número, una cadena u otro objeto hashable.
- **Método add_edge.** El método `add_edge()` añade una arista al grafo entre dos nodos especificados. Si los nodos no existen previamente, se añaden al grafo.
- **Método get_edge_attributes.** El método `get_edge_attributes()` retorna los atributos de las aristas en el grafo. Esto puede incluir pesos, etiquetas u otros datos asociados a las aristas.
- **Método get_node_attributes.** Similar a `get_edge_attributes`, `get_node_attributes()` recupera los atributos asociados a los nodos en el grafo, como colores, tamaños o etiquetas personalizadas.
- **Método draw_networkx.** El método `draw_networkx()` se utiliza para visualizar el grafo. Permite una amplia gama de personalizaciones para la representación gráfica de los nodos, aristas y etiquetas.
- **Método draw_networkx_edge_labels.** `draw_networkx_edge_labels()` se usa para añadir etiquetas a las aristas en la visualización de un grafo. Esto es útil para mostrar pesos, tipos de relaciones o cualquier otro dato relevante asociado a las aristas.

A continuación, se muestra una porción de código fuente para representar, con *NetworkX* el siguiente grafo:

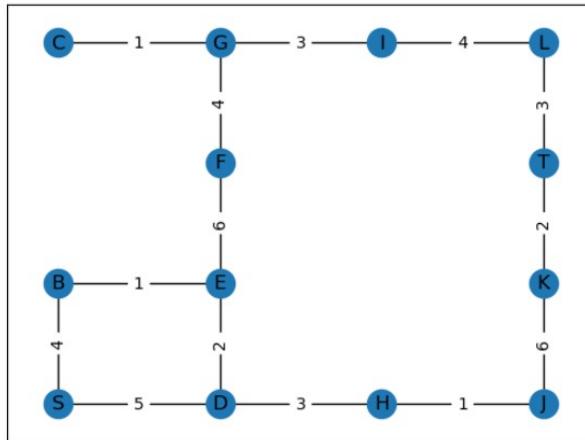


Figura 10: Ejemplo de un grafo con *NetworkX*

```
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_node('C', pos=(1,4))
g.add_node('D', pos=(2,1))
g.add_node('E', pos=(2,2))
g.add_node('F', pos=(2,3))
g.add_node('G', pos=(2,4))
g.add_node('H', pos=(3,1))
g.add_node('I', pos=(3,4))
g.add_node('J', pos=(4,1))
g.add_node('K', pos=(4,2))
g.add_node('T', pos=(4,3))
g.add_node('L', pos=(4,4))

g.add_edge('S', 'B', lenght=4)
g.add_edge('S', 'D', lenght=5)
g.add_edge('B', 'E', lenght=1)
g.add_edge('C', 'G', lenght=1)
g.add_edge('D', 'E', lenght=2)
g.add_edge('D', 'H', lenght=3)
g.add_edge('E', 'F', lenght=6)
g.add_edge('F', 'G', lenght=4)
g.add_edge('G', 'I', lenght=3)
g.add_edge('H', 'J', lenght=1)
g.add_edge('I', 'L', lenght=4)
g.add_edge('J', 'K', lenght=6)
g.add_edge('K', 'T', lenght=2)
g.add_edge('T', 'L', lenght=3)
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)
```

Más información sobre ejemplos de *NetworkX* en:

https://networkx.org/documentation/latest/auto_examples/index.html

6.2. ALGORITMO A*

El algoritmo A* es un algoritmo de búsqueda informada, ampliamente utilizado en la planificación de rutas y navegación. Combina características de los algoritmos de búsqueda de costo uniforme y búsqueda voraz, utilizando una función de evaluación $f(n) = g(n) + h(n)$, donde:

- $g(n)$ es el costo del camino desde el nodo inicial hasta n .
- $h(n)$ es una función heurística que estima el costo más bajo desde n hasta el objetivo.

Características del algoritmo A*

- Dado un grafo que representa un mapa, se busca el camino de un punto de origen a un destino.
- Este proceso se realiza mediante un algoritmo basado en *Forward Chaining*, que realiza una exploración en amplitud sin realizar “vuelta atrás”.
- La eficiencia del algoritmo se ve mejorada mediante el uso de una función heurística que estima el coste desde cada posible origen hasta el objetivo.
- Para garantizar la optimalidad del camino encontrado, la función heurística debe siempre subestimar el coste real; es decir, el valor de la heurística es menor o igual al coste real de alcanzar el objetivo. Un ejemplo de esto sería: “Tardo 1h en llegar a Albacete y son 100 kilómetros de distancia”.
- El algoritmo utiliza dos listas principales para su funcionamiento:
 - ABIERTA:** Contiene los nodos candidatos a formar parte de la ruta.
 - CERRADA:** Contiene los nodos que han sido seleccionados durante el proceso de búsqueda.

Con estas características el algoritmo A* se implementa en dos pasos:

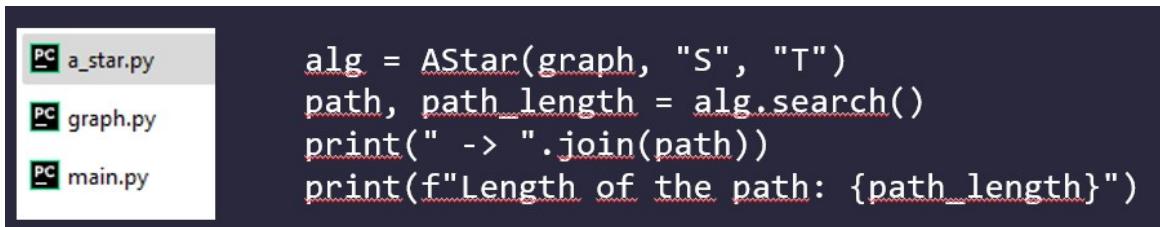
- 1) **Inicialización:** Se calcula la heurística del nodo origen y se añade a la lista ABIERTA
- 2) **Búsqueda:** A continuación se muestra un pseudocódigo para realizar la búsqueda

```
#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 > anterior
        si el nodo_hijo no est á ni en la lista ABIERTA ni en la CERRADA
            nodo_hijo.padre=nodo_seleccionado
            añadir nodo_hijo a la lista ABIERTA
```

Puedes explorar los ficheros que se adjuntan a este curso de especialización para ver cómo se ejecuta una versión del algoritmo A* por el grafo definido en la sección anterior:



```
a_star.py
graph.py
main.py

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

Figura 11: Ejemplo de código Python para el algoritmo A*

7. TEORÍA DE JUEGOS

La teoría de juegos es un área de las matemáticas aplicadas que utiliza modelos para analizar interacciones entre agentes racionales que toman decisiones dentro de un contexto estratégico. Estudia la toma de decisiones en situaciones donde el resultado para un participante depende de sus propias acciones y de las acciones de los demás. En otras palabras, se puede describir como el estudio de modelos matemáticos para interacciones estratégicas con agentes racionales.

7.1. DINÁMICAS DE JUEGOS CON INTELIGENCIA ARTIFICIAL

La aplicación de la teoría de juegos en el ámbito de la inteligencia artificial (IA) introduce una variedad de dinámicas interesantes que pueden ser clasificadas de la siguiente manera:

- **Simetría y Asimetría:** Los juegos pueden ser simétricos, donde todos los jugadores tienen las mismas estrategias disponibles, o asimétricos, donde las estrategias difieren entre los jugadores.
- **Información Perfecta o Imperfecta:** En los juegos de información perfecta, todos los jugadores conocen todas las jugadas realizadas. En contraste, en los juegos de información imperfecta, alguna información es desconocida, creando un escenario de incertidumbre.
- **Juegos Simultáneos o Secuenciales:** Los juegos simultáneos requieren que los jugadores realicen sus movimientos sin conocer los movimientos de los otros jugadores. Por otro lado, en los juegos secuenciales, los jugadores hacen sus movimientos uno después del otro, con conocimiento de las acciones previas.
- **Cooperativos:** Estos juegos permiten la formación de coaliciones y la cooperación entre jugadores para lograr un beneficio mutuo.
- **Suma Cero vs. No Suma Cero:** En los juegos de suma cero, las ganancias y las pérdidas de todos los jugadores suman cero, significando que lo que un jugador gana es exactamente la pérdida del otro. En los juegos de no suma cero, todos los jugadores pueden ganar o perder simultáneamente.



Figura 12: Dinámicas en teoría de juegos

Estas dinámicas son cruciales para el diseño y análisis de algoritmos de IA que participan en situaciones estratégicas y juegos, ya que dictan el conjunto de estrategias y enfoques computacionales apropiados para su participación exitosa.

7.2. UN EJEMPLO DE ÉXITO REAL: DEEP BLUE

Deep Blue, una supercomputadora de ajedrez creada por IBM, es un ejemplo destacado de la aplicación de la teoría de juegos en el diseño de inteligencia artificial. La victoria de Deep Blue sobre el campeón mundial de ajedrez Garry Kasparov en 1997 fue un hito significativo en el campo de la IA. La teoría de juegos se aplicó en Deep Blue de la siguiente manera:

- **Árbol de Juego:** Deep Blue utilizaba una técnica conocida como búsqueda de árbol de juego, que es una aplicación directa de la teoría de juegos. Analizaba millones de posibles movimientos y sus consecuencias, una estrategia que se alinea con el concepto de juegos secuenciales de información perfecta.
- **Evaluación de Posiciones:** Utilizando una función de evaluación compleja, Deep Blue podía aplicar principios de la teoría de juegos para evaluar y comparar las posiciones resultantes de los movimientos posibles, seleccionando la jugada que maximizaba su ventaja estratégica.
- **Minimax y Poda Alfa-Beta:** Estas técnicas son fundamentales en la teoría de juegos y fueron utilizadas por Deep Blue para minimizar la máxima ganancia posible de su oponente mientras maximiza la suya. La poda alfa-beta permitió a la supercomputadora descartar muchas posiciones que no necesitaba considerar, aumentando la eficiencia de su búsqueda.
- **Preparación de Aperturas y Finales:** Al estudiar miles de aperturas y finales de partidas de ajedrez, Deep Blue implementó el conocimiento estratégico de la teoría de juegos para tomar decisiones informadas desde el comienzo hasta el final de cada juego.

El éxito de Deep Blue contra Kasparov fue un testimonio de cómo la teoría de juegos, combinada con el poder de procesamiento computacional y el aprendizaje de máquina, puede resultar en un sistema de IA capaz de desempeñar tareas cognitivas complejas, como jugar al ajedrez a un nivel mundial.



Figura 13: Kasparov vs Deep Blue

7.3. EL ALGORITMO MINIMAX

El algoritmo *Minimax* es un algoritmo de decisión utilizado en teoría de juegos, lógica y ciencias de la computación para minimizar la posible pérdida en un escenario de peor caso (maximizar la pérdida mínima) o, alternativamente, maximizar la posible ganancia de un oponente. Es ampliamente utilizado en juegos de dos jugadores como ajedrez, damas, y tic-tac-toe, donde los jugadores se turnan.

El algoritmo funciona de la siguiente manera:

- **Árbol de Juego:** Se construye un árbol de juego completo a partir del estado actual del juego. Cada nodo representa un estado del juego y cada rama representa un movimiento posible.
- **Nodos Terminales:** Los nodos terminales del árbol representan los estados del juego donde se ha alcanzado un resultado final, ya sea una victoria, una derrota o un empate.
- **Función de Evaluación:** Para cada nodo terminal, se aplica una función de evaluación que asigna un valor numérico a la posición desde la perspectiva de un jugador.
- **Minimizar o Maximizar:** Los jugadores alternan capas del árbol, con un jugador tratando de maximizar el puntaje (maximizador) y el otro tratando de minimizar el puntaje (minimizador). El maximizador elige la jugada que lleva al mejor resultado, mientras que el minimizador elige la jugada que lleva al peor resultado para el maximizador.
- **Podar Alfa-Beta:** A menudo se utiliza en conjunto con el algoritmo Minimax para reducir el número de nodos evaluados en el árbol de búsqueda. Esta técnica recorta las ramas del árbol que no van a influir en la decisión final, mejorando la eficiencia del algoritmo.

A través de la utilización de *Minimax*, una IA puede planificar movimientos considerando todas las posibles respuestas del oponente y eligiendo la estrategia que resulte en el mejor resultado asegurado, dado el mejor juego del oponente. Esto hace que *Minimax* sea especialmente poderoso en juegos de suma cero de información perfecta.

La idea de *Minimax*

El *minimax* se basa en los siguientes conceptos para su implementación:

- 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.

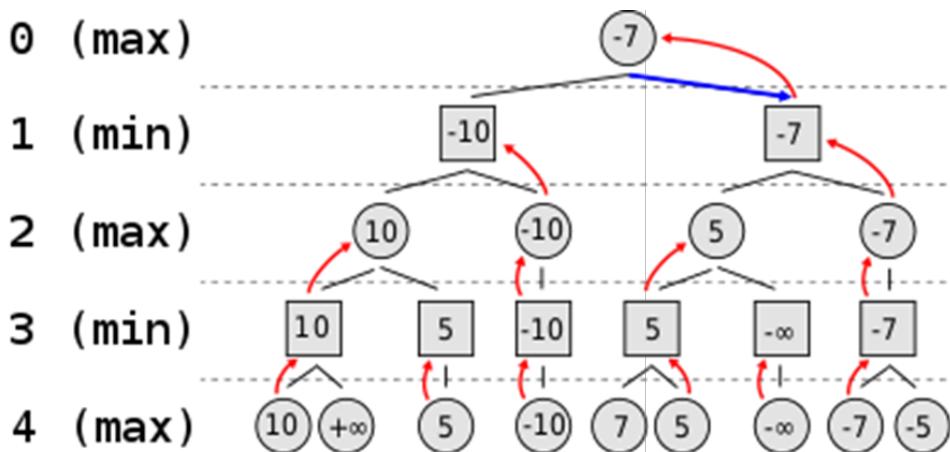


Figura 14: Ejemplo de cálculo de un árbol minimax

7.4. CÁLCULO DE VALORES EN UN ÁRBOL MINIMAX

El cálculo de los valores en un árbol *Minimax* se lleva a cabo mediante un proceso recursivo que comienza en los nodos terminales y asciende hacia el nodo raíz. Aquí se presenta un ejemplo simplificado del procedimiento:

Supongamos que tenemos un árbol de juego para *tic-tac-toe* con un nivel de nodos terminales que representan victorias, derrotas o empates. A cada uno de estos nodos se les asigna un valor: +1 para una victoria, 0 para un empate y -1 para una derrota.

- 1) **Asignar Valores a Nodos Terminales:** En el nivel más bajo del árbol, asignamos a cada nodo terminal su valor correspondiente según el resultado del juego.
- 2) **Aplicar Minimax en Nodos Intermedios:** En cada nivel intermedio, el algoritmo decide si está en un nivel de minimización o maximización.
 - En un nivel de maximización*, elegimos el valor más alto de entre los nodos hijos.
 - En un nivel de minimización*, seleccionamos el valor más bajo de entre los nodos hijos.
- 3) **Propagar Valores hacia Arriba:** Este proceso se repite, subiendo a través del árbol, hasta que se alcanza el nodo raíz.

4) **Movimiento Óptimo:** El valor en el nodo raíz dictará la mejor jugada posible para el jugador inicial, suponiendo que ambos jugadores juegan de manera óptima.

Por ejemplo, si el jugador con el turno es el maximizador y sus movimientos potenciales lo llevan a estados del juego con valores de -1, -1 y +1 respectivamente, elegirá el movimiento que resulta en +1.

Este método de cálculo asegura que, en cada punto del juego, se considera la mejor jugada posible teniendo en cuenta la estrategia óptima del oponente, lo cual es la esencia del algoritmo *Minimax*.

A continuación, proponemos un ejemplo de esquema *Minimax* en Python:

```
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

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

7.5. PODA ALFA-BETA EN EL ALGORITMO MINIMAX

La poda *alfa-beta* es una mejora del algoritmo *Minimax* que reduce significativamente el número de nodos que se evalúan en el árbol de búsqueda. Esta técnica se basa en dos valores: *alfa* y *beta*.

- **Alfa:** Es el valor de la mejor opción (más alta) encontrada hasta ahora en el camino hacia el nodo actual, para el maximizador. Inicialmente, alfa es $-\infty$.

- **Beta:** Es el valor de la mejor opción (más baja) encontrada hasta ahora en el camino hacia el nodo actual, para el minimizador. Inicialmente, beta es $+\infty$.

Durante el proceso de búsqueda, la poda *alfa-beta* utiliza estos dos valores para descartar ramas del árbol que no van a influir en la decisión final.

- Si en cualquier punto alfa es mayor o igual a beta, se corta (poda) esa rama del árbol, ya que no es necesario explorarla más. Esto se debe a que sabemos que el oponente ya ha encontrado una opción mejor en otro lugar del árbol.
- Cuando estamos evaluando un movimiento para el maximizador y encontramos un valor que es mayor que el actual valor alfa, actualizamos alfa con este nuevo valor. Si estamos en el nodo del minimizador y encontramos un valor que es menor que el actual valor beta, actualizamos beta con este nuevo valor.

La poda *alfa-beta* no afecta al resultado final del algoritmo *Minimax*, simplemente elimina partes del árbol que no necesitan ser exploradas, lo que resulta en una búsqueda mucho más rápida y eficiente.

Por ejemplo, si el maximizador tiene una opción con un valor alfa de 10 y el minimizador está considerando dos opciones, una con un valor de 15 y la otra aún no evaluada, el minimizador puede descartar la segunda opción sin evaluarla porque el maximizador ya tiene una opción mejor (10 en lugar de 15).

Esta técnica es especialmente poderosa en juegos complejos como el ajedrez, donde el número de posibles movimientos es muy grande.

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