INSTITUTO POLITÉCNICO NACIONAL

ESCOM

INTELIGENCIA ARTIFICIAL

LAB 2: BUSQUEDA NO INFORMADA P1

MARTÍNEZ CHÁVEZ JORGE ALEXIS

6CV3

20 SEPTIEMBRE 2024

Introducción

La búsqueda en profundidad (DFS) es una técnica de búsqueda no informada, que se utiliza comúnmente para explorar grafos o árboles en problemas de búsqueda. La idea principal bajando lo más profundo posible en una rama antes de retroceder y buscar otras alternativas. Aplicaciones de la búsqueda en profundidad se puede aplicar en una gran variedad de problemas, desde resolver laberintos hasta recorrer grafos o árboles. También se utiliza en la búsqueda de caminos, detección de ciclos y para encontrar soluciones a problemas de tipo "puzzle".

Algoritmos

Biblioteca

En la primera parte debemos de hacer la biblioteca para poder trabajar con las pilas, colas y listas.

Primero vamos a trabajar la pila en donde el ultimo objeto en entrar es el primer objeto en salir, para ello creamos un nodo en donde tiene como parámetros el valor y el siguiente objeto o nodo, que en el caso de las pilas será el nodo que se encuentra en la cima. Le creamos la función para ver si esta vacía, para ello simplemeten comparamos si esta None, entonces nos dicia que esta vacía y nos regresara el booleano de cual esta el estado.

```
class Pila:
    def __init__(self):
        self.cima = None

    def esta_vacia(self):
        return self.cima is None
```

Ahora le creamos la función para poder insertar nuevos objetos, recibe como parametros a la misma pila, y el nuevo nodo que estamos creado. Creamos el nuevo nodo y se le toma el parametro que esta recibiendo, y a nodo siguiente se le asigna el valor del nodo que esta en la cima de la misma pila.

```
    def push(self, objeto):
        nuevo_nodo = Nodo(objeto)
        nuevo_nodo.siguiente = self.cima
        self.cima = nuevo_nodo
```

Ahora para poder sacar un elemento primero debemos de comprobar si no esta vacía, para ello mandamos llamar la función que hemos creado, en caso de que no este vacía entonces sacara el ultimo elemento creado o la cima. Entonces simplemente guardaremos temporalmente el nodo cima como nodo removido, y al nodo cima le vamos asignar el nodo siguiente del nodo removido.

```
    def pop(self):
        if self.esta_vacia():
            return None
        nodo_removido = self.cima
        self.cima = self.cima.siguiente
        return nodo_removido.valor
```

Ahora simplemente para poder consultar haremos dos funciones, la primera para poder consultar cual es la cima y la otra para poder consultar toda la pila. En el caso de la cima retornaremos la cima y simplemete la imprimiremos. En el caso de toda la pila haremos un ciclo while en donde haremos un nodo de ayuda que es el actual al cual le asignaremos la cima, en caso de que ya no haya cima es que ya recorrimos toda la pila entonces terminaremos e iremos ir imprimiendo cada uno de los nodos.

```
    def ver_cima(self):
        if self.esta_vacia():
            return None
        return self.cima.valor

    def mostrar_pila(self):
        actual = self.cima
        elementos = []
        while actual:
            elementos.append(actual.valor)
            actual = actual.siguiente
        return elementos
```

Ahora hacemos una prueba, primero creamos la pila. Una vez creada le incertamos 3 objetos, despues pedimos que nos imprima toda la pila, depues hacemos un pop para poder sacar el ultimo objeto y volvemos a mostrar toda la pila para comprobar que realmente se hizo.

```
# Creando una pila
pila = Pila()
pila.push("objeto 1")
pila.push("objeto 2")
pila.push("objeto 3")
print("Pila:", pila.mostrar_pila())
print("Pop:", pila.pop())
print("Pila:", pila.mostrar_pila())
```

```
Pila: ['objeto 3', 'objeto 2', 'objeto 1']
Pop: objeto 3
Pila: ['objeto 2', 'objeto 1']
```

Ahora vamos a trabajar las colas en donde el primero entrar es el ultimo en salir, igual debemos de iniciar la pila en donde parametros es el primer nodo y el ultimo. Ahora hacemos el metodo para poder comprobar si la cola esta vacía para ello es simplemente hacer un booleano que regrese si esta vacía o no.

```
class Cola:
    def __init__(self):
        self.frente = None
        self.ultimo = None

    def esta_vacia(self):
        return self.frente is None
```

Ahora para poder insertar se recibe a la misma cola y al nuevo nodo que vamos a insertar con el valor que tiene, y comprobaremos que no este vacía, el caso de que si entonces al nodo frente y a ultimo le vamos a asignar el nuevo nodo que estamos recibiendo. En el caso de que no este vacía, entonces al valor de nodo siguiente del ultimo nodo le vamos a asignar el valor del nuevo nodo y el valor de ultimo nodo lo igualaremos con el del nodo recibido ya que ahora este es el ultimo.

```
def insertar(self, objeto):
    nuevo_nodo = Nodo(objeto)
    if self.esta_vacia():
        self.frente = self.ultimo = nuevo_nodo
    else:
        self.ultimo.siguiente = nuevo_nodo
        self.ultimo = nuevo_nodo
```

Ahora para eliminar un objeto, se elimina lo que este gurdado como el nodo frente, para ello primero comprobamos que no este vacío, luego a nodo removido le asignamos el nodo frente y al nodo frente ahora le asignaremos el nodo siguiente que tenia guardado este nodo frente.

```
def insertar(self, objeto):
    nuevo_nodo = Nodo(objeto)
    if self.esta_vacia():
        self.frente = self.ultimo = nuevo_nodo
    else:
        self.ultimo.siguiente = nuevo_nodo
        self.ultimo = nuevo_nodo
```

Ahora hacemos el recorrido de la cola para poderla imprimirla, iremos nodo por nodo y lo iremos almacenando en un arreglo y luego simplemente imprimirla.

```
def recorrer(self):
    actual = self.frente
    elementos = []
    while actual:
        elementos.append(actual.valor)
        actual = actual.siguiente
    return elementos
```

Ahora hacemos una prueba en donde hacemos la incerción de 3 objetos, imprimir la cola, depues eliminar el elemento que en este caso es el primer ingresado y depues volver a imprimir la cola para comprobar que se realizo la operación.

```python
# Creando una cola
cola = Cola()
cola.insertar("objeto A")
cola.insertar("objeto B")
cola.insertar("objeto C")
print("Cola:", cola.recorrer())
print("Quitar:", cola.quitar())
print("Cola:", cola.recorrer())
```

```
Cola: ['objeto A', 'objeto B', 'objeto C']
Quitar: objeto A
Cola: ['objeto B', 'objeto C']
```

Ahora para la lista general unicamente tendra como parametro el nodo primero, creamos la función para poder comprobar que no este vacía la lista simplemete con el booleano de que el nodo de inicio no sea vacío.

```python
class ListaGenerica:
    def __init__(self):
        self.primero = None

    def esta_vacia(self):
        return self.primero is None
```

Ahora para poder insertar un nuevo objeto primero en el caso de que este vacía la lista simplemente al nodo primero le vamos a asignar el valor del nuevo nodo, pero en caso de que no este vacía entonces nos ayudamos de un nodo que llamamos actula y almacenamos el valor del nodo primero e iremos recorriendo toda la lista hasta llegar al final esto con un bucle hasta que no haya un siguiente, entonces a este ultimo nodo en el espacio de nodo siguiente le asignaremos el valor del nuevo nodo.

```python
    def insertar(self, objeto):
        nuevo_nodo = Nodo(objeto)
        if self.esta_vacia():
            self.primero = nuevo_nodo
        else:
            actual = self.primero
            while actual.siguiente:
                actual = actual.siguiente
            actual.siguiente = nuevo_nodo
```

Ahora para poder eliminar un objeto primero comprobamos que no este vacía, despues debemos de hacer un recorrido para eliminar, ya que ahora podemos eliminar un nodo que este en cualquier posición. En caso de que el objeto a eliminar sea el primero objeto simplemente el nodo que esta la posición de siguiente nodo del nodo primero, se lo asignamos al nodo primero ya que este pasara a ser el nuevo nodo primero. En

caso de que el nodo se encuntre en otra posición entonces vamos a ayudarnos de un nuevo que llamaremos como anterior, cuando hayamos encontrado el nodo entonces a la variable de siguiente del nodo anterior le asignaremos el siguiente del actual, es decir como una cadena uniremos los eslabones del que quitamos.

```
    def eliminar(self, objeto):
        if self.esta_vacia():
            return False

        if self.primero.valor == objeto:
            self.primero = self.primero.siguiente
            return True

        actual = self.primero
        anterior = None
        while actual:
            if actual.valor == objeto:
                anterior.siguiente = actual.siguiente
                return True
            anterior = actual
            actual = actual.siguiente

        return False
```

Ahora haremos la prueba en donde insertamos 3 objetos, imprimiremos la lista eliminamos el de en medio y volvemos a imprimir la lista para comprobar que se realizo la operación.

```
# Creando una lista genérica
lista = ListaGenerica()
lista.insertar("elemento 1")
lista.insertar("elemento 2")
lista.insertar("elemento 3")
print("Lista:", lista.recorrer())
print("Eliminar 'elemento 2':", lista.eliminar("elemento 2"))
print("Lista:", lista.recorrer())
```

```
Lista: ['elemento 1', 'elemento 2', 'elemento 3']
Eliminar 'elemento 2': True
Lista: ['elemento 1', 'elemento 3']
```

4-Puzzle

Primero debemos de crear cada uno de los nodos, dentro del nodo le vamos a crear el estado, que básicamente es el como se encuentra el rompecabezas en ese momento, le vamos a asignar un padre que básicamente es la dirección del nodo padre y este nos servirá para mas tarde poder obtener el camino solución, tendremos también la acción que es el movimiento que se realizo para poder llegar al estado actual. También le crearemos una función propia del nodo para poder hacer el recorrido inverso y encontrar la solución, la acción será un bucle en donde ira almacenando en un arreglo los nodos solución e ira buscando hacía arriba mientras el parámetro de nodo padre se diferente de "None", una vez que haya guardado el arreglo simplemente lo invertiremos para que este en el orden de inicio a fin.

```python
#ESTRUCTURA DEL NODO
class Nodo:
    def __init__(self, estado, padre=None, accion=None):
        self.estado = estado      # es el tablero
        self.padre = padre        # nodo padre para ver el camino
        self.accion = accion      # que movimiento se hizo

    def obtener_camino(self):
        #Reconstruir el camino desde el nodo raíz hasta el nodo actual
        camino = []
        nodo_actual = self
        while nodo_actual.padre is not None:
            camino.append(nodo_actual.accion)
            nodo_actual = nodo_actual.padre
        camino.reverse()  # Invertimos el camino para que sea desde el inicio
        return camino
```

Ahora creamos la estrucutura de la pila basica en donde podremos tener la acciones de pode iniciar la pila en donde tiene como parametro el arreglo de los nodos, podremos hacer el push de nodos, el pop de los mismos y la función para comprobar si esta vacía la pila.

```python
#PILA
class Pila:
    def __init__(self):
        self.elementos = []

    def push(self, elemento):
        self.elementos.append(elemento)

    def pop(self):
        if not self.esta_vacia():
            return self.elementos.pop()
        else:
            return None

    def esta_vacia(self):
        return len(self.elementos) == 0
```

Ahora ya empezamos con la ejecución del puzzle, primero crearemos un estado iicial el cual es un estado aleatorio, recoredemo que es un simple arreglo del 0 al 3 en donde el 0 representa el estado vacío y debemos de ordenar de menor a mayor.

```
#GENERAMOS UN ESTADO INICIAL DEL JUEGO
def generar_estado_aleatorio():
    estado = [0, 1, 2, 3]
    random.shuffle(estado)
    return estado
```

He imprimimos este estado inicial, para mejor visualización vamos a representar el tablero como un tablero de 2x2.

```
#IMPRIMIR EL TABLERO
def imprimir_tablero(estado):
    print(f"{estado[0]} {estado[1]}")
    print(f"{estado[2]} {estado[3]}")
    print()
```

Ejecutamos el problema en donde creamos el estado inicial con la función que ya habiamos creado y mandamos llamar una nueva función que es la función principal del puzzle, donde le enviamos como parametro el estado inicial y el estado objetivo del tablero.

```
#EJECUTAMOS EL PROBLEMA
def resolver_puzzle():
    estado_inicial = generar_estado_aleatorio()  # Estado inicial aleatorio
    estado_objetivo = [1, 2, 3, 0]  # Estado objetivo (solución)

    print("Estado inicial del 4-puzzle:")
    imprimir_tablero(estado_inicial)

    # Ejecutamos DFS para resolver el puzzle
    solucion = puzzle4(estado_inicial, estado_objetivo)
```

En esta función empezamos creado principalmente 2 lista, la primera para los nodos frontera, que son los nodos que nos faltan visitar; la segunda los nodos visitados para no repetir estados y hacer un bucle infinito. Una vez creada la pila de frontera le

hacemos un push del nodo inicial con el estado que recibimos.

```
#HACEMOS LA BUSQUEDA DE LA SOLUCIÓN
def puzzle4(estado_inicial, estado_objetivo):
    # Cremaos la lista que son los nodo frontera ( nodos hijos ) y le asignamos una nueva pila
    frontera = Pila()
    nodo_inicial = Nodo(estado_inicial)
    frontera.push(nodo_inicial)

    # Usamos un conjunto para los nodos visitados
    visitados = set()
```

Ya ahora comenzamos con un bucle para podre empezar a crear el arbol y los nodos, este sera con un while que se seguira ejecutando mientras la pila de nodos frontera no este vacía ya que en cada ejecución hara un pop del nodo de hasta arriba. Comienza haciendo el pop del nodo, y guarda en una variable el estado del tablero del nodo actual; ahora comparamos si este estado es la solución, en caso de que si, va a retornar la lista que le regresa la función del nodo para hacer la inversa del camino solución; en caso de que este no sea la solución, entonces debera de crear los hijos de este nodo y agregarlos a la pila y agregar el nodo actual a la lista de los estados visitados. Ahora con otro ciclo crea los posibles movimientos los cuales explicaremos despues, si este nuevo movimiento no esta dentro de la lista de visitados entonces va a crear el nuevo nodo donde le asigna el estado que se crea con el movimiento, le asigna como nodo padre el nodo actual y el movimeinto que se hizo para llegar a este estado.

```
    # Bucle para poder crear a los hijos
    while not frontera.esta_vacia():
        nodo_actual = frontera.pop()
        estado_actual = nodo_actual.estado

        # Si encontramos la solución, regresamos el camino
        if estado_actual == estado_objetivo:
            return nodo_actual.obtener_camino()

        # Añadimos el estado actual a los visitados
        visitados.add(tuple(estado_actual))

        # Creamos a los nodos hijos
        for direccion in ['arriba', 'abajo', 'izquierda', 'derecha']:
            nuevo_estado = mover_ficha(estado_actual, direccion)
            #Verificamos que no hayamos ya creado ese estado
            if tuple(nuevo_estado) not in visitados:
                nuevo_nodo = Nodo(nuevo_estado, nodo_actual, direccion)
                frontera.push(nuevo_nodo)

    # Si no se encuentra solución
    return None
```

Para crear los movimientos los hacemos con el bucle de arriba, derecha izquierda y abajo; los cuales son los posibles moviemitos, pero al ser de 2x2 no se pueden crear todos los movimientos, entonces hacemos la comparación. Para moverse hacía arriba solo se puede si el indice del espacio vacío es mayor a 1, hacía abajo si el index es menor a 1, a la izquierda si el indice es 1 o 3 es decir un numero impar, y a la derecha si es un numero par.

```python
#CREAMOS LOS POSIBLES MOVIMIENTOS DE LA FICHA
def mover_ficha(estado, direccion):
    # Encontramos la posición del espacio vacío (0)
    index_espacio = estado.index(0)
    nuevo_estado = estado[:]

    # Definimos las posibles direcciones y los movimientos válidos
    if direccion == 'arriba' and index_espacio > 1:  # Solo podemos mover hacia arriba si no estamos
        nuevo_estado[index_espacio], nuevo_estado[index_espacio - 2] = nuevo_estado[index_espacio - 2
    elif direccion == 'abajo' and index_espacio < 2:  # Solo podemos mover hacia abajo si no estamos
        nuevo_estado[index_espacio], nuevo_estado[index_espacio + 2] = nuevo_estado[index_espacio + 2
    elif direccion == 'izquierda' and index_espacio % 2 != 0:  # Solo podemos mover hacia la izquierd
        nuevo_estado[index_espacio], nuevo_estado[index_espacio - 1] = nuevo_estado[index_espacio - 1
    elif direccion == 'derecha' and index_espacio % 2 != 1:  # Solo podemos mover hacia la derecha si
        nuevo_estado[index_espacio], nuevo_estado[index_espacio + 1] = nuevo_estado[index_espacio + 1

    return nuevo_estado
```

Por ultimo si la función de puzzle encontro una solución, esta se lo regresa a la función principal, el cual se encargara de hacer las respectivas impresiones de la solución agregando el numero de la instrucción asi como el movimiento que se debe de realizar.

```python
    # Ejecutamos DFS para resolver el puzzle
    solucion = puzzle4(estado_inicial, estado_objetivo)

    # Mostramos el resultado paso a paso
    if solucion:
        print("Se encontró una solución. Movimientos a realizar:")
        estado_actual = estado_inicial
        for i, movimiento in enumerate(solucion):
            print(f"Movimiento {i+1}: {movimiento}")
            estado_actual = mover_ficha(estado_actual, movimiento)
            imprimir_tablero(estado_actual)
    else:
        print("No se encontró ninguna solución.")
```

Ahora hacemos un ejemplo en ejecución y nos mostrara lo siguiente.

```
.exe c:/Users/jorge/OneDrive/Documentos/JORGE/ESCOM/9_SEMESTRE/IA/LABORATORIO/2_BUSQUEDA_NO_INFORMADA_P1/4puzzle.py
Estado inicial del 4-puzzle:
0 3
2 1

Se encontró una solución. Movimientos a realizar:
Movimiento 1: derecha
3 0
2 1

Movimiento 2: abajo
3 1
2 0

Movimiento 3: izquierda
3 1
0 2

Movimiento 4: arriba
0 1
3 2

Movimiento 5: derecha
1 0
3 2

Movimiento 6: abajo
1 2
3 0
```

Laberinto

Primero creamos el tablero en donde los 0 nos representan los espacios vacíos y los 1 nos representa que existen paredes. Así como las posiciones de inicio y final que es nuestro nodo objetivo.

```
laberinto = [
    [1, 0, 1, 1, 1],
    [1, 0, 0, 0, 1],
    [1, 1, 1, 0, 1],
    [1, 0, 0, 0, 0],
    [1, 1, 1, 1, 1]
]

# Posición de inicio y salida
inicio = (0, 1)  # Coordenadas (fila, columna) de inicio
fin = (3, 4)     # Coordenadas (fila, columna) de salida
```

En este caso vamos a simplificarlo y no crear las funciones de la pila, ocuparemos las funciones de add y pop que ya tiene Python, pero si creamos la función para poder imprimir el laberinto, en donde simplemente son 2 bucles para poder recorres la matriz bidimensional e ira imprimiendo posición por posición.

```
#MOSTRAR EL LABERINTO
def imprimir_laberinto(laberinto, posicion_actual):
    for i in range(len(laberinto)):
        for j in range(len(laberinto[i])):
            if (i, j) == posicion_actual:
                print("P", end=" ")
            else:
                print(laberinto[i][j], end=" ")
        print()
    print()
```

Ahora ejecutamos la solucoión del problema, en donde mandaremos llamar la función de resolver el laberinto y nos regresara una lista con las posiciónes que debemos de seguir para poder resolverlo.

```
# Llamada a la función principal para resolver el laberinto
def solucion_laberinto():
    camino_solucion = resolver_laberinto(laberinto, inicio, fin)

    # Imprimir el resultado
    if camino_solucion:
        print("Se encontró un camino. El laberinto resuelto es:")
    else:
        print("No se pudo resolver el laberinto.")

solucion_laberinto()
```

En la función de resolver el laberinto recibe como parametros el laberinto y las posiciones de inicio y la objetivo, ahora creamos la pila la cual va a tener el nodo y le ingresamos el inicio, tambien creamos la lista para los nodos ya visitados.

```
#FUNCION QUE CREA LOS NODOS HIJOS
def resolver_laberinto(laberinto, inicio, fin):
    pila = [Nodo(inicio)]
    visitados = set()
```

Ahora empezamos con el ciclo para buscar la solución, primero le hacemos un pop a la pila para obtener el ultimo nodo ingresado, lo comparamos con la posicion objetivo, si esta es la solución entonces hemos llegado a la salida e imprimimos la solución.

```
    while pila:
        nodo_actual = pila.pop()
        fila_actual, columna_actual = nodo_actual.posicion

        # Imprimir el laberinto con la posición actual
        imprimir_laberinto(laberinto, nodo_actual.posicion)

        # Si llegamos a la posición final, se reconstruye el camino y se imprime
        if nodo_actual.posicion == fin:
            print("¡Has llegado a la salida!")
            return obtener_camino(nodo_actual)
```

En caso de que no sea la solución entonces lo agregamos a la lista de nodos visiitados y le creamos lo posibles hijos de este nodo, si se mueve hacía arriba es restarle una posición a la fila, si se mueve hacía abajo es sumarle una posición a la fila, si se mueve a la izquierda es restale una posición a la columna y si se mueve a la derecha es sumarle una posición a la columna. Una vez que se hayan creado todos los

movimientos posibles, entonces lo vamos a agregar a la pila frontera.

```
        # Marcar la posición actual como visitada
        visitados.add(nodo_actual.posicion)

        # Movimientos posibles: arriba, abajo, izquierda, derecha
        movimientos = [
            (fila_actual - 1, columna_actual),  # Arriba
            (fila_actual + 1, columna_actual),  # Abajo
            (fila_actual, columna_actual - 1),  # Izquierda
            (fila_actual, columna_actual + 1)   # Derecha
        ]

        # Agregar los movimientos válidos a la pila, creando un nuevo nodo para cada movimiento
        for movimiento in movimientos:
            fila, columna = movimiento
            if es_movimiento_valido(laberinto, fila, columna, visitados):
                nuevo_nodo = Nodo(movimiento, nodo_actual)  # Crear un nuevo nodo enlazado al nodo act
                pila.append(nuevo_nodo)

    print("No se encontró solución.")
    return None
```

Para poder comprobar si el movimiento es valido debemos de crear una función y poder comparar que la suma o resta ya sea en columna o fila no debe se superar los parametros de la orilla o que no exista una pared o que el estado del laberinto que se esta creando no este dentro de la lista de ya visitados.

```
# MOVIMIENTO VALIDO
def es_movimiento_valido(laberinto, fila, columna, visitados):
    if 0 <= fila < len(laberinto) and 0 <= columna < len(laberinto[0]):
        if laberinto[fila][columna] == 0 and (fila, columna) not in visitados:
            return True
    return False
```

Cunaod lleguemos a la solución objetivo simplemente lo indicaremos y haremos la impresión del laberinto.

```
#MOSTRAR EL LABERINTO
def imprimir_laberinto(laberinto, posicion_actual):
    for i in range(len(laberinto)):
        for j in range(len(laberinto[i])):
            if (i, j) == posicion_actual:
                print("P", end=" ")
            else:
                print(laberinto[i][j], end=" ")
        print()
    print()
```

Ahora hacemos la prueba de ejecución.

```
PS C:\Users\jorge\OneDrive\Documentos\JORGE\ESCOM\9_SEMESTRE> & C:/Users/jorge/AppData/Local/Programs/Pyt
.exe c:/Users/jorge/OneDrive/Documentos/JORGE/ESCOM/9_SEMESTRE/IA/LABORATORIO/2_BUSQUEDA_NO_INFORMADA_P1/
1 P 1 1 1
1 0 0 0 1
1 1 1 0 1
1 0 0 0 0
1 1 1 1 1

1 0 1 1 1
1 P 0 0 1
1 1 1 0 1
1 0 0 0 0
1 1 1 1 1

1 0 1 1 1
1 0 P 0 1
1 1 1 0 1
1 0 0 0 0
1 1 1 1 1

1 0 1 1 1
1 0 0 P 1
1 1 1 0 1
1 0 0 0 0
1 1 1 1 1

1 0 1 1 1
1 0 0 0 1
1 1 1 P 1
1 0 0 0 0
1 1 1 1 1

1 0 1 1 1
1 0 0 0 1
1 1 1 0 1
1 0 0 P 0
1 1 1 1 1
```

```
1 0 1 1 1
1 0 0 0 1
1 1 1 0 1
1 0 0 0 P
1 1 1 1 1

¡Has llegado a la salida!
Se encontró un camino. El laberinto resuelto es:
PS C:\Users\jorge\OneDrive\Documentos\JORGE\ESCOM\9_SEMESTRE>
```

Enlace

Se comparte el siguiente enlace: https://github.com/Jorge300403/IA_6CV3_MartinezChavez/tree/main/LAB_2_BUSQUEDA_NO_INFORMADA_P1 que redirecciona a la carpeta correspondiente a esta sesión de laboratorio donde se encuentran todos los códigos realizados.