A logo with text on it

Description automatically generatedA logo with text and symbols

Description automatically generated

**Sistemas de Búsqueda y Razonamiento**

**Alumnos:** Adrian Alejandro González Domínguez – 359834

Héctor Daniel Medrano Meza — 361345

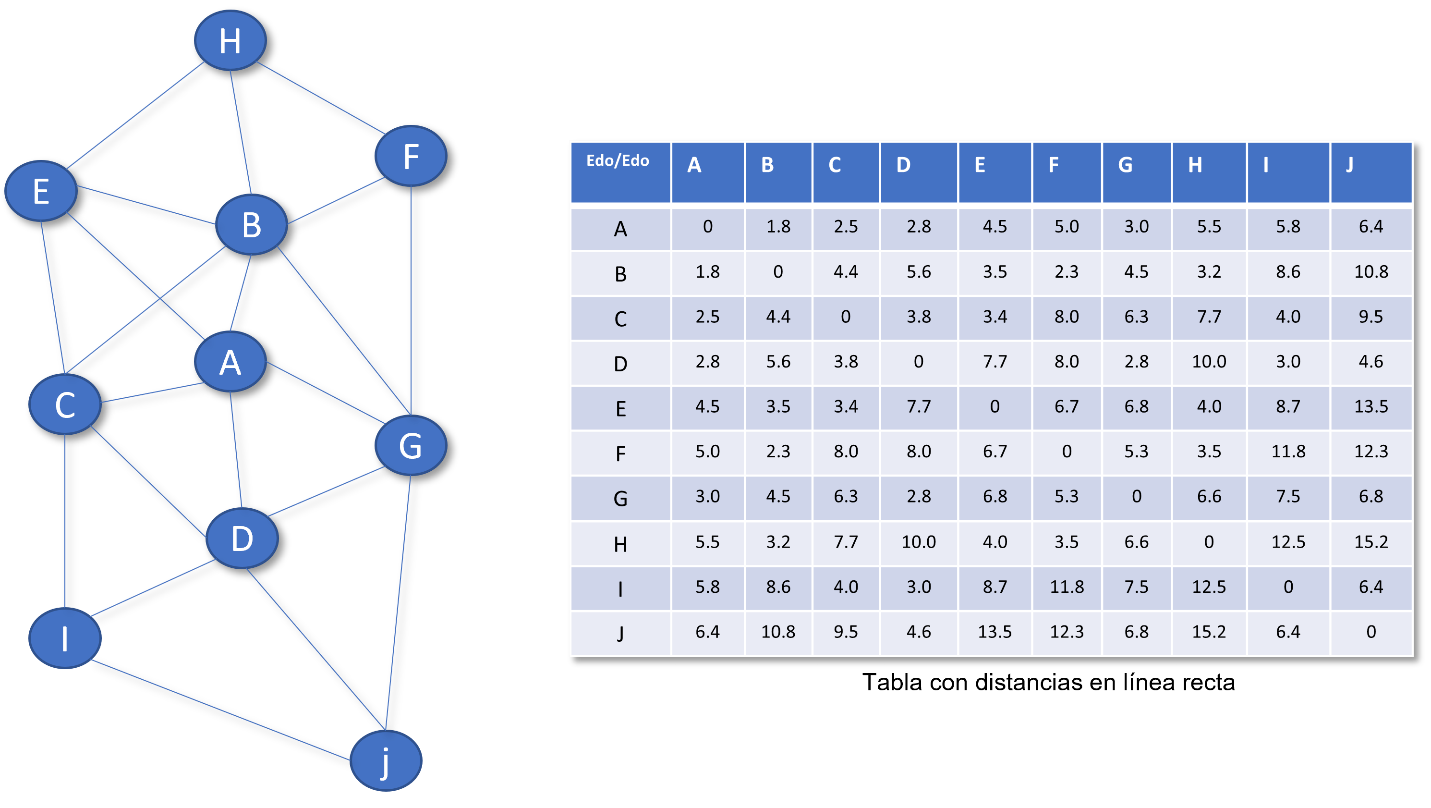
Manuel Abraham Escudero Moreno — 355208  
 Emiliano Rivera Armendáriz - 358193

**Docente:** Ricardo Majalca Martinez

**Fecha:** 2024-05-27

(a) Supongamos el siguiente espacio estado, al que le acompaña una tabla de distancias en línea recta entre cualquier par de estados.

**Proyecto final**



Supongamos que tenemos al estado de inicio 𝑖𝑛𝑖 y tenemos al estado 𝑚𝑒𝑡𝑎, porque se los hemos pedido al usuario de nuestro programa. Entonces la distancia en línea recta entre cualquier estado

𝐴 y 𝐵 es:

𝑑𝑙𝑟[𝑒𝑑𝑜, 𝑚𝑒𝑡𝑎]

Luego supongamos que hemos decidido que nuestra función de costo es para llegar a un estado

𝑒𝑑𝑜𝑘 desde un estado 𝑒𝑑𝑜𝑘−1 es:

𝑔(𝑒𝑑𝑜𝑘) = 𝑑𝑙𝑟[𝑒𝑑𝑜𝑘−1, 𝑒𝑑𝑜𝑘] + 𝑔(𝑒𝑑𝑜𝑘−1)

Naturalmente que 𝑔(𝑖𝑛𝑖) = 𝑑𝑙𝑟[𝑖𝑛𝑖, 𝑖𝑛𝑖] = 0. Además, también hemos decidido que usaremos cono función de valor heurístico a esta misma tabla de distancias en línea recta. Por ejemplo. El valor heurístico de cualquier estado 𝑒𝑑𝑜 es su distancia en línea recta hacia la 𝑚𝑒𝑡𝑎:

ℎ(𝑒𝑑𝑜) = 𝑑𝑙𝑟[𝑒𝑑𝑜, 𝑚𝑒𝑡𝑎]

Así que todo lo que necesitamos para implementar cualquier técnica de búsqueda informada lo tenemos ya en la tabla anterior. Así las cosas, implementemos usando el lenguaje de nuestra preferencia las siguientes técnicas de búsqueda informada:

1. Búsqueda primero por lo mejor

2. Búsqueda avara

3. Búsqueda A\*

Presentemos nuestros resultados en un reporte en formato PDF que incluya una breve descripción del problema, una breve descripción de nuestra implementación, el código del script o programa que hemos usado para implementar la solución, así como una captura de como el programa pide la información de entrada (estados de inicio y meta) y como nos reporta el camino u orden de visita para llegar del estado de inicio al meta.

1. Búsqueda primero por lo mejor

**Código**

import time

# =========================================================

# Implementaciones relacionadas al algoritmo Primero por lo Mejor

# =========================================================

def best\_first(start\_node, goal\_node):

    current = (None, None, None) # Tupla encargada de llevar registro del nodo actual (1), nodo desde el que se llegó (2), y costo (3)

    open = [] # Lista de tuplas encargada de llevar registro de la lista Abiertos. Nodo (1), nodo desde el que se llegó (2), y costo (3)

    closed = [] # Lista de tuplas encargada de llevar registro de la lista Cerrados. Nodo (1), nodo desde el que se llegó (2), y costo (3)

    path = []

    print()

    print("Actual: -")

    print(f"Abiertos: [{start\_node.name}\_{start\_node.name}: {straight\_line\_distance[start\_node.name, start\_node.name]}]")

    print("Cerrados: []")

    new\_tuple = (start\_node, start\_node, straight\_line\_distance[start\_node.name, start\_node.name])

    open.append(new\_tuple)

    while current[0] is not goal\_node:

        open\_lowest = open.pop(get\_lowest\_tuple\_index(open))

        current = (open\_lowest[0], open\_lowest[1], open\_lowest[2])

        for neighbor in current[0].neighbors:

            # Si el nodo no ha sido cerrado, se crea otra tupla y se añade a Abiertos

            if not node\_is\_in\_closed(neighbor, closed):

                new\_tuple = (neighbor, current[0], straight\_line\_distance[neighbor.name, current[0].name]+current[2])

                open.append(new\_tuple)

            open = clean\_open\_from\_duplicates(open) # Se buscan los nodos duplicados dentro de Abiertos para dejar solo uno, el de menor costo

        closed.append(current)

        print\_step(current, open, closed)

    current = closed[len(closed) - 1]

    while current[0] is not start\_node:

        path.append(current[0].name)

        for tuple in closed:

            if tuple[0] == current[1]:

                current = tuple

    path.append(start\_node.name)

    path = path[::-1]

    print()

    print(f"El mejor camino para llegar del nodo {start\_node.name} al {goal\_node.name} es:")

    for i in range(len(path)):

        print(f"{path[i]}", end="" if i == len(path) - 1 else " -> ")

    print()

def get\_lowest\_tuple\_index(tuple\_array):

    lowest = float('inf')

    index\_lowest = None

    for i in range(len(tuple\_array)):

        # Se busca el tercer valor de las tuplas, el cual represeta el costo, para buscar el mínimo

        if tuple\_array[i][2] < lowest:

            lowest = tuple\_array[i][2]

            index\_lowest = i

    return index\_lowest

def node\_is\_in\_closed(node\_to\_check, closed):

    for tuple in closed:

        if node\_to\_check.name == tuple[0].name:

            return True

    return False

def clean\_open\_from\_duplicates(open):

    nodes\_accounted\_for = []

    clean\_open = [] # Arreglo de tuplas donde se guardan los valores minimos de cada nodo

    for tuple in open:

        if tuple[0] not in nodes\_accounted\_for: # Si el nodo no ha sido tomado en cuenta, automáticamente se considera el menor y se mete a la lista

            nodes\_accounted\_for.append(tuple[0])

            clean\_open.append(tuple)

        else: # Si el nodo ya está metido en la lista, significa que debe de compararse con el otro para ver cuál tiene menor peso

            for i in range(len(clean\_open)):

                if clean\_open[i][0].name == tuple[0].name:

                    if tuple[2] < clean\_open[i][2]: # Si el costo del nodo es menor al del mismo nodo ya registrado, lo reemplaza

                        clean\_open[i] = tuple

                    break

    return clean\_open

def print\_step(current, open, closed):

    # Impimir actual

    print()

    print(f"Actual: {current[0].name}\_{current[1].name}: {current[2]}")

    # Imprimir lista Abiertos

    print(f"Abiertos: [", end="")

    for tuple in open:

        print(f"{tuple[0].name}\_{tuple[1].name}: {tuple[2]}", end=", ")

    print(f"]")

    # Imprimir lista Cerrados

    print(f"Cerrados: [", end="")

    for tuple in closed:

        print(f"{tuple[0].name}\_{tuple[1].name}: {tuple[2]}", end=", ")

    print(f"]")

# =========================================================

# Implementaciones relacionadas a los nodos

# =========================================================

def get\_node(node\_list, name):

    for node in node\_list:

        if node.name == name:

            return node

    return None

class Node:

    def \_\_init\_\_(self, name):

        self.name = name

        self.neighbors = []

    def add\_neighbor(self, neighbor):

        self.neighbors.append(neighbor)

    def print\_neighbors(self):

        print(f'Neighbors of {self.name}:')

        for neighbor in self.neighbors:

            print(f'  {neighbor.name}')

        print()

# =========================================================

# Creación del mapa de nodos y sus distancias entre sí

# =========================================================

node\_names = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']

node\_list = [Node(name) for name in node\_names]

node\_list[0].add\_neighbor(node\_list[1])  # A -> B

node\_list[0].add\_neighbor(node\_list[2])  # A -> C

node\_list[0].add\_neighbor(node\_list[3])  # A -> D

node\_list[0].add\_neighbor(node\_list[4])  # A -> E

node\_list[0].add\_neighbor(node\_list[6])  # A -> G

node\_list[1].add\_neighbor(node\_list[0])  # B -> A

node\_list[1].add\_neighbor(node\_list[2])  # B -> C

node\_list[1].add\_neighbor(node\_list[4])  # B -> E

node\_list[1].add\_neighbor(node\_list[5])  # B -> F

node\_list[1].add\_neighbor(node\_list[6])  # B -> G

node\_list[1].add\_neighbor(node\_list[7])  # B -> H

node\_list[2].add\_neighbor(node\_list[0])  # C -> A

node\_list[2].add\_neighbor(node\_list[1])  # C -> B

node\_list[2].add\_neighbor(node\_list[3])  # C -> D

node\_list[2].add\_neighbor(node\_list[4])  # C -> E

node\_list[2].add\_neighbor(node\_list[8])  # C -> I

node\_list[3].add\_neighbor(node\_list[0])  # D -> A

node\_list[3].add\_neighbor(node\_list[2])  # D -> C

node\_list[3].add\_neighbor(node\_list[6])  # D -> G

node\_list[3].add\_neighbor(node\_list[8])  # D -> I

node\_list[3].add\_neighbor(node\_list[9])  # D -> J

node\_list[4].add\_neighbor(node\_list[0])  # E -> A

node\_list[4].add\_neighbor(node\_list[1])  # E -> B

node\_list[4].add\_neighbor(node\_list[2])  # E -> C

node\_list[4].add\_neighbor(node\_list[7])  # E -> H

node\_list[5].add\_neighbor(node\_list[1])  # F -> B

node\_list[5].add\_neighbor(node\_list[6])  # F -> G

node\_list[5].add\_neighbor(node\_list[7])  # F -> H

node\_list[6].add\_neighbor(node\_list[0])  # G -> A

node\_list[6].add\_neighbor(node\_list[1])  # G -> B

node\_list[6].add\_neighbor(node\_list[3])  # G -> D

node\_list[6].add\_neighbor(node\_list[5])  # G -> F

node\_list[6].add\_neighbor(node\_list[9])  # G -> J

node\_list[7].add\_neighbor(node\_list[1])  # H -> B

node\_list[7].add\_neighbor(node\_list[4])  # H -> E

node\_list[7].add\_neighbor(node\_list[5])  # H -> F

node\_list[8].add\_neighbor(node\_list[2])  # I -> C

node\_list[8].add\_neighbor(node\_list[3])  # I -> D

node\_list[8].add\_neighbor(node\_list[9])  # I -> J

node\_list[9].add\_neighbor(node\_list[3])  # J -> D

node\_list[9].add\_neighbor(node\_list[6])  # J -> G

node\_list[9].add\_neighbor(node\_list[8])  # J -> I

straight\_line\_distance = {

    ('A', 'A'): 0, ('A', 'B'): 1.8, ('A', 'C'): 2.5, ('A', 'D'): 2.8, ('A', 'E'): 4.5, ('A', 'F'): 5.0,

    ('A', 'G'): 3.0, ('A', 'H'): 5.5, ('A', 'I'): 5.8, ('A', 'J'): 6.4,

    ('B', 'A'): 1.8, ('B', 'B'): 0, ('B', 'C'): 4.4, ('B', 'D'): 5.6, ('B', 'E'): 3.5, ('B', 'F'): 2.3,

    ('B', 'G'): 4.5, ('B', 'H'): 3.2, ('B', 'I'): 8.6, ('B', 'J'): 10.8,

    ('C', 'A'): 2.5, ('C', 'B'): 4.4, ('C', 'C'): 0, ('C', 'D'): 3.8, ('C', 'E'): 3.4,

    ('C', 'F'): 8.0, ('C', 'G'): 6.3, ('C', 'H'): 7.7, ('C', 'I'): 4.0, ('C', 'J'): 9.5,

    ('D', 'A'): 2.8, ('D', 'B'): 5.6, ('D', 'C'): 3.8, ('D', 'D'): 0, ('D', 'E'): 7.7,

    ('D', 'F'): 8.0, ('D', 'G'): 2.8, ('D', 'H'): 10.0, ('D', 'I'): 3.0, ('D', 'J'): 4.6,

    ('E', 'A'): 4.5, ('E', 'B'): 3.5, ('E', 'C'): 3.4, ('E', 'D'): 7.7, ('E', 'E'): 0,

    ('E', 'F'): 6.7, ('E', 'G'): 6.8, ('E', 'H'): 4.0, ('E', 'I'): 8.7, ('E', 'J'): 13.5,

    ('F', 'A'): 5.0, ('F', 'B'): 2.3, ('F', 'C'): 8.0, ('F', 'D'): 8.0, ('F', 'E'): 6.7,

    ('F', 'F'): 0, ('F', 'G'): 5.3, ('F', 'H'): 3.5, ('F', 'I'): 11.8, ('F', 'J'): 12.3,

    ('G', 'A'): 3.0, ('G', 'B'): 4.5, ('G', 'C'): 6.3, ('G', 'D'): 2.8, ('G', 'E'): 6.8,

    ('G', 'F'): 5.3, ('G', 'G'): 0, ('G', 'H'): 6.6, ('G', 'I'): 7.5, ('G', 'J'): 6.8,

    ('H', 'A'): 5.5, ('H', 'B'): 3.2, ('H', 'C'): 7.7, ('H', 'D'): 10.0, ('H', 'E'): 4.0,

    ('H', 'F'): 3.5, ('H', 'G'): 6.6, ('H', 'H'): 0, ('H', 'I'): 12.5, ('H', 'J'): 15.2,

    ('I', 'A'): 5.8, ('I', 'B'): 8.6, ('I', 'C'): 4.0, ('I', 'D'): 3.0, ('I', 'E'): 8.7,

    ('I', 'F'): 11.8, ('I', 'G'): 7.5, ('I', 'H'): 12.5, ('I', 'I'): 0, ('I', 'J'): 6.4,

    ('J', 'A'): 6.4, ('J', 'B'): 10.8, ('J', 'C'): 9.5, ('J', 'D'): 4.6, ('J', 'E'): 13.5,

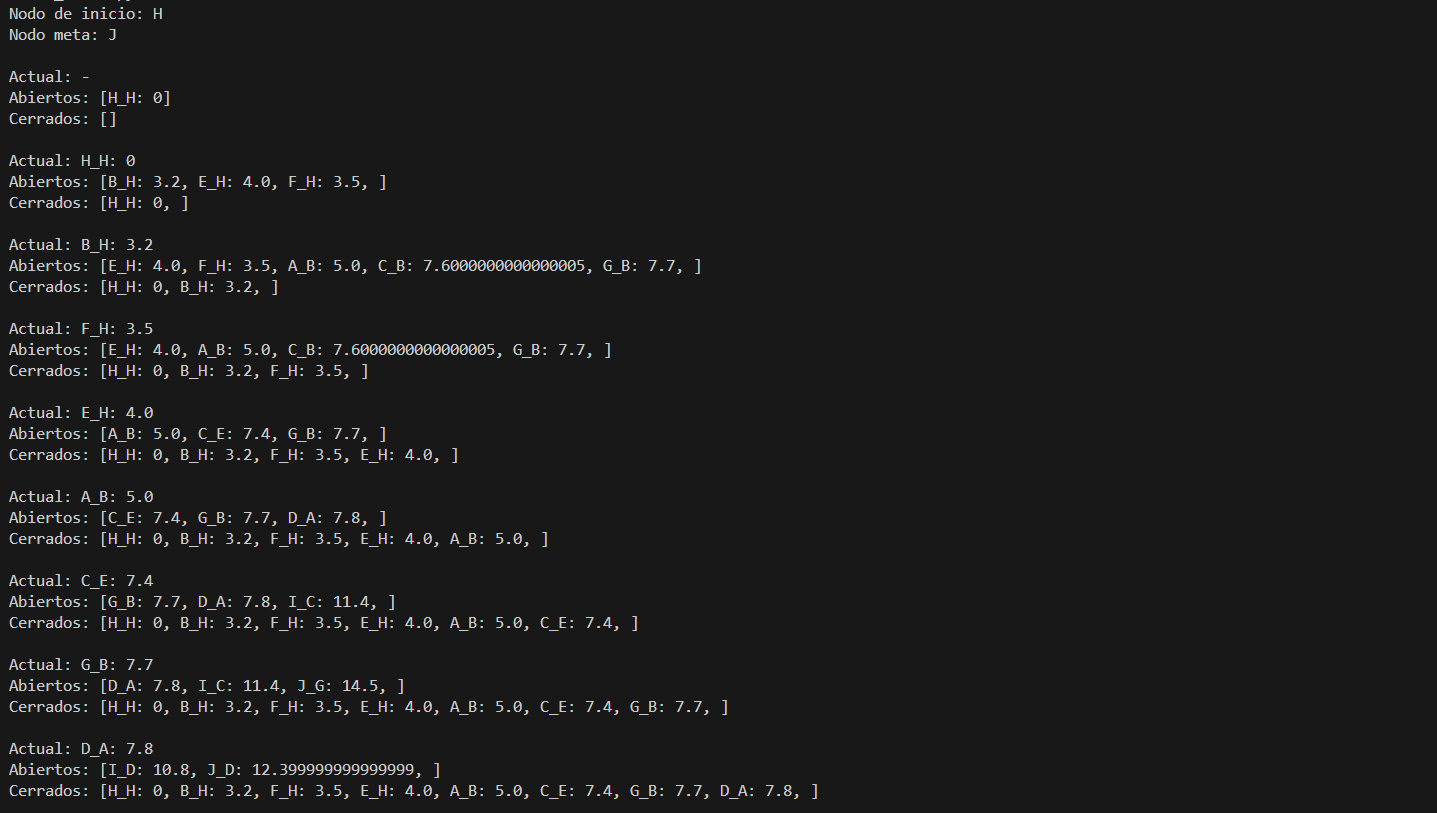
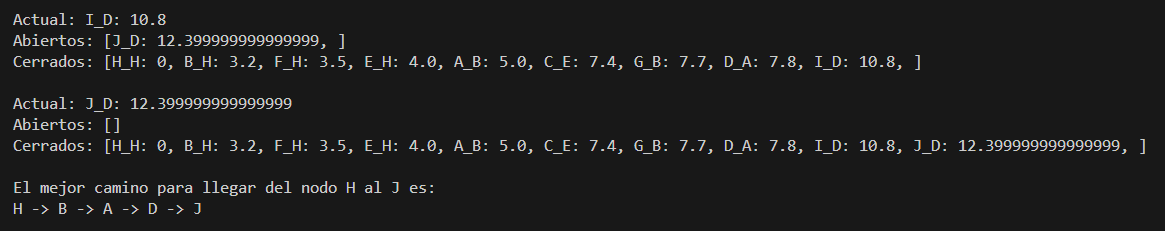
    ('J', 'F'): 12.3, ('J', 'G'): 6.8, ('J', 'H'): 15.2, ('J', 'I'): 6.4, ('J', 'J'): 0

}

start\_node = input("Nodo de inicio: ")

goal\_node = input("Nodo meta: ")

best\_first(get\_node(node\_list, start\_node.upper()), get\_node(node\_list, goal\_node.upper()))

**Ejecución**  
  


1. Búsqueda avara

**Código**

En la implementación greedy o avara, casi todo el código se mantiene igual, así que solo explicaremos las cosas que cambian.

A la hora el valor hasta un nodo, se realizará con la distancia en línea recta del nodo en cuestión al nodo destino. Esto aplica al iniciar la búsqueda,

new\_tuple = (start\_node, start\_node, straight\_line\_distance[start\_node.name, goal\_node.name])

open.append(new\_tuple)

y en el paso donde agregamos los vecinos a abiertos

new\_tuple = (neighbor, current[0], straight\_line\_distance[neighbor.name, goal\_node.name])

open.append(new\_tuple)

import time

# =========================================================

# Implementaciones relacionadas al algoritmo aváro

# =========================================================

def greedy(start\_node, goal\_node):

    current = (None, None, None) # Tupla encargada de llevar registro del nodo actual (1), nodo desde el que se llegó (2) y distancia en línea recta al nodo destino (3)

    open = [] # Lista de tuplas encargada de llevar registro de la lista Abiertos. Nodo (1), nodo desde el que se llegó (2) y distancia en línea recta al nodo destino (3)

    closed = [] # Lista de tuplas encargada de llevar registro de la lista Cerrados. Nodo (1), nodo desde el que se llegó (2) y distancia en línea recta al nodo destino (3)

    path = []

    print()

    print("Actual: -")

    print(f"Abiertos: [{start\_node.name}\_{start\_node.name}: {straight\_line\_distance[start\_node.name, goal\_node.name]}]")

    print("Cerrados: []")

    new\_tuple = (start\_node, start\_node, straight\_line\_distance[start\_node.name, goal\_node.name])

    open.append(new\_tuple)

    while current[0] is not goal\_node:

        open\_lowest = open.pop(get\_lowest\_tuple\_index(open))

        current = (open\_lowest[0], open\_lowest[1], open\_lowest[2])

        for neighbor in current[0].neighbors:

            # Si el nodo no ha sido cerrado, se crea otra tupla y se añade a Abiertos

            if not node\_is\_in\_closed(neighbor, closed):

                new\_tuple = (neighbor, current[0], straight\_line\_distance[neighbor.name, goal\_node.name])

                open.append(new\_tuple)

            open = clean\_open\_from\_duplicates(open) # Se buscan los nodos duplicados dentro de Abiertos para dejar solo uno, el de menor costo

        closed.append(current)

        print\_step(current, open, closed)

    current = closed[len(closed) - 1]

    while current[0] is not start\_node:

        path.append(current[0].name)

        for tuple in closed:

            if tuple[0] == current[1]:

                current = tuple

    path.append(start\_node.name)

    path = path[::-1]

    print()

    print(f"El mejor camino para llegar del nodo {start\_node.name} al {goal\_node.name} es:")

    for i in range(len(path)):

        print(f"{path[i]}", end="" if i == len(path) - 1 else " -> ")

    print()

def get\_lowest\_tuple\_index(tuple\_array):

    lowest = float('inf')

    index\_lowest = None

    for i in range(len(tuple\_array)):

        # Se busca el tercer valor de las tuplas, el cual represeta el costo, para buscar el mínimo

        if tuple\_array[i][2] < lowest:

            lowest = tuple\_array[i][2]

            index\_lowest = i

    return index\_lowest

def node\_is\_in\_closed(node\_to\_check, closed):

    for tuple in closed:

        if node\_to\_check.name == tuple[0].name:

            return True

    return False

En cada iteración haremos el llamado a la función clean\_open\_from\_duplicates la cual, si el último nodo agregado es repetido, lo eliminará de la lista. Aquí no es necesario comparar la procedencia del nodo, ya que el valor siempre es el mismo, así que, si se repite, solamente lo saltamos.

def clean\_open\_from\_duplicates(open):

    nodes\_accounted\_for = []

    clean\_open = [] # Arreglo de tuplas donde se guardan los valores minimos de cada nodo

    for tuple in open:

        if tuple[0] not in nodes\_accounted\_for: # Si el nodo no ha sido tomado en cuenta se mete a la lista

            nodes\_accounted\_for.append(tuple[0])

            clean\_open.append(tuple)

        else: # Si el nodo ya está metido en la lista, solo lo ignoramos ya que el que se encuentra tiene el mismo peso.

            pass

    return clean\_open

def print\_step(current, open, closed):

    # Impimir actual

    print()

    print(f"Actual: {current[0].name}\_{current[1].name}: {current[2]}")

    # Imprimir lista Abiertos

    print(f"Abiertos: [", end="")

    for tuple in open:

        print(f"{tuple[0].name}\_{tuple[1].name}: {tuple[2]}", end=", ")

    print(f"]")

    # Imprimir lista Cerrados

    print(f"Cerrados: [", end="")

    for tuple in closed:

        print(f"{tuple[0].name}\_{tuple[1].name}: {tuple[2]}", end=", ")

    print(f"]")

Dado que los nodos, la creación de nodos y del grafo es la misma, nos la saltamos, sin embargo, si forma parte del código. Ya por último pedimos nodo de inicio y el nodo meta, y hacemos la búsqueda avara.

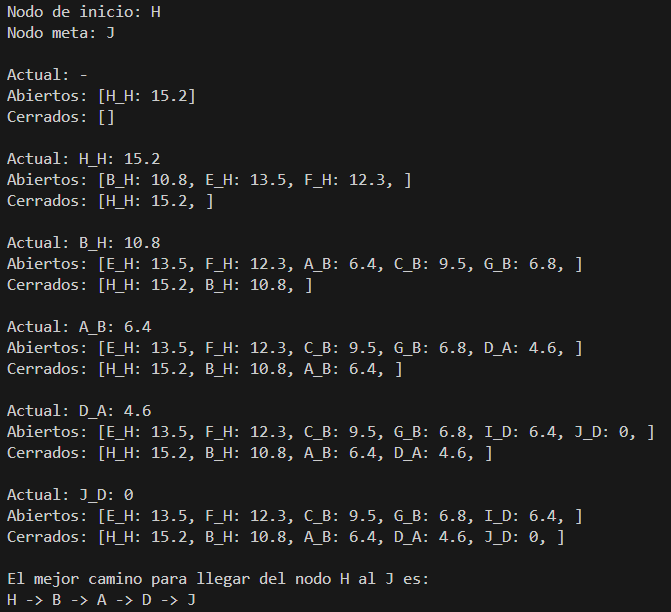
start\_node = input("Nodo de inicio: ")

goal\_node = input("Nodo meta: ")

greedy(get\_node(node\_list, start\_node.upper()), get\_node(node\_list, goal\_node.upper()))

**Ejecución**

La ejecución imprime cada paso de la búsqueda, imprimiendo imprimiendo de cada nodo su padre y el su valor heurístico. De cada iteración imprime el actual, los abiertos y cerrados. Al final hace la retropropagación para obtener la respuesta del camino usado.



3. Búsqueda A\*

**Código**

En la implementación greedy o avara, casi todo el código se mantiene igual, así que solo explicaremos las cosas que cambian.

A la hora el valor hasta un nodo, se realizará con la distancia en línea recta del nodo en cuestión al nodo destino. Esto aplica al iniciar la búsqueda,

new\_tuple = (

        start\_node,

        start\_node,

        straight\_line\_distance[start\_node.name, start\_node.name],

        straight\_line\_distance[start\_node.name, goal\_node.name],

        straight\_line\_distance[start\_node.name, start\_node.name] + straight\_line\_distance[start\_node.name, goal\_node.name]

)

open.append(new\_tuple)

y en el paso donde agregamos los vecinos a abiertos

new\_tuple = (

        neighbor,

        current[0],

        straight\_line\_distance[neighbor.name, current[0].name]+current[2],

        straight\_line\_distance[neighbor.name, goal\_node.name],

        straight\_line\_distance[neighbor.name, current[0].name] + current[2] + straight\_line\_distance[neighbor.name, goal\_node.name]

)

open.append(new\_tuple)

import time

# =========================================================

# Implementaciones relacionadas al algoritmo A estrella

# =========================================================

def a\_star(start\_node, goal\_node):

    current = (None, None, None, None, None) # Tupla encargada de llevar registro del nodo actual (1), nodo desde el que se llegó (2), el costo (3), el valor heurístico (4) y la suma de estos 2(5)

    open = [] # Lista de tuplas encargada de llevar registro de la lista Abiertos. Nodo (1), nodo desde el que se llegó (2), el costo (3), el valor heurístico (4) y la suma de estos 2(5)

    closed = [] # Lista de tuplas encargada de llevar registro de la lista Cerrados. Nodo (1), nodo desde el que se llegó (2), el costo (3), el valor heurístico (4) y la suma de estos 2(5)

    path = []

    print()

    print("Actual: -")

    print(f"Abiertos: [{start\_node.name}\_{start\_node.name}: {straight\_line\_distance[start\_node.name, start\_node.name]}: {straight\_line\_distance[start\_node.name, goal\_node.name]}: {straight\_line\_distance[start\_node.name, start\_node.name]+straight\_line\_distance[start\_node.name, goal\_node.name]}]")

    print("Cerrados: []")

    new\_tuple = (

        start\_node,

        start\_node,

        straight\_line\_distance[start\_node.name, start\_node.name],

        straight\_line\_distance[start\_node.name, goal\_node.name],

        straight\_line\_distance[start\_node.name, start\_node.name]+straight\_line\_distance[start\_node.name, goal\_node.name]

    )

    open.append(new\_tuple)

    while current[0] is not goal\_node:

        open\_lowest = open.pop(get\_lowest\_tuple\_index(open))

        current = (open\_lowest[0], open\_lowest[1], open\_lowest[2], open\_lowest[3], open\_lowest[4])

        for neighbor in current[0].neighbors:

            # Si el nodo no ha sido cerrado, se crea otra tupla y se añade a Abiertos

            if not node\_is\_in\_closed(neighbor, closed):

                new\_tuple = (

                    neighbor,

                    current[0],

                    straight\_line\_distance[neighbor.name, current[0].name]+current[2],

                    straight\_line\_distance[neighbor.name, goal\_node.name],

                    straight\_line\_distance[neighbor.name, current[0].name]+current[2]+straight\_line\_distance[neighbor.name, goal\_node.name]

                )

                open.append(new\_tuple)

            open = clean\_open\_from\_duplicates(open) # Se buscan los nodos duplicados dentro de Abiertos para dejar solo uno, el de menor costo

        closed.append(current)

        print\_step(current, open, closed)

    current = closed[len(closed) - 1]

    while current[0] is not start\_node:

        path.append(current[0].name)

        for tuple in closed:

            if tuple[0] == current[1]:

                current = tuple

    path.append(start\_node.name)

    path = path[::-1]

    print()

    print(f"El mejor camino para llegar del nodo {start\_node.name} al {goal\_node.name} es:")

    for i in range(len(path)):

        print(f"{path[i]}", end="" if i == len(path) - 1 else " -> ")

    print()

def get\_lowest\_tuple\_index(tuple\_array):

    lowest = float('inf')

    index\_lowest = None

    for i in range(len(tuple\_array)):

        # Se busca el tercer valor de las tuplas, el cual represeta el costo, para buscar el mínimo

        if tuple\_array[i][2] < lowest:

            lowest = tuple\_array[i][2]

            index\_lowest = i

    return index\_lowest

def node\_is\_in\_closed(node\_to\_check, closed):

    for tuple in closed:

        if node\_to\_check.name == tuple[0].name:

            return True

    return False

En cada iteración haremos el llamado a la función clean\_open\_from\_duplicates la cual, si el último nodo agregado es repetido, lo eliminará de la lista. Cuando encontramos que un nodo está repetido, procederá a buscar la anterior incidencia, y la comparará para ver si el nodo incidente cuestión tiene un mejor valor que el nodo ya guardado, si lo tiene, lo remplazará, en caso contrario, saltará al nodo incidente.

def clean\_open\_from\_duplicates(open):

    nodes\_accounted\_for = []

    clean\_open = [] # Arreglo de tuplas donde se guardan los valores minimos de cada nodo

    for tuple in open:

        if tuple[0] not in nodes\_accounted\_for: # Si el nodo no ha sido tomado en cuenta, automáticamente se considera el menor y se mete a la lista

            nodes\_accounted\_for.append(tuple[0])

            clean\_open.append(tuple)

        else: # Si el nodo ya está metido en la lista, significa que debe de compararse con el otro para ver cuál tiene menor peso

            for i in range(len(clean\_open)):

                if clean\_open[i][0].name == tuple[0].name:

                    if tuple[4] < clean\_open[i][4]: # Si el costo del nodo es menor al del mismo nodo ya registrado, lo reemplaza

                        clean\_open[i] = tuple

                    break

    return clean\_open

def print\_step(current, open, closed):

    # Impimir actual

    print()

    print(f"Actual: {current[0].name}\_{current[1].name}: {current[2]}: {current[3]}: {current[4]}")

    # Imprimir lista Abiertos

    print(f"Abiertos: [", end="")

    for tuple in open:

        print(f"{tuple[0].name}\_{tuple[1].name}: {tuple[2]}: {tuple[3]}: {tuple[4]}", end=", ")

    print(f"]")

    # Imprimir lista Cerrados

    print(f"Cerrados: [", end="")

    for tuple in closed:

        print(f"{tuple[0].name}\_{tuple[1].name}: {tuple[2]}: {tuple[3]}: {tuple[4]}", end=", ")

    print(f"]")

Dado que los nodos, la creación de nodos y del grafo es la misma, nos la saltamos, sin embargo, si forma parte del código. Ya por último pedimos nodo de inicio y el nodo meta, y hacemos la búsqueda A estrella.

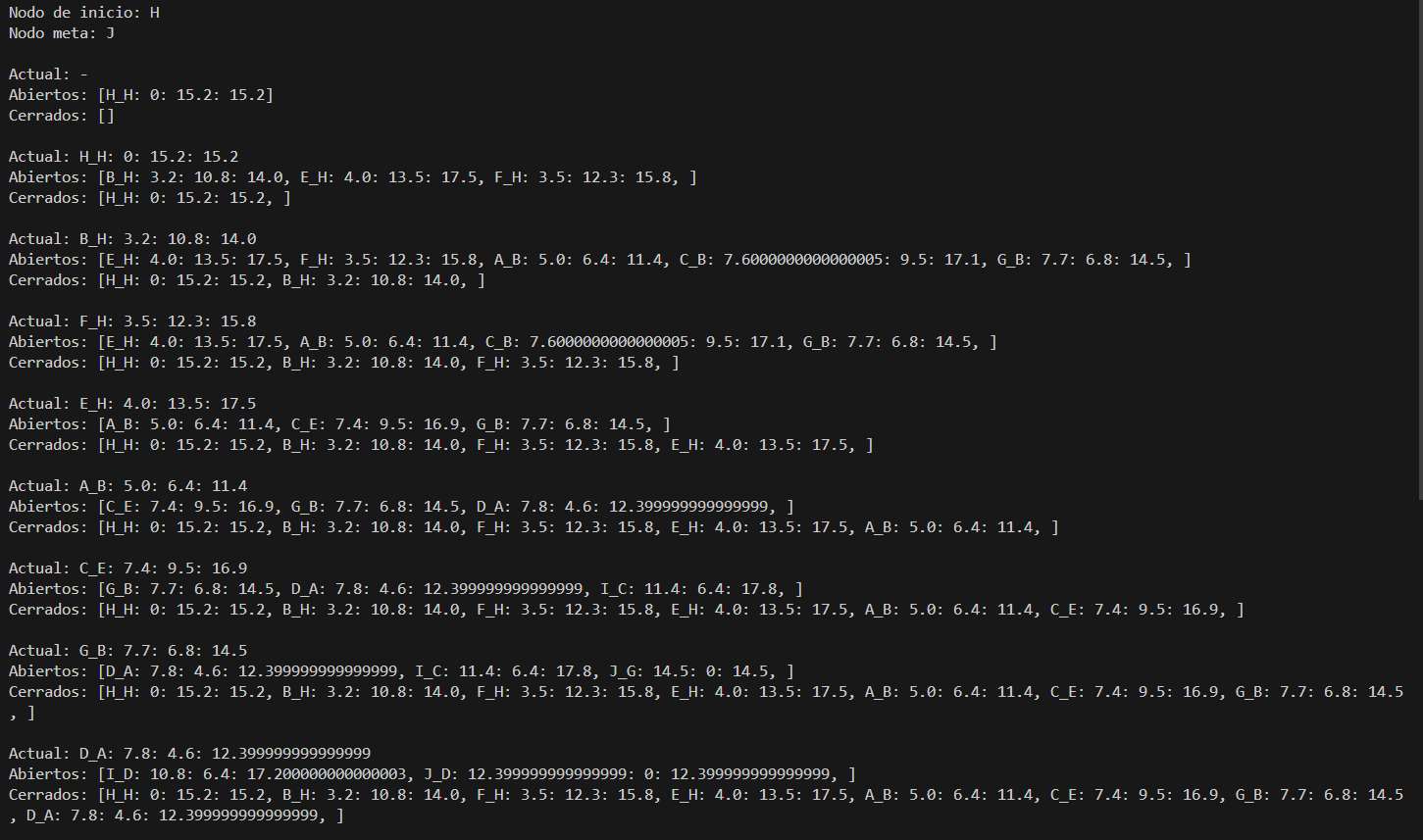
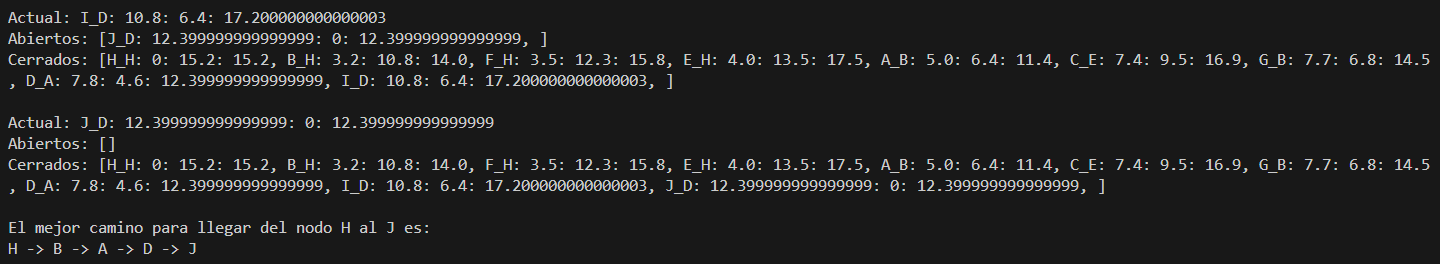
start\_node = input("Nodo de inicio: ")

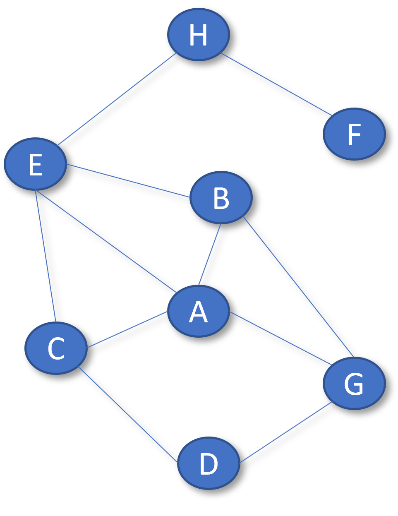
goal\_node = input("Nodo meta: ")

a\_star(get\_node(node\_list, start\_node.upper()), get\_node(node\_list, goal\_node.upper()))

**Ejecución**

La ejecución imprime cada paso de la búsqueda, imprimiendo imprimiendo de cada nodo su padre y el su valor heurístico. De cada iteración imprime el actual, los abiertos y cerrados. Al final hace la retropropagación para obtener la respuesta del camino usado.

(b) Sea un problema como el siguiente:

Debemos resolver este problema utilizando para ello:

1. Primero en amplitud

2. Primero en profundidad

El programa que resuelve el problema deberá pedir el estado de inicio y al final deberá desplegar el resultado de la búsqueda, que depende según la búsqueda implementada

**Código**

Para resolverlo optamos por realizarlo en C++. Este fue el primer problema que realizamos y pensamos que debía ser una solución general, así que tiene la interacción con el usuario para que pueda agregar sus grafos específicos. Entonces tenemos el inicio del archivo, donde definimos librerías, variables y una función que nos ayuda a imprimir los resultados.

#include <iostream>

#include <stdio.h>

#include <list>

#include <string>

#include <iomanip>

using namespace std;

const char separator = ' ';

const int currentWidth = 8;

const int listWidth = 30;

template <typename T>

void printElement(T t, const int &width)

{

    cout << left << setw(width) << setfill(separator) << t << "|";

}

Después creamos la clase (o estructura) Node, la cual contiene 2 atributos, el tag o nombre del nodo,y una lista de apuntadores a nodos, que es la lista que guardará las ubicaciones de memoria de los nodos vecinos. Además define métodos que nos ayudarán a agregar nodos a una lista de nodos, encontrar un nodo en una lista de nodos, conectar 2 nodos de una misma lista entre sí, imprimir los nodos con sus vecinos, y convertir la lista de nodos a un string.

struct Node

{

    string tag = "";

    list<Node \*> neighbors;

    static void addNode(list<Node \*> &nodes)

    {

        Node \*newNode = new Node;

        cout << "Enter tag for the new node: ";

        cin >> newNode->tag;

        nodes.push\_back(newNode);

        cout << "Node added successfully." << "\n";

    }

    static Node \*findByTag(list<Node \*> &nodes, string tag)

    {

        for (list<Node \*>::iterator node\_it = nodes.begin(); node\_it != nodes.end(); node\_it++)

        {

            if ((\*node\_it)->tag == tag)

                return (\*node\_it);

        }

        return NULL;

    }

    static void connectNodes(list<Node \*> &nodes)

    {

        string tag1, tag2;

        cout << "Enter tag of first node: ";

        cin >> tag1;

        cout << "Enter tag of second node: ";

        cin >> tag2;

        Node \*node1 = findByTag(nodes, tag1);

        Node \*node2 = findByTag(nodes, tag2);

        if (node1 && node2)

        {

            node1->neighbors.push\_back(node2);

            node2->neighbors.push\_back(node1);

            cout << "Nodes connected successfully." << "\n";

        }

        else if (node1 == NULL)

        {

            cout << "Node " << tag1 << " does not exists." << "\n";

        }

        else if (node2 == NULL)

        {

            cout << "Node " << tag2 << " does not exists." << "\n";

        }

    }

    static void displayNodesWithNeighbors(list<Node \*> &nodes)

    {

        cout << "Nodes:" << "\n";

        for (list<Node \*>::iterator node\_it = nodes.begin(); node\_it != nodes.end(); node\_it++)

        {

            cout << "Node tag: " << (\*node\_it)->tag << "\n";

            cout << "Neighbors: ";

            for (list<Node \*>::iterator neighbor\_it = (\*node\_it)->neighbors.begin(); neighbor\_it != (\*node\_it)->neighbors.end(); neighbor\_it++)

            {

                cout << (\*neighbor\_it)->tag << " ";

            }

            cout << "\n";

        }

    }

    static string nodesToString(list<Node \*> &nodes)

    {

        string nodes\_str = "[ ";

        for (list<Node \*>::iterator node\_it = nodes.begin(); node\_it != nodes.end(); node\_it++)

        {

            nodes\_str += (\*node\_it)->tag + ", ";

        }

        nodes\_str += "]";

        return nodes\_str;

    }

};

Despues, definimos otra clase GeneralSearchTree que nos servirá para realizar las 2 soluciones para el árbol general de búsqueda, es decir, búsqueda en amplitud, y búsqueda en profundidad. Entre los atributos privados de la clase tendrá 2 variables booleanas para determinar cuál búsqueda se realizará, además de tener los métodos que servirán para realizar la búsqueda. El método next nos regresa el siguiente nodo a visitar. Dado que el comportamiento depende del algoritmo, si es búsqueda en amplitud, devolveremos el primer nodo en la lista, si es búsqueda en profundidad devolveremos el último nodo en la lista, la lista que le pasaremos a next será la lista de abiertos. El método add añade un nodo al final de una lista. Y expand agregará los nodos vecinos del nodo actual que no estén ya en la lista de abiertos o en la lista de cerrados..

struct GeneralSearchTree

{

private:

    bool firstInAmplitude = false;

    bool firstInDepth = false;

    Node \*next(list<Node \*> &nodes\_list)

    {

        Node \*res = NULL;

        if (firstInAmplitude)

        {

            res = nodes\_list.front();

            nodes\_list.pop\_front();

        }

        else if(firstInDepth)

        {

            res = nodes\_list.back();

            nodes\_list.pop\_back();

        }

        return res;

    }

    void add(list<Node \*> &nodes\_list, Node \*node)

    {

        nodes\_list.push\_back(node);

    }

    void expand(Node \*current)

    {

        for (list<Node \*>::iterator neighbor\_it = current->neighbors.begin(); neighbor\_it != current->neighbors.end(); neighbor\_it++)

        {

            if (Node::findByTag(opened\_nodes, (\*neighbor\_it)->tag) == NULL &&

                Node::findByTag(closed\_nodes, (\*neighbor\_it)->tag) == NULL)

            {

                add(opened\_nodes, (\*neighbor\_it));

            }

        }

    }

Continuando con los métodos y atributos públicos, tenemos ambas listas, la de abiertos y cerrados. El constructor recibirá las siglas del algoritmo a usar, lo que cambiará las variables mencionadas anteriormente.

Como el punto primordial, está el método search se encargará de realizar la búsqueda desde el nodo de inicio hasta el nodo de destino. También se encarga de imprimir en consola cada una de las iteraciones de la búsqueda.

Por último el método initSearch únicamente es para la interacción con el usuario, obtendrá los nodos indicados por el usuario, y realizará la búsqueda del árbol general de búsqueda con el algoritmo indicado.

public:

    GeneralSearchTree(string algorithm)

    {

        if (algorithm == "BFS")

        {

            firstInAmplitude = true;

        }

        else if (algorithm == "DFS")

        {

            firstInDepth = true;

        }

    }

    list<Node \*> opened\_nodes;

    list<Node \*> closed\_nodes;

    list<Node \*> search(Node \*start, Node \*destination)

    {

        add(opened\_nodes, start);

        Node \*current = NULL;

        cout << "|";

        printElement("=========", currentWidth);

        printElement("==============================", listWidth);

        printElement("==============================", listWidth);

        cout << "\n";

        cout << "|";

        printElement(" Current ", currentWidth);

        printElement(" Opened ", listWidth);

        printElement(" Closed ", listWidth);

        cout << "\n";

        cout << "|";

        printElement("=========", currentWidth);

        printElement("==============================", listWidth);

        printElement("==============================", listWidth);

        cout << "\n";

        cout << "| ";

        printElement("-", currentWidth);

        printElement(Node::nodesToString(opened\_nodes), listWidth);

        printElement(Node::nodesToString(closed\_nodes), listWidth);

        cout << "\n";

        while (!opened\_nodes.empty())

        {

            current = next(opened\_nodes);

            if (current == destination)

            {

                add(closed\_nodes, current);

                cout << "| ";

                printElement(current->tag, currentWidth);

                printElement(Node::nodesToString(opened\_nodes), listWidth);

                printElement(Node::nodesToString(closed\_nodes), listWidth);

                cout << "\n";

                cout << "====FOUND====\n";

                return closed\_nodes;

            }

            add(closed\_nodes, current);

            expand(current);

            cout << "| ";

            printElement(current->tag, currentWidth);

            printElement(Node::nodesToString(opened\_nodes), listWidth);

            printElement(Node::nodesToString(closed\_nodes), listWidth);

            cout << "\n";

        }

        cout << "| ";

        printElement(current->tag, currentWidth);

        printElement(Node::nodesToString(opened\_nodes), listWidth);

        printElement(Node::nodesToString(closed\_nodes), listWidth);

        cout << ""====NOT FOUND====\n";

        return closed\_nodes;

    }

    static void initSearch(list<Node \*> &graph, string algorithm)

    {

        string startTag, destTag;

        cout << "Enter tag of start node: ";

        cin >> startTag;

        cout << "Enter tag of destination node: ";

        cin >> destTag;

        Node \*start = Node::findByTag(graph, startTag);

        Node \*dest = Node::findByTag(graph, destTag);

        GeneralSearchTree search = GeneralSearchTree(algorithm);

        if (start && dest)

        {

            search.search(start, dest);

        }

        else if (start == NULL)

        {

            cout << "Node " << startTag << " does not exists." << "\n";

        }

        else if (dest == NULL)

        {

            cout << "Node " << destTag << " does not exists." << "\n";

        }

    }

};

En eso consiste la solución, la función main se encargará de la interacción con el usuario, dando de 6 operaciones a realizar, agregar nodo al grafo, conectar 2 nodos del grafo, imprimir los nodos con sus vecinos, realizar la búsqueda en profundidad, realizar la búsqueda en amplitud, y salir del programa.

Una vez que el usuario sale del programa, se limpia la memoria del grafo.

int main()

{

    list<Node \*> graph;

    char choice;

    do

    {

        cout << "\nMenu:" << "\n";

        cout << "1. Add Node" << "\n";

        cout << "2. Connect Nodes" << "\n";

        cout << "3. Display Nodes with neighbors" << "\n";

        cout << "4. Depth First Search" << endl;

        cout << "5. Breadth First Search" << endl;

        cout << "6. Exit" << "\n";

        cout << "Enter your choice: ";

        cin >> choice;

        switch (choice)

        {

        case '1':

            Node::addNode(graph);

            break;

        case '2':

            Node::connectNodes(graph);

            break;

        case '3':

            Node::displayNodesWithNeighbors(graph);

            break;

        case '4':

        {

            GeneralSearchTree::initSearch(graph, "DFS");

            break;

        }

        case '5':

        {

            GeneralSearchTree::initSearch(graph, "BFS");

            break;

        }

        default:

            cout << "Invalid choice. Please try again." << "\n";

            break;

        }

    } while (choice != '6');

*// Clean up allocated memory for nodes*

    for (list<Node \*>::iterator node\_it = graph.begin(); node\_it != graph.end(); node\_it++)

    {

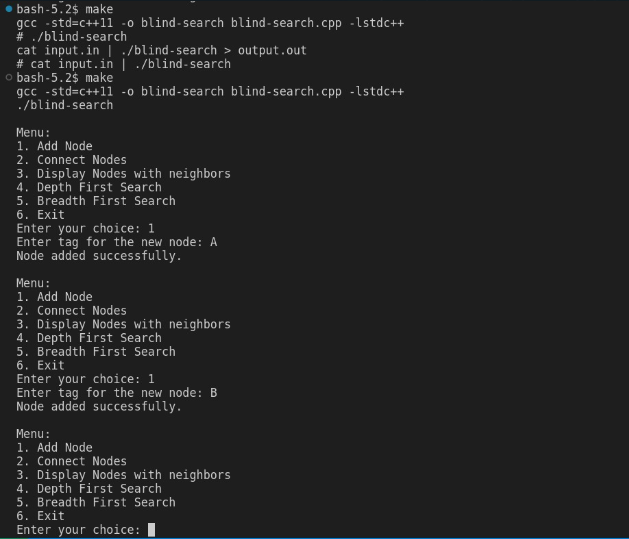
        delete (\*node\_it);

    }

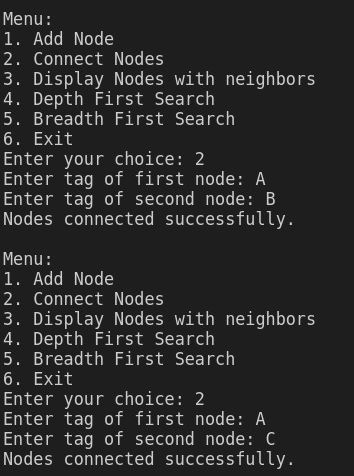
    return 0;

}

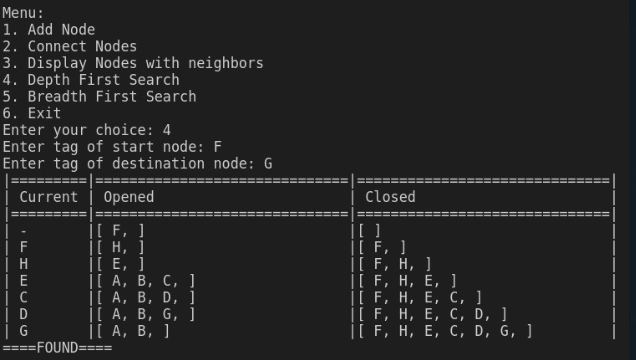
**Ejecución**

Se agregan los nodos del problema

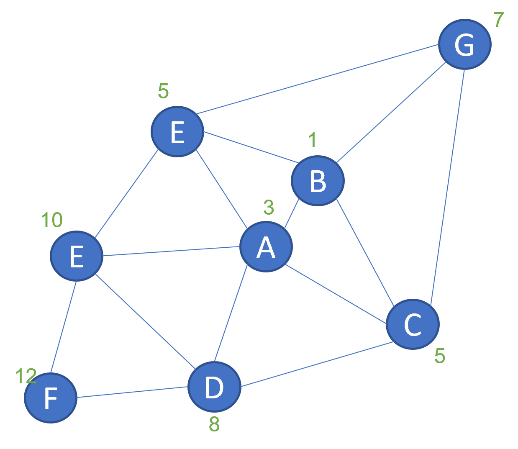
Después se agregan las conexiones  
A screenshot of a computer

Description automatically generated

Y resolvemos el problema utilizando:

1. Primero en amplitud desde el nodo F hasta el nodo G  
   
2. Primero en profundidad desde el nodo F hasta el nodo G  
   

(c) Se tiene el siguiente espacio estado:

Utilizando las siguientes técnicas de búsqueda local:

1. Templado simulado

Encontremos el mejor estado que se puede localizar iniciando desde cualquier estado. El programa deberá preguntarnos por el estado de inicio y responder con el mejor estado posible

**Código**

En esta implementación se realizó la búsqueda del mejor estado en un grafo mediante el algoritmo de templado simulado. El grafo está definido por el diccionario ‘problema’, que indica los vecinos de cada estado, y los valores asociados a cada estado están en el diccionario ´valores´. El algoritmo comienza en un estado inicial, luego recorre los vecinos del estado actual, eligiendo el mejor vecino basado en una función de probabilidad que depende de una temperatura decreciente. La tasa de enfriamiento de la temperatura está controlada por el parámetro ‘alpha’, que multiplica la temperatura actual en cada iteración, reduciéndola gradualmente. Este proceso continua hasta que la temperatura alcanza un valor mínimo, devolviendo el mejor estado encontrado y la temperatura en la que se encontró.

import random

import math

problema=dict([("A",["B","C","D","E","H"]), ("B",["A","C","G","H"]),

               ("C",["A","B","D","G"]), ("D",["A","C","E","F"]),

               ("E",["A","D","F","H"]), ("F",["D","E"]),

               ("G",["B","C","H"]), ("H",["A","B","E","G"]) ])

valores=dict([("A",3), ("B",1), ("C",5), ("D",8), ("E",10), ("F",12), ("G",7), ("H",5)])

*# Hay 2 E que se repiten en el documento asi que cambie la E que tiene valor de 5 por una H*

def f(estado):

    return valores[estado]

def vecindad(estado):

    return problema[estado]

def simulated\_tempering(tini, tfin, alpha, ini, f, vecindad):

    t = tini

    actual = ini

    mejor = None

    fmejor = -10

    t\_mejor = t

    while t >= tfin:

        vecinos = vecindad(actual)

        nuevo = max(vecinos, key=f)

        if f(nuevo) > f(actual):

            actual = nuevo

        else:

            r = random.random()

            p = math.exp((f(nuevo) - f(actual)) / t)

            if r < p:

                actual = nuevo

        if f(actual) > fmejor:

            mejor = actual

            fmejor = f(actual)

            t\_mejor = t

        print(f"Temperatura: {t:.2f}, Estado Actual: {actual}, Valor: {f(actual)}")

        t \*= alpha

    return mejor, t\_mejor

if \_\_name\_\_ == "\_\_main\_\_":

    tini = 100

    tfin = 1

    alpha = random.random()

    ini = None

    ini = input("Ingresa el estado de inicio: ")

    mejor\_estado, temp\_mejor\_estado = simulated\_tempering(tini, tfin, alpha, ini, f, vecindad)

    print(f"El mejor estado encontrado es: {mejor\_estado} con un valor de: {f(mejor\_estado)}")

    print(f"El mejor estado se encontró a una temperatura de: {temp\_mejor\_estado:.2f}")

**Ejecución**

