Universidad Autónoma de Chihuahua

Facultad de Ingeniería

SISTEMAS DE BÚSQUEDA Y RAZONAMIENTO

Docente: Majalca Martínez Ricardo

Grupo: 6CC2

Proyecto Final

Gerardo Esteban Jurado Carrera 273880

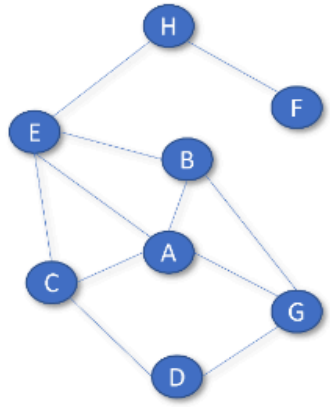
Erick Fernando Nevarez Avila 357664

José Ángel Ortiz Meraz 353195

Ingeniería en Ciencias de la Computación

27/05/2024

(a).- 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

Lenguaje Utilizado: C + +. Por Erick Fernando Nevarez Avila.

```
//Código realizado el dia 27 de mayo del 2024
//Hecho por Erick Fernando Nevarez Avila 357664

#include <iostream>
#include <map>
#include <set>
#include <queue>
#include <stack>
#include <vector>

using namespace std;

// Estructura de un Nodo
struct Nodo {
    string nombre;

    set<string> vecinos;

};

//Procedimiento de insercion de nodos al nuestro mapa (El grafo)
//Es necesario mandarle como referencia los 2 nodos que necesitamos
     relacionar de manera bilateral
//Ademas necesitamos enviarle el mapa al cual van a ser insertados,
     los grafos no son dirigidos, en
// caso de que sea dirigido, podriamos hacer otro procedimiento para
     solo insertar una conexion del nodo A al B
void agregarArista(map<string, Nodo>& grafo, const string& nodo1,
     const string& nodo2) {
    grafo[nodo1].vecinos.insert(nodo2);
    grafo[nodo2].vecinos.insert(nodo1);
}
```

```cpp
//Nuestra busqueda en amplitud, requiere el grafo con los nodos
     contenidos dentro del mapa, necesita un nodo
//  inicio y un nodo objetivo
vector<string>  busquedaAmplitud(map<string,  Nodo>&  grafo,  const
     string& inicio, const string& objetivo) {
      //En cada busqueda necesitamos un contenedor de los nodos
     visitados, para no quedarnos en bucles sin fin
    set<string> visitados;
     //La siguiente cola nos permitira mantener un orden del camino
     de la busqueda
    //Nuestra cola es de tipo vector de strings
     // le insertamos nuestro primer nodo a la cola y le a adimos
     nuestro nodo inicio

    queue<vector<string>> cola;
    cola.push({inicio});
    //La busqueda va a continuar si es que todavia existen nodos por
     visitar (nuestra cola no este vacia)
    while (!cola.empty()) {
          //Extraemos la referencia del primer elemento de la cola
     para continuar desde el nodo siguiente a manipular
         //  despues vamos a sacar el nodo a manipular de la cola,
     para que no lo vayamos a manipular despues otra vez
       vector<string> camino = cola.front();
       cola.pop();

        //Del camino que extragimos de la cola, sacamos el nodo que
     esta al final del camino (es el frente de la cola,
       //  solo que como es un vector este se postra al final)
       string nodo = camino.back();

        //Realizamos una verificacion si es que el nodo que estamos
     buscando no es el que estamos manipulando
         //  si es asi retornamos el camino que tenemos hasta el
     momento
       if (nodo == objetivo) {
           return camino;
       }
          //En caso de que no sea el nodo que buscamos, vamos a
     verificar que el nodo no este dentro de nuestro
        //  arreglo de visitados, si es que llegamos al final del
     vector de visitados (visitados.end()), vamos
        // a asumir que no ha sido visitado nuestro nodo y vamos a
     continuar
       if (visitados.find(nodo) == visitados.end()) {
           //Vamos a insertar el nodo en el vector de visitados
           visitados.insert(nodo);
```

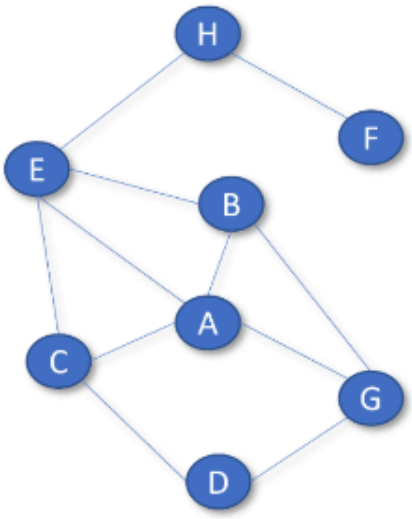
```
            //Luego iteramos entre todos los nodos "hijos" o vecinos como los llamamos nosotros y vamos a verificar
            //  que no esten en el vector de visitados y si es asi y no estan, vamos a insertar en un nuevo camino el camino que
            //  ya teniamos y luego vamos a meter hasta el final este nodo, para cuando ya hayamos verificado todos los hijos
            //  de los nodos de la anterior generacion, vamos a tener ya en nuestro camino, los siguientes nodos a buscar
            for (const string& vecino : grafo[nodo].vecinos) {
                if (visitados.find(vecino) == visitados.end()) {
                    vector<string> nuevoCamino = camino;
                    nuevoCamino.push_back(vecino);
                       //Por eso a?adimos a la cola, este camino que luego vamos a consultar
                    cola.push(nuevoCamino);
                }
            }
        }
    }
    return {}; //En caso de no encontrar nuestro nodo objetivo vamos a retornar un vector vacio representado por {}
}



//En el caso de la busqueda por profundidad vamos a buscar nodo por nodo sin importar los hijos, sino hasta llegar
//  al final de esa rama donde no existan hijos no visitados
vector<string> busquedaProfundidad(map<string, Nodo>& grafo, const string& inicio, const string& objetivo) {

    //El proceso es igual a como lo manejamos en la busqueda por amplitud, solamente que los caminos que vamos a revisar
    //  en la siguiente iteracion no seran lo que se hayan quedado en la cola (los primeros manipulados), sino el nodo
    //  hijo que hayamos manipulado
    set<string> visitados;
    // Vamos a hacer una piia la cual nos ayuda a manipular el ultimo valor introducido, ya que no nos importan los caminos
    //  que estaban antes sino el camino del nodo hijo, la pila nos ayuda a esto
    stack<vector<string>> pila;

    //A?adimos a la pila nuestro nodo inicial
    pila.push({inicio});

    while (!pila.empty()) {
```

```
   //En el vector camino metemos la referencia del elemento mas
arriba de la pila, ya que buscamos manipular los
   //  siguientes caminos, no los anteriores
   vector<string> camino = pila.top();
   pila.pop();
   //Nuestro vector de camino guarda al final el nodo siguiente
a manipular (es una de las propiedades de los vectores)
     //  por lo tanto guardamos en la variable string "nodo"
nuestro ultimo elemento del camino
   string nodo = camino.back();

   //verificamos si el nodo actual es igual al del objetivo, si
es asi retornamos el camino
   if (nodo == objetivo) {
       return camino;
   }

       //En caso de que no sea el nodo que buscamos, vamos a
verificar que el nodo no este dentro de nuestro
    //  arreglo de visitados, si es que llegamos al final del
vector de visitados (visitados.end()), vamos
    // a asumir que no ha sido visitado nuestro nodo y vamos a
continuar
   if (visitados.find(nodo) == visitados.end()) {

       //Insertamos el nodo manipulado dentro de nuestro vector
de visitados
       visitados.insert(nodo);

       //Luego iteratemos en cada uno de los vecinos (hijos) de
nuestro nodo
       for (const string& vecino : grafo[nodo].vecinos) {
            //Realizamos la misma verificacion de que este nodo
hijo no este en el vector de verificados
           if (visitados.find(vecino) == visitados.end()) {

                //En caso de que no este entre los visitados,
vamos a guardar en nuestra variable nuevoCamino
                  //  nuestro camino actual y vamos a a adir
nuestro vecino al final del camino (recordemos la
                  //  propiedad de los vectores) y por ultimo
a adiremos arriba de nuestra pila este camino
               vector<string> nuevoCamino = camino;
               nuevoCamino.push_back(vecino);
               pila.push(nuevoCamino);
                  //El camino siguiente a manipular va a ser el
que esta mas arriba en la pila, como nosotros
```

```
                    //  enviamos nuestro nuevo camino arriba en la
    pila, este camino sera el siguiente en ser consultado
                }
            }
             //El camino que estamos manipulando no es como en el de
    amplitud, en donde el camino de los hijos se iban al
             // final de la fila, sino que en este caso los caminos
    de los hijos se van al inicio, haciendo que sean los
             //  siguientes en manipularse y podamos verificar cada
    uno de los caminos posibles en profundidad

        }
    }
    return {}; // No se encontr\ufffd un camino
}

int main() {
    map<string, Nodo> grafo;

    // Agregamos las aristas de nuestro grafo
    agregarArista(grafo, "F", "H");
    agregarArista(grafo, "H", "E");
    agregarArista(grafo, "E", "B");
    agregarArista(grafo, "E", "A");
    agregarArista(grafo, "E", "C");
    agregarArista(grafo, "A", "B");
    agregarArista(grafo, "A", "C");
    agregarArista(grafo, "A", "G");
    agregarArista(grafo, "D", "G");
    agregarArista(grafo, "D", "C");

    string inicio, objetivo;

    //Pedimos al usuario el nodo de inicio y el nodo final
    cout << "Ingrese el nodo de inicio: ";
    cin >> inicio;
    cout << "Ingrese el nodo que desea buscar: ";
    cin >> objetivo;

     //Realizamos las respectivas busquedas en amplitud y luego en
     profundidad
     vector<string> caminoAmplitud = busquedaAmplitud(grafo, inicio,
     objetivo);
    if (!caminoAmplitud.empty()) {
        cout << "Busqueda por Amplitud"<<endl;
        cout << "Camino de " << inicio << " a " << objetivo << ": ";
        for (const string& nodo : caminoAmplitud) {
            cout << nodo << " ";
```

```
        }
        cout << endl;
    } else {
            cout << "No hay un camino de " << inicio << " a " <<
     objetivo << " (BFS)." << endl;
    }


     vector<string> caminoProfundidad = busquedaProfundidad(grafo,
     inicio, objetivo);
    if (!caminoProfundidad.empty()) {
        cout << "Busqueda por Profundidad"<<endl;
        cout << "Camino de " << inicio << " a " << objetivo << ": ";
        for (const string& nodo : caminoProfundidad) {
            cout << nodo << " ";
        }
        cout << endl;
    } else {
            cout << "No hay un camino de " << inicio << " a " <<
     objetivo << " (DFS)." << endl;
    }
    return 0;
}
```

(b).- Tenemos un problema con el siguiente espacio estado:



Edo/Edo	A	B	C	D	E	F	G	H
A	0	1.8	2.5	2.8	4.5	5.0	3.0	5.5
B	1.8	0	4.4	5.6	3.5	2.3	4.5	3.2
C	2.5	4.4	0	3.8	3.4	8.0	6.3	7.7
D	2.8	5.6	3.8	0	7.7	8.0	2.8	10.0
E	4.5	3.5	3.4	7.7	0	6.7	6.8	4.0
F	5.0	2.3	8.0	8.0	6.7	0	5.3	3.5
G	3.0	4.5	6.3	2.8	6.8	5.3	0	6.6
H	5.5	3.2	7.7	10.0	4.0	3.5	6.6	0

Tabla con distancias en línea recta

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 edo_k desde un estado edo_{k-1} es:

$$g(edo_k) = dlr[edo_{k-1}, edo_k] + g(edo_{k-1})$$

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

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.

Implementación del Algoritmo de Búsqueda Primero por lo Mejor

Descripción del Problema

El problema consiste en encontrar el camino óptimo desde un estado inicial a un estado meta en un grafo no dirigido. El grafo contiene varios nodos (representados por letras de 'A' a 'H') y aristas con distancias en línea recta (dlr) entre ellos. La búsqueda debe realizarse utilizando el algoritmo de búsqueda primero por lo mejor, que prioriza la exploración de nodos basándose en una función heurística. En este caso, la heurística es la distancia en línea recta (dlr) desde el nodo actual al nodo meta.

Descripción de la Implementación

La implementación se realizó en el lenguaje de programación C++. A continuación, se describen las partes clave del código:

Estructura de Datos

- **Estructura Nodo**: Representa cada nodo del grafo y almacena el estado (nombre del nodo), el costo g (costo acumulado desde el nodo inicial), el costo h (heurística, distancia en línea recta al nodo meta) y un puntero al nodo padre para reconstruir el camino.
- **Mapas y Vectores**: Se utilizan estructuras unordered_map para almacenar las distancias en línea recta entre los nodos (dlr) y las conexiones del grafo (grafo).

Funciones

- **heuristica**: Calcula la heurística para un nodo dado, en este caso, la distancia en línea recta desde el nodo actual al nodo meta.
- **reconstruir_camino**: Reconstruye el camino desde el nodo inicial al nodo meta utilizando los punteros a los nodos padres.
- **busqueda_primero_por_lo_mejor**: Implementa el algoritmo de búsqueda primero por lo mejor. Utiliza una cola de prioridad para gestionar los nodos a explorar, priorizando aquellos con menor costo total (costo_g + costo_h).

Algoritmo

1. **Inicialización**: Se inicializa la cola de prioridad con el nodo inicial y se establece su costo g a 0. La heurística se calcula usando la función heuristica.
2. **Exploración de Nodos**: Mientras haya nodos en la cola de prioridad, se extrae el nodo con menor costo total. Si el nodo es el meta, se reconstruye y muestra el camino.

3. **Actualización de Costos**: Para cada vecino del nodo actual, se calcula el costo g tentativo. Si es menor que el costo g conocido, se actualiza y el vecino se añade a la cola de prioridad.
4. **Terminación**: Si se encuentra el nodo meta, se muestra el camino. Si se exploran todos los nodos y no se encuentra el meta, se indica que no hay camino.

Captura de pantalla de implementación del código

```
c:\Users\gejc2\.vscode\extensions\ms-vscode.cpptools-1.20.5-win32-x64\
0ut0z.bnz' '--stdout=Microsoft-MIEngine-Out-tqidgdry.olz' '--stderr=Mi
z' '--dbgExe=C:\msys64\ucrt64\bin\gdb.exe' '--interpreter=mi'
Ingrese el estado inicial: A
Ingrese el estado meta: H
A E H
PS C:\Users\gejc2\OneDrive\Escritorio\My\UNI\UACH\CIENCIAS DE LA COMPU
```

Lenguaje Utilizado: C ++. Por Gerardo Esteban Jurado Carrera.

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <limits>
#include <algorithm>

using namespace std;


struct Nodo {
```

```
    char estado;
    float costo_g;
    float costo_h;
    Nodo* padre;


     Nodo(char estado, float costo_g, float costo_h, Nodo* padre =
     nullptr)
             : estado(estado), costo_g(costo_g), costo_h(costo_h),
     padre(padre) {}

    // Sobrecarga del operador '>' para ordenar los nodos en la cola
     de prioridad
    // La comparación se basa en la suma de costo_g y costo_h
    bool operator>(const Nodo& otro) const {
        return (costo_g + costo_h) > (otro.costo_g + otro.costo_h);
    }
};

// Mapa para almacenar las distancias en línea recta (dlr) entre
     nodos
// Este mapa es un unordered_map donde la clave es un carácter que
     representa el nodo
// El valor es otro unordered_map que contiene las distancias a los
     nodos vecinos
unordered_map<char, unordered_map<char, float>> dlr = {
     {'A', {{'B', 1.8}, {'C', 2.5}, {'D', 2.8}, {'E', 4.5}, {'F',
     5.0}, {'G', 3.0}, {'H', 5.5}}},
     {'B', {{'A', 1.8}, {'C', 4.4}, {'D', 5.6}, {'E', 3.5}, {'F',
     2.3}, {'G', 4.5}, {'H', 3.2}}},
     {'C', {{'A', 2.5}, {'B', 4.4}, {'D', 3.8}, {'E', 3.4}, {'F',
     8.0}, {'G', 6.3}, {'H', 7.7}}},
     {'D', {{'A', 2.8}, {'B', 5.6}, {'C', 3.8}, {'E', 7.7}, {'F',
     8.0}, {'G', 2.8}, {'H', 10.0}}},
     {'E', {{'A', 4.5}, {'B', 3.5}, {'C', 3.4}, {'D', 7.7}, {'F',
     6.7}, {'G', 6.8}, {'H', 4.0}}},
     {'F', {{'A', 5.0}, {'B', 2.3}, {'C', 8.0}, {'D', 8.0}, {'E',
     6.7}, {'G', 5.3}, {'H', 3.5}}},
     {'G', {{'A', 3.0}, {'B', 4.5}, {'C', 6.3}, {'D', 2.8}, {'E',
     6.8}, {'F', 5.3}, {'H', 6.6}}},
     {'H', {{'A', 5.5}, {'B', 3.2}, {'C', 7.7}, {'D', 10.0}, {'E',
     4.0}, {'F', 3.5}, {'G', 6.6}}}
};
```

```
// Mapa para almacenar las conexiones entre los nodos en el grafo
// Este mapa es un unordered_map donde la clave es un carácter que
     representa el nodo
// El valor es un vector de caracteres que representan los nodos
     vecinos
unordered_map<char, vector<char>> grafo = {
    {'A', {'B', 'C', 'D', 'E', 'G'}},
    {'E', {'H'}},
    {'H', {'F'}}
};

// Función heurística que devuelve la distancia en línea recta entre
     dos nodos
// Esta función se utiliza para estimar el costo restante desde el
     nodo actual hasta el nodo meta
float heuristica(char actual, char meta) {
    return dlr[actual][meta];
}

// Función para reconstruir e imprimir el camino desde el nodo
     inicial hasta el nodo meta
// Esta función sigue los punteros a los padres desde el nodo meta
     hasta el nodo inicial
void reconstruir_camino(Nodo* nodo) {
    vector<char> camino;  // Vector para almacenar el camino
    while (nodo) {
        camino.push_back(nodo->estado);
        nodo = nodo->padre;
    }
     reverse(camino.begin(), camino.end());   // Invierte el camino
     para obtener el orden correcto
    for (char estado : camino) {
        cout << estado << " ";
    }
    cout << endl;
}

// Implementación del algoritmo de búsqueda primero por lo mejor
// Esta función busca el camino óptimo desde el nodo inicial hasta
     el nodo meta
void busqueda_primero_por_lo_mejor(char inicio, char meta) {
```

```
    // Cola de prioridad para nodos a explorar, ordenada por el
    costo total (g + h)
priority_queue<Nodo, vector<Nodo>, greater<Nodo>> abiertos;
// Mapa para almacenar todos los nodos visitados
unordered_map<char, Nodo*> cerrados;

// Añadir el nodo inicial al conjunto abierto con costo g = 0 y
    heurística calculada
abiertos.emplace(inicio, 0.0, heuristica(inicio, meta));
// Almacenar el nodo inicial en el mapa de todos los nodos
    cerrados[inicio] = new Nodo(inicio, 0.0, heuristica(inicio,
    meta));

// Bucle principal de búsqueda
while (!abiertos.empty()) {
      // Obtener el nodo con el menor costo total (g + h) del
    conjunto abierto
    Nodo actual = abiertos.top();
    abiertos.pop();

    // Verificar si se ha alcanzado el nodo meta
    if (actual.estado == meta) {
          reconstruir_camino(&actual);  // Si se alcanza el nodo
    meta, reconstruir el camino
        return;
    }

    // Explorar los vecinos del nodo actual
    for (char vecino : grafo[actual.estado]) {
          // Calcular el costo g (costo acumulado desde el nodo
    inicial hasta el vecino)
                        float costo_g = actual.costo_g +
    dlr[actual.estado][vecino];
          // Calcular el costo h (heurística estimada desde el
    vecino hasta el nodo meta)
        float costo_h = heuristica(vecino, meta);
         // Crear un nuevo nodo vecino con los costos calculados
    y el nodo actual como padre
         Nodo* nodo_vecino = new Nodo(vecino, costo_g, costo_h,
    new Nodo(actual));
        // Añadir el vecino al conjunto abierto
        abiertos.push(*nodo_vecino);
```

```
                // Almacenar el nodo vecino en el mapa de todos los
     nodos
            cerrados[vecino] = nodo_vecino;
        }
    }

    cout << "No se encontró un camino." << endl;
}

int main() {
    char inicio, meta;
    cout << "Ingrese el estado inicial: ";
    cin >> inicio;
    cout << "Ingrese el estado meta: ";
    cin >> meta;
    busqueda_primero_por_lo_mejor(inicio, meta);
}
```

Implementación del Algoritmo de Búsqueda Avara

Descripción del Problema

El objetivo de este problema es encontrar el camino óptimo entre dos nodos en un grafo no dirigido utilizando el algoritmo de búsqueda avara. El grafo contiene nodos etiquetados con letras y aristas con distancias en línea recta (dlr). El usuario ingresa un nodo inicial y un nodo meta, y el algoritmo debe calcular el camino más corto desde el nodo inicial hasta el nodo meta basándose únicamente en una heurística que estima la distancia restante al objetivo.

Descripción de la Implementación

La implementación se realizó en el lenguaje de programación C++. A continuación, se describen las partes clave del código:

Estructuras y Datos:

- **Estructura Nodo**: Se define una estructura Nodo que contiene el estado del nodo (letra), el costo heurístico (costo_h), y un puntero al nodo padre.
- **dlr:** Un unordered_map que contiene las distancias en línea recta entre los nodos, organizadas por cada nodo y sus vecinos.
- **grafo:** Un unordered_map que representa las conexiones entre los nodos en el grafo, es decir, los vecinos de cada nodo.

Funciones:

- **heuristica:** Esta función toma dos nodos (actual y meta) y devuelve la distancia heurística entre ellos, que es la distancia en línea recta almacenada en dlr.
- **reconstruir_camino:** Esta función reconstruye el camino desde el nodo inicial hasta el nodo meta utilizando los punteros a los padres almacenados en cada nodo, e imprime el camino resultante.
- **busqueda_avara:** Esta es la función principal que implementa el algoritmo de búsqueda avara. Utiliza una cola de prioridad (priority_queue) para mantener los nodos a explorar ordenados por su costo heurístico. Explora los nodos basándose únicamente en la heurística, priorizando aquellos que están más cerca del nodo meta según la heurística.

Captura de pantalla de implementación del código

```
c:\Users\gejc2\.vscode\extensions\ms-vscode.cpptools-1.20.5-win32-x64\debugAdapters\
ydlbl.3zx' '--stdout=Microsoft-MIEngine-Out-kaxrgz1g.czh' '--stderr=Microsoft-MIEngi
f' '--dbgExe=C:\msys64\ucrt64\bin\gdb.exe' '--interpreter=mi'
Ingrese el estado inicial: A
Ingrese el estado meta: F
A E H F
PS C:\Users\gejc2\OneDrive\Escritorio\My\UNI\UACH\CIENCIAS DE LA COMPUTACION\SEMESTR
```

Lenguaje Utilizado: C ++. Por Gerardo Esteban Jurado Carrera.

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <limits>
#include <algorithm>

using namespace std;


struct Nodo {
    char estado;
    float costo_h;
    Nodo* padre;

    Nodo(char estado, float costo_h, Nodo* padre = nullptr)
        : estado(estado), costo_h(costo_h), padre(padre) {}

    // Sobrecarga del operador '>' para ordenar los nodos en la cola
     de prioridad
    // La comparación se basa en el costo_h
```

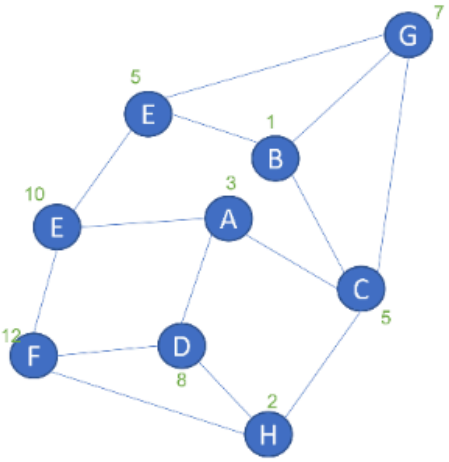
```
    bool operator>(const Nodo& otro) const {
        return costo_h > otro.costo_h;
    }
};

// Mapa para almacenar las distancias en línea recta (dlr) entre
    nodos
// Este mapa es un unordered_map donde la clave es un carácter que
    representa el nodo
// El valor es otro unordered_map que contiene las distancias a los
    nodos vecinos
unordered_map<char, unordered_map<char, float>> dlr = {
    {'A', {{'B', 1.8}, {'C', 2.5}, {'D', 2.8}, {'E', 4.5}, {'F',
    5.0}, {'G', 3.0}, {'H', 5.5}}},
    {'B', {{'A', 1.8}, {'C', 4.4}, {'D', 5.6}, {'E', 3.5}, {'F',
    2.3}, {'G', 4.5}, {'H', 3.2}}},
    {'C', {{'A', 2.5}, {'B', 4.4}, {'D', 3.8}, {'E', 3.4}, {'F',
    8.0}, {'G', 6.3}, {'H', 7.7}}},
    {'D', {{'A', 2.8}, {'B', 5.6}, {'C', 3.8}, {'E', 7.7}, {'F',
    8.0}, {'G', 2.8}, {'H', 10.0}}},
    {'E', {{'A', 4.5}, {'B', 3.5}, {'C', 3.4}, {'D', 7.7}, {'F',
    6.7}, {'G', 6.8}, {'H', 4.0}}},
    {'F', {{'A', 5.0}, {'B', 2.3}, {'C', 8.0}, {'D', 8.0}, {'E',
    6.7}, {'G', 5.3}, {'H', 3.5}}},
    {'G', {{'A', 3.0}, {'B', 4.5}, {'C', 6.3}, {'D', 2.8}, {'E',
    6.8}, {'F', 5.3}, {'H', 6.6}}},
    {'H', {{'A', 5.5}, {'B', 3.2}, {'C', 7.7}, {'D', 10.0}, {'E',
    4.0}, {'F', 3.5}, {'G', 6.6}}}
};

// Mapa para almacenar las conexiones entre los nodos en el grafo
// Este mapa es un unordered_map donde la clave es un carácter que
    representa el nodo
// El valor es un vector de caracteres que representan los nodos
    vecinos
unordered_map<char, vector<char>> grafo = {
    {'A', {'B', 'C', 'D', 'E', 'G'}},
    {'E', {'H'}},
    {'H', {'F'}}
};

// Función heurística que devuelve la distancia en línea recta entre
    dos nodos
```

```
// Esta función se utiliza para estimar el costo restante desde el
     nodo actual hasta el nodo meta
float heuristica(char actual, char meta) {
    return dlr[actual][meta];
}

// Función para reconstruir e imprimir el camino desde el nodo
     inicial hasta el nodo meta
// Esta función sigue los punteros a los padres desde el nodo meta
     hasta el nodo inicial
void reconstruir_camino(Nodo* nodo) {
    vector<char> camino;  // Vector para almacenar el camino
    while (nodo) {
        camino.push_back(nodo->estado);
        nodo = nodo->padre;
    }
     reverse(camino.begin(), camino.end());  // Invierte el camino
     para obtener el orden correcto
    for (char estado : camino) {
        cout << estado << " ";
    }
    cout << endl;
}

// Implementación del algoritmo de búsqueda avara
// Esta función busca el camino desde el nodo inicial hasta el nodo
     meta
void busqueda_avara(char inicio, char meta) {
     // Cola de prioridad para nodos a explorar, ordenada por el
     costo heurístico (h)
    priority_queue<Nodo, vector<Nodo>, greater<Nodo>> abiertos;
    // Mapa para almacenar todos los nodos visitados
    unordered_map<char, Nodo*> cerrados;

     // Añadir el nodo inicial al conjunto abierto con heurística
     calculada
    abiertos.emplace(inicio, heuristica(inicio, meta));
    // Almacenar el nodo inicial en el mapa de todos los nodos
    cerrados[inicio] = new Nodo(inicio, heuristica(inicio, meta));

    // Bucle principal de búsqueda
    while (!abiertos.empty()) {
```

```
          // Obtener el nodo con el menor costo heurístico (h) del
     conjunto abierto
        Nodo actual = abiertos.top();
        abiertos.pop();

        // Verificar si se ha alcanzado el nodo meta
        if (actual.estado == meta) {
              reconstruir_camino(&actual);  // Si se alcanza el nodo
     meta, reconstruir el camino
            return;
        }

        // Explorar los vecinos del nodo actual
        for (char vecino : grafo[actual.estado]) {
              // Calcular el costo h (heurística estimada desde el
     vecino hasta el nodo meta)
            float costo_h = heuristica(vecino, meta);
              // Crear un nuevo nodo vecino con el costo heurístico
     calculado y el nodo actual como padre
                Nodo* nodo_vecino = new Nodo(vecino, costo_h, new
     Nodo(actual));
            // Añadir el vecino al conjunto abierto
            abiertos.push(*nodo_vecino);
               // Almacenar el nodo vecino en el mapa de todos los
     nodos
            cerrados[vecino] = nodo_vecino;
        }
    }


    cout << "No se encontró un camino." << endl;
}

int main() {
    char inicio, meta;
    cout << "Ingrese el estado inicial: ";
    cin >> inicio;
    cout << "Ingrese el estado meta: ";
    cin >> meta;

    busqueda_avara(inicio, meta);
}
```

(c).- Se tiene el siguiente espacio estado:

Utilizando las siguientes técnicas de búsqueda local:



1. Búsqueda tabú

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

Lenguaje Utilizado: JavaScript. Por José Ángel Ortiz Meraz.
Consideraciones:

1. *Deberá instalar prompt desde la terminal con: npm install prompt-sync*

```
Anngelo@DESKTOP-E06B266 MINGW64 ~
$ npm install prompt-sync

added 3 packages in 3s
```

2. *Para correr el código ejecute en terminal: node Busqueda-Tabu.js*

```
Anngelo@DESKTOP-E06B266 MINGW64 ~/Downloads
$ node Busqueda-Tabu.js
Ingrese el nodo de inicio (A-I):G
El mejor estado posible es: B con un valor de: 1
```

O bien, utilice un compilador online sin necesidad de instalar paquetes adicionales.
https://www.programiz.com/javascript/online-compiler/

```
const prompt = require('prompt-sync')();
// Debemos definir el grafo con sus nodos, sus valores y los nodos
vecinos de cada nodo
const graph = {
    A: { value: 3, neighbors: ['E', 'D', 'C'] },
    B: { value: 1, neighbors: ['I', 'G', 'C'] },
    C: { value: 5, neighbors: ['B', 'G', 'H', 'A'] },
    D: { value: 8, neighbors: ['A', 'F', 'H'] },
    E: { value: 10, neighbors: ['I', 'A', 'F'] },
    F: { value: 12, neighbors: ['E', 'D', 'H'] },
    G: { value: 7, neighbors: ['I', 'B', 'C'] },
    H: { value: 2, neighbors: ['C', 'D', 'F'] },
```

```
    I: { value: 5, neighbors: ['G', 'B', 'E'] }
};

// Definimos la función de búsqueda tabú, que recibirá como parámetro el nodo desde el que se comenzará la búsqueda
function tabuSearch(startNode) {
    let currentNode = startNode; // Nodo actual
       let bestNode = currentNode; // Nodo con mejor utilidad encontrado (cuando recién arrancamos el programa es el mismo nodo de inicio)
     let tabuList = []; // Lista tabú guardar los nodos prohibidos, es decir, los nodos que ya no podemos visitar porque ya los hemos visitado una vez
    let iterations = 0;
    let maxIterations = Object.keys(graph).length * 2; // Máximo de iteraciones, lo que nos dará una chance más grande de que el algoritmo explore el grafo, pero no tanta como para colapsar en tiempo de ejecución

    while (iterations < maxIterations) {
          let neighbors = graph[currentNode].neighbors; // Vecinos del nodo actual
         let bestNeighbor = null; // Mejor utilidad del nodo vecino no visitado, es decir, no tabú encontrado
          let bestValue = Infinity; // Valor del mejor vecino, como buscamos el mínimo ponemos nuestra variable a un valor alto para que cualquier valor de los nodos sea menor que el valor inicial

        // Evaluaamos cada nodo vecino del nodo actual
        neighbors.forEach(neighbor => {
              // Si el nodo vecino no está en la lista tabú (nodos prohibidos) y tiene un valor menor que el mejor valor encontrado entonces
                            if (!tabuList.includes(neighbor) && graph[neighbor].value < bestValue) {
                  bestNeighbor = neighbor; // Actualizamos el mejor nodo vecino
                 bestValue = graph[neighbor].value; // Actualizamos el mejor valor utilidad
             }
        });
```

```
            // Si no encuentra ningún vecino válido, terminamos la
búsqueda
        if (bestNeighbor === null) {
            break;
        }

        currentNode = bestNeighbor; // Movemos al mejor nodo vecino
que hemos encontrado

        // Si el valor del nodo actual es mejor que el mejor valor
encontrado o utilidad entonces
        if (graph[currentNode].value < graph[bestNode].value) {
            bestNode = currentNode; // Actualizamos el mejor nodo
        }

        // Agregamos el nodo actual a la lista tabú o lista de nodos
prohibidos
        tabuList.push(currentNode);

            iterations++; // Vamos incrementando el valor de las
iteraciones que hemos hecho
    }

      return bestNode; // Retornamos oregresamos el mejor nodo
encontrado o utilidad
}

function getStartNode() {
    while (true) {
            const startNode = prompt("Ingrese el nodo de inicio
(A-I):").toUpperCase();
        if (graph[startNode]) {
                return startNode; // Sólo si el nodo es válido lo
retornamos
        } else {
                alert("Nodo no válido. Por favor, ingrese un nodo
válido (A-I)."); // De lo contrario, mostramos una alerta
        }
    }
}

// Solicitamos que el usuario digite el nodo de inicio del cual
quiere empezar la búsqueda tabú
```

```
const startNode = getStartNode();
const bestNode = tabuSearch(startNode);
console.log(`El mejor estado posible es: ${bestNode} con un valor
de: ${graph[bestNode].value}`);
```