GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: RECORRIDO A LO ANCHO (BFS)

1. Introducción

La búsqueda en amplitud o a lo ancho es uno de los algoritmos de búsqueda básicos y esenciales en grafos.

Como resultado de cómo funciona el algoritmo, la ruta encontrada por la búsqueda primero en amplitud a cualquier nodo es la ruta más corta a ese nodo, es decir, la ruta que contiene la menor cantidad de aristas en gráficos no ponderados.

2. Conocimientos previos

2.1. Cola

Una cola es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción push se realiza por un extremo y la operación de extracción pull por el otro. También se le llama estructura FIFO (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir.

La particularidad de una estructura de datos de cola es el hecho de que solo podemos acceder al primer y al último elemento de la estructura. Así mismo, los elementos solo se pueden eliminar por el principio y solo se pueden añadir por el final de la cola.

2.1.1. C++

En el caso de C++ para utilizar la cola incluimos la biblioteca *queue* que nos permite utilizar la cola propia del lenguaje la cual cuenta con las siguientes funcionalidades

- **queue::empty():** Devuelve si la cola está vacía.
- **queue::size():** Devuelve el tamaño de la cola.
- **queue::swap():** Intercambia el contenido de dos colas, pero las colas deben ser del mismo tipo, aunque los tamaños pueden diferir.
- **queue::emplace():** Inserta un nuevo elemento en el contenedor de la cola, el nuevo elemento se agrega al final de la cola.
- **queue::front():** Devuelve una referencia al primer elemento de la cola.
- **queue::back():** Devuelve una referencia al último elemento de la cola.
- **queue::push(g):** Agrega el elemento 'gál final de la cola.
- **queue::pop():** Elimina el primer elemento de la cola.

```
#include <iostream>
#include <queue>

using namespace std;
```

```
void showq(queue<int> gq){
   queue<int> g = gq;
   while (!g.empty()) {
     cout << '\t' << g.front();
     g.pop();
   }
   cout << '\n';
}

int main(){
  queue<int> gquiz;
  gquiz.push(10);
  gquiz.push(20);
  gquiz.push(30);

  cout << "The queue gquiz is : ";
  showq(gquiz);

  cout << "\ngquiz.size() : " << gquiz.size();
  cout << "\ngquiz.front() : " << gquiz.front();
  cout << "\ngquiz.back() : " << gquiz.back();
  cout << "\ngquiz.pop() : ";
  gquiz.pop();
  showq(gquiz);
  return 0;
}
```

2.1.2. Java

En Java, Queue es una interfaz que forma parte del paquete java.util. La interfaz de Queue amplía la interfaz de Colllection de Java.

Para usar una cola en Java, primero debemos importar la interfaz de la cola de la siguiente manera:

```
importar java.util.queue;
```

O

```
importar java.util.*;
```

Una vez importado, podemos crear una cola como se muestra a continuación:

```
Queue<String> str_queue = new LinkedList<> ();
```

Como Queue es una interfaz, usamos una clase LinkedList que implementa la interfaz Queue para crear un objeto de cola.

Del mismo modo, podemos crear una cola con otras clases concretas.

```
Queue<String> str_pqueue = new PriorityQueue<> ();
Queue<Integer> int_queue = new ArrayDeque<> ();
```

Ahora que se creó el objeto de la cola, podemos inicializar el objeto de la cola al proporcionarle los valores a través del método de agregar, como se muestra a continuación.

```
str_queue.add("uno");
str_queue.add("dos");
str_queue.add("tres");
```

- **add boolean add(E e):** Agrega el elemento e a la cola al final (cola) de la cola sin violar las restricciones de capacidad. Devuelve verdadero si tiene éxito o IllegalStateException si la capacidad está agotada.
- **peek E peek():** Devuelve la cabeza (frente) de la cola sin eliminarla.
- **element E element():** Realiza la misma operación que el método peek(). Lanza NoSuchElementException cuando la cola está vacía.
- **remove E remove():** Elimina la cabeza de la cola y la devuelve. Lanza NoSuchElementException si la cola está vacía.
- **poll E poll():** Elimina la cabeza de la cola y la devuelve. Si la cola está vacía, devuelve nulo.
- **size int size():** Devuelve el tamaño o el número de elementos en la cola.
- **Offer boolean offer(E e):** Inserta el nuevo elemento e en la cola sin violar las restricciones de capacidad.

Puede tener otro grupo de operaciones las cuales va a depender de con que clase se instancie la interfaz Queue en si.

```
import java.util.*;

public class Main {
  public static void main(String[] args) {
        Queue<String> str_queue = new LinkedList<>();
        str_queue.add("one");
        str_queue.add("two");
        str_queue.add("three");
        str_queue.add("four");

        System.out.println("The Queue contents:" + str_queue);
  }
}
```

3. Desarrollo

El algoritmo toma como entrada un grafo no ponderado y la identificación del vértice de origen s. El grafo de entrada puede ser dirigido o no dirigido, no importa el algoritmo.

El algoritmo se puede entender como un fuego que se propaga en el grafo: en el paso cero solo la fuente s está en llamas. A cada paso, el fuego que arde en cada vértice se propaga a todos sus vecinos. En una iteración del algoritmo, el *anillo de fuego* se expande en ancho en una unidad (de ahí el nombre del algoritmo).

Más precisamente, el algoritmo se puede establecer de la siguiente manera: Crear una cola q que contendrá los vértices a procesar y un arreglo booleano $used[]$ que indica para cada vértice, si ha sido encendido (o visitado) o no.

Inicialmente, adicione la fuente s a la cola y establezca $used[s] = true$, y para todos los demás vértices v set $used[v] = false$. Luego, repita hasta que la cola esté vacía y, en cada iteración, extraiga un vértice desde el frente de la cola. Iterar a través de todos las aristas que salen de este vértice y si algunos de estos bordes van a vértices que aún no están encendidos, prende fuego y colócalos en la cola.

Como resultado, cuando la cola está vacía, el *anillo de fuego* contiene todos los vértices accesibles desde la fuente s, con cada vértice alcanzado de la manera más corta posible. También puede calcular las longitudes de las rutas más cortas (que solo requiere mantener una matriz de longitudes de ruta $d[]$), así como guardar información para restaurar todas estas rutas más cortas (para esto, es necesario mantener una matriz de $padresp[]$, que almacena para cada vértice el vértice desde el que llegamos)

El siguiente ejemplo ilustra el funcionamiento del algoritmo BFS sobre un grafo de ejemplo. La secuencia de ilustraciones va de izquierda a derecha y de arriba hacia abajo. El algoritmo comienza por el nodo 0.

.

3.1. BFS sobre tablero

Una variante de este algoritmo es desarrollarlo sobre una matriz que en disimiles situaciones actua como un mapa de donde una celda se puede a unas de sus vecinas acorde a al concepto de vecinas que plantee el problema. Para este tipo de sistuaciones se trabaja con una matriz de visitados para saber cual celda de la matriz ha sido visitado y cual no.

Además se utiliza una estructura para representar la celda de la matriz de la cual siempre en todas las ocasiones se almacena la fila y columna de dicha celda. El otro elemento en este tipo de situaciones son las llamadas matrices direccionales que ayudan de una forma más practica dada una celda hallar todas las celdas vecinas a esta dependiendo del problema. En la implementación del guía se pone un ejemplo de este caso muy peculiar pero super utilizado para resolver problemas.

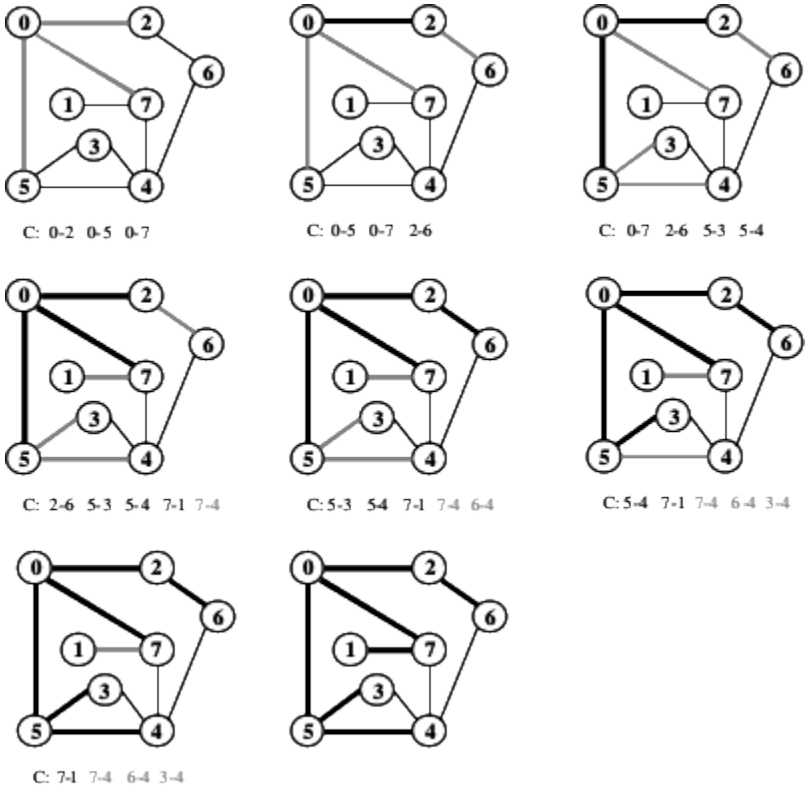

Figura 1: Ejecucción del algoritmo BFS.

4. Implementacion

4.1. C++

```
#include <queue>
#define MAX_N 5001
using namespace std;

struct Node{
   vector<int> adj;
};

Node graf[MAX_N];
bool mark[MAX_N];

inline void BFS(int start){
  queue<int> bfs_queue;
  bfs_queue.push(start);
  while (!bfs_queue.empty()){
    int xt = bfs_queue.front();
    bfs_queue.pop();
```

```
        mark[xt] = true;
        for(int i=0;i<graf[xt].adj.size();i++){
      if(!mark[graf[xt].adj[i]]){
        bfs_queue.push(graf[xt].adj[i]);
        mark[graf[xt].adj[i]] = true;
      }
    }
  }
}
```

4.1.1. BFS sobre tablero

```
#include <queue>
#define MAX 110
#define REP(i,n) for(int i=0;i<(int)n;++i)

/*Matrices direccionales en este caso dada una celda se puede mover hacia una
   de la celda vecinas en la vertical u horizontal el primer arreglo indica
   desplazamiento en la filas mientras el segundo es la columna el primer par
    ordenado hace referencia a la posicion a la derecha de la celda que este
   parado en ese momento, el segundo par es la celda a la izquierda mientras
   tercer y cuarto par son las celdas de arriba y abajo respectivamente */
const int mov_r[]={ 0, 0,-1, 1};
const int mov_c[]={ 1,-1, 0, 0};

/*Estructura para representar la celda*/
struct Cell{
  int r,c;
  Cell(int _r=0,int _c=0){
    r=_r;
    c=_c;
  }
};

int nrows,ncolumns;
bool maps[MAX][MAX];
Cell tmp;

/*Funcion tipica para validar que una supesta celda este dentro de la matriz
   que trabajo*/
bool validCell(int _r, int _c){
  return (1<=_r && _r<=nrows && 1<=_c && _c<=ncolumns);
}


void bfs(Cell _start){

   queue<Cell> visit;
```

```
    Cell current,next;
    maps[_start.r][_start.c]=true;
    visit.push(_start);

    while(!visit.empty()){
      current=visit.front();
      visit.pop();
      REP(i,4){
         /*A partir de una celda actual (current) y las matrices de direccion
            genero cada una de las posibles
         vecinas (next) en cada iteracion del ciclo*/
        next.c=current.c+mov_c[i];
        next.r=current.r+mov_r[i];

        /*Cheqeo que la vecina este dentro de la matriz*/
        if(validCell(next.r,next.c)){
           if(maps[next.r][next.c]==false){
             maps[next.r][next.c]=true;
             visit.push(next);
           }
        }
      }
  }
}
```

4.2. Java

```
import java.io.*;
import java.util.*;

class Graph{
  private int V; // Numero de vertices
  private LinkedList<Integer> adj[]; //Lista de adyacencia

  // Constructor
  Graph(int v){
    V = v;
    adj = new LinkedList[v];
    for (int i=0; i<v; ++i)
    adj[i] = new LinkedList();
  }

  // Funcion para adicionar una arista al grafo
  void addEdge(int v,int w){
    adj[v].add(w);
  }

  void BFS(int s){
```

```
    boolean visited[] = new boolean[V];

    Queue<Integer> queue = new LinkedList<Integer>();

    visited[s]=true;
    queue.add(s);

    while (queue.size() != 0){
          n = queue.poll();
          for (int i = 0; i < adj[n].size(); i++){
                a = adj[n].get(i);
                if(!visited[a]){
          nodes[a] = true;
          queue.add(a);
        }
          }
        }
  }
}
```

5. Complejidad

Este algoritmo tiene una complejidad de O(V+E) donde V es la cantidad de vértices del grafo y E las aristas. El algoritmo recibe como parámetro el nodo inicial por el cual se inicia el BFS.

6. Aplicaciones

En los ejercicios y problemas de concursos para resolverlos utilizando el algoritmo BFS hay que partir de las aplicaciones o usos que se le puede dar a este en diferentes situaciones donde la utilización de este algoritmo es parte de la solución global al problema.

- Encuentre la ruta más corta desde un vertices a otros vértices en un grafo no ponderado.
- Encuentra todas las componentes conexa en un grafo no dirigido en tiempo O(n+m): Para hacer esto, solo ejecutamos BFS comenzando desde cada vértice, excepto los vértices que ya han sido visitados en ejecuciones anteriores. Por lo tanto, realizamos BFS normal desde cada uno de los vértices, pero no reiniciamos el arreglo de visitados utilizados cada vez que obtenemos una nueva componente conectada, y el tiempo total de ejecución seguirá siendo O (n + m) (realizando múltiples BFS en el gráfico sin poner a cero el arreglo $used[]$ se llama una serie de búsquedas primero en amplitud).
- Encontrar una solución a un problema o un juego con el menor número de movimientos, si cada estado del juego se puede representar por un vértice del grafo, y las transiciones de un estado al otro son las aristas del grafo.
- Encontrar la ruta más corta en un gráfico con pesos 0 o 1: Esto requiere solo una pequeña

modificación a la búsqueda normal en anchura: si la arista actual de peso cero y la distancia al vértice es más corta que la distancia encontrada actual, agregue esto vértice no hacia atrás, sino hacia el frente de la cola.

- Encontrar el ciclo más corto en un grafo no ponderado dirigido: Inicie una búsqueda en anchura desde cada vértice. Tan pronto como intentamos volver del vértice actual al vértice de origen, hemos encontrado el ciclo más corto que contiene el vértice de origen. En este punto, podemos detener el BFS y comenzar un nuevo BFS desde el siguiente vértice. De todos esos ciclos (como máximo uno de cada BFS) elija el más corto.
- Encuentre todos las aristas que se encuentran en cualquier camino más corto entre un par de vértices dados (a, b). Para hacer esto, ejecute dos búsquedas primero en amplitud: una desde a y otra desde b. Sea $d_a[]$ la matriz que contiene las distancias más cortas obtenidas del primer BFS (de a) y $d_b[]$ sea la matriz que contiene las distancias más cortas obtenidas del segundo BFS de b. Ahora, para cada arista (u, v) es fácil verificar si esa arista se encuentra en algún camino más corto entre a y b: el criterio es la condición $d_a[u] + 1 + d_b[v] = d_a[b]$.
- Encuentre la ruta más corta de longitud uniforme desde un vértice de origen s hasta un vértice de destino t en un grafo no ponderado: para esto, debemos construir un grafo auxiliar, cuyos vértices son el estado (v, c), donde v - el nodo actual, c=0 o c=1 - la paridad actual. Cualquier borde (a,b) del grafo original en esta nueva columna se convertirá en dos aristas ((u,0),(v,1)) y ((u,1),(v,0)). Después de eso, ejecutamos un BFS para encontrar la ruta más corta desde el vértice inicial (s,0) hasta el vértice final (t,0).

7. Ejercicios propuestos

A continuación una lista de ejercicios que se resuelven utilizando esta técnica:

- Codeforces - Shortest Path.
- Codeforces - Police Stations
- Codeforces - Okabe and City.
- Codeforces - Bear and Forgotten Tree 2.
- Codeforces - Cycle in Maze.
- DMOJ - Manchas en el Papel
- DMOJ - Escondidas
- DMOJ - Closing the Farm
- DMOJ - Avoid the lake
- DMOJ - Profundidad de las Páginas
- DMOJ - El salto del caballo
- DMOJ - En Hombros