Búsqueda de Profundidad (DFS)

Lorenz Francisco Equihua Loeck

September 2023

Índice

Antecedentes

Grafos

Un grafo G se define como un par ordenado (V, E), donde V representa un conjunto de elementos conocidos como vértices o nodos, y E representa un conjunto de pares no ordenados de vértices, denominados aristas o arcos [14][1]. En el contexto del algoritmo de búsqueda profunda, es esencial comprender la estructura y las relaciones dentro de un grafo.

Los grafos se representan visualmente mediante diagramas en el plano, donde cada vértice se simboliza con un punto o un pequeño círculo, y cada arista se representa mediante una curva o una línea recta que conecta sus extremos. Dos vértices se consideran adyacentes cuando existe una arista que los conecta; en este caso, ambos vértices se consideran incidentes en esa arista. Además, un vértice se clasifica como aislado si no está relacionado con ninguna arista en el grafo [9].

Para ilustrar esto, consideremos el siguiente ejemplo:

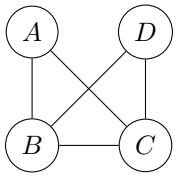

Figura 1

En este ejemplo, tenemos 4 vértices (A, B, C, D) y 5 aristas que los conectan. Podemos definir formalmente el grafo como:

$$V = \{A, B, C, D\} \text{ y}$$
$$E = \{e_1 = (A, B), e_2 = (B, C), e_3 = (C, D), e_4 = (A, C), e_5 = (B, D)\}.$$

Camino, Camino Cerrado, Camino Simple, Paseo, Ciclo

En la teoría de grafos, se utilizan varios conceptos importantes para describir rutas y ciclos en un grafo $G = (V, E)$.

I. **Camino en G:** Un camino en G es una sucesión finita y alternante de vértices y aristas de G que comienza en el vértice V_0 y termina en el vértice V_n. La sucesión se representa como $V_0, e_1, V_1, e_2, V_2, \ldots, V_{n-1}, e_n, V_n$, donde n es la longitud del camino. Un camino se denota por su secuencia de vértices $(V_0, V_1, \ldots, V_n)$ [9].

II. **Camino Cerrado:** Un camino se considera cerrado cuando $V_0 = V_n$. Si V_0 y V_n no son iguales, se dice que el camino es de V_0 a V_n [9].

[1] `https://w3.ual.es/~btorreci/tr-grafos.pdf`, consultada el 04 de septiembre de 2023

III. **Camino Simple:** Un camino simple es un camino en el cual todos sus vértices son distintos [9].

IV. **Paseo:** Un paseo es un camino en el cual todas sus aristas son distintas [9].

V. **Ciclo:** Un ciclo es un camino cerrado de longitud 3 o más en el cual todos sus vértices son distintos, excepto $V_0 = V_n$. Un ciclo de longitud k se denomina k-ciclo [9].

Grafo Conexo

Un grafo $G = (V, E)$ se considera conexo si entre cualquier par de vértices existe al menos un camino que los conecta [9]. A continuación, se presentan ejemplos de dos grafos para ilustrar la noción de conexidad:

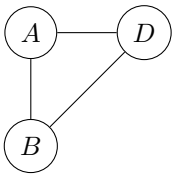

Figura 2: Grafo conexo.

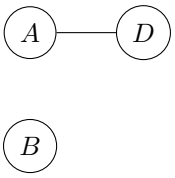

Figura 3: Grafo no conexo.

En el Grafo 1, se puede observar que existe al menos un camino que conecta cualquier par de vértices (por ejemplo, los vértices A y D están conectados por el camino $A \to D$). Por lo tanto, el Grafo 1 es un grafo conexo.

Por otro lado, en el Grafo 2, no existe un camino que conecte todos los vértices. Por ejemplo, los vértices A y B no están conectados por ningún camino. Por lo tanto, el Grafo 2 no es conexo.

Lista de Adyacencia

Una lista de adyacencia es una representación de todas las aristas o arcos de un grafo mediante una lista [6][2]. Consiste en una estructura de datos que almacena la información de las conexiones entre vértices en un grafo. En una lista de adyacencia, cada vértice del grafo tiene una entrada asociada que contiene una lista de los vértices adyacentes a ese vértice.

Consideremos el siguiente grafo G como ejemplo:

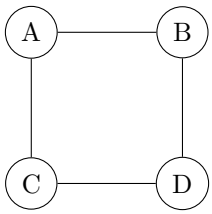

Figura 4

Su lista de adyacencia se representa de la siguiente manera:

Vértice	Vértices Adyacentes
A	B, C
B	A, D
C	A, D
D	B, C

Figura 5: Lista de Adayencia

[2] `https://www.youtube.com/watch?v=KHyGn4cRjN4`, consultado el 04 de septiembre de 2023

Grafo Dirigido o Dígrafo

Un grafo dirigido o dígrafo $D = (V_D, E_D)$ se define como una dupla que consiste en [9]:

I. V_D: Un conjunto de vértices o nodos.

II. E_D: Un conjunto de pares ordenados de vértices llamados arcos o aristas dirigidas.

A continuación, se muestra un ejemplo de un grafo dirigido:

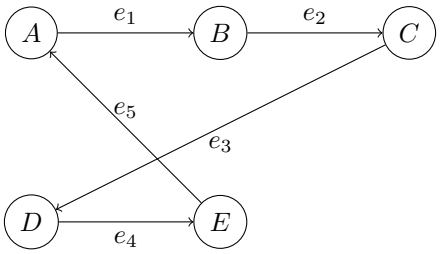

Figura 6

Formalmente, el grafo se define como:

$$V = \{A, B, C, D, E\},$$
$$E = \{e_1 = (A, B), e_2 = (B, C), e_3 = (C, D), e_4 = (D, E), e_5 = (E, A)\}.$$

Árbol

Un árbol es un tipo especial de grafo en el que cualquier par de vértices está conectado por exactamente un camino [13][3]. En un árbol, se cumple la siguiente definición:

- Cualquier par de vértices está conectado por un camino único.
- Hay un nodo, especialmente designado, llamado la raíz del árbol T.
- Dado un nodo n y una serie de árboles $T_1, T_2, ..., T_k$ con raíces $n_1, n_2, ..., n_k$, podemos crear un nuevo árbol haciendo que n sea el padre de los nodos $n_1, n_2, ..., n_k$. Decimos entonces que n es la raíz del nuevo árbol, $T_1, T_2, ..., T_k$ son subárboles de la raíz y a los nodos $n_1, n_2, ..., n_k$ son hijos del nodo n[12][4].
- Los nodos que no son raíces de otros subárboles se denominan hojas y no tienen sucesores o hijos.

A continuación, se muestra un ejemplo de un árbol:

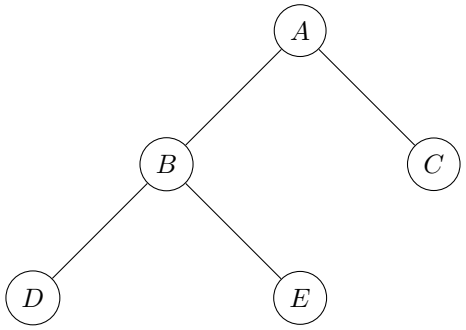

Figura 7

En este ejemplo, el vértice A es la raíz del árbol, y el subárbol es $T_1 = \{B, D, E\}$, donde B es la raíz del subárbol. Los vértices D, E y C son hojas, ya que no tienen sucesores o hijos. El árbol ilustrado cumple con todas las propiedades de un árbol.

Pila o Stack

Una pila es una lista ordenada o estructura de datos que permite almacenar y recuperar datos siguiendo el principio UEPS (Ultimo en Entrar, Primero en

[3] `https://docplayer.es/26258138-Teoria-de-grafos-ingenieria-de-sistemas.html`, consultado el 04 de septiembre de 2023

[4] `https://www.dlsi.ua.es/asignaturas/p2ir/teoria/l03/lessonh.html`, consultado el 13 de septiembre de 2023.

Salir) [10][5]. Esto significa que el último elemento que se agrega a la pila es el primero en ser recuperado.

Puede entenderse mejor este concepto utilizando la analogía de lavar platos después de una cena. Cuando se apilan los platos, el primer plato que se lavará es el último que se colocó en la pila de platos, y el último plato en ser agregado será el último en ser lavado.

Una pila es ampliamente utilizada en programación y estructuras de datos para llevar a cabo diversas operaciones, como la gestión de llamadas de funciones, la inversión de cadenas, la evaluación de expresiones matemáticas, entre otros.

[5] `https://es.wikipedia.org/wiki/Pila_%28inform%C3%A1tica%29`, consultado el 05 de septiembre de 2023

Búsqueda en grafos

En teoría de grafos, existen varios problemas y algoritmos importantes. Algunos de estos problemas incluyen[16][6]:

- **Conectividad**: Determinar si dos vértices de un grafo están conectados.
- **Ruta Más Corta**: Encontrar la ruta más corta entre dos vértices de un grafo.
- **Camino Hamiltoniano**: Encontrar un ciclo en un grafo que pase por todos los vértices una y solo una vez.

Para abordar estos problemas, se utilizan algoritmos de búsqueda, como la Búsqueda en Anchura (BFS) y la Búsqueda en Profundidad (DFS). Estos algoritmos recorren el grafo de manera diferente[8][7]:

- **Búsqueda en Anchura (BFS)**: Recorre el grafo por niveles, visitando todos los vecinos de un vértice antes de avanzar al siguiente nivel. Esto significa que se exploran primero todos los vértices a una distancia de 1 del vértice inicial, luego a distancia 2, y así sucesivamente.
- **Búsqueda en Profundidad (DFS)**: Sigue primero una ruta desde el inicio hasta el nodo final (vértice), luego otra ruta de principio a fin, y así sucesivamente, explorando todas las ramas antes de retroceder.

A continuación, se muestra un ejemplo de cómo estos algoritmos recorren el siguiente grafo:

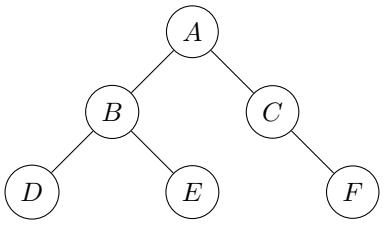

Figura 8

Recorrido BFS: A, B, C, D, E, F.
Recorrido DFS: A, B, D, E, C, F.

[6] `https://www.grapheverywhere.com/teoria-de-grafos/`, consultado el 04 de septiembre de 2023

[7] `https://es.gadget-info.com/difference-between-bfs`, consultado el 04 de septiembre de 2023

Búsqueda de Profundidad

Depth First Search (DFS) o búsqueda en profundidad es un algoritmo de búsqueda no informada utilizado para recorrer todos los nodos de un grafo o árbol de manera ordenada, pero no uniforme[4][8]. El algoritmo búsqueda en profundidad empieza en un nodo raíz (que puede ser arbitrario en el caso de un grafo) y explora lo más posible a lo largo de cada rama antes de retroceder[5][9]. Para hacer esto, se necesita una memoria adicional, usualmente una pila, para guardar los nodos descubiertos hasta el momento a lo largo de un camino especificado. El algoritmo visita cada nodo una sola vez y marca los nodos ya visitados para evitar ciclos infinitos.

Existen dos formas de implementar el algoritmo búsqueda en profundidad: una utilizando recursión y otra sin ella. En este reporte, se analiza únicamente la versión sin recursión.

Para ejemplarizar el algoritmo suponga el siguiente grafo:

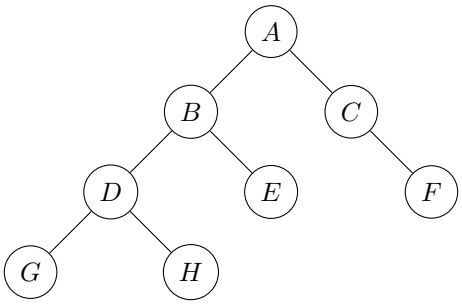

Figura 9: Grafo inicial.

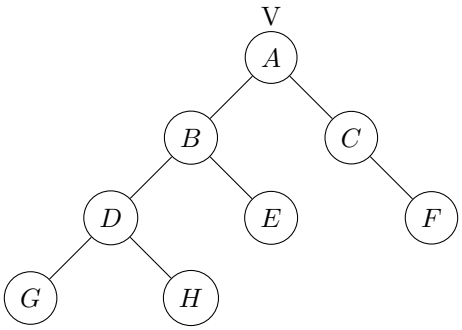

Figura 10: Paso 1.

[8] `https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_profundidad`, consultado el 05 de septiembre de 2023

[9] `https://www.techiedelight.com/es/dfs-interview-questions/`, consultado el 05 de septiembre de 2023

1. Marcaremos con una V los vértices que ya hayamos visitado y marcaremos la arista de color rojo cuando regresemos al vértice anterior. Comenzaremos nuestro algoritmo en el vértice $V(A)$ y comenzaremos a aplicar el algoritmo, por lo que marcaremos los vértices $V(B)$, $V(D)$, $V(G)$. Figura 10

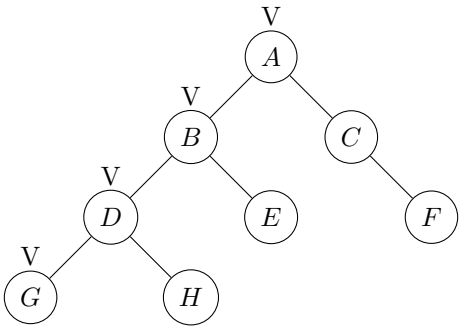

Figura 11: Paso 2.

2. Como el vértice $V(G)$ ya no tiene hijos, regresaremos al vértice $V(D)$, que todavía tiene un vértice hijo por explorar. Figura 11

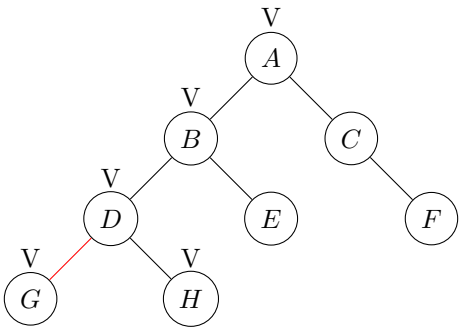

Figura 12: Paso 3.

3. Como el vértice $V(H)$ ya no tiene hijos, regresaremos al vértice $V(D)$, el cual ya no tiene vértices por explorar, por lo que regresamos al vértice $V(B)$. En el cual podemos visitar al vértice $V(E)$. Figura 12

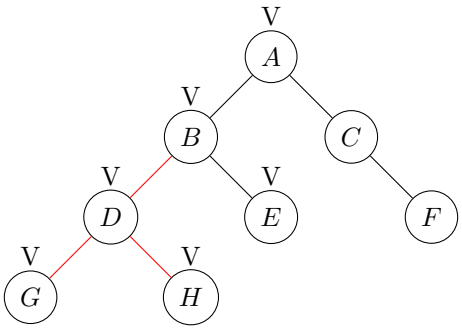

Figura 13: Paso 4.

4. Como el vértice $V(E)$ ya no tiene hijos y del vértice $V(B)$ ya no hay vértices por explorar, nos podemos regresar hasta el vértice $V(A)$ en el cual podemos comenzar a explorar el vértice $V(C)$ y subsecuentemente el vértice $V(F)$. Y así terminar de explorar todo el grafo. Figura 13

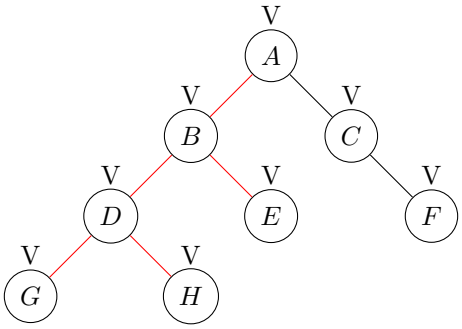

Figura 14: Paso 5.

5. Es importante notar que las aristas (A, C) y (C, F) no se marcan en rojo, ya que el algoritmo termina antes de poder regresar por esos caminos.

Pseudocódigo

A continuación se presenta el pseudocódigo del algoritmo de búsqueda en profundidad utilizado para recorrer un grafo.[1][10].

Algorithm 0: DFS(Grafo, verticeInicial)

```
   Data: Grafo, verticeInicial
   Result: Nada
   // Crear una pila para el recorrido
 1 Crear una pila llamada PILA;
   // Crear un conjunto para rastrear los nodos visitados
 2 Crear un conjunto llamado VISITADO;
   // Colocar el vértice inicial en la pila
 3 Poner verticeInicial en PILA;
 4 while PILA no esté vacía do
       // Extraer un vértice de la pila
 5     Sacar u de PILA;
 6     if u no está en VISITADO then
           // Marcar el vértice, u, como visitado y agregarlo al
              conjunto VISITADO
 7         Marcar u como visitado y ponerlo en VISITADO;
           // Explorar los vértices adyacentes
 8         for cada v adyacente a u en Grafo do
               // Agregar los vértices adyacentes, v, a la pila
                  para su posterior exploración
 9             Poner v en PILA;
```

[10] https://codebreakerscorp.wordpress.com/2011/03/05/algoritmo-de-busqueda-depth-first-search/ consultado el 05 de septiembre de 2023

Complejidad

A continuación, se presenta un análisis de la complejidad del algoritmo de búsqueda en profundidad aplicado a un grafo.

Algorithm 0: DFS(Grafo, verticeInicial)

```
   Data: Grafo, verticeInicial
   Result: Nada
   // Inicialización
1  Crear una pila llamada PILA;                                  // O(1)
2  Crear un conjunto llamado VISITADO;                           // O(1)
3  Poner verticeInicial en PILA;                                 // O(1)
4  while PILA no esté vacía do
   |  ;                                                         // O(V)
5  |  Sacar u de PILA;                                          // O(1)
6  |  if u no está en VISITADO then
7  |  |  Marcar u como visitado y ponerlo en VISITADO;          // O(1)
8  |  |  for cada v adyacente a u en Grafo do
   |  |  |  ;                                                   // O(E)
9  |  |  |  Poner v en PILA;                                    // O(1)
```

La complejidad total del algoritmo de búsqueda en profundidad se puede calcular sumando las complejidades individuales de sus operaciones. La inicialización tiene un costo constante $O(1)$. El bucle principal se ejecuta una vez por cada vértice $|V|$ en el grafo, y dentro del bucle, la operación de examinar los vértices adyacentes tiene un costo proporcional al número de aristas $|E|$. Por lo tanto, la complejidad total es $O(|V| + |E|)$[2].

Código del Programa

El código fuente utilizado en este proyecto está disponible en el siguiente enlace: `https://github.com/lfeq/DFS`.

Es importante señalar que el código que se analiza en este reporte puede haber sido ligeramente modificado para la presentación en clase. Las funciones adicionales que se han agregado al repositorio se encuentran separadas del código analizado en este reporte mediante el uso de `#pragma region extras` dentro de los archivos correspondientes.

Explicación del Programa

En esta sección se presenta la implementación de la clase `Graph` y el algoritmo de búsqueda en profundidad no recursivo[11][11]. El código fuente se encuentra en el archivo `Grafo.hpp`.

Listing 1: Grafo.hpp

```cpp
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

class Graph {
private:
    int n; // Number of vertices
    vector<vector<int>> adjacency_list; // Adjacency
       list

public:
    Graph(int n) : n(n), adjacency_list(n) {}

    // Add an edge from u to v
    void add_edge(int u, int v) {
        adjacency_list[u].push_back(v);
    }

    // Implement the non-recursive DFS algorithm
    void dfs_non_recursive(int s) {
        vector<bool> visited(n, false); // Create a
           vector to mark visited vertices
        stack<int> stack; // Create a stack
        stack.push(s); // Put the initial vertex into
           the stack
        while (!stack.empty()) { // While the stack is
            not empty
            int u = stack.top();
            stack.pop();
            if (!visited[u]) {  // If the vertex has
               not been visited
                visited[u] = true; // Mark it as
                   visited
                cout << u << " "; // Print the vertex
```

[11] `https://www.geeksforgeeks.org/iterative-depth-first-traversal/`, consultado el 05 de septiembre de 2023

```
31                  for (int v : adjacency_list[u]) { //
                        Traverse vertices adjacent to u
32                      stack.push(v); // Put them into
                            the stack
33                  }
34              }
35          }
36      }
37  };
```

En las líneas del código se describen las funcionalidades de la clase `Graph`. En primer lugar, se incluyen las bibliotecas necesarias en las líneas 1 a 5. La clase `Graph` (Grafo) se define en la línea 7.

Dentro de la clase `Graph`, se declaran dos variables miembro privadas: `n`, que representa el número de vértices en el grafo, y `adjacency_list`, que es una lista de adyacencia bidimensional utilizada para almacenar las aristas.

Por ejemplo, si el grafo tiene 3 vértices, la variable `adjacency_list` podría tener la siguiente estructura:

```
adjacency_list[0] -> [1, 2]   % Vértice 0 tiene aristas a los vértices 1 y 2
adjacency_list[1] -> [0, 3]   % Vértice 1 tiene aristas a los vértices 0 y 3
adjacency_list[2] -> [4]      % Vértice 2 tiene una arista al vértice 4
```

A continuación, se presentan las funciones públicas de la clase `Graph`:

- `Graph(int n)`: El constructor de la clase recibe el número de vértices que tendrá el grafo y crea la lista de adyacencia `adjacency_list` con el tamaño especificado.
- `add_edge(int u, int v)`: La función `add_edge` agrega una arista desde el vértice `u` hasta el vértice `v` en la lista de adyacencia.
- `dfs_non_recursive(int s)`: La función `dfs_non_recursive` implementa el algoritmo de búsqueda en profundidad no recursivo. Toma como parámetro el vértice de inicio `s` y realiza la búsqueda en profundidad a partir de ese vértice.

Las líneas 16 a 18 muestran la implementación de la función `add_edge`, que agrega una arista desde el vértice `u` hasta el vértice `v` en la lista de adyacencia.

Las líneas 20 a 35 muestran la implementación de la función `dfs_non_recursive`, que realiza el recorrido en profundidad de manera no recursiva a partir del vértice especificado.

Resultados del Programa

El siguiente fragmento de código en `main.cpp` muestra cómo crear un grafo de ejemplo y ejecutar el algoritmo de búsqueda en profundidad no recursivo desde un vértice inicial. También se ilustra el grafo de ejemplo junto con las secuencias obtenidas durante la ejecución.

Listing 2: main.cpp

```cpp
#include "Graph.hpp"

int main() {
  // Crear un grafo de ejemplo con 6 vertices y 7 aristas
  Graph g(6);
  g.add_edge(0, 1);
  g.add_edge(0, 2);
  g.add_edge(0, 3);
  g.add_edge(1, 3);
  g.add_edge(2, 4);
  g.add_edge(3, 5);
  g.add_edge(4, 5);
  g.add_edge(5, 0);

  // Ejecutar el algoritmo DFS no recursivo desde el vertice 0
  cout << "DFS desde el vertice 0:" << endl;
  g.dfs_non_recursive(0);
  cout << endl;

  return 0;
}
```

El grafo creado en este ejemplo se visualiza de la siguiente manera:

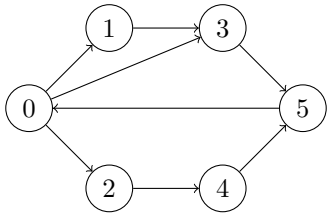

Figura 15

El resultado al ejecutar el algoritmo de búsqueda profunda desde el vértice $V(0)$ es:

```
$ DFS desde el vértice 0:
$ 0 3 5 2 4 1
```

Cambiando el vértice inicial al vértice $V(3)$, el resultado del algoritmo es:

```
$ DFS desde el vértice 3:
$ 3 5 0 2 4 1
```

Conclusiones

Prueba en un Grafo Grande

Con el fin de evaluar el rendimiento del algoritmo de búsqueda en profundidad no recursivo en un contexto más amplio, se realizó una prueba utilizando un grafo considerablemente grande. El grafo se construyó con 100,000 nodos y 200,000 aristas generadas aleatoriamente. Aunque este enfoque no garantiza que todos los nodos estén conectados, proporciona una prueba sólida del algoritmo en una escala significativamente mayor.

Para generar las aristas, se utilizó un enfoque aleatorio, seleccionando dos nodos al azar y asegurándose de evitar autovínculos (self-loops). Este proceso se repitió 200,000 veces para construir el grafo. Si bien esta metodología puede resultar en algunos nodos no conectados, simplificó la tarea de generar un grafo de gran tamaño.

Listing 3: main.cpp

```
#include "Graph.hpp"

void RunBigGraphDemo() {
                // Create a graph with 100 nodes
                int num_nodes = 100000;
                Graph g(num_nodes);

                // Generate random edges
                srand(static_cast<unsigned int>(time(
                    nullptr)));
                int num_edges = 200000; // Adjust the
                    number of edges as needed

                for (int i = 0; i < num_edges; ++i) {
                        int u = rand() % num_nodes;
                        int v = rand() % num_nodes;

                        if (u != v) { // Avoid self-
                            loops
                                g.add_edge(u, v);
                        }
                }
                // Start measuring time
                auto start_time = chrono::
                    high_resolution_clock::now();

                // Perform DFS
                cout << "DFS from vertex 66:" << endl;
                g.dfs_non_recursive(66, false);
```

```
26
27                  // End measuring time
28                  auto end_time = chrono::
                        high_resolution_clock::now();
29                  auto duration = chrono::duration_cast<
                        chrono::milliseconds>(end_time -
                        start_time);
30
31  }
```

A continuación se presentan los tiempos de ejecución obtenidos en cinco ejecuciones:

```
$ DFS desde el vértice 66:
$ Tiempo tomado por DFS no recursivo: 8 milisegundos
$ DFS desde el vértice 66:
$ Tiempo tomado por DFS no recursivo: 8 milisegundos
$ DFS desde el vértice 66:
$ Tiempo tomado por DFS no recursivo: 8 milisegundos
$ DFS desde el vértice 66:
$ Tiempo tomado por DFS no recursivo: 8 milisegundos
$ DFS desde el vértice 66:
$ Tiempo tomado por DFS no recursivo: 8 milisegundos
```

Estos resultados demuestran que, incluso en un grafo de gran tamaño, el algoritmo de búsqueda en profundidad no recursivo es altamente eficiente.

Experimentos Futuros

Para futuros experimentos, se contempla la posibilidad de llevar a cabo una comparativa entre la versión no recursiva del algoritmo de búsqueda en profundidad y su contraparte recursiva. Además, se considera valioso realizar una evaluación comparativa entre el algoritmo de búsqueda en profundidad y el algoritmo de búsqueda en amplitud en términos de tiempo de ejecución. Estos experimentos tienen el potencial de ofrecer una visión más completa de las fortalezas y limitaciones de cada algoritmo en diversos contextos.

Ventajas y Desventajas

Entre las ventajas notables del algoritmo de búsqueda en profundidad se encuentra su capacidad para detectar ciclos en grafos y encontrar todos los caminos posibles entre dos vértices. Sin embargo, es importante destacar que este algoritmo no garantiza encontrar el camino más corto entre dos vértices y puede ser menos eficiente en comparación con algoritmos como BFS en grafos muy grandes.

Posibles Mejoras

Una mejora potencial en el uso del algoritmo de búsqueda en profundidad en otros contextos podría implicar adaptar la estructura de la clase 'Graph'. En lugar de utilizar un arreglo dinámico para guardar los vértices adyacentes, se podría crear una nueva clase, por ejemplo, 'Node', que almacene datos y un arreglo de vértices adyacentes. Luego, la clase 'Graph' se podría ajustar para utilizar la clase 'Node' en lugar de la matriz de adyacencia. Esto permitiría el uso del grafo como una estructura de datos flexible, aunque no necesariamente mejoraría la eficiencia del algoritmo en sí.

Aplicaciones

El algoritmo de búsqueda de profundidad tiene una variedad de aplicaciones más allá de su uso común para el recorrido de grafos. Una de estas aplicaciones interesantes es la creación de laberintos[15][12]. Para lograr esto se tendría que representar una cuadrícula de cualquier tamaño como un grafo, una vez que se tiene el grafo tomamos un punto arbitrario en las orillas como el punto de partida. A partir de ese punto, en vez de agregar todos los nodos adyacentes, se agregaría solamente un nodo al azar, con la condición de que no haya sido visitado antes.

Además, el algoritmo de búsqueda de profundidad se puede utilizar para resolver laberintos[3][13]. Aunque el algoritmo de búsqueda de profundidad no se considera el algoritmo más eficiente para encontrar la salida en laberintos, existen otros algoritmos diseñados específicamente para esta tarea.

Otras aplicaciones importantes del algoritmo de búsqueda de profundidad se pueden encontrar en la página de Wikipedia sobre el algoritmo[7][14].

[12] `https://github.com/MuMashhour/Maze-solver`
[13] `https://codepen.io/Owlree/pen/PPomzo`
[14] `https://en.wikipedia.org/wiki/Depth-first_search#Applications`

Referencias

[1] *ALGORITMO DE BÚSQUEDA: DEPTH FIRST SEARCH*. URL: https://codebreakerscorp.wordpress.com/2011/03/05/algoritmo-de-busqueda-depth-first-search/.

[2] *Algoritmo DFS*. URL: https://personales.unican.es/ruizvc/scheme/grafos.alg/DFS.pdf.

[3] Robert Badea. *Maze Generator + BFS/DFS*. URL: https://codepen.io/Owlree/pen/PPomzo.

[4] *Búsqueda en profundidad*. URL: https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_profundidad.

[5] *Búsqueda en profundidad primero (DFS): preguntas de la entrevista y problemas de práctica*. URL: https://www.techiedelight.com/es/dfs-interview-questions/.

[6] CHOCHY. *Algoritmos de Grafos - Lista de Adyacencia*. URL: https://www.youtube.com/watch?v=KHyGn4cRjN4.

[7] *Depth-first search*. URL: https://en.wikipedia.org/wiki/Depth-first_search#Applications.

[8] *Diferencia entre BFS y DFS*. URL: https://es.gadget-info.com/difference-between-bfs.

[9] Lorenz Equihua. *Mis Apuntes de la Clase de Análisis de Algoritmos, impartida por Miguel Ángel Márquez Hidalgo*. Apunte personal. 2023.

[10] *FIFO y LIFO (contabilidad)*. URL: https://es.wikipedia.org/wiki/FIFO_y_LIFO_(contabilidad).

[11] GeeksforGeeks. *Iterative Depth First Traversal of Graph*. URL: https://www.geeksforgeeks.org/iterative-depth-first-traversal/.

[12] Ing. Viviana Semprún Ing. Daniel Zambrano. *TEMA 3: Tipos Abstractos de Datos: Árboles, Grafos*. URL: https://www.dlsi.ua.es/asignaturas/p2ir/teoria/l03/lessonh.html.

[13] Ing. Viviana Semprún Ing. Daniel Zambrano. *TEORÍA DE GRAFOS Ingeniería de Sistemas*. URL: https://docplayer.es/26258138-Teoria-de-grafos-ingenieria-de-sistemas.html.

[14] Blas Torrecillas Jover. *Grafos*. URL: https://w3.ual.es/~btorreci/tr-grafos.pdf.

[15] MuMashhour. *Maze-solver*. URL: https://github.com/MuMashhour/Maze-solver.

[16] *Teoría de grafos*. URL: https://www.grapheverywhere.com/teoria-de-grafos/.