Informe de Laboratorio 07

Tema: Algoritmos para Grafos: Accesibilidad

Nota

Estudiante	Escuela	Asignatura
Reyser Julio Zapata Butron rzapata@unsa.edu.pe	Escuela Profesional de Ingeniería de Sistemas	Análisis Y Diseño de Algoritmos Semestre: IV Código: 1702231

Laboratorio	Tema	Duración
07	Algoritmos para Grafos: Accesibilidad	02 horas

Semestre académico	Fecha de inicio	Fecha de entrega
2024 - B	26 noviembre 2024	26 noviembre 2024

1. Código base

El siguiente código presentado, es la base para la realización de los ejerción propuestos, añadiendo los métodos requeridos.

```
#include <iostream>
#include <vector>

using namespace std;

typedef int vertex;
typedef struct
{
  int V;
  vector<vector<int>> adj;
} Graph;

static int visited[1000];
static void reachR(Graph &G, vertex v);

bool GRAPHreach(Graph &G, vertex s, vertex t)
{
  for (vertex v = 0; v < G.V; ++v)
    visited[v] = 0;

  reachR(G, s);

```

```
  if (visited[t] == 1)
    return true;
  else
    return false;
}

static void reachR(Graph &G, vertex v)
{
  visited[v] = 1;

  for (vertex w = 0; w < G.V; ++w)
  {
    if (G.adj[v][w] == 1 && visited[w] == 0)
      reachR(G, w);
  }
}

int main()
{
  return 0;
}
```

Listing 1: base.cpp

2. Ejercicios Propuestos

2.1. Permutación de vecinos. Repite los ejemplos C y D anteriores suponiendo que el grafo G está representado por las listas de adyacencia:

0	2 3 4
1	
2	1 4
3	4 5
4	1 5
5	1

```
#include <iostream>
#include <vector>

using namespace std;

typedef int vertex;
typedef struct
{
  int V;
  vector<vector<int>> adj;
} Graph;

static int visited[1000];
static void reachR(Graph &G, vertex v);

bool GRAPHreach(Graph &G, vertex s, vertex t)
{
  for (vertex v = 0; v < G.V; ++v)
    visited[v] = 0;

  reachR(G, s);

```

```
  if (visited[t] == 1)
    return true;
  else
    return false;
}

static void reachR(Graph &G, vertex v)
{
  visited[v] = 1;

  for (vertex w = 0; w < G.V; ++w)
  {
    if (G.adj[v][w] == 1 && visited[w] == 0)
      reachR(G, w);
  }
}

int main()
{
  Graph G;
  G.V = 6;
  G.adj = vector<vector<int>>(G.V, vector<int>(G.V, 0));

  G.adj[0][2] = 1;
  G.adj[0][3] = 1;
  G.adj[0][4] = 1;
  G.adj[2][1] = 1;
  G.adj[2][4] = 1;
  G.adj[3][4] = 1;
  G.adj[3][5] = 1;
  G.adj[4][1] = 1;
  G.adj[4][5] = 1;
  G.adj[5][1] = 1;

  if (GRAPHreach(G, 0, 5))
  {
    cout << "\n[+] El vertice 5 esta al alcance del vertice 0." << endl;
  }
  else
  {
    cout << "\n[-] El vertice 5 NO esta al alcance del vertice 0." << endl;
  }

  if (GRAPHreach(G, 2, 3))
  {
    cout << "\n[+] El vertice 3 esta al alcance del vertice 2." << endl;
  }
  else
  {
    cout << "\n[-] El vertice 3 NO esta al alcance del vertice 2." << endl;
  }

  cout << endl;
  return 0;
}
```

Listing 2: exercise1.cpp

Ejecución del código

```
> .\exercise1.exe
[+] El vertice 5 esta al alcance del vertice 0.
[-] El vertice 3 NO esta al alcance del vertice 2.
```

Explicación del Código

El código implementa una búsqueda en un grafo **dirigido** utilizando el algoritmo de búsqueda en profundidad (DFS). La estructura del grafo está representada por una matriz de adyacencia, donde un valor de 1 indica la existencia de un arco dirigido entre dos vértices. La función `GRAPHreach` recibe dos vértices s y t, y determina si hay un camino dirigido entre ellos.

Funcionamiento

1. **Inicialización:** La función `GRAPHreach` inicializa el vector `visited[]` para marcar los vértices visitados durante la búsqueda. Luego, llama a la función recursiva `reachR` comenzando desde el vértice s.
2. **Búsqueda recursiva (DFS):** La función `reachR` recorre los vértices adyacentes al vértice v y, si un vértice adyacente no ha sido visitado, realiza una llamada recursiva para continuar la búsqueda.
3. **Determinación del camino:** Si durante la búsqueda el vértice t es alcanzado, la función `GRAPHreach` retorna `true`, indicando que t está al alcance de s. De lo contrario, retorna `false`.

Complejidad Temporal

La complejidad temporal del algoritmo es $O(V + E)$, donde V es el número de vértices y E es el número de arcos. Esto se debe a que cada vértice es visitado una sola vez, y cada arco es examinado una vez.

2.2. Versión ansiosa de la función. Escriba una variante de la función GRAPHreach() que se detenga inmediatamente (y devuelva true) al descubrir que t está al alcance de s. (El código de reachR() para esta variante es más complicado que el de la versión ansiosa discutida anteriormente). Para hacer el ejercicio más interesante, imprime un camino de s a t antes de devolver true.

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

using namespace std;

typedef int vertex;
typedef struct
{
  int V;
  vector<vector<int>> adj;
} Graph;

static int visited[1000];
static bool reachR(Graph &G, vertex v, vertex t);

static vector<vertex> path;
```

```
18
19 bool GRAPHreach(Graph &G, vertex s, vertex t)
20 {
21   for (vertex v = 0; v < G.V; ++v)
22     visited[v] = 0;
23
24   path.clear();
25
26   if (reachR(G, s, t))
27   {
28     cout << "[+] Camino encontrado de " << s << " a " << t << ": ";
29     for (vertex v : path)
30       cout << v << " ";
31     cout << endl;
32     return true;
33   }
34   else
35   {
36     cout << "\n[-] No hay camino de " << s << " a " << t << "." << endl;
37     return false;
38   }
39 }
40
41 static bool reachR(Graph &G, vertex v, vertex t)
42 {
43   visited[v] = 1;
44
45   path.push_back(v);
46
47   if (v == t)
48     return true;
49
50   for (vertex w = 0; w < G.V; ++w)
51   {
52     if (G.adj[v][w] == 1 && visited[w] == 0)
53     {
54       if (reachR(G, w, t))
55         return true;
56     }
57   }
58
59   path.pop_back();
60   return false;
61 }
62
63 int main()
64 {
65   Graph G;
66   G.V = 6;
67   G.adj = vector<vector<int>>(G.V, vector<int>(G.V, 0));
68
69   G.adj[0][2] = 1;
70   G.adj[0][3] = 1;
71   G.adj[0][4] = 1;
72   G.adj[2][1] = 1;
73   G.adj[2][4] = 1;
74   G.adj[3][4] = 1;
75   G.adj[3][5] = 1;
76   G.adj[4][1] = 1;
77   G.adj[4][5] = 1;
78   G.adj[5][1] = 1;
79
80   if (GRAPHreach(G, 0, 5))
81   {
82     cout << "\n[+] El vertice 5 esta al alcance del vertice 0." << endl;
```

```
  }
  else
  {
    cout << "\n[-] El vertice 5 NO esta al alcance del vertice 0." << endl;
  }

  if (GRAPHreach(G, 2, 3))
  {
    cout << "\n[+] El vertice 3 esta al alcance del vertice 2." << endl;
  }
  else
  {
    cout << "\n[-] El vertice 3 NO esta al alcance del vertice 2." << endl;
  }

  cout << endl;
  return 0;
}
```

Listing 3: exercise2.cpp

Ejecución del código

```
> .\exercise2.exe
[+] Camino encontrado de 0 a 5: 0 2 4 5

[+] El vertice 5 esta al alcance del vertice 0.

[-] No hay camino de 2 a 3.

[-] El vertice 3 NO esta al alcance del vertice 2.
```

Explicación de los Cambios en el Código

En este ejercicio, la función `GRAPHreach()` fue modificada para detenerse inmediatamente después de encontrar el vértice t y para imprimir el camino de s a t. A continuación se explican los cambios con fragmentos de código.

Función `GRAPHreach()`

La principal modificación en esta función es que ahora se imprime el camino de s a t si se encuentra un camino, y la función se detiene de inmediato al encontrar t.

```
bool GRAPHreach(Graph &G, vertex s, vertex t) {
    for (vertex v = 0; v < G.V; ++v)
        visited[v] = 0;
    path.clear();  // Se limpia el vector de camino

    if (reachR(G, s, t)) {
        cout << "[+] Camino encontrado de " << s << " a " << t << ": ";
        for (vertex v : path)  // Imprime el camino
            cout << v << " ";
        cout << endl;
        return true;
    } else {
```

```
            cout << "\n[-] No hay camino de " << s << " a " << t << "." << endl;
            return false;
        }
    }
```

Aquí, si la función `reachR()` retorna `true`, se imprime el camino almacenado en el vector `path[]`.

Función `reachR()`

La función `reachR()` fue modificada para que, en cuanto se encuentre el vértice t, se devuelva `true` inmediatamente, y el vértice actual se agregue a `path[]`. Además, si el vértice t no es alcanzado por el camino actual, se elimina el vértice de `path[]`.

```
static bool reachR(Graph &G, vertex v, vertex t) {
    visited[v] = 1;        // Marca el vértice como visitado
    path.push_back(v);     // Agrega el vértice al camino

    if (v == t)            // Si encontramos el vértice t
        return true;

    for (vertex w = 0; w < G.V; ++w) {
        if (G.adj[v][w] == 1 && visited[w] == 0) {
            if (reachR(G, w, t))  // Llamada recursiva
                return true;
        }
    }

    path.pop_back();  // Si no encontramos t, deshacemos el último paso
    return false;
}
```

Las modificaciones clave son:

- `path.push_back(v)`: Añade el vértice v al camino.
- `if (v == t)`: Si el vértice actual es t, se retorna `true` y el camino es impreso.
- `path.pop_back()`: Si no se encuentra un camino hacia t, el vértice actual se elimina de `path[]`.

3. Repositorio de Github

- Repositorio de Github donde se encuentra el actual laboratorio
  `https://github.com/ReyserLynnn/ada-lab-b-24b/tree/main/laboratorio07/src`
- Repositorio de Github donde se encuentran los laboratorios del curso
  `https://github.com/ReyserLynnn/ada-lab-b-24b.git`

4. Conclusión

En este laboratorio, aprendí a trabajar con algoritmos de búsqueda en grafos, usando recursión y una versión más eficiente que se detiene al encontrar el destino. También entendí cómo seguir el camino y mostrarlo, evitando seguir explorando una vez que ya encontré lo que buscaba.