Fundamentos de Estructuras de Datos y Algoritmos **Semestre 2023-1**

Laboratorio 15: 02 de Junio

Profesores : Jérémy Barbay, José Fuentes *Ayudante: Vicente Lermanda*

Objetivos

- Familiarizarse con las implementaciones de grafos.
- Recorrer grafos y encontrar componentes fuertemente conexas.

Representación de Grafos

Matriz de adyacencia

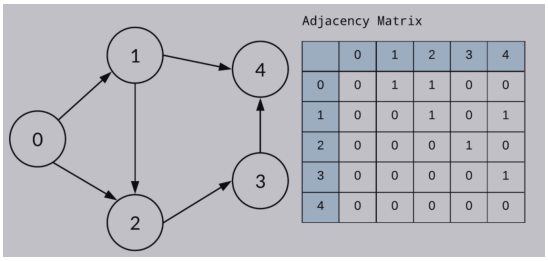

Figure 15.1: Representación de un grafo dirigido como matriz de adyacencia.

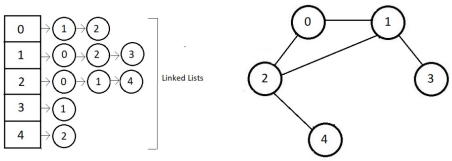

Figure 15.2: Representación de un grafo no dirigido como lista de adyacencia.

Lista de adyacencia

Operaciones

Solo con las representaciones ya podemos hacer algunas consultas!

- Conocer la *vecindad* de un vértice v.

- Saber si dos vértices están conectados.
- Listar todas las aristas (u, v) de un grafo. Cuál es la complejidad de estas en cada representación?
  - Recordar que acceder a una posición en un arreglo o matriz tiene complejidad $O(1)$.
  - En una lista esto tiene complejidad $O(n)$.

```
int matriz_adyacencia[NODOS][NODOS];

// caso grafo no dirigido
void agregar_arista(int nodo1, int nodo2){
matriz_adyacencia[nodo1][nodo2] = 1;
matriz_adyacencia[nodo2][nodo1] = 1;
}

bool nodos_conectados(int nodo1, int nodo2){
return matriz_adyacencia[nodo1][nodo2];
}

void listar_vecinos(int nodo){
for(int i = 0; i < NODOS; i++)
if(matriz_adyacencia[nodo][i] == 1)
cout << i << endl;
}

int main(){


return 0;
}
```

```
#include <vector>
#include <algorithm>

using namespace std;

vector<vector<int>> adj_list(NODOS,vector<int>());

// Grafo dirigido
void agregar_arista(int nodo1, int nodo2){
    adj_list[nodo1].push_back(nodo2);
}

bool nodos_conectados(int nodo1, int nodo2){
for(int nodo_vecino : adj_list[nodo1])
if(nodo_vecino == nodo2)
return true;
return false;
}

void listar_vecinos(int nodo){
```

```
for(int nodo_vecino : adj_list[nodo])
cout<<nodo_vecino<<endl;
}
int main(){
vector<pair<int, int>> lista_aristas;
int a, b;
for(int i = 0; i < cantidad_aristas; i++){
cin>>a>>b;
lista_aristas.push_back({a,b});
}
// imprimir primer arista
cout<<lista_aristas[0].first<<" "<<lista_aristas[0].second<<endl;

// Se ordena en base al elemento first
sort(lista_aristas.begin(),lista_aristas.end());
return 0;
}
```

Material útil [https://github.com/stevenhalim/cpbook-code/https://github.com/stevenhalim/cpbook-code/]

Recorrido de Grafos

BFS

```
Algorithm BFS(G, start):
  empty queue Q
  empty list DIST to keep track of the distance of the nodes

  DIST[start] = 0;

  Enqueue start into Q
  Add start into V

  While Q is not empty:
    node = Dequeue from Q
    Visit the node

    For each neighbor of node:
      If neighbor is INF in DIST:
        Enqueue neighbor into Q
        DIST[neighbor] = DIST[node] + 1;
```

DFS

Iterativo

```
Algorithm IterativeDFS(G, v):
```

```
  Initialize an empty stack S
  Push v onto S

  While S is not empty:
    Pop the top node t from S
    If t is not visited:
      Mark t as visited
      For each neighbor n of t in G:
        Push n onto S
```

Recursivo

```
Algorithm RecursiveDFS(G, v):
  Mark v as visited
  For each neighbor n of v in G:
    If n is not visited:
      Call RecursiveDFS(G, n)
```

Componentes Fuertemente Conexas

Componente Conexa

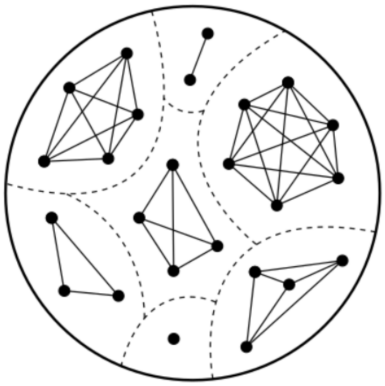

Figure 15.3: Componentes conexas en un grafo

Encontrar SCC

Algoritmo de Kosaraju

```
Algorithm Kosaraju(G):
  Initialize an empty stack S
  Mark all vertices as not visited

  For each vertex v in G:
```

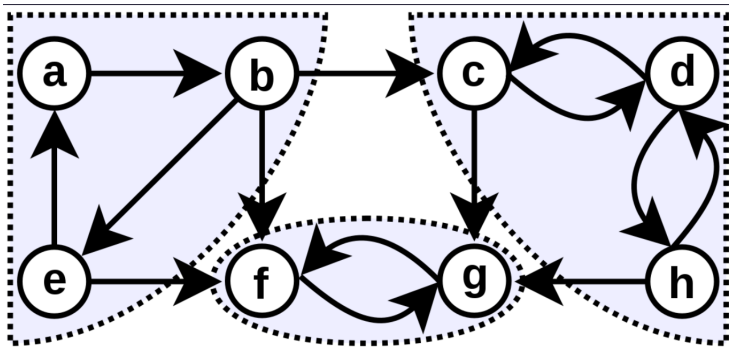

Figure 15.4: Ejemplo de componentes fuertemente conexas

```
  If v is not visited:
    Run DFS on v to fill the stack S

Create a transposed graph G'

Mark all vertices as not visited for the second DFS run

While S is not empty:
  Get the top vertex v from S
  Pop v from S

  If v is not visited in the transposed graph:
    Run DFS on v in the transposed graph
    Each DFS run will give a strongly connected component

End
```