Recorridos en grafos
DFS

Algoritmos y Estructuras de Datos III

2do cuat 2022

Objetivo: recorrer sistemáticamente los vértices de un grafo sin repetir

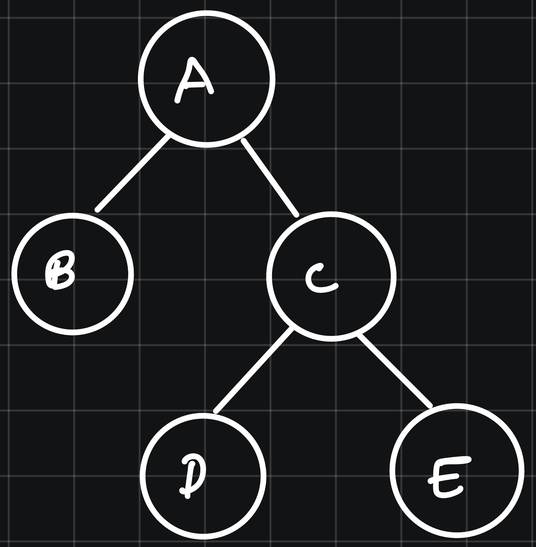
- Para buscar elementos
- Para procesar elementos
- Para determinar características del grafo en función de los árboles resultantes

2 Algoritmos

- Depth First Search
(recorre en profundidad)

- Breadth First Search
(recorre en anchura)

DFS



Para terminar con A
debo terminar con B y C,
Para terminar con C
debo terminar con D y E...

- La exploración de un vértice termina cuando todos sus hijos fueron explorados

- El recorrido toma el siguiente vértice a recorrer de una pila (ie. el último que fuera agregado)

- Si comenzamos el DFS en un vértice $v \in G$, los vértices que recorramos serán los alcanzables por v

El código se puede escribir de manera recursiva

```cpp
//RECURSIVO
void dfs(vector<vector<int>>& ady, int source, vector<bool>& visitado) {
    visitado[source] = true;
    for (int u : ady[source]) {
        if (!visitado[u]) {
            dfs(ady, u, visitado);
        }
    }
}
```

También se puede escribir de forma iterativa, haciendo explícita la pila del stack de llamados

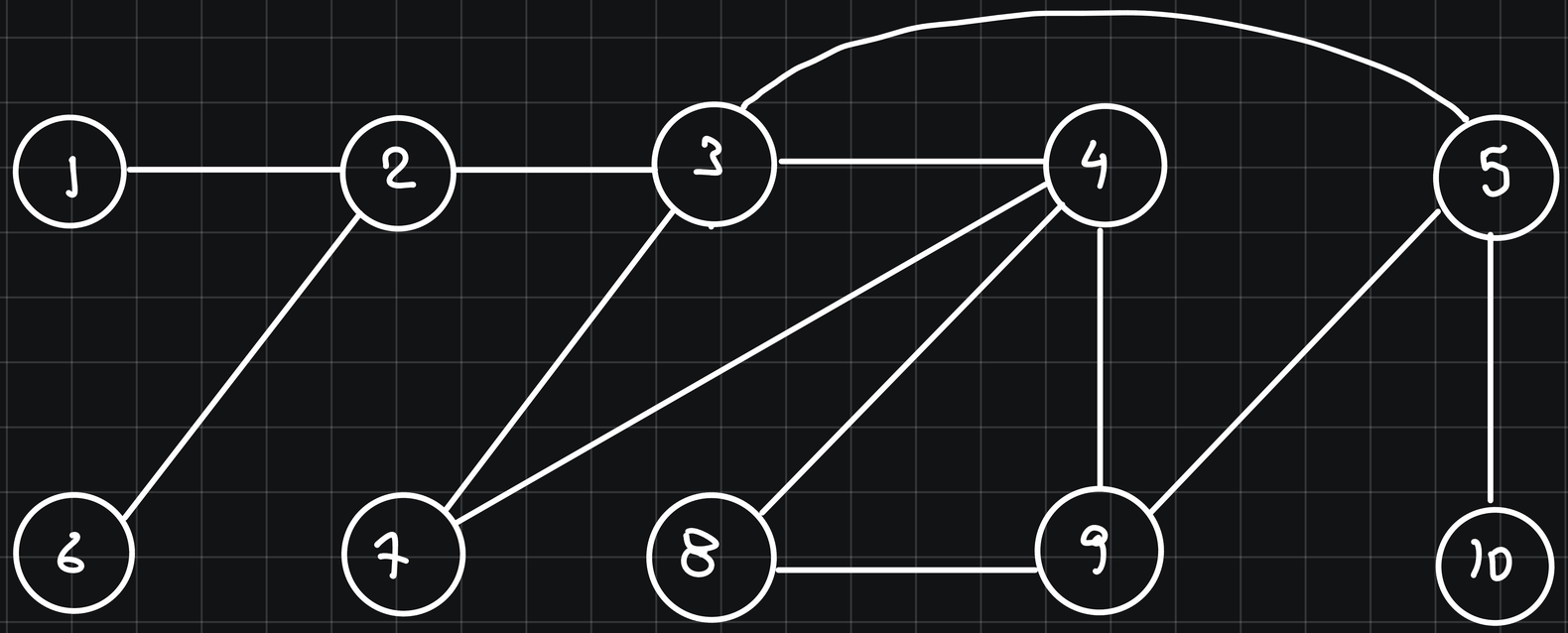
```cpp
//ITERATIVO
void dfs_iter(const vector<vector<int>>& ady, const int source, vector<bool>& visitado){
    stack<int> pila; pila.push(source);
    while (!pila.empty()) {
        int u = pila.top();
        pila.pop();
        if (!visitado[u]) {
            visitado[u] = true;
        }
        for (int w : ady[u]) {
            if (!visitado[w]) {
                pila.push(w);
            }
        }
    }
}
```

Podemos usar un vector de parents en vez del de visitados para representar el árbol obtenido

```cpp
//RECURSIVO CON PARENT
void dfs_parent(vector<vector<int>>& ady, int source, vector<int>& parents) {
    for (int u : ady[source]) {
        if (parents[u]==-1) {
            parents[u]=source;
            dfs_parent(ady, u, parents);
        }
    }
}
```

(Al llamarlo, se debe establecer en el vector de parents que el parent del vértice que vamos a poner como raíz del árbol DFS es él mismo)

Ejemplo

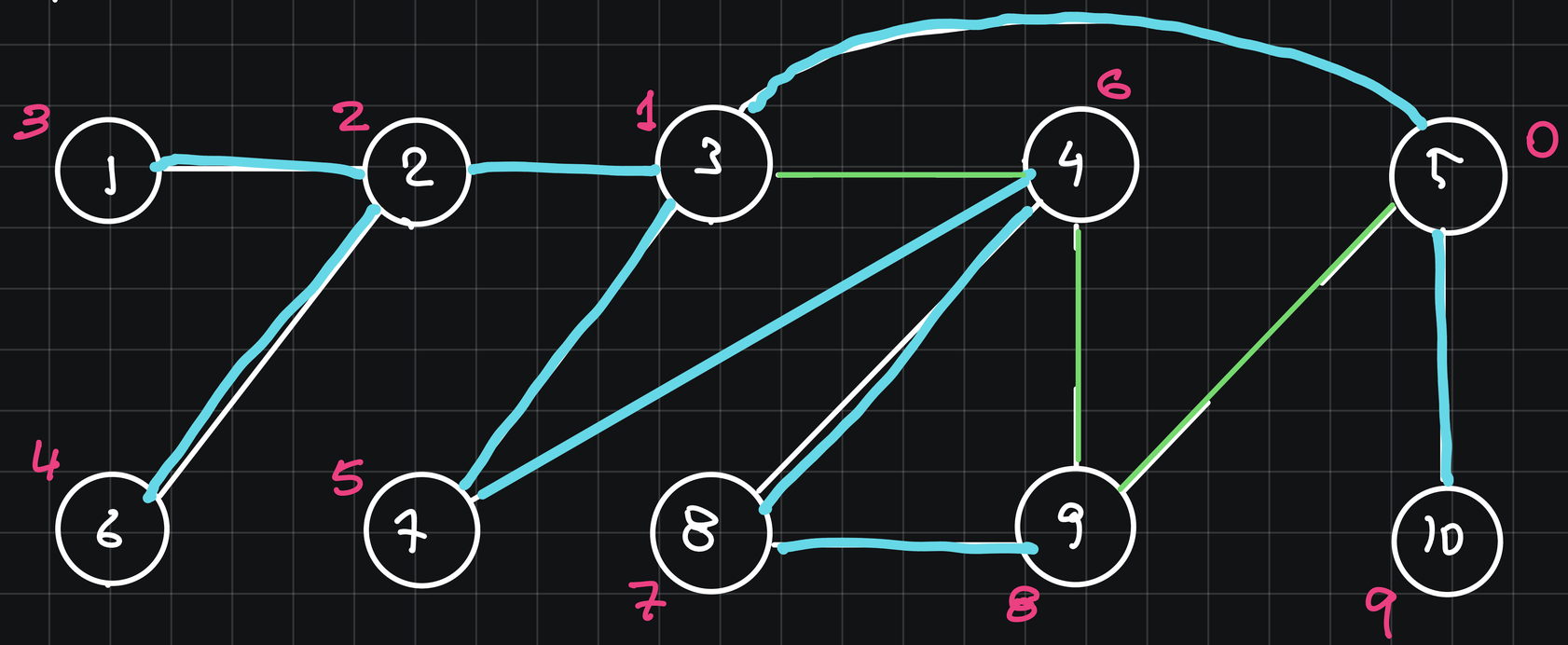


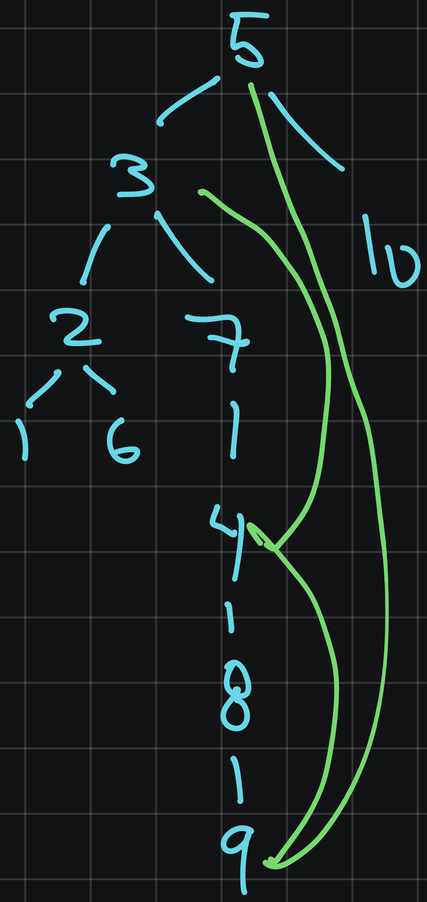
1 → 2
2 → 1 6 3 11
3 → 2 7 5 4
4 → 7 8 9 3
5 → 3 9 10

6 → 2
7 → 3 4
8 → 4 9
9 → 8 4 5
10 → 5
11 → 2

Supongamos DFS enraizado en 5.

Ejemplo





Tree edges

Back Edges

Órden de exploración

En grafos no conexos
Podemos correr DFS múltiples veces
y determinar las componentes conexas

```
//Si quisiera detectar otras componentes conexas
for (int i = 0;i<n+1;i++) {
    if(padres[i]==-1) {
        padres[i]=i;
        dfs_parent(grafo, i, padres);
    }
}
```

Se puede llevar registro del recorrido usando 3 colores para los vértices: Blanco, Gris y Negro.

- Blanco es el color de los vértices aún explorar
- gris es el color de los vértices aún en exploración
- negro es el color de los vértices ya explorados.

Se agrega, además del color, un timer: Este valor indica el momento en el que un vértice pasó de Blanco a Gris

```cpp
//RECURSIVO CON TIMER Y COLORES
void dfs_timer(const vector<vector<int>>& ady,const int source,
               int & current_time, vector<int>& time_in, vector<int>& color) {
    color[source] = 1;
    time_in[source] = current_time++;
    for (int u : ady[source]) {
        if (color[u]==0) {
            dfs_timer(ady, u, current_time, time_in, color);
        }
    }
    color[source]=2;
}
```

(con 0=Blanco, 1=Gris, 2=negro)

Invariantes:

LRCS: Un vértice u es ancestro de un vértice v sii al momento de cambiar el color de u de Blanco a Gris se puede alcanzar v desde u por medio de un camino de vértices blancos.

22.9 "white path theorem"

Alternativamente, propiedades más fáciles:

Obs 1 - Al momento de cambiar un vértice v de Blanco a Gris, todo vértice $w \neq v$ es ancestro de v sii es de color Gris.

Obs 2 - Al momento de v de pasar de gris a negro, todos los descendientes de v son vértices de color negro (No vale la vuelta!)

Tipos de aristas:

- Tree Edges (aristas del árbol)
- Back Edges (aristas que conectan ancestros con descendientes)

Propiedad:

En un grafo no dirigido G, toda arista de G es Tree edge o Back edge respecto al árbol DFS de G

(En grafos dirigidos puede no ser cierto, ya que no da igual cuál de los vértices se explora primero. En esos casos, además de las Back Edges, hay Forward Edges y Cross Edges)

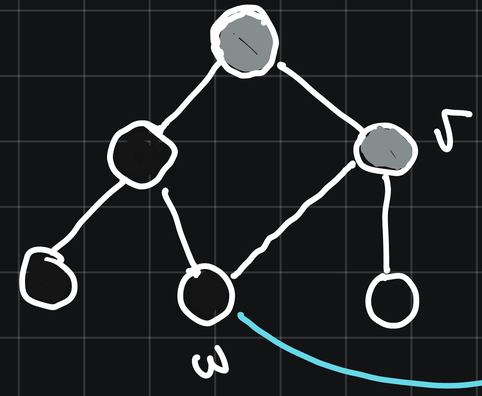
Para ver esta propiedad, podemos pensar en los colores posibles del vértice del otro lado de la arista al momento de explorarla: Supongamos que estoy explorando el vértice v y estoy por considerar, de entre sus vecinos, a w. Consideremos el color de w.

- Si es blanco, (v,w) es una tree edge y w será descendiente de v.
- Si es gris, w es ancestro de v (obs. 1), y la arista (v,w) es back edge.
- Si es negro, es un descendiente (y también es una back-edge)

¿Por qué? la vuelta de la obs. 2 no vale...

- Si es negro, es un descendiente (y también es una back-edge)

Si estoy explorando el vecindario de v y encuentro el vértice w negro, pero este no es descendiente de v, debería ser porque w fue explorado en una rama del DFS recorrida previamente, pero si ese fuera el caso, la propia arista (v,w) hubiera hecho que v perteneciera a esa rama (lo cual no es posible)



Si ya pasé por w, ¿por qué no se habría agregado a v como hijo?

Nota: El color del vértice también identifica si la back-edge que encontré une al vértice con un ancestro o con su antecesor. S. el grafo ya está pintado, puedo usar el timer.

Ejercicio:

Un puente es una arista de G tal que si es removida la cantidad de componentes conexas de G se incrementa.

Queremos detectar los puentes de un grafo G dado

(se puede hacer también con Puntos de articulación, es parecido)

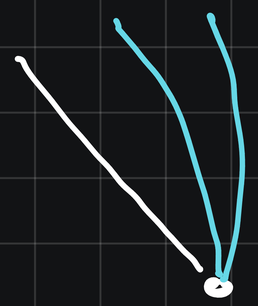
Obs 1: Todo puente necesariamente es un Tree edge

(Si remuevo todos los back edges, la cantidad de componentes conexas no cambia)

¿Qué característica tiene una tree edge e que es puente? No tiene una back-edge que la "cubra", ie. que sirva de camino alternativo si quitara a e.

Función recursiva para contar cantidad de back edges que cubren a una tree edge. $(p(v), v)$

Caso base: v hoja



$dp(v)$ = back edges que salen de v

caso recursivo.

$dp(v)$ = back edges que salen de v

$+$ (back edges que cubren a los hijos de v − back edges que terminan en v)

Escrito en Código:

(Obs: Hacer esto en cada componente conexa)

```cpp
//DETECCIÓN DE PUENTES
int dp(const vector<vector<int>>& ady, int source,
    const vector<int>& time_in, const vector<int>& parents, vector<bool>& puente_con_parent) {
    int cantidad = 0;
    for (int u : ady[source]) {
        if (parents[u]==source) {
            cantidad +=dp(ady, u, time_in, parents, puente_con_parent);
        } else {
            if (time_in[source]>time_in[u] and parents[source]!=u) {
                cantidad++;
            }
            if (time_in[source]<time_in[u]) {
                cantidad--;
            }
        }
    }
    if (cantidad==0 and parents[source]!=source) {
        puente_con_parent[source] = true;
    }
    return cantidad;
}
```

No es la única manera! de hecho, el algoritmo propuesto en la ayuda de la práctica es distinto.