

Teoría de Grafos

MSc. (c) Jhosimar George Arias Figueroa

jariasf03@gmail.com

State University of Campinas
Institute of Computing

Contenido

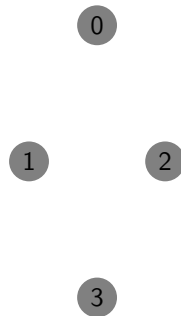
- 1 Introducción
- 2 Representación
 - Matriz de adyacencia
 - Lista de adyacencia
- 3 Recorrido de grafos
 - Breadth-first Search
 - Depth-first Search

Qué es un grafo?

Qué es un grafo?

- Vértices

- Intersecciones de caminos
- Computadores
- Pisos en una casa
- Objetos



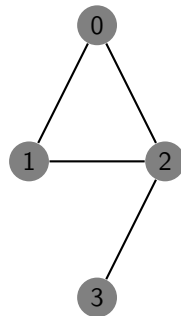
Qué es un grafo?

- Vértices

- Intersecciones de caminos
- Computadores
- Pisos en una casa
- Objetos

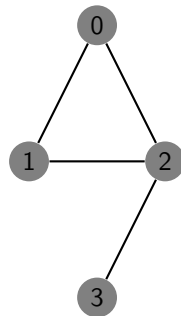
- Aristas

- Caminos
- Cables Ethernet
- Gradas o elevadores
- Relaciones entre objetos



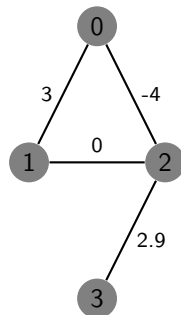
Tipos de aristas

- Sin peso



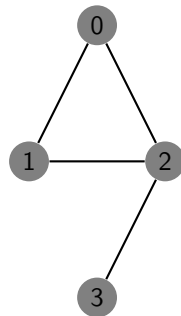
Tipos de aristas

- Sin peso o Con peso



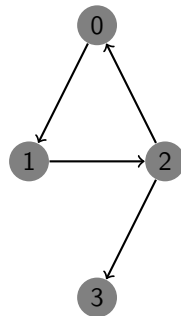
Tipos de aristas

- Sin peso o Con peso
- No dirigidos

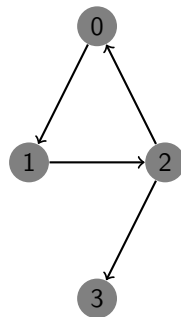


Tipos de aristas

- Sin peso o Con peso
- No dirigidos o Dirigidos

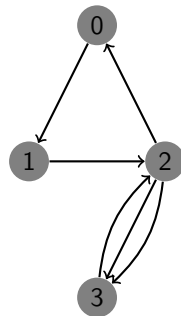


Multigrafos



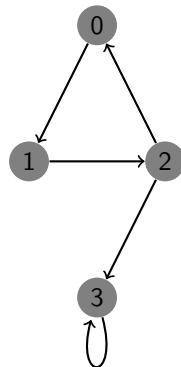
Multigrafos

- Múltiples aristas



Multigrafos

- Múltiples aristas
- Self-loops



Contenido

1 Introducción

2 Representación

- Matriz de adyacencia
- Lista de adyacencia

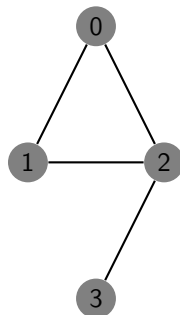
3 Recorrido de grafos

- Breadth-first Search
- Depth-first Search

Matriz de adyacencia

	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0

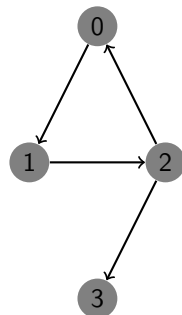
```
int adj[4][4];
adj[0][1] = adj[1][0] = 1;
adj[0][2] = adj[2][0] = 1;
adj[1][2] = adj[2][1] = 1;
adj[2][3] = adj[3][2] = 1;
```



Matriz de adyacencia (dirigido)

	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0

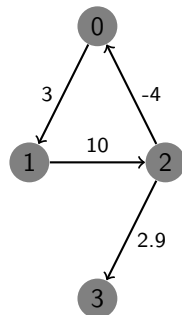
```
int adj[4][4];  
adj[0][1] = 1;  
adj[1][2] = 1;  
adj[2][0] = 1;  
adj[2][3] = 1;
```



Matriz de adyacencia (dirigido con pesos)

	0	1	2	3
0	0	3	0	0
1	0	0	10	0
2	-4	0	0	2.9
3	0	0	0	0

```
float adj[4][4];  
adj[0][1] = 3;  
adj[1][2] = 10;  
adj[2][0] = -4;  
adj[2][3] = 2.9;
```



Contenido

1 Introducción

2 Representación

- Matriz de adyacencia
- Lista de adyacencia

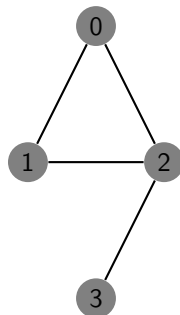
3 Recorrido de grafos

- Breadth-first Search
- Depth-first Search

Lista de adyacencia

```
0: 1, 2
1: 0, 2
2: 0, 1, 3
3: 2
```

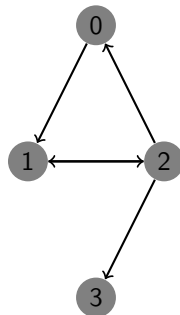
```
vector<int> adj[4];
adj[0].push_back(1);
adj[0].push_back(2);
adj[1].push_back(0);
adj[1].push_back(2);
adj[2].push_back(0);
adj[2].push_back(1);
adj[2].push_back(2);
adj[3].push_back(2);
```



Lista de adyacencia (dirigido)

```
0: 1
1: 2
2: 0, 1, 3
3:
```

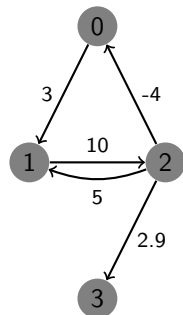
```
vector<int> adj[4];
adj[0].push_back(1);
adj[1].push_back(2);
adj[2].push_back(0);
adj[2].push_back(1);
adj[2].push_back(3);
```



Lista de adyacencia (dirigido con pesos)

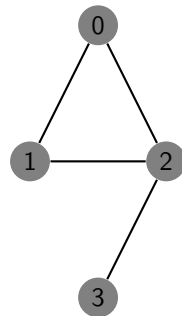
0: (1,3)
1: (2,10)
2: (0,-4), (1,5), (3,2.9)
3:

```
vector< pair<int,float> > adj[4];  
adj[0].push_back(make_pair(1,3));  
adj[1].push_back(make_pair(2,10));  
adj[2].push_back(make_pair(0,-4));  
adj[2].push_back(make_pair(1,5));  
adj[2].push_back(make_pair(3,2.9));
```



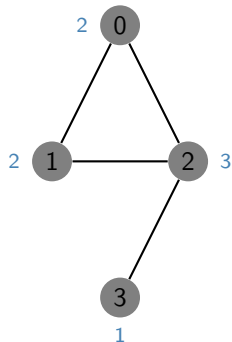
Propiedades de vértices (grafos no dirigidos)

- Grado de un vértice
 - Número de vértices adyacentes



Propiedades de vértices (grafos no dirigidos)

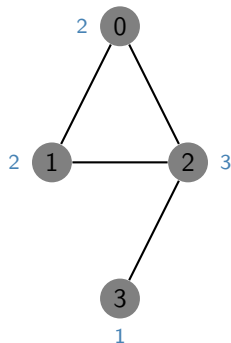
- Grado de un vértice
 - Número de vértices adyacentes



Propiedades de Vértices (grafos no dirigidos)

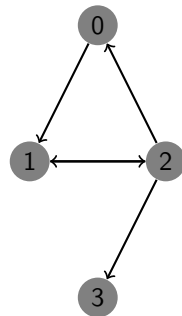
```
0: 1, 2
1: 0, 2
2: 0, 1, 3
3: 2
```

```
adj[0].size() // 2
adj[1].size() // 2
adj[2].size() // 3
adj[3].size() // 1
```



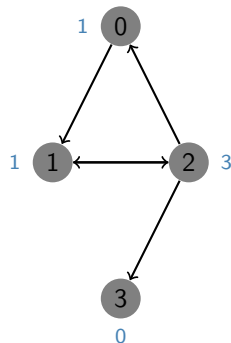
Propiedades de Vértices (grafos dirigidos)

- Grado de salida de un vértice (outdegree)
 - Número de aristas salientes



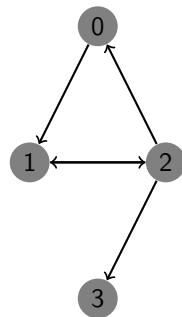
Propiedades de Vértices (grafos dirigidos)

- Grado de salida de un vértice (outdegree)
 - Número de aristas salientes



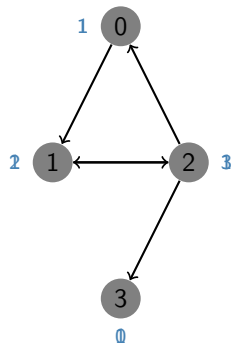
Propiedades de Vértices (grafos dirigidos)

- Grado de salida de un vértice (outdegree)
 - Número de aristas salientes
- Grado de entrada de un vértice (indegree)
 - Número de aristas entrantes



Propiedades de Vértices (grafos dirigidos)

- Grado de salida de un vértice (outdegree)
 - Número de aristas salientes
- Grado de entrada de un vértice (indegree)
 - Número de aristas entrantes



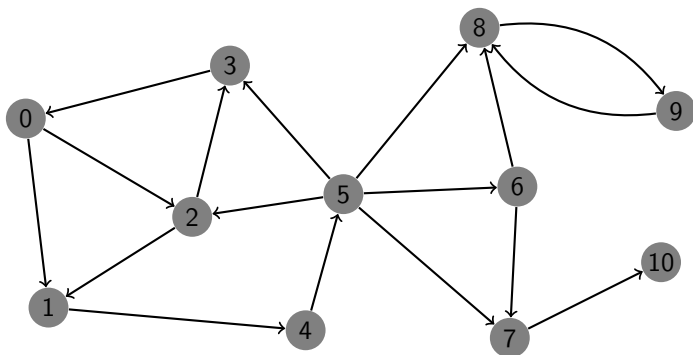
Contenido

- 1 Introducción
- 2 Representación
 - Matriz de adyacencia
 - Lista de adyacencia
- 3 Recorrido de grafos
 - Breadth-first Search
 - Depth-first Search

Breadth-first Search (BFS)

- El algoritmo de búsqueda por anchura en un grafo(BFS) es un algoritmo muy útil en concursos de programación.
- Permite realizar el recorrido en un grafo.
- Trabaja en grafos dirigidos y no dirigidos.
- Las aristas deben tener el mismo valor.
- Hace uso de la estructura de datos **Cola**.

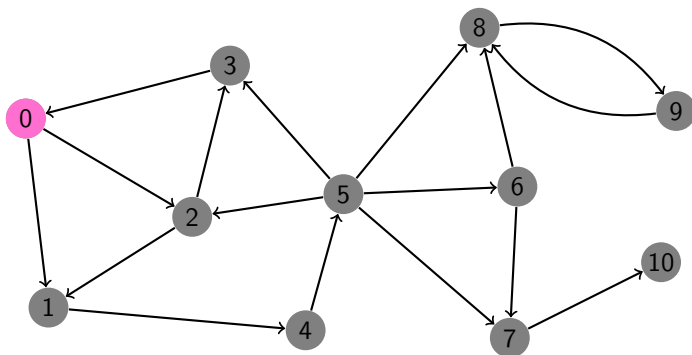
Breadth-first search



Queue:

	0	1	2	3	4	5	6	7	8	9	10
marked	0	0	0	0	0	0	0	0	0	0	0

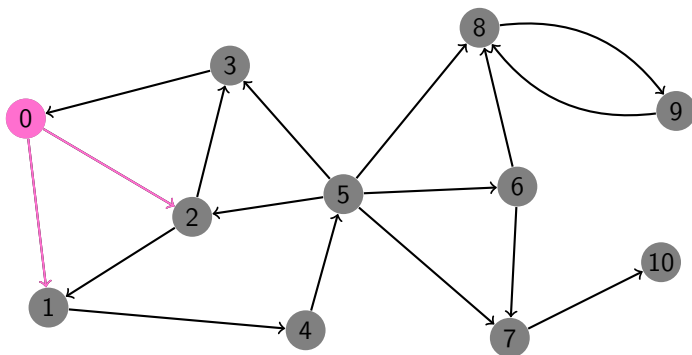
Breadth-first search



Queue: 0

	0	1	2	3	4	5	6	7	8	9	10
marked	1	0	0	0	0	0	0	0	0	0	0

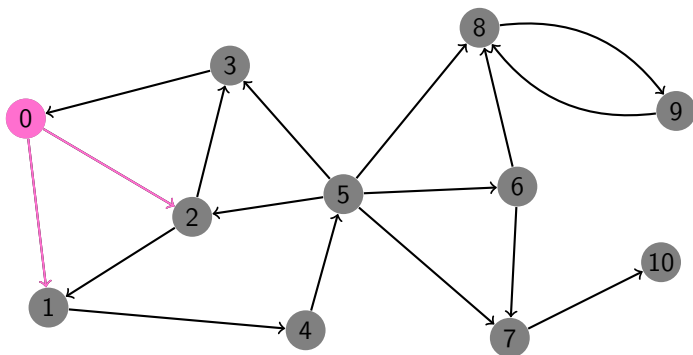
Breadth-first search



Queue: 0

	0	1	2	3	4	5	6	7	8	9	10
marked	1	0	0	0	0	0	0	0	0	0	0

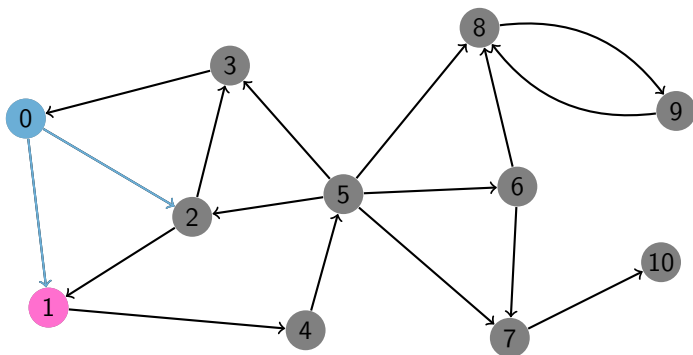
Breadth-first search



Queue: 0 1 2

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	0	0	0	0	0	0	0

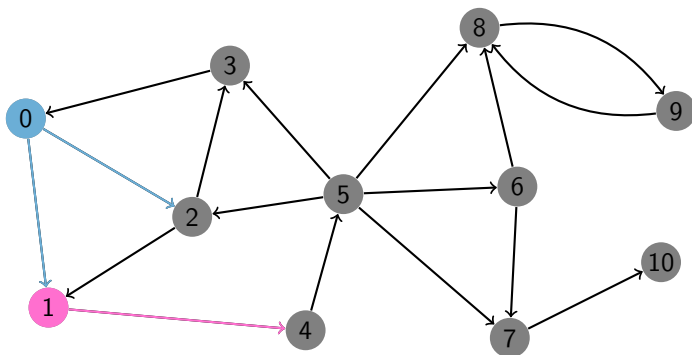
Breadth-first search



Queue: 1 2

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	0	0	0	0	0	0	0

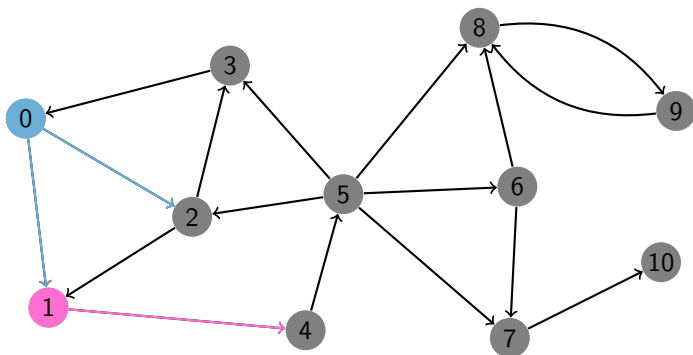
Breadth-first search



Queue: 1 2

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	0	0	0	0	0	0	0

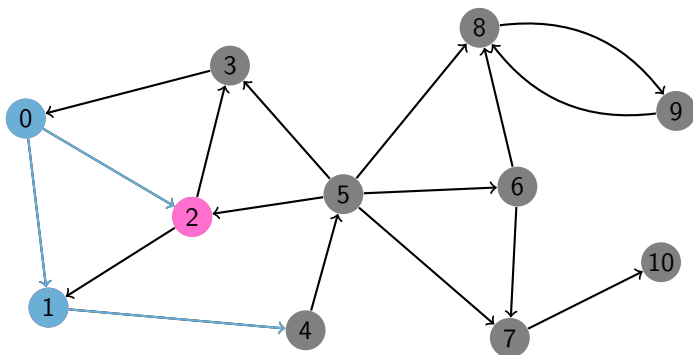
Breadth-first search



Queue: 1 2 4

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	1	0	0	0	0	0	0

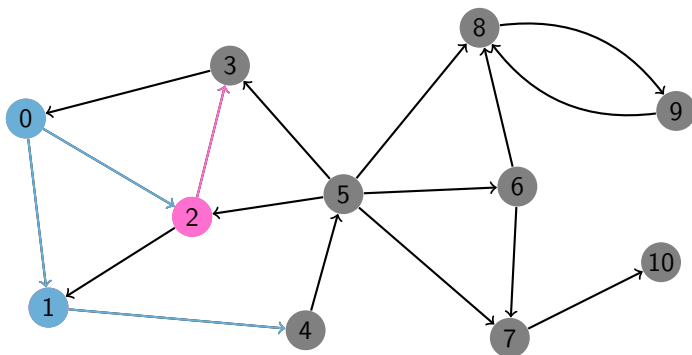
Breadth-first search



Queue: 2 4

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	1	0	0	0	0	0	0

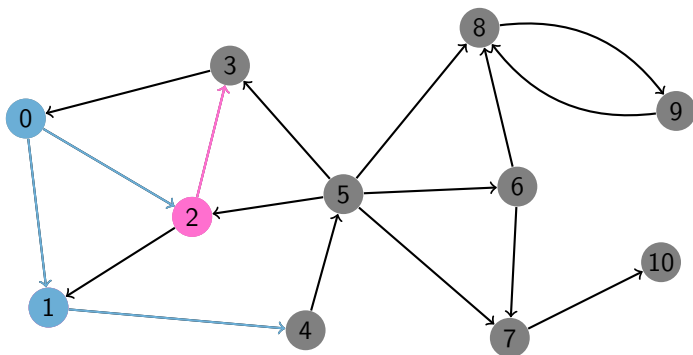
Breadth-first search



Queue: 2 4

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	1	0	0	0	0	0	0

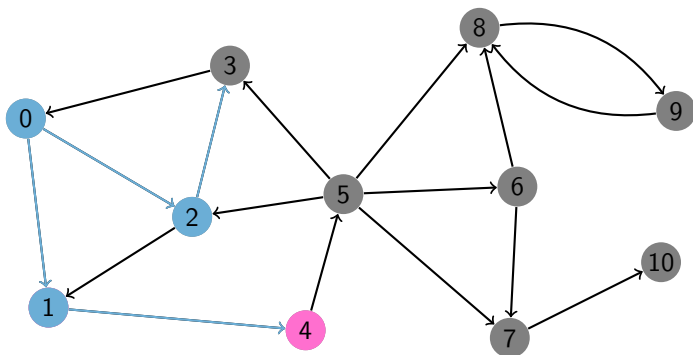
Breadth-first search



Queue: 2 4 3

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	0	0	0	0	0	0

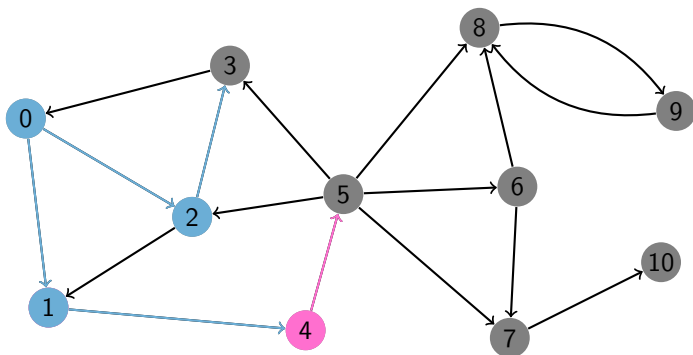
Breadth-first search



Queue: 4 3

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	0	0	0	0	0	0

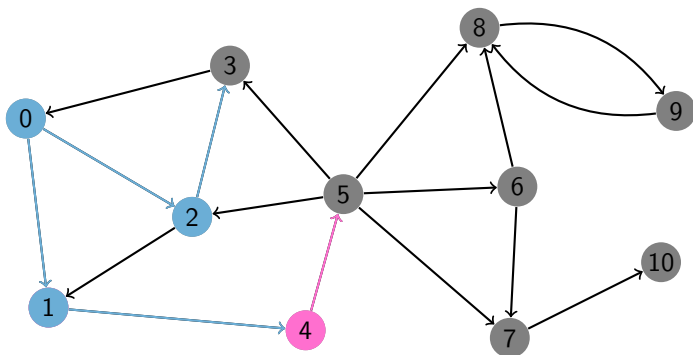
Breadth-first search



Queue: 4 3

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	0	0	0	0	0	0

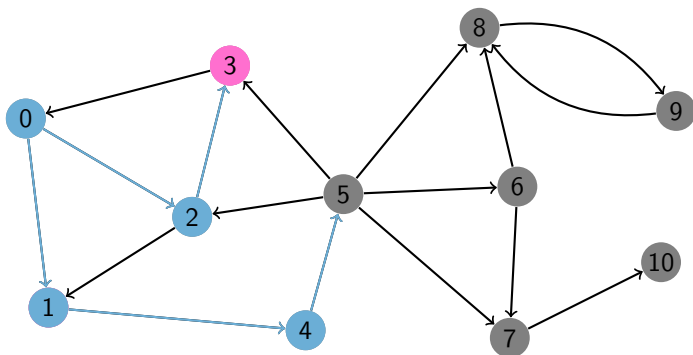
Breadth-first search



Queue: 4 3 5

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	0	0	0	0	0

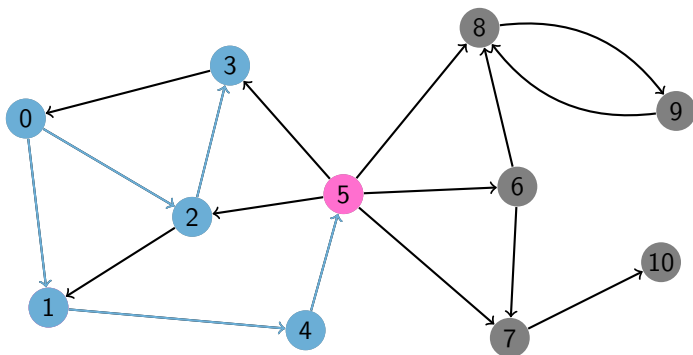
Breadth-first search



Queue: 3 5

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	0	0	0	0	0

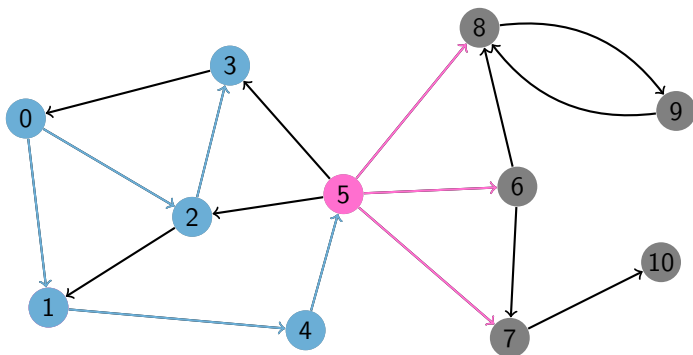
Breadth-first search



Queue: 5

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	0	0	0	0	0

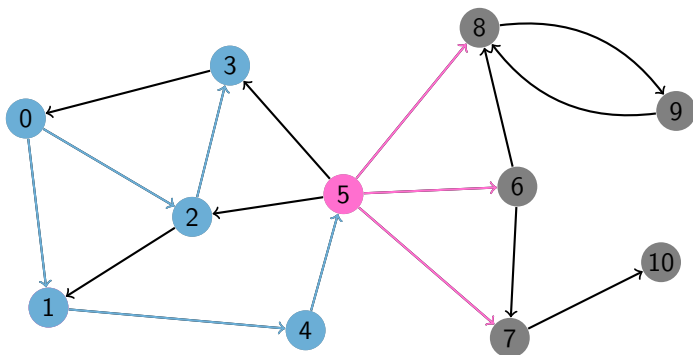
Breadth-first search



Queue: 5

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	0	0	0	0	0

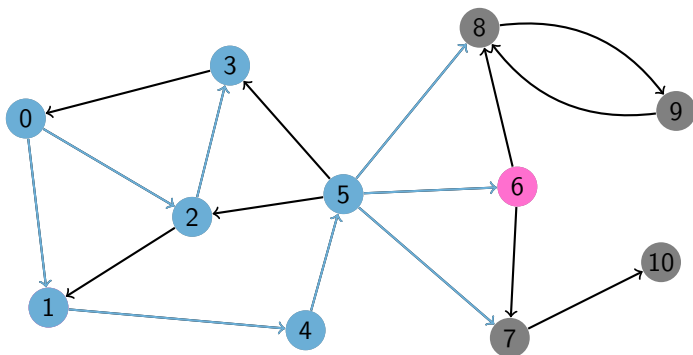
Breadth-first search



Queue: 5 6 7 8

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	0

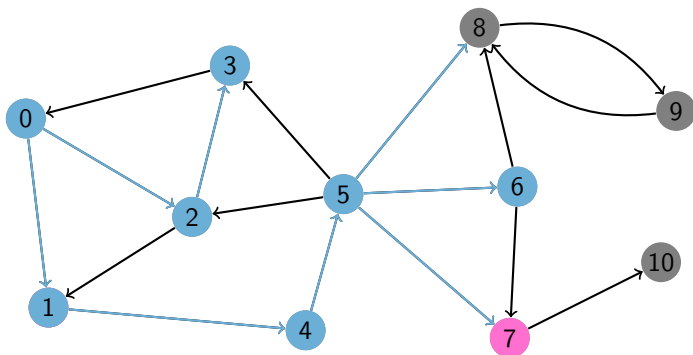
Breadth-first search



Queue: 6 7 8

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	0

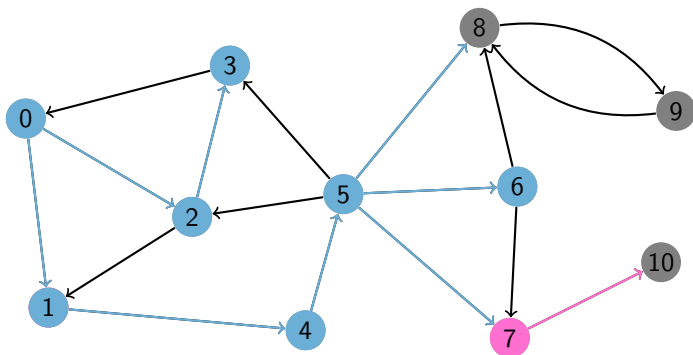
Breadth-first search



Queue: 7 8

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	0

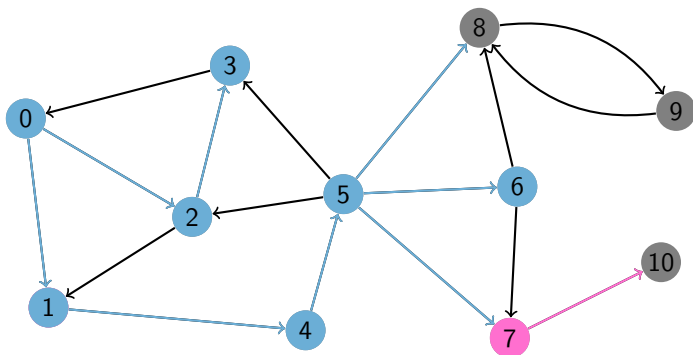
Breadth-first search



Queue: 7 8

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	0

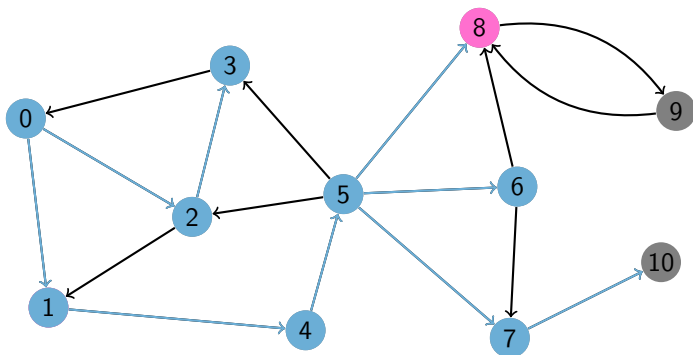
Breadth-first search



Queue: 7 8 10

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	1

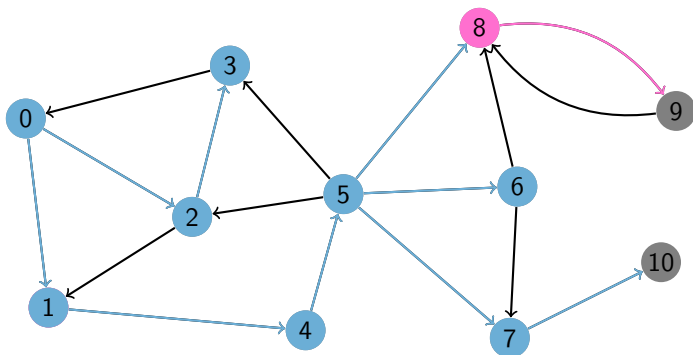
Breadth-first search



Queue: 8 10

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	1

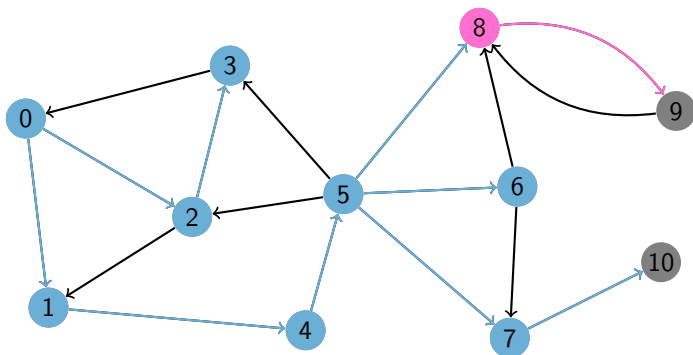
Breadth-first search



Queue: 8 10

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	1

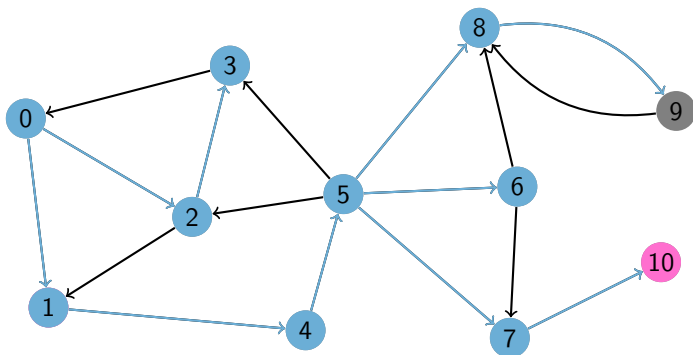
Breadth-first search



Queue: 8 10 9

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	1

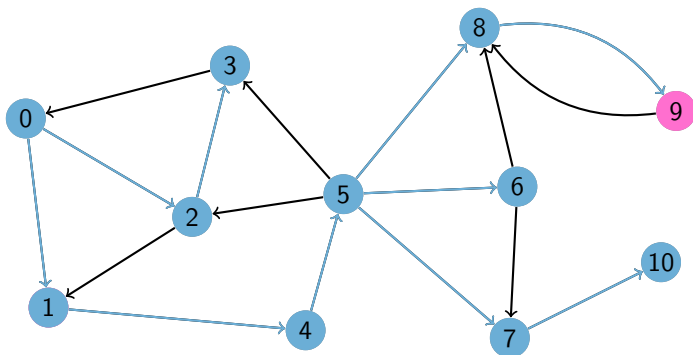
Breadth-first search



Queue: 10 9

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	1

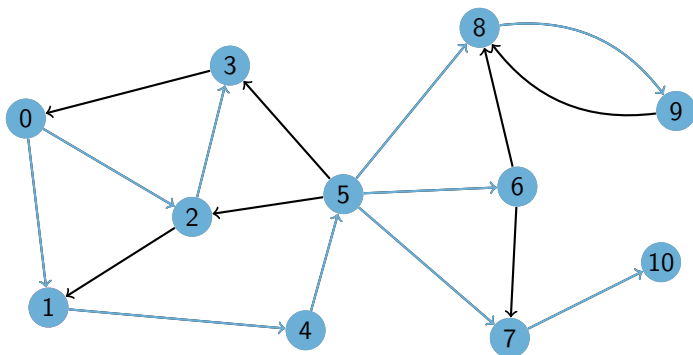
Breadth-first search



Queue: 9

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	1

Breadth-first search



Queue:

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	1

Breadth-first search

```
#define MAX 500
vector<int> ady[ MAX ];    //lista de adyacencia
bool visitado[ MAX ];     //para nodos visitados

void bfs(int inicial){
    queue<int> Q;
    Q.push( inicial );
    visitado[inicial] = true;

    while( !Q.empty() ){
        int actual = Q.front(); Q.pop();
        for( int i = 0 ; i < ady[ actual ].size() ; ++i ){
            //vemos adyacentes de nodo actual
            int adyacente = ady[actual][i];
            //si no esta visitado insertamos en la cola
            if( !visitado[ adyacente ] ){
                Q.push( adyacente );
                visitado[ adyacente ] = true;
            }
        }
    }
}
```

Complejidad

Considerando V = número de vértices y E = número de aristas:

- Usando Listas de Adyacencia: $O(V + E)$
- Usando Matriz de Adyacencia: $O(V^2)$

Aplicaciones

- Encontrar el camino más corto en un grafo con aristas de igual peso.
- Revisar si un grafo es bipartito.
- Encontrar componentes conexas en un grafo.
- Encontrar el ordenamiento topológico en un grafo.
- Encontrar el diámetro de un árbol.

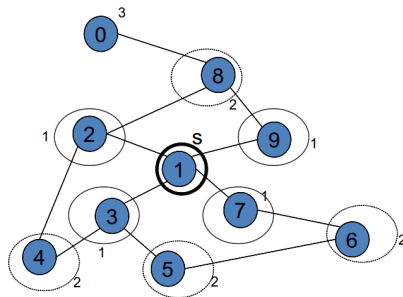
Camino más corto

- Tenemos un grafo sin pesos, y queremos encontrar la ruta mas corta desde A hacia B .
- Es decir, queremos encontrar un camino desde A hacia B con el mínimo número de aristas.
- Breadth-first search recorre los vértices en orden creciente de distancia desde el vértice inicial.
- Solo hacemos un único recorrido desde A , hasta encontrar B .
- O permitimos la búsqueda continuar por todo el grafo, y entonces encontramos las rutas más cortas desde A a todos los otros vértices.

Camino más corto

Problema: Dado un grafo G y un vértice inicial S , encontrar todos los vértices V que son alcanzables desde S determinando la distancia mas corta en G desde S a V .

Que entendemos por "**distancia**"? *El número de aristas en una ruta de S a V .*



Ejemplo:

Consideremos $S =$ vértice 1

- Nodos a distancia 1?
2, 3, 7, 9
- Nodos a distancia 2?
8, 6, 5, 4
- Nodos a distancia 3?
0

Camino más corto

```
//Impresion de camino mas corto
void printPath( int destino ){
    if( previo[ destino ] == -1 ){ //si estamos en vertice inicial
        printf("%d" , destino ); return;
    }
    printPath( previo[destino ] );
    printf(" %d" , destino );
}

void bfs_shortest_path(int inicial){
    queue<int> Q;
    Q.push( inicial );
    visitado[inicial] = true;
    distancia[inicial] = 0;           //distancia inicial es 0
    previo[ inicial ] = -1;          //previo de inicial es vacio
    while( !Q.empty() ){
        int actual = Q.front(); Q.pop();
        for( int i = 0 ; i < ady[ actual ].size() ; ++i ){
            //vemos adyacentes de nodo actual
            int adyacente = ady[actual][i];
            //si no esta visitado insertamos en cola
            if( !visitado[ adyacente ] ){
                Q.push( adyacente );
                visitado[ adyacente ] = true;
                distancia[ adyacente ] = distancia[ actual ] + 1;
                previo[ adyacente ] = actual;
            }
        }
    }
}

for( int i = 0 ; i < V ; ++i )
    printf("%d: %d\n" , i , distancia[i]);
```

Ejemplo Aplicativo

Tenemos una matriz de caracteres que representa un laberinto 2D
un '#' implica un muro, un '.' implica un espacio libre, un 'E'
indica la entrada del laberinto y una 'S' indica una salida.

```

. . . . . E
. #####
. # . . . . .
. # . S . . . S
. ### . # . #
. # . . . # . #
. # . ### . #
. . . . .

```

¿Cuánto mide la ruta más corta para escapar?


Solución

- El problema puede ser resuelto por BFS al tener un estado inicial y final e ir avanzando por algún adyacente que no sea '#' . Los posibles adyacentes para movernos es hacia arriba, abajo, izquierda y derecha.
- Haremos uso de la técnica de revisión de adyacentes aprendido en el capítulo de Ad Hoc.

Solución




Así por ejemplo si estoy en la posición siguiente (círculo) y suponiendo que sea la coordenada (5,3):

```

. . . . . E
. #####
. # . . . . .
. # . S . . . S
. ### . # . #
. # .  . # . #
. # . ### . #
. . . . .

```

```

. . . . . E
. #####
. # . . . . .
. # . S . . . S
. ###  . # . #
. # .  . # . #
. # .  ### . #
. . . . .

```

Solución

Primeramente declaramos nuestro laberinto como una matriz, en este caso tenemos un grafo implícito, también declaramos un arreglo de Visitado que nos indicará si se visito o no un estado determinado y en nuestros arreglos para la revisión de adyacentes:

```
char ady[ MAX ][ MAX ];    //laberinto
bool visitado[ MAX ][ MAX ]; //arreglo de estados visitados
int dx[ 4 ] = {0, 0, 1, -1 }; //incremento en coordenada x
int dy[ 4 ] = {1, -1, 0, 0 }; //incremento en coordenada y
```

Solución

Cada vez que trabajamos con BFS trabajamos con estados, desde un estado inicial podemos llegar al estado final por medio de varios estados, por ello una forma de manejar los diferentes estados y los valores que se tiene hasta ese estado podemos realizar una estructura **Estado**:

```
struct Estado{  
    int x; // Fila del estado  
    int y; // Columna del estado  
    int d; // Distancia del estado  
    Estado(int _x , int _y , int _d) : x(_x), y(_y), d(_d){}  
    Estado(){}  
};
```

Solución

Ahora seguimos los pasos vistos anteriormente en la explicación del recorrido:

```
//marcamos como no visitado todos los vertices
memset( visitado , false , sizeof( visitado ) );

Estado inicial( x , y , 0 ) ; //Estado inicial, distancia = 0
queue<Estado> Q;              //Cola de todos los posibles Estados por los
                              //que se pase para llegar al destino
Q.push( inicial );           //Insertamos el estado inicial en la Cola.

visitado[ x ][ y ] = 1;      //Marcamos como visitado la posicion inicial
```

```
.....E
.#####
.#.....
.#.S...S
.###.#.#
.#...#.#
.###.#.#
```

Solución

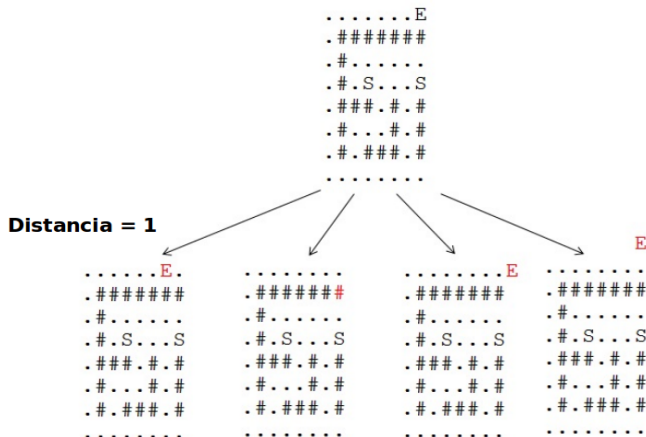
```

int BFS( int x , int y ){
    //marcamos como no visitado todos los vertices
    memset( visitado , false , sizeof( visitado ) );
    Estado inicial( x , y , 0 ) ; //Estado inicial, distancia = 0
    queue<Estado> Q;
    Q.push( inicial ); //Insertamos el estado inicial en la Cola.
    visitado[ x ][ y ] = 1; //Marcamos como visitado la posicion inicial

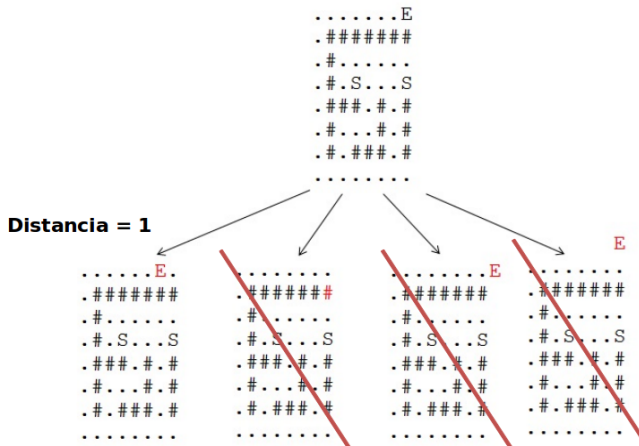
    while( !Q.empty() ){
        Estado actual = Q.front(); Q.pop();
        //Si se llego al destino (punto final)
        if( ady[actual.x][actual.y]== 'S'){
            print( actual.x , actual.y );
            return actual.d;
        }
        //Recorremos los posibles adyacentes
        for( int i = 0; i < 4; ++i ){
            int nx = dx[ i ] + actual.x;
            int ny = dy[ i ] + actual.y;
            //Comprobamos movimientos invalidos
            if( nx >= 0 && nx < h && ny >= 0 && ny < w
                && ady[ nx ][ ny ] != '#' && !visitado[ nx ][ ny ] ){
                //Creamos estado adyacente y aumentamos en 1 la distancia recorrida
                Estado adyacente( nx , ny , actual.d + 1 );
                Q.push( adyacente );
                prev[ nx ][ ny ] = actual;
                visitado[ nx ][ ny ] = 1;
            }
        }
    }
    return -1;
}

```

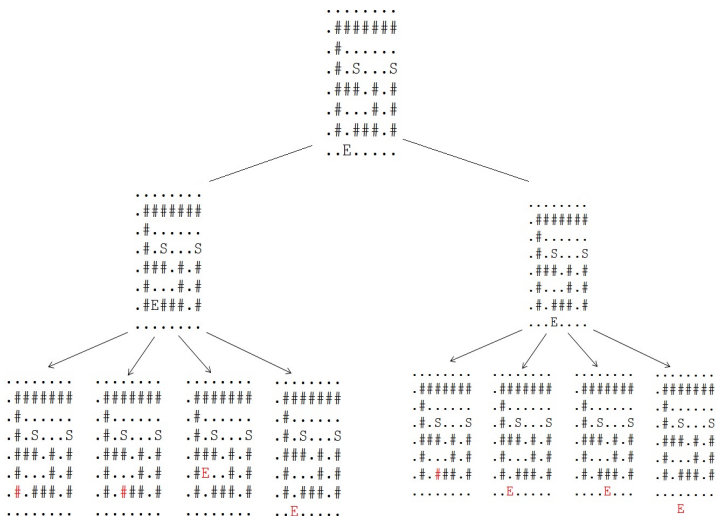
Solución



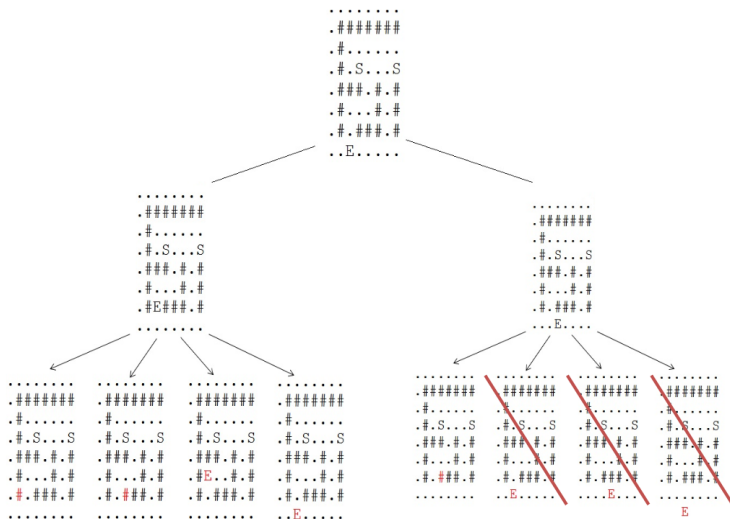
Solución



Solución



Solución



Impresión de camino

Supongamos ahora que el problema nos da una tarea adicional, la cual es mostrar el camino mas corto de E a S.

```

. . . . .
. #####
. # . . . .
. # . S . . S
. ### . # . #
. # . . . # . #
. # . ### . #
. . . . .

```

Puntos Importantes

- Hay que definir que es un estado según el enunciado del problema. En el problema anterior un estado es nuestra posición con respecto al tablero (x,y) .
- Hay que escoger una estructura de datos adecuada para saber que estados ya hemos visitado.

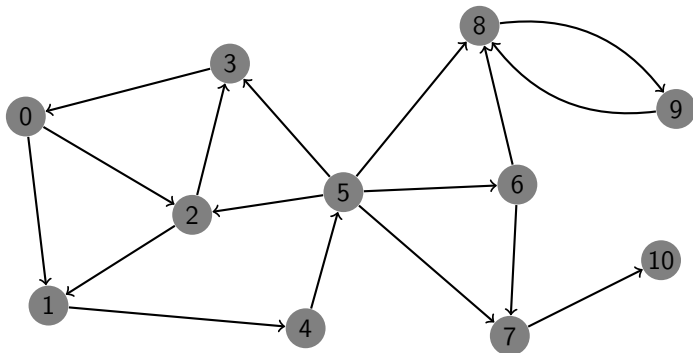
Contenido

- 1 Introducción
- 2 Representación
 - Matriz de adyacencia
 - Lista de adyacencia
- 3 Recorrido de grafos
 - Breadth-first Search
 - Depth-first Search

Depth-first Search (DFS)

- Permite realizar el recorrido en un grafo.
- Trabaja en grafos dirigidos y no dirigidos.
- Hace uso de la estructura de datos **Pila**.
- Usualmente implementado usando recursión.

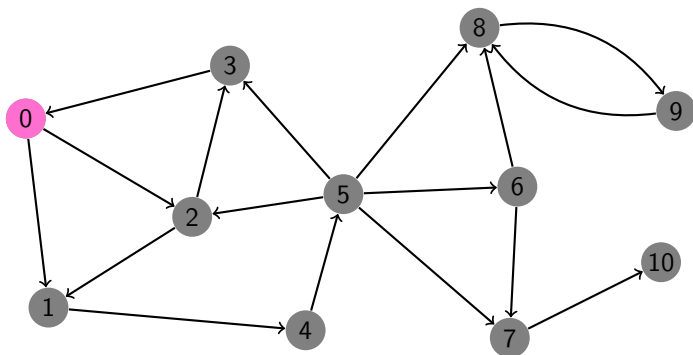
Depth-first search



Stack: |

	0	1	2	3	4	5	6	7	8	9	10
marked	0	0	0	0	0	0	0	0	0	0	0

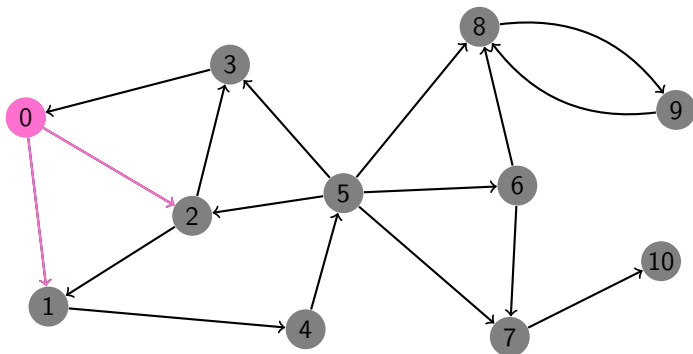
Depth-first search



Stack: 0 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	0	0	0	0	0	0	0	0	0	0

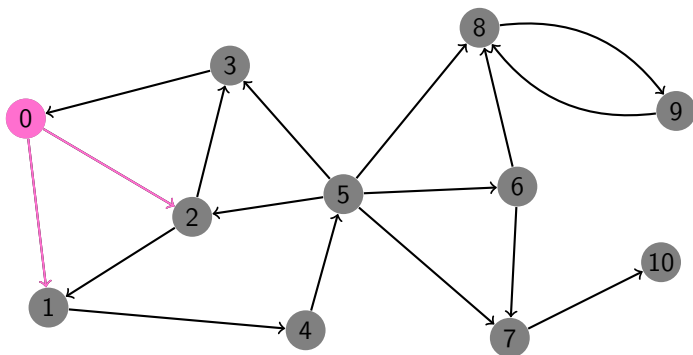
Depth-first search



Stack: 0 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	0	0	0	0	0	0	0	0	0	0

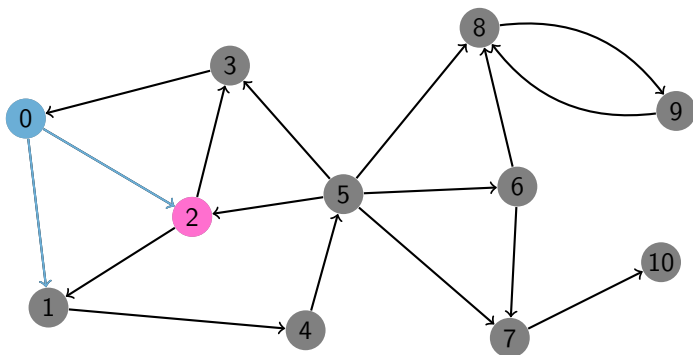
Depth-first search



Stack: 0 | 2 1

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	0	0	0	0	0	0	0

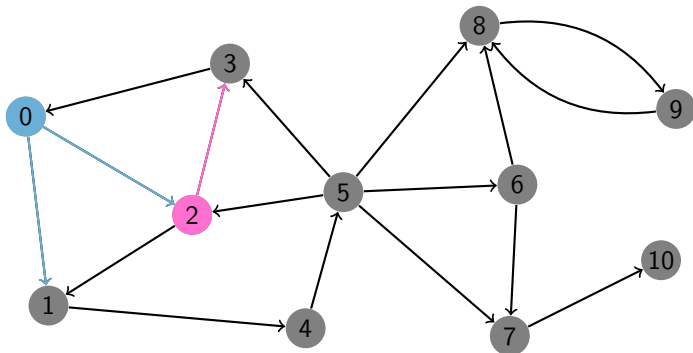
Depth-first search



Stack: 2 | 1

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	0	0	0	0	0	0	0

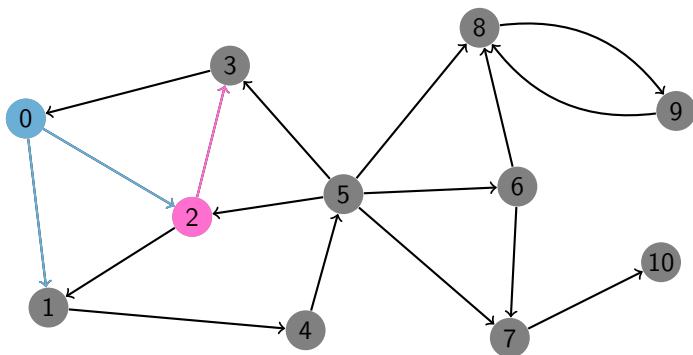
Depth-first search



Stack: 2 | 1

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	0	0	0	0	0	0	0	0

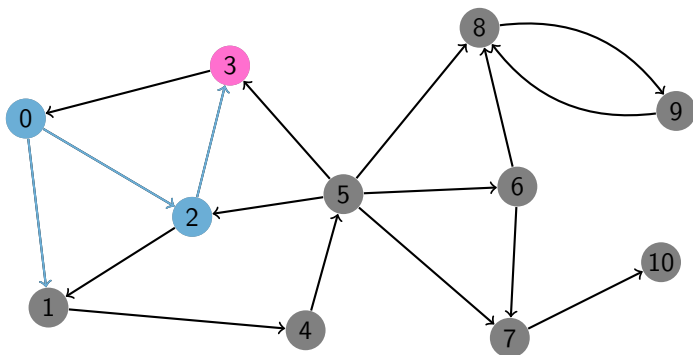
Depth-first search



Stack: 2 | 3 1

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	0	0	0	0	0	0	0

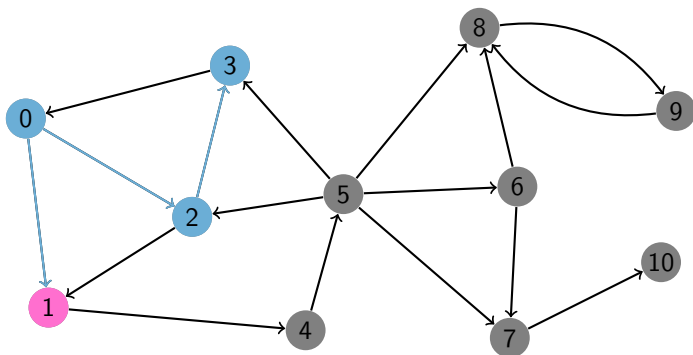
Depth-first search



Stack: 3 | 1

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	0	0	0	0	0	0	0

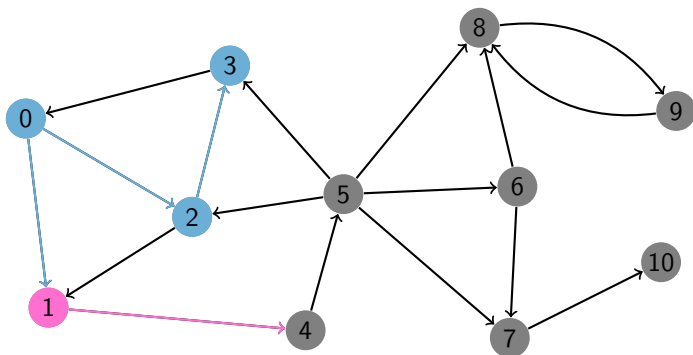
Depth-first search



Stack: 1 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	0	0	0	0	0	0	0

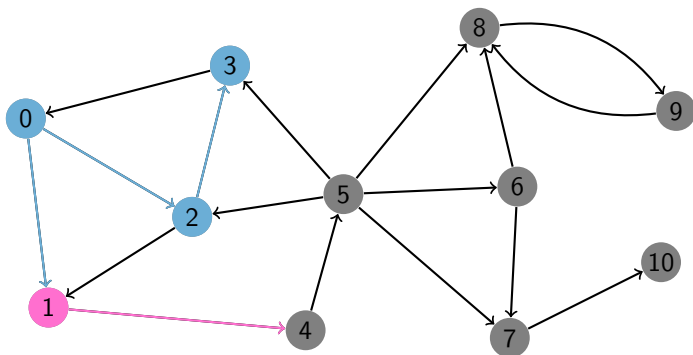
Depth-first search



Stack: 1 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	0	0	0	0	0	0	0

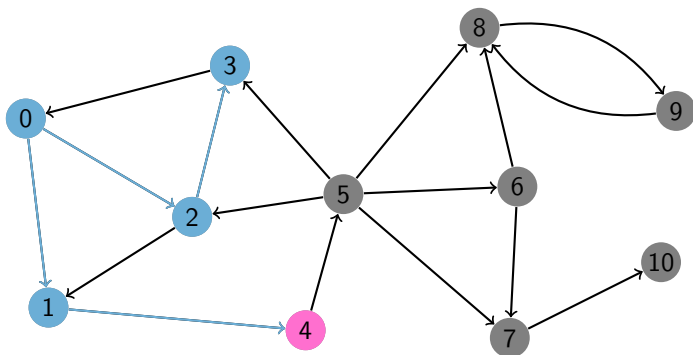
Depth-first search



Stack: 1 | 4

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	0	0	0	0	0	0

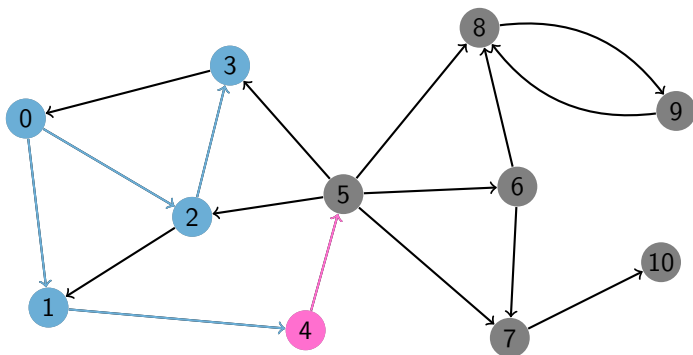
Depth-first search



Stack: 4 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	0	0	0	0	0	0

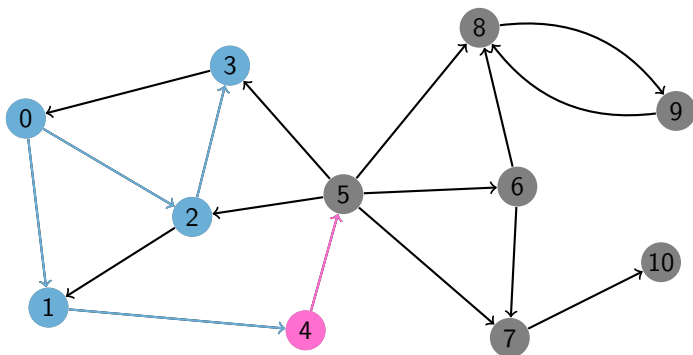
Depth-first search



Stack: 4 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	0	0	0	0	0	0

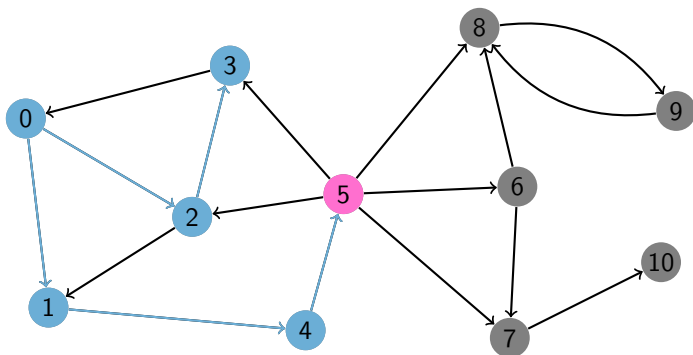
Depth-first search



Stack: 4 | 5

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	0	0	0	0	0

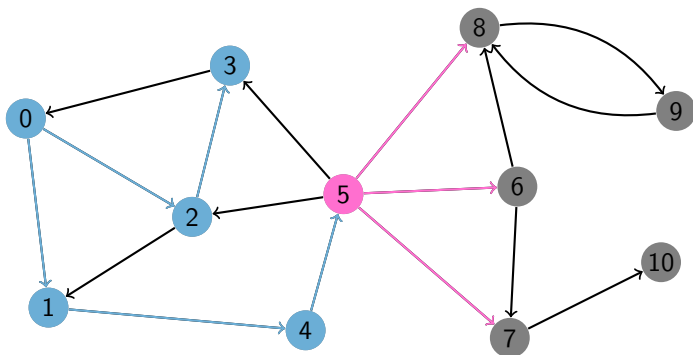
Depth-first search



Stack: 5 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	0	0	0	0	0

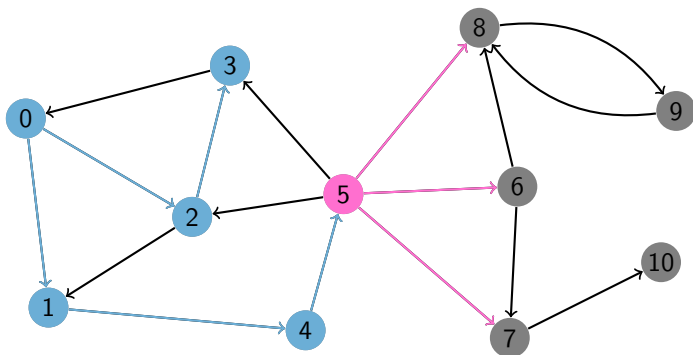
Depth-first search



Stack: 5 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	0	0	0	0	0

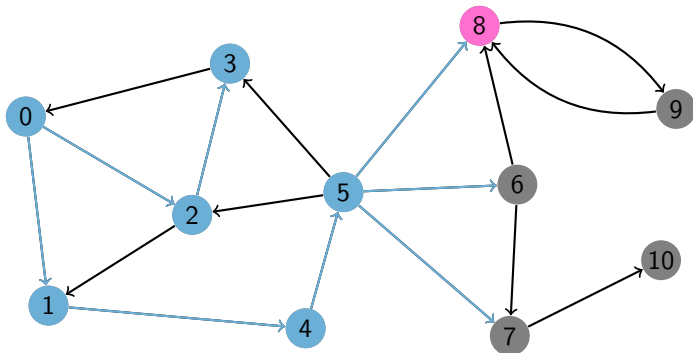
Depth-first search



Stack: 5 | 8 6 7

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	0

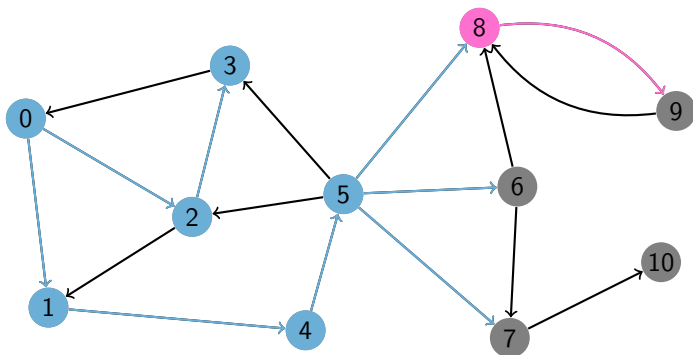
Depth-first search



Stack: 8 | 6 7

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	0

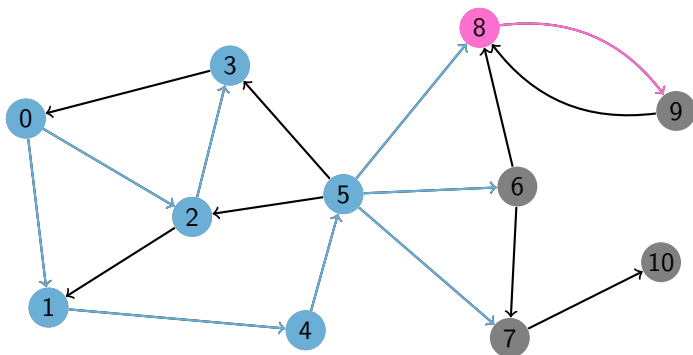
Depth-first search



Stack: 8 | 6 7

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	0	0

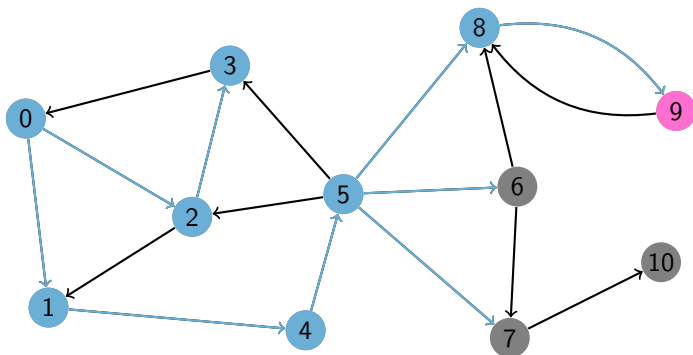
Depth-first search



Stack: 8 | 9 6 7

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	0

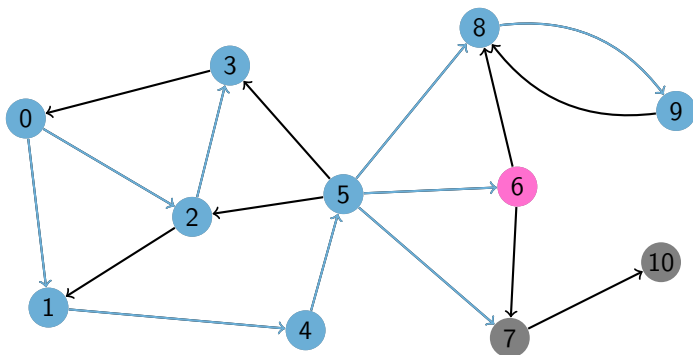
Depth-first search



Stack: 9 | 6 7

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	0

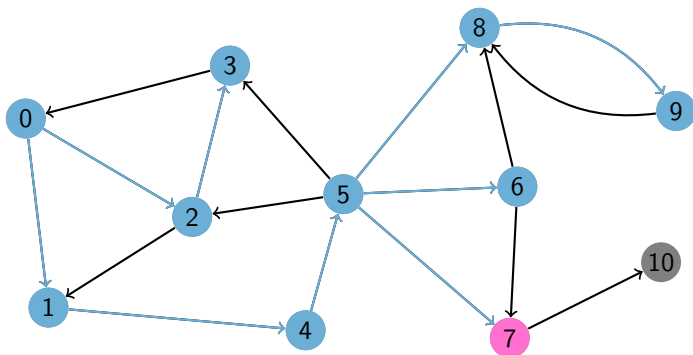
Depth-first search



Stack: 6 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	0

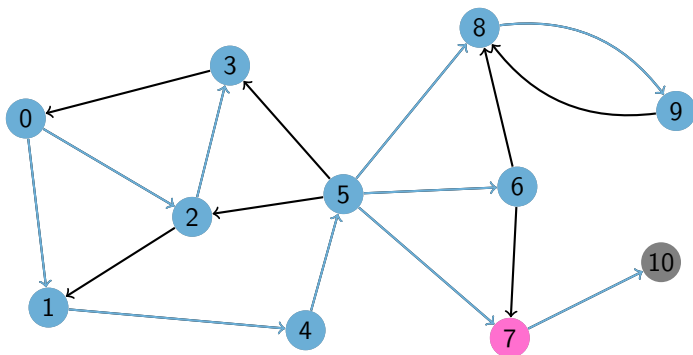
Depth-first search



Stack: 7 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	0

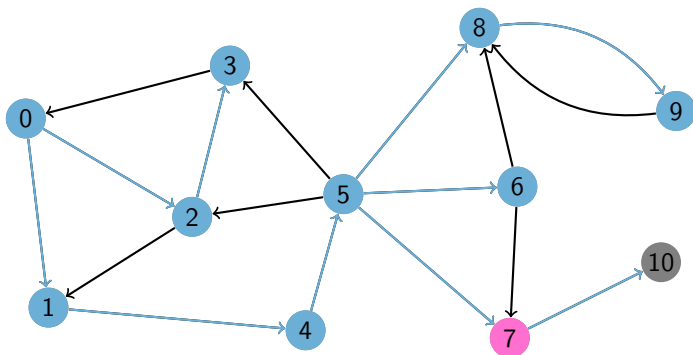
Depth-first search



Stack: 7 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	0

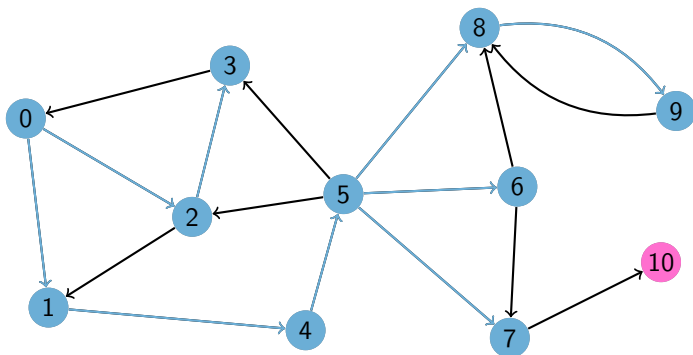
Depth-first search



Stack: 7 | 10

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	1

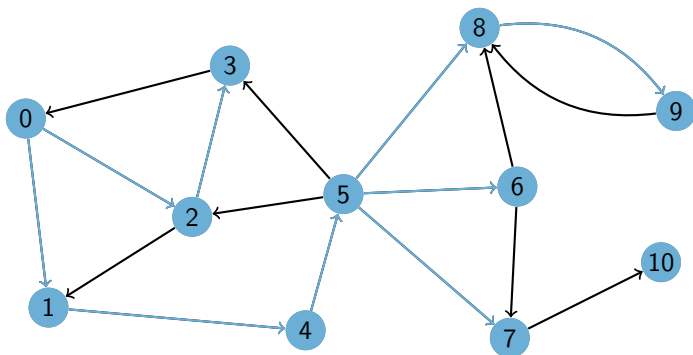
Depth-first search



Stack: 10 |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	1

Depth-first search



Stack: |

	0	1	2	3	4	5	6	7	8	9	10
marked	1	1	1	1	1	1	1	1	1	1	1

Depth-first search (iterativo)

```
#define MAX 500
vector<int> ady[ MAX ]; //lista de adyacencia
bool visitado[ MAX ]; //para nodos visitados

void dfs(int inicial){
    stack<int> S;
    S.push( inicial );
    visitado[ inicial ] = true;

    while( !S.empty() ){
        int actual = S.top(); S.pop();
        for(int i = 0 ; i < ady[ actual ].size() ; ++i ){
            int adyacente = ady[ actual ][ i ];
            //si no esta visitado insertamos en pila
            if( !visitado[ adyacente ] ){
                visitado[ adyacente ] = true;
                S.push( adyacente );
            }
        }
    }
}
```

Depth-first search (recursivo)

```
#define MAX 500
vector<int> ady[ MAX ];    //lista de adyacencia
bool visitado[ MAX ];     //para nodos visitados

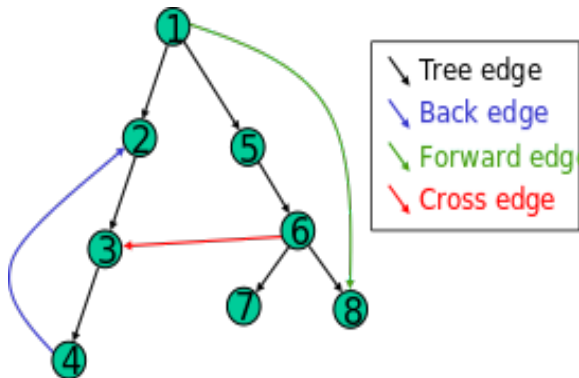
bool first;
void dfs( int u ){
    ( first )? printf("%d" , u ) : printf("->%d" , u );
    first = false;
    visitado[ u ] = true;
    for( int v = 0 ; v < ady[ u ].size(); ++v ){
        if( !visitado[ ady[ u ][ v ] ] ){
            dfs( ady[ u ][ v ] );
        }
    }
}
```

Árbol de búsqueda en profundidad

Cuando realizamos una búsqueda en profundidad desde un cierto vértice, el camino que tomamos forma un árbol. Existen 4 tipos de aristas en el recorrido, considerando la arista (u,v) , arista desde el vértice u hacia el vértice v , tenemos:

- Si v es visitado por primera vez, la arista es llamada **tree edge**.
- Si v ya fue visitado, tenemos lo siguiente:
 - Si v es un ancestro de u , la arista es llamada **backward edge**.
 - Si v es un descendiente de u , la arista es llamada **forward edge**.
 - Si v no es ninguno de los casos anteriores, la arista es llamada **cross edge**.

Árbol de búsqueda en profundidad



Árbol de búsqueda en profundidad

- Este árbol puede ser analizado para obtener información muy útil.
- Por ejemplo: un *backward edge* representa un ciclo en el grafo original.
- Si no hay *backward edges*, entonces el grafo no posee ciclos (grafo acíclico).

DFS vs BFS

Aplicaciones	DFS	BFS
Bosques de Expansión, componentes conexas, recorridos, ciclos, revisar si grafo es bipartito	SI	SI
Camino más corto	NO	SI
Componentes Biconexas	SI	NO
Ordenamiento Topológico	SI	SI