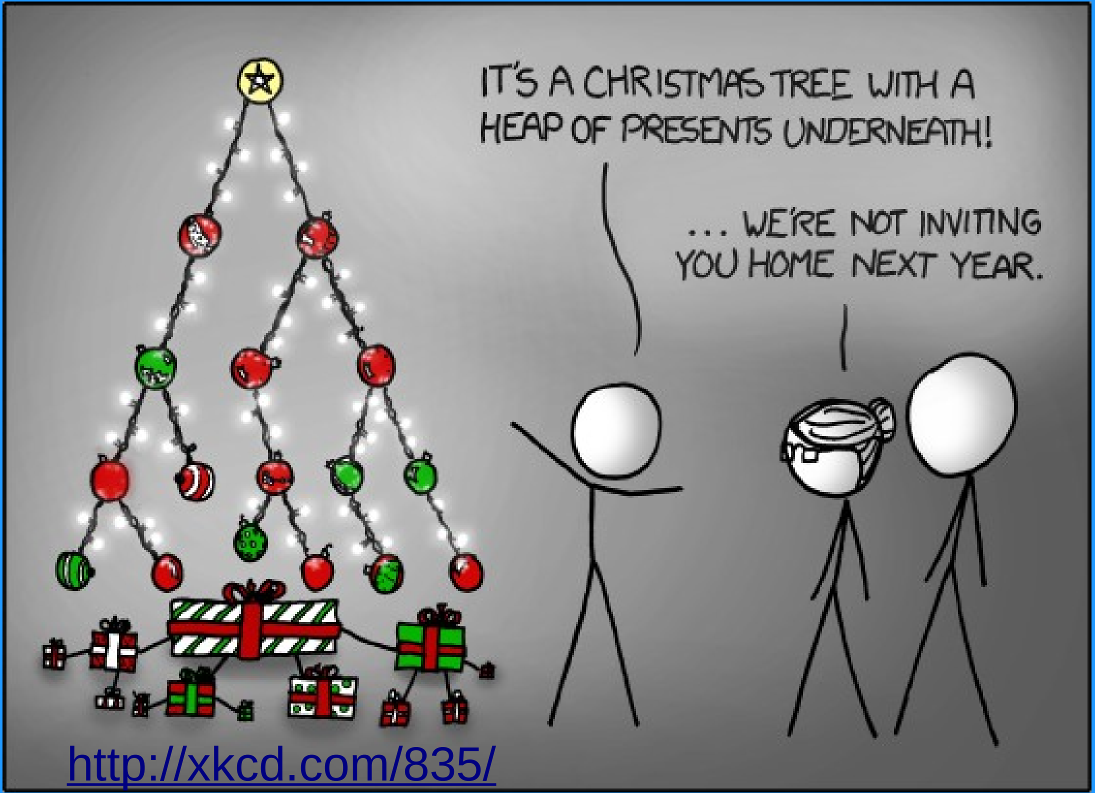
Estructuras de Datos

2020



http://xkcd.com/835/

Grafos

Definiciones

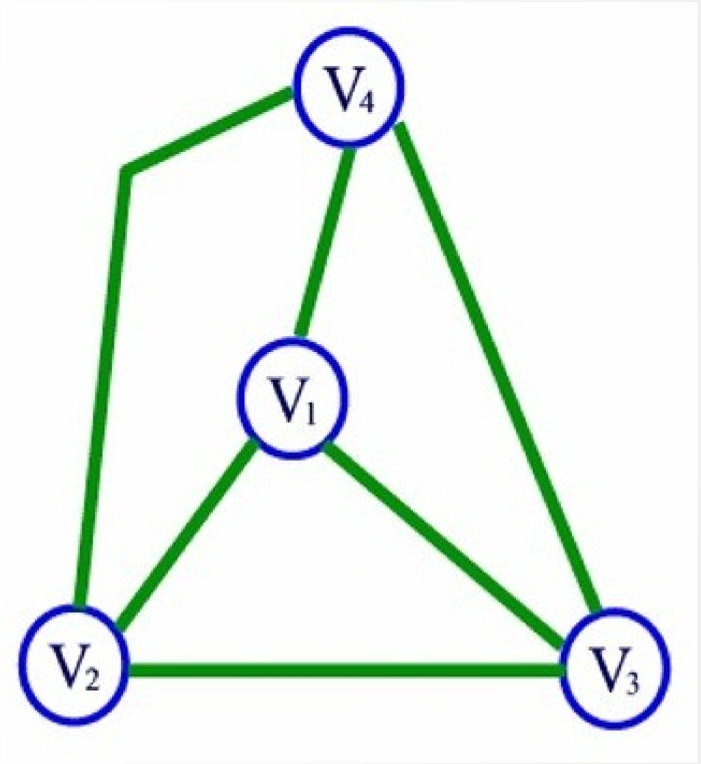
Grafos

Un grafo es:

-Un conjunto de Vértices no vacíos

+

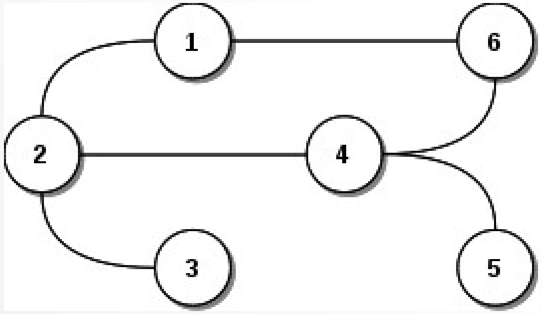
-Un conjunto de Aristas G= {V,A}

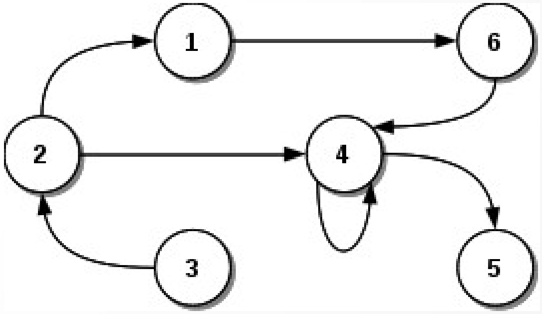


Grafos

-Si las aristas son pares no ordenados de vértices → el grafo es no dirigido

-Si las aristas son pares ordenados de vértices → el grafo es dirigido

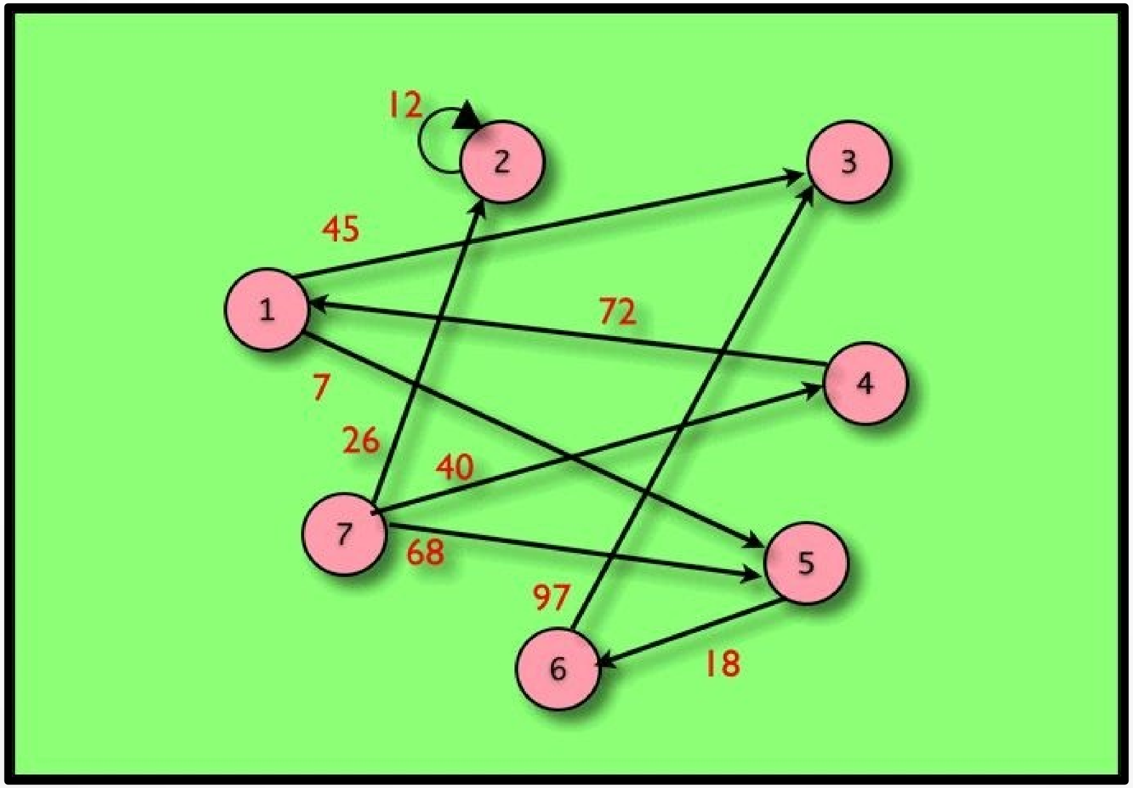




Grafos

Grafo Pesado Es un Grafo en el que las aristas tienen un peso asignado.

Un Grafo pesado puede ser dirigido o no dirigido.



Grafos

Grado:

Grado de Salida: es el número de vértices adyacentes al vértice v, o sea la cantidad de aristas que “salen” de un vértice.

Grado de Entrada: es el número de vértices incidentes al vértice v, o sea la cantidad de aristas que “llegan” de un vértice.

Grafos

Camino:

Es una secuencia de vértices v1,v2,. . . ,vn tal que

v1-> v2,v2->v3 . . . o

(v1,v2),(v2,v3) . . .

Pertenecen al conjunto de aristas del grafo

Es un camino de v1 a vn

Grafos

Longitud del Camino:

Está dado por la cantidad de aristas que contienen un camino. Cada vértice tiene un camino a si mismo de longitud cero.

Ciclo: Camino en el que el primer y último vértice coinciden.

Grafos

Ciclo:

Para digrafo: Camino de longitud mayor que 1 que empieza y termina con el mismo vértice.

Para Grafo: Ciclo de longitud al menos 3

Grafo Acíclico: es un grafo sin ciclos

Grafos

Grafo Conexo:

Para digrafo:

Fuertemente Conexo: si existe un camino para cada par de vértices.

Debilmente Conexo: Si existe un camino para cualquier par de vértices sin tener en cuenta la orientación de los arcos.

Para Grafo: Si existe camino entre cada par de vértices

Grafos

Grafo Completo:

Si existe una arista entre cada par de vértices es completo.

Componente fuertemente conexa (para digrafo):

Es un conjunto de vértices en el que hay un camino desde cualquier vértice en el conjunto a otro del conjunto.

TAD Grafo
Especificación

Especificación. “Molde General”

TAD Nombre TAD

Igualdad Observacional

Usa

Parámetro Formal

Géneros

observadores básicos

Generadores

otras operaciones

Axiomas

Exporta

Especificación. Grafo

TAD Grafo <a>

Igualdad Observacional

Si **a** y **b** son dos grafos

a es igual a **b** si se cumple que: Tienen los mismos vértices y las mismas conexiones entre ellos.

Usa

Natural, Bool, None, Lista<a>

Parámetro Formal

a

Géneros

Grafo<a>

Especificación. Grafo

```
observadores básicos

tamaño(Grafo<a>) → Natural

es_vacío(Grafo<a>) → Bool


{Pre: el vértice pertenece al grafo}

adyacentes(Grafo<a>, a) → Lista<a>


{Pre: los vértices pertenecen al grafo}

Es_adyacente(Grafo<a>, a, a) → Bool


vértices(Grafo<a>) → Lista<a>
```

Especificación. Grafo

Generador

```
vacío() → Grafo<a>
{Post: El grafo retornado esta vacío}
```

Especificación. Grafo

```
Otras Operaciones


agregar_vértice(Grafo<a>, a ) → None
{Pos: el grafo no esta vacío}


{Pre: el vértice pertenece al grafo}
borrar_vértice(Grafo<a>, a ) → None


agregar_arista(Grafo<a>, a, a ) → None
borrar_arista(Grafo<a>, a, a ) → Bool
```

Especificación. Grafo

Axiomas

vacío(): Crea un grafo (sin elementos)

tamaño(Grafo<a> g) → Natural: Retorna/devuelve la cantidad de vértices del grafo g.

es_vacío(Grafo<a> g) → Bool: Retorna/devuelve verdadero si el grafo g esta vacío y falso en caso contrario

adyacentes(Grafo<a> g, a v) → Lista<a>: Retorna una lista con los adyacentes del vértice v.

es_adyacente(Grafo<a> g, a v, a j) → Bool: Retorna/devuelve verdadero si en el grafo g v y j son adyacentes y falso en caso contrario

vértices(Grafo<a> g): retorna una lista con todos los vértices del grafo g.

Especificación. Grafo

Axiomas

agregar_vértice(Grafo<a> g, a v) → None: agrega el vértice v al grafo g

borrar_vértice(Grafo<a> g, a v) → None: borra el vértice v del grafo g

agregar_arista(Grafo<a> g, a v, a j) → None: agrega una arista entre v y j en el grafo g

borrar_arista(Grafo<a> g, a v, a j) → Bool: retorna verdadero si pudo borrar la arista entre v y j en el grafo g y falso en caso contrario.

Especificación. Grafo

Exporta

```
Grafo<a>, vacío, tamaño, es_vacío, adyacentes,
es_adyacente, agregar_vertice, borrar_vertice,
agregar_arista, borrar_arista
```

TAD Grafo
Algunas implementaciones

Matriz de Adyacencias

Grafos. M. de Adyacencias

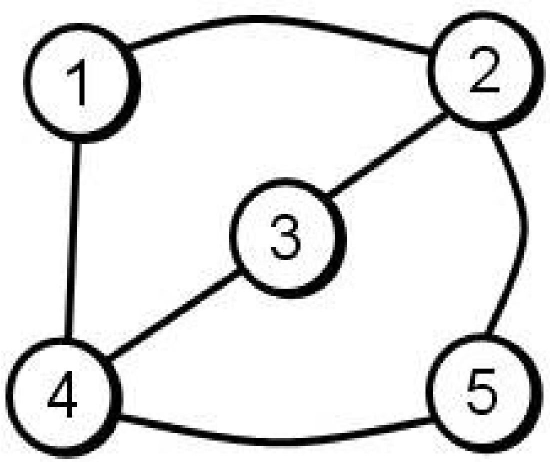
Para un grafo de N vértices usamos una matriz A de N * N tal que:

-A[i,j] es True si y solo si hay una arista del vértice i al j

-A[i,j] es False en caso contrario

La matriz de adyacencias para un grafo no dirigido es simétrica si A[i,j]=A[j,i]. En este caso alcanza con almacenar la mitad.

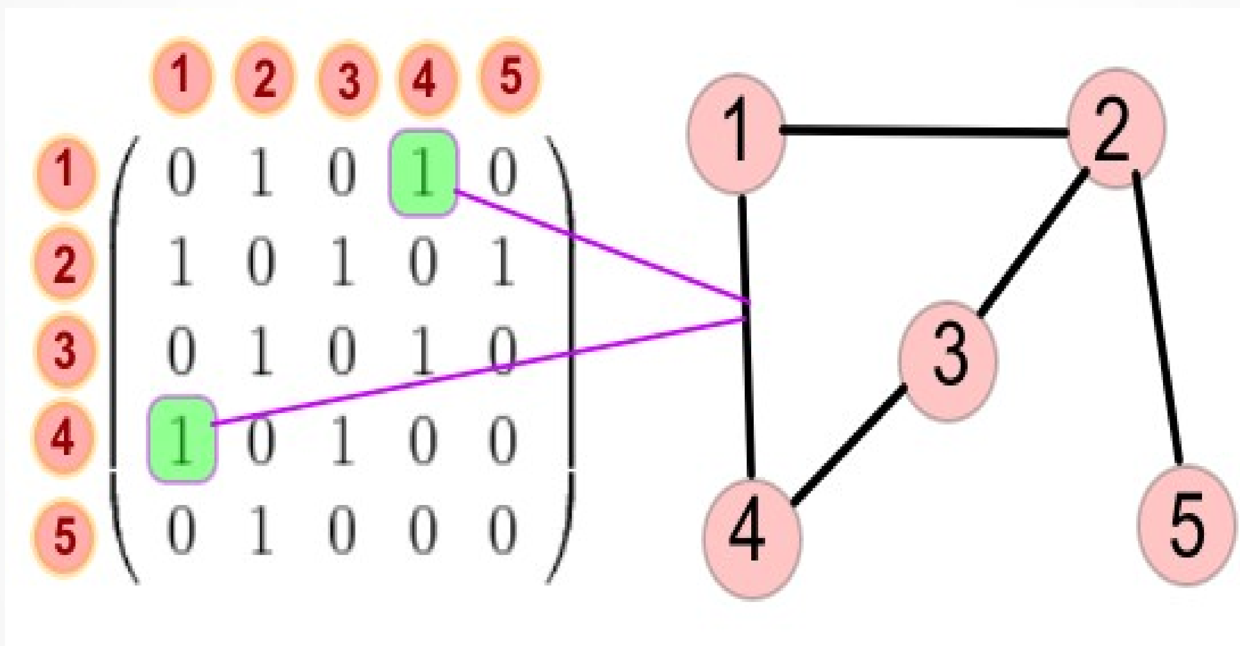
Grafos. M. de Adyacencias



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

Usamos 0 y 1 en lugar de False y True

Grafos. M. de Adyacencias



Usamos 0 y 1 en lugar de False y True

Grafos. M. de Adyacencias

En caso de ser Grafo Pesado (en el que las aristas tienen un peso asignado)

A[i,j]=0 si no existe arista

A[i,j]=peso si existe arista

Grafo. Estructura Interna

```
class AdjacencyMatrixGraph:

    __slots__ = ['_vertices', '_edges']

def __init__(self):
      self._vertices = []
      self._edges = []
```

Usamos una lista para almacenar los
Valores de los vértices
Y una” lista de listas” (matriz) para
Las aristas y sus pesos

Grafo. Operaciones

```
def is_empty(self):
        return len(self._vertices) == 0


def __len__(self):
        return len(self._vertice)


def vertices(self):
        return list(range(len(self._vertices)))


def get_vertex(self, vertex):
        return self._vertices[vertex]


def set_vertex(self, vertex, value):
        self._vertices[vertex] = value
```

Grafo. Implementación

```
def __getitem__(self, key):
        if isinstance(key, tuple):
            return self.get_edge(key[0], key[1])
        return self.get_vertex(key)
```

Puede ser una de estas opciones :
G[índice]→ valor vértice
G[valor1,valor2]→ valor del peso

```
def __setitem__(self, key, value):
        if isinstance(key, tuple):
            return self.set_edge(key[0], key[1], value)
        return self.set_vertex(key, value)
```

Puede ser una de estas opciones :
G[índice]= valor vértice
G[valor1,valor2]=valor del peso

Grafo. Implementación

```
def __delitem__(self, key):
        if isinstance(key, tuple):
            self.erase_edge(key[0], key[1])
        else:
            self.erase_vertex(key)
```

Grafo. Implementación

```
def add_vertex(self, value):
        vertex = len(self._vertices)
        self._vertices.append(value)
        for edges in self._edges:
            edges.append(None)
        self._edges.append([None] * (vertex + 1))
        return vertex



def erase_vertex(self, vertex):
        assert 0 <= vertex < len(self._vertices)
        del self._vertices[vertex]
        del self._edges[vertex]
        for edges in self._edges:
            del edges[vertex]
```

Grafo. Implementación

```
def get_edge(self, from_, to):
        return self._edges[from_][to]


def set_edge(self, from_, to, weight):
        self._edges[from_][to] = weight


def add_edge(self, from_, to, weight):
        self._edges[from_][to] = weight


def erase_edge(self, from_, to):
        if self._edges[from_][to] is not None:
            self._edges[from_][to] = None
            return True
        return False
```

Grafo. Implementación

```
def is_adjacent(self, from_, to):
        return self._edges[from_][to] is not None

def adjacents(self, from_):
        result = []
        for i in range(len(self._vertices)):
            if self._edges[from_][i] is not None:
                result.append(i)
        return result

def __eq__(self, other):
    return self._vertices == other._vertices and self._edges ==
other.edges
```

Grafo. Implementación

```
def copy(self):

        result = AdjacencyMatrixGraph()
        result._vertices = self._vertices.copy()
        result._edges = []
        for edge in self._edges:
              result._edges.append(edge.copy())
        return result



def __repr__(self):
    result = '( '
    for v in self.vertices():
        for w in self.adjacents(v):
            result += '{0}-({2})->{1}, '.format(v, w,

self.get_edge(v, w))
    return result + ') : AdjacencyMatrixGraph'
```

Grafos. M. de Adyacencias

¿ Qué les parece la implementación propuesta?

¿Qué pasa si el grafo a representar tiene pocas aristas?

No usamos gran parte de la matriz

Lista de Adyacencias

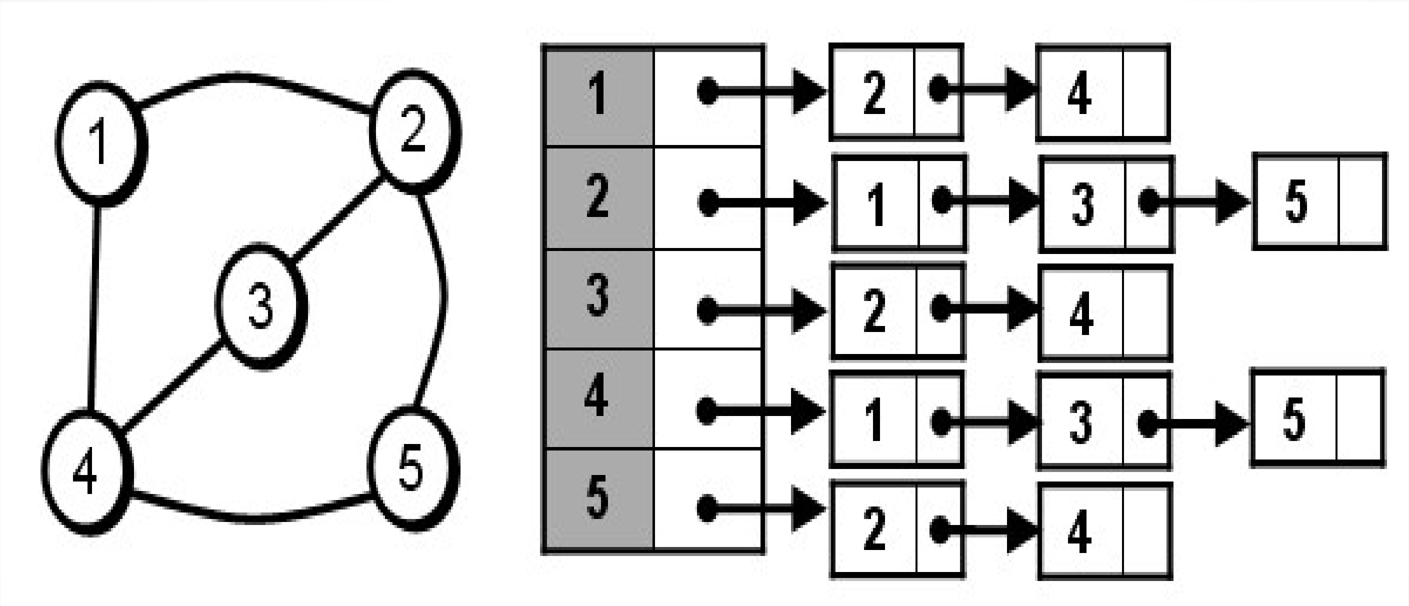
Grafos. Lista de Adyacencias

Almacenamos los adyacentes a cada nodo en una lista enlazada.

Entonces:

Hay un nodo en la lista para el vértice i si y solo si hay una arista del vértice i al vértice j.

Grafos. Lista de Adyacencias



Grafos. Lista de Adyacencias

Con está implementación solucionamos el “tema espacio”.

Una desventaja es que puede llevar O(N) determinar si existe una arista del vértice i al j, ya que puede haber N vértices en la lista de adyacencias del vértice i.

Grafo. Estructura Interna

```
from listas import SinglyLinkedList
from dataclasses import dataclass
from typing import Any

class AdjacencyListGraph:
    @dataclass
    class _Vertex:
        value: Any
        Edges:SinglyLinkedList

    @dataclass
    class _Edge:
        to: int
        weight: int

    __slots__ = ['_vertices']

    def __init__(self):
        self._vertices = []
```

Lista con los valores de los
Vértices y sus listas
de adyacentes

La lista de adyacentes
Va a tener el vértice adyacente
+ el peso

Grafo. Implementación

```
def is_empty(self):
        return len(self._vertices) == 0

def __len__(self):
        return len(self._vertices)

def vertices(self):
        return list(range(len(self._vertices)))
```

Grafo. Implementación

```
def get_vertex(self, vertex):

        return self._vertices[vertex].value



def set_vertex(self, vertex, value):

        self._vertices[vertex].value = value



def add_vertex(self, value):
    vertex = len(self._vertices)
    self._vertices.append(self._Vertex(value, SinglyLinkedList()))
    return vertex
```

Grafo. Implementación

```
def erase_vertex(self, vertex):

        assert 0 <= vertex < len(self._vertices)
        del self._vertices[vertex]
        for _, edges in self._vertices:
            prev = edges.before_begin()
            p = prev.next()
            while p != edges.end():
                if p.value.to == vertex:
                    p = edges.erase_after(prev)
                else:
                    if p.value.to > vertex:
                        p.value.to -= 1
                    prev.advance()
                    p.advance()
```

Borramos el vértice
De la lista y tenemos que
Actualizar todas las listas
De adyacentes de los
demás vértices

Grafo. Implementación

```
def __getitem__(self, key):
        if isinstance(key, tuple):
            return self.get_edge(key[0], key[1])
        return self.get_vertex(key)


def __setitem__(self, key, value):
        if isinstance(key, tuple):
            return self.set_edge(key[0], key[1], value)
        return self.set_vertex(key, value)


def __delitem__(self, key):
        if isinstance(key, tuple):
            self.erase_edge(key[0], key[1])
        else:
            self.erase_vertex(key)
```

Grafo. Implementación

```
def get_edge(self, from_, to):

        for edge in self._vertices[from_].edges:
            if edge.to == to:
                return edge.weight
        raise IndexError('No existe esa arista')



def set_edge(self, from_, to, weight):

        for edge in self._vertices[from_].edges:
            if edge.to == to:
                edge.weight = weight
                return
        raise IndexError('No existe esa arista')
```

Grafo.   Implementación

```
def add_edge(self, from_, to, weight):

        edges = self._vertices[from_].edges
        prev = edges.before_begin()
        p = prev.next()
        while p != edges.end():
            if p.value.to == to:
                p.value.weight = weight
                return
            elif p.value.to > to:
                break
            prev.advance()
            p.advance()
        edges.insert_after(prev, self._Edge(to, weight))
```

Buscamos la posición
Porque mantenemos
La lista ordenada

Grafo.   Implementación

```
def erase_edge(self, from_, to):

        edges = self._vertices[from_].edges
        prev = edges.before_begin()
        p = prev.next()
        while p != edges.end():
            if p.value.to == to:
                edges.erase_after(prev)
                return True
            elif p.value.to > to:
                break;
            prev.advance()
            p.advance()
        return False
```

Grafo. Implementación

```
def is_adjacent(self, from_, to):
        for edge in self._vertices[from_].edges:
            if edge.to == to:
                return True
            elif edge.to > to:
                break
        return False


def adjacents(self, from_):
        result = []
        for to, _ in self._vertices[from_].edges:
            result.append(to)
        return result
```

Grafo. Implementación

```
def __eq__(self, other):
        if len(self._vertices) != len(other._vertices):
            return False

        for vx, vy in zip(self._vertices, other._vertices):
            if vx.value != vy.value or vx.edges != vy.edges:
                return False

        return True


def __repr__(self):
        result = '( '
        for v in self.vertices():
            for w in self.adjacents(v):
                result += '{0}-({2})->{1}, '.format(v, w,
self.get_edge(v, w))
        return result + ') : AdjacencyListGraph'
```

Grafo. Implementación

```
def copy(self):

        result = AdjacencyListGraph()
        for vertex in self._vertices:
            new_list = SinglyLinkedList()
            p = new_list.before_begin()
            for edge in vertex.edges:
                new_edge = self._Edge(edge.to, edge.weight)
                p = new_list.insert_after(p, new_edge)
            new_vertex = self._Vertex(vertex.value, new_list)
            result._vertices.append(new_vertex)
        return result
```

No podemos usar el
Copy de las listas
Porque quedan las
Referencias a los mismos
valores

Grafos. Comparación

¿Qué implementación es mejor?

Dependiendo de la aplicación las operaciones más frecuentes serán:

1- dados dos vértices i y j determinar si existe arista

2-Encontrar los vértices adyacentes a un vértice i

Grafos. Comparación

-La Operación 1- es más eficiente con la Matriz ya que solo debemos ver A[i,j]. Con listas de adyacentes deberíamos recorrer la lista del vértice i.

-La Operación 2 es más eficiente con la Lista ya que con la matriz deberíamos recorrer toda i-ésima columna. Para un grafo de N vértices la i-ésima columna de la matriz siempre tiene N entradas, mientras que la lista puede tener menos.

Grafos. Comparación

¿Qué implementación es mejor con respecto al espacio?

En la Matriz cada posición tiene un valor booleano o entero, mientras que la Lista contiene el valor y el puntero.

Pero la Matriz tiene N^2 entradas, mientras que en la lista el número de entradas es el número de aristas del grafo (en general $< N^2$)

→ La Lista requiere menos espacio

Recorridos

Búsqueda en Profundidad DFS

Grafos. DFS

-Vamos a tener una estructura para saber que vértices ya fueron visitados.

DFS trabaja seleccionando un vértice start de D como vértice de partida.

start se marca como visitado, después se recorre cada vértice no visitado adyacente a start usando DFS.

Una vez recorridos todos los adyacentes de start si algún vértice de D quedó sin visitar, se selecciona el nuevo vértice y se repite el proceso

Grafo. DFS

```
def dfs(graph, start):

    visited = set()
    pending = list()
    vertices = [start] + graph.vertices()

    for vertex in vertices:
        if vertex in visited:
            continue

        pending.append(vertex)
        visited.add(vertex)
        while pending:
            current = pending.pop()
            yield current, graph.get_vertex(current)

            for adjacent in graph.adjacents(current):
                if adjacent not in visited:
                    pending.append(adjacent)
                    visited.add(adjacent)
```

Usamos la lista como
Una **pila** para almacenar
Los nodos pendientes

Búsqueda en Amplitud BFS

Grafos. BFS

Se denomina en amplitud porque desde cada vértice start que se visita se busca en forma tan amplia como sea posible, visitando todos los vértices adyacentes a start.

Es decir no vamos a “disparar” el recorrido desde ningún vértice adyacente a start hasta haber visitado todos los adyacentes a start

Es similar al recorrido de un árbol por niveles.

La implementación iterativa utiliza una cola.

Grafo. BFS

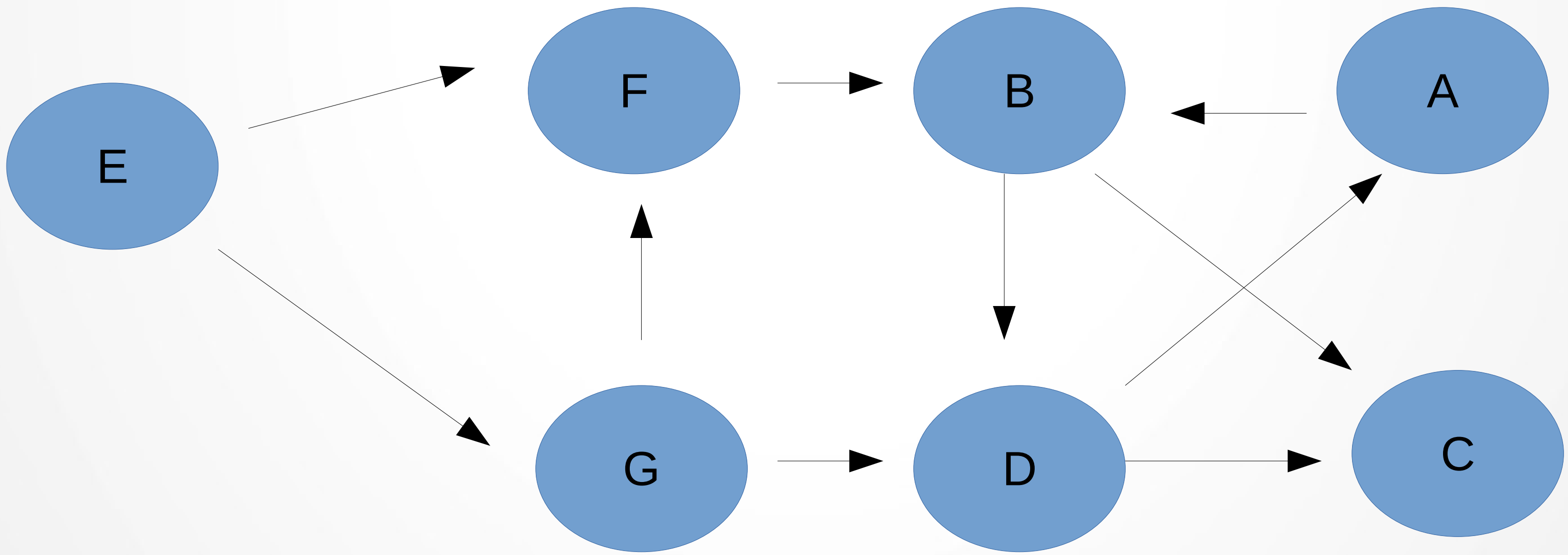
```
def bfs(graph, start):
    from collections import deque

    visited = set()
    pending = deque()
    vertices = [start] + graph.vertices()

    for vertex in vertices:
        if vertex in visited:
            continue

        pending.append(vertex)
        visited.add(vertex)
        while pending:
            current = pending.popleft()
            yield current, graph.get_vertex(current)

            for adjacent in graph.adjacents(current):
                if adjacent not in visited:
                    pending.append(adjacent)
                    visited.add(adjacent)
```

Usamos **cola doblemente terminada** para almacenar Los nodos pendientes

Grafos.

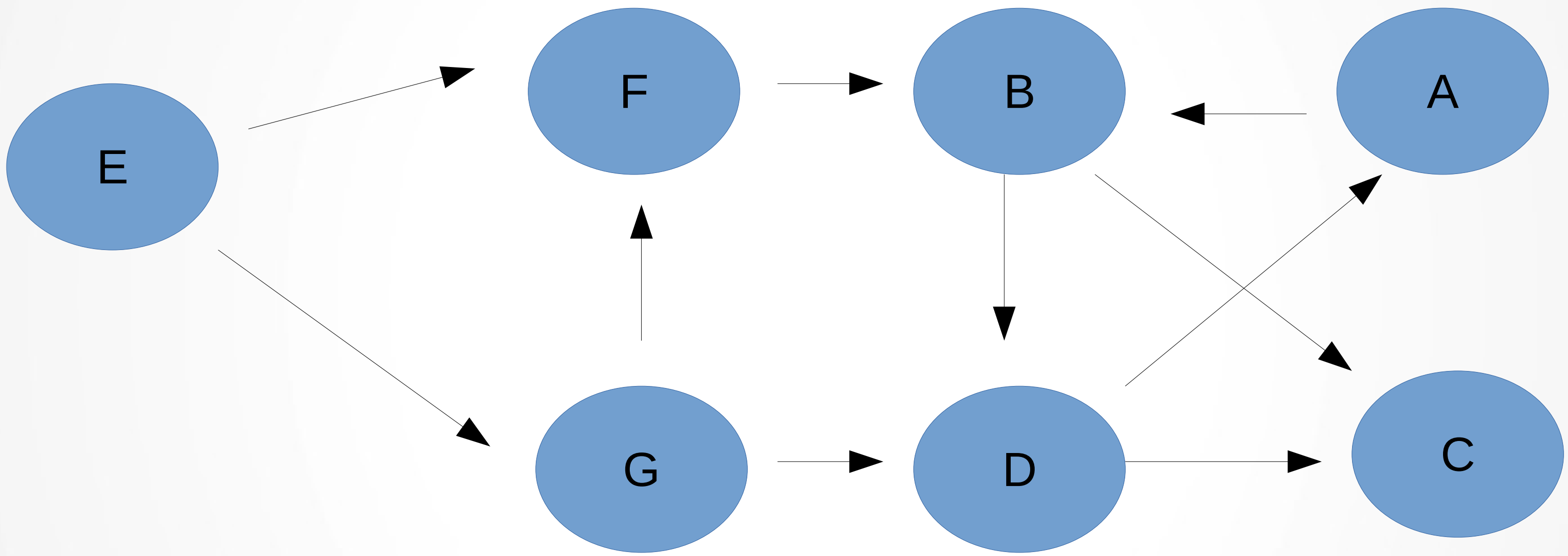
La complejidad de bfs es la misma que para dfs.

Vemos un ejemplo:



Grafos.

Vemos un ejemplo:

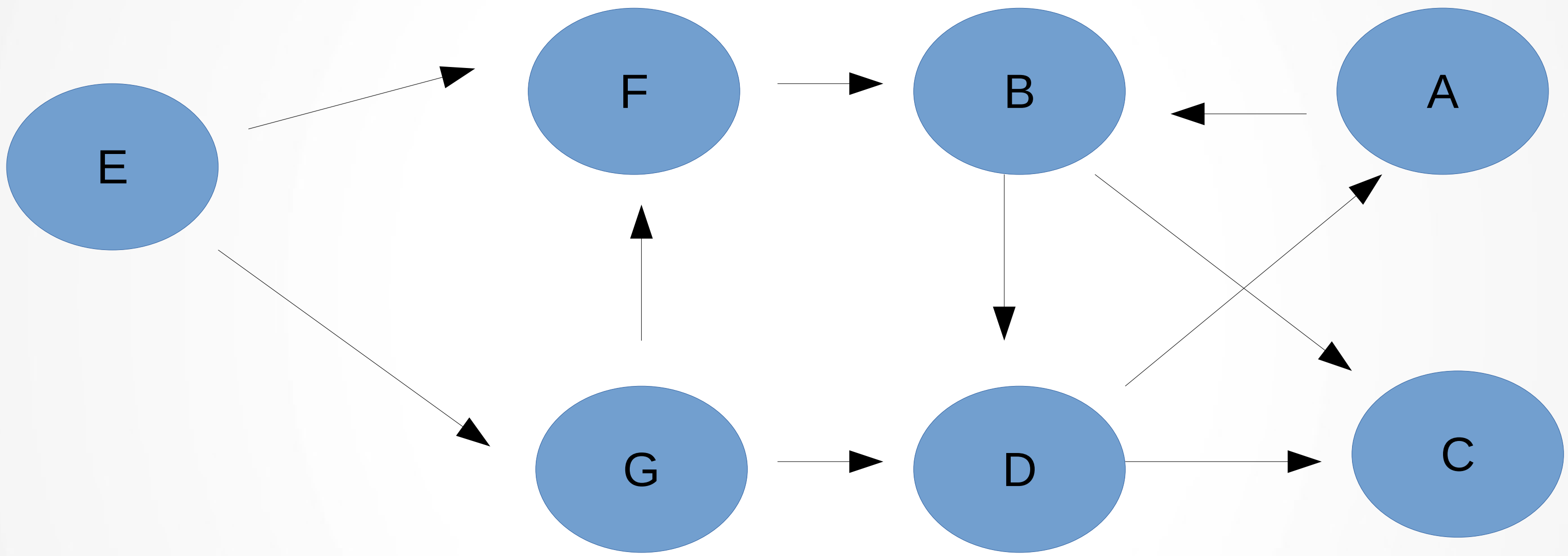


Con dfs(en Profundidad) y comenzando desde E:

Un posible recorrido sería E-F-B-C-D-A-G

Grafos.

Vemos un ejemplo:



Con bfs(en Amplitud) y comenzando desde E:

Un posible recorrido sería E-F-G-B-D-C-A

Grafos. Encontrar un camino

Dado un digrafo ciclico

¿Cómo encontrarían un camino entre dos nodos dados from y to?

-Podemos usar dfs o bfs
-Es posible almacenar el camino en una estructura para poder retornarlo
-Nodo inicio from
-Si en algún momento llegamos a to entonces Encontramos un camino
-No debemos recorrer todo el grafo. Solo el subgrafo Conectado al origen

Grafo. Existe Camino?

```
def connected(graph, from, to):
    visited = set()
    pending = deque()
    pending.append(from)
    visited.add(from)
    while pending:
        current = pending.popleft()
        if current == to:
            return True
        for adj in graph.adjacents(current):
            if adj not in visited:
                pending.append(adj)
                visited.add(adj)
    return False
```

Usamos BFS para el recorrido
Pero también podríamos
haber usado
DFS

Orden Topológico

Grafos. O. Topologico

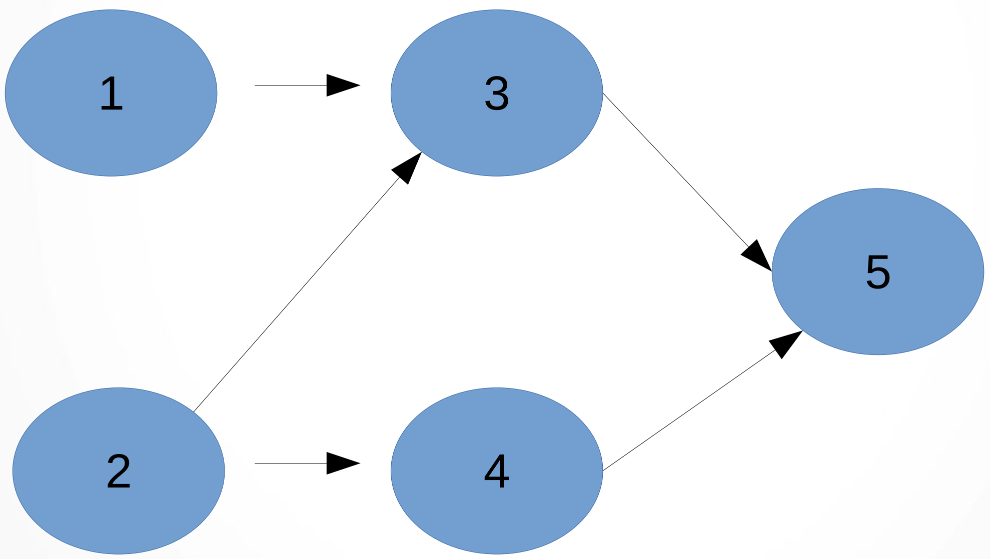
Es un proceso de asignación de un orden lineal a los vértices de un grafo dirigido aciclico

Tal que si existe un arco del vértice i al vértice j, i aparece antes que j en el ordenamiento lineal.

Grafos. S. Topologico

Vemos un ejemplo: 1, 2, 3, 4, 5

Podría ser también: 2, 1, 3, 4, 5



Grafo. O. Topologico

```
from collections import deque, defaultdict

def topological_sort(graph):

    inputs = defaultdict(int)
    for vertex in graph.vertices():
        for adjacent in graph.adjacents(vertex):
            inputs[adjacent] += 1

    pending = deque()
    for vertex in graph.vertices():
        if inputs[vertex] == 0:
            pending.append(vertex)

    ...
```

En caso de no existir
El valor asociado
A la clave le asigna
Valor 0 para los int

En inputs vamos a tener “cuantas flechas llegan” a cada vértice
En pending tenemos los vértices que ya no tienen flechas de entrada

Grafo. O. Topologico

```
...

result = []

while pending:
    vertex = pending.popleft()
    result.append(vertex)

    for adjacent in graph.adjacents(vertex):
        inputs[adjacent] -= 1
        if inputs[adjacent] == 0:
            pending.append(adjacent)

for i in inputs.values():
    if i != 0:
        raise ValueError('El grafo dado tiene ciclo(s).')

return result
```

Grafos. Camino Mínimo

Y si ahora por ejemplo quisiéramos encontrar el camino "más corto" entre v y w...

El Algoritmo de Dijkstra nos ayuda a encontrar los caminos "más cortos" desde un nodo de inicio hasta cada uno de los otros nodos del grafo...

Camino más corto

(Algoritmo de Dijkstra)

Grafos. Camino Mínimo

Dado un digrafo acíclico en el cual cada arista tiene un peso no negativo.

La longitud del camino es la suma de los costos de todos los arcos del camino.

Sea **X** el nodo inicial, un vector **D** de tamaño **N** guardará al final del algoritmo las distancias mínimas desde **X** al resto de los **N** nodos.

Grafos. Camino Mínimo

1) Inicializamos todas las distancias en **D** con un valor infinito (o, si no fuera posible, el máximo representable) ya que al principio son todas desconocidas, exceptuando la de **x** que ponemos en 0 (debido a que la distancia de **x** a **x** es 0).

2) Sea **a** = **x** (tomamos **a** como nodo actual).

3) Recorremos todos los nodos adyacentes de **a**, excepto aquellos nodos marcados como vistos. Llamaremos a estos nodos no marcados $\mathbf{v}_i$.

Grafos. Camino Mínimo

4) Para el nodo actual, calculamos la distancia tentativa desde dicho nodo a sus vecinos con la siguiente fórmula: $dt(\mathbf{v}_i) = D[\mathbf{a}] + d(\mathbf{a}, \mathbf{v}_i)$.

Si la distancia tentativa es menor que la distancia almacenada en el vector, actualizamos el vector con esta distancia tentativa.

Es decir: Si $dt(\mathbf{v}_i) < D[\mathbf{v}_i] \rightarrow D[\mathbf{v}_i] = dt(\mathbf{v}_i)$

Grafos. Camino Mínimo

5) Marcamos como visto al nodo **a**.

6) Mientras existan nodos no marcados como vistos, tomamos como próximo nodo actual al de menor valor en **D** (puede hacerse almacenando los valores en una cola de prioridad) y volvemos al paso 3).

Ver visualización en:
http://www.cs.usfca.edu/~galles/visualization/Dijkstra.html

Grafo. Dijkstra.

```
def shortest_path(graph, start, max_cost=float('inf')):
    Data = namedlist('Data', 'cost visited prev')

    data = dict()
    for vertex in graph.vertices():
        data[vertex] = Data(max_cost, False, start)

    data[start].cost = 0
    for adjacent in graph.adjacents(start):
        data[adjacent].cost = graph.get_edge(start, 
adjacent)

    ...
```

Grafo. Dijkstra.

```
def shortest_path(graph, start, max_cost=float('inf')):
    ..
 while True:
  min_cost = max_cost
  for vertex in graph.vertices():
        cost, visited, _ = data[vertex]
        if not visited and cost < min_cost:
            min_cost = cost
            min_vertex = vert
  if min_cost == max_cost:
        break
  data[min_vertex].visited = True
  accumulated = data[min_vertex].cost
  for adjacent in graph.adjacents(min_vertex):
   entry = data[adjacent]
   total_cost = accumulated + graph.get_edge(min_vertex, adjacent)
        if total_cost < entry.cost:
            entry.cost = total_cost
            entry.prev = min_vertex
```

Grafo. Dijkstra.

```
result = dict()
for vertex in graph.vertices():
    path = []
    current = vertex
    while data[current].visited:
        path.append(current)
        if current == start:
            break
        current = data[current].prev
    result[vertex] = (data[vertex].cost, path[::-1])

return result
```

Grafo. Dijkstra.

La complejidad del algoritmo es $O(N^2)$ porque se visita una única vez a cada nodo ($O(N)$) y por cada nodo visitado, es necesario encontrar el nodo que tiene la mínima distancia almacenada en D (otra ves, $O(N)$), por lo que en total es de $O(NxN)$ o $O(N^2)$.

Grafo. Dijkstra.

La complejidad **puede reducirse** usando una mejor manera para encontrar el mínimo en cada iteración (con orden O(log N) en lugar de O(N)). Así, la complejidad total del algoritmo sería de O(N log(N)).

Para esto puede usarse un montículo o heap, para que en cada iteración se disponga del mínimo en O(log N) operaciones.