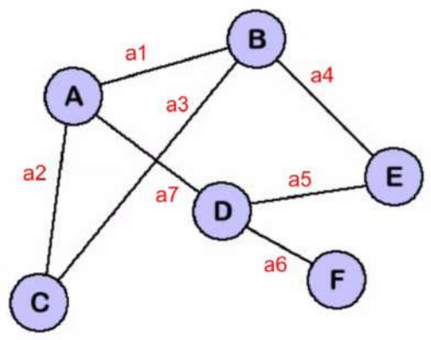
GRAFOS

E = Aristas / V = Vertices

REPRESENTACIONES

Matriz de incidencia

	A	B	C	D	E	F
a1	1	1	0	0	0	0
a2	1	0	1	0	0	0
a3	0	1	1	0	0	0
a4	0	1	0	0	1	0
a5	0	0	0	1	1	0
a6	0	0	0	1	0	1
a7	1	0	0	1	0	0

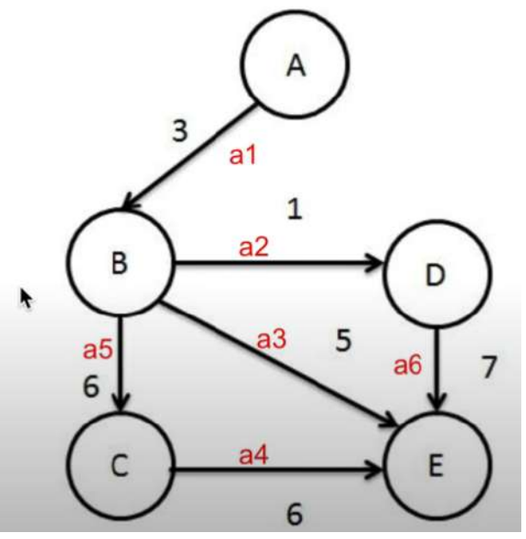


Costos

- ¿En Espacio? O(V*E)
- ¿Agregar un vértice? O(V*E)
- ¿Agregar una arista? O(V*E)
- ¿Ver si dos vértices son adyacentes? O(E)
- ¿Obtener los adyacentes de un vértice? O(E)

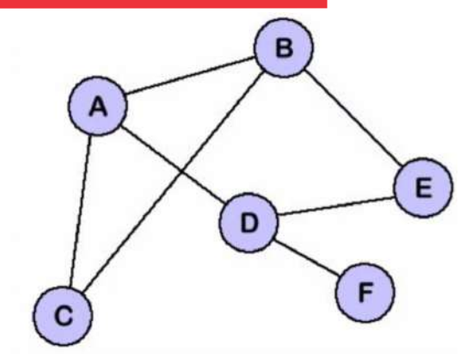
Matriz de incidencia (II) (con pesos y dirigido)

	A	B	C	D	E
a1	-3	3	0	0	0
a2	0	-1	0	1	0
a3	0	-5	0	0	5
a4	0	0	-6	0	6
a5	0	-6	6	0	0
a6	0	0	0	-7	7



Matriz de Adyacencia

	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	1	0	1	0
C	1	1	0	0	0	0
D	1	0	0	0	1	1
E	0	1	0	1	0	0
F	0	0	0	1	0	0

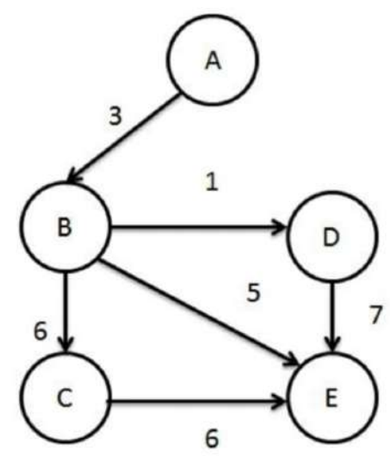


Costos

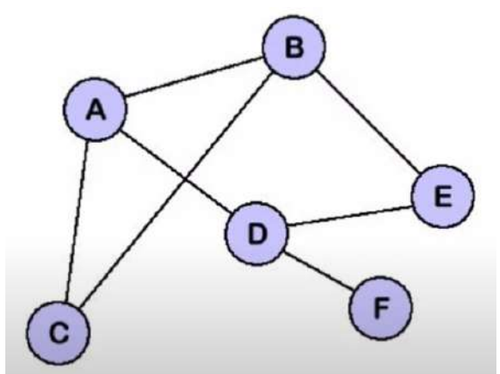
- ¿En Espacio? O(V^2)
- ¿Agregar un vértice? O(V^2)
- ¿Agregar una arista? O(1)
- ¿Ver si dos vértices son adyacentes? O(1)
- ¿Obtener los adyacentes de un vértice? O(V)

Matriz de Adyacencia (II) (con pesos y dirigido)

	A	B	C	D	E
A	0	3	0	0	0
B	0	0	6	1	5
C	0	0	0	06	
D	0	0	0	0	7
E	0	0	0	0	0



Listas de adyacencia



elem | adyacentes.

B → A → C → E
A → B → D → C
D → E → F → A
E → .....

Costos

- ¿En Espacio? O(V+E)
- ¿Agregar un vértice? O(1) u O(V)
- ¿Agregar una arista? O(V)
- ¿Ver si dos vértices son adyacentes? O(V)
- ¿Obtener los adyacentes de un vértice? O(V)

Matriz de Adyacencia ++ Listas de adyacencia (II)

Cuánto cuesta:

- ¿En Espacio? O(V+E)
- ¿Agregar un vértice? O(1)
- ¿Agregar una arista? O(1)
- ¿Ver si dos vértices son adyacentes? O(1)
- ¿Obtener los adyacentes de un vértice? O(1) u O(V)

(Diccionario de diccionario)

RECORRIDOS

En Anchura : BFS (dirigido y no dirigido)

Codigo Complejidad = O(V+E)

```
def bfs(grafo, origen):
     visitados = set()
     padres = {}
     orden = {}
     q = Cola()
     visitados.agregar(origen)
     padres[origen] = None
     orden[origen] = 0
     q.encolar(origen)
while !q.esta_vacia():
     v = q.desencolar()
     for w in grafo.adyacentes(v):
          if w not in visitados:
               visitados.agregar(w)
               padre[w] = v
               orden[w] = orden[v] + 1
               q.encolar(w)
return padre, orden
```

En Profundidad: DFS

Codigo Complejidad = O(V+E)

```
def dfs(grafo, v, visitados, padre, orden):
      visitados.agregar(v)
      for w in grafo.adyacentes(v):
            if w not in visitados:
                  padre[w] = v
                  orden[w] = orden[v] + 1
                  dfs(grafo, w, visitados, padre, orden)

def recorrido_dfs_completo(grafo, origen):
      visitados = set()
      padres = {}
      orden = {}
      padre[origen] = None
      orden[padre] = 0
      dfs(grafo, origen, visitados, padre, orden)
      return padre, orden
```

ORDEN TOPOLOGICO

Tareas a realizar en general (algunas deben hacerse antes que otras)
Se usan algoritmos de tipo BFS y DFS para resolver los distintos problemas

Algoritmo de Tarjan

Usos

Componentes Fuertemente Conexas

Codigo

```
def cfc(grafo, v, visitados, pila, apilados, orden, mas_bajo, cfcs, *indice):
      visitados.add(v)
      pila.apilar(v)
      apilados.add(v)
      Mas_bajo[v] = orden[v]
      for w in grafo.adyacentes(v):
            if w not in visitados:
                  orden[w] = *indice + 1
                  *indice++
                  cfc(grafo, w, visitados, pila, apilados, orden, mas_bajo, cfcs)
                  mas_bajo[v] = min(mas_bajo[v], mas_bajo[w])
            else if w in apilados:
                  mas_bajo[v] = min(mas_bajo[v], orden[w])
      if mas_bajo[v] == orden[v]:
            nueva_cfc = []
            do:
                  w = pila.desapilar()
                  apilados.remove(w)
                  nueva_cfc.append(w)
            while w != v
            cfcs.append(nueva_cfc)
```

Complejidad = O(V+E)

CAMINOS MINIMOS

- Grafo no pesado - BFS
  (puede ser a un destino especifico o hacia todos los vertices)

- Grafos pesados - Algoritmo de Dijkstra ⟶ Complejidad = O(E Log V)

  Una modificacion sobre BFS
  Vamos a usar un Heap de minimos + actualizar distancias

Codigo

```
def camino_minimo(grafo, origen):
      dist = {}
      padre = {}
      for v in grafo:
            distancia[v] = infinito
      dist[origen] = 0
      padre[origen] = None
      q = heap_crear()
      q.encolar(origen, 0)
      while not q.esta_vacia():
            v = q.desencolar()
            for w in grafo.adyacentes(v):
                  if dist[v] + grafo.peso_union(v, w) < dist[w]:
                        dist[w] = dist[v] + grafo.peso_union(v, w)
                        padre[w] = v
                        q.encolar(w, dist[w]) # o: q.actualizar(w, dist[w])
      return padre, distancia
```

(dirigido y no dirigido)

Grafo conexo

Pesos Positivos

Se puede usar para un vertice en particular

- Grafos dirigidos con pesos negativos - Algoritmo de Bellman-Ford

Codigo

```
def camino_minimo_bf(grafo, origen):
      dist = {}
      padre = {}
      for v in grafo:
            distancia[v] = infinito
      dist[origen] = 0
      padre[origen] = None
      aristas = obtener_aristas(grafo)
      for i in range(len(grafo)):
            for v, w, peso in aristas:
                  if dist[v] + peso < dist[w]:
                        padre[w] = v
                        dist[w] = dist[v] + peso

      for v, w, peso in aristas:
            if dist[v] + peso < dist[w]:
                  return None # Hay un ciclo negativo (lanzar excep)
      return padre, dist
```

Complejidad = O(V * E)

ARBOLES DE TENDIDO MINIMO

- Un grafo con los mismos vertices, con la minima cantidad de aristas para que se mantenga conexo

- MST - Algoritmo de Prim

Pasos

Complejidad = O(E Log V)

1. Comienza en un vértice aleatorio, encola en un heap todas sus aristas.
2. El vértice origen queda como visitado
3. Mientras el heap no está vacío, saca una arista.
4. Si ambos vértices de la arista fueron visitados, se descarta la arista (formaría un ciclo).
5. Caso contrario, se agrega la arista al árbol, y se encolan todas las aristas del nuevo vértice visitado.
6. Volvemos al paso 3.

Codigo

```
def mst_prim(grafo):
    vertice = grafo.vertice_aleatorio()
    visitados = set()
    visitados.agregar(vertice)
    q = heap_crear()
    arbol = grafo_crear(grafo.obtener_vertices())
    for w in grafo.adyacentes(v):
        q.encolar((v, w) , grafo.peso_arista(v, w))
    while not q.esta_vacia():
        (v, w) = q.desencolar()
        if w in visitados:
            continue
        arbol.agregar_arista(v, w, grafo.peso_arista(v, w))
        visitados.agregar(w)
        for u in grafo.adyacentes(w):
            if u not in visitados: q.encolar((w, u), grafo.peso_arista(w, u))
return arbol
```

- MST - Algoritmo de Kruskal

Se usa el TDA UnionFind ("Todo O(1)")

Pasos

1. Ordenamos las aristas de menor a mayor
2. Por cada arista (en ese orden), si los vértices no están en una misma componente conexa (dentro del árbol), agregamos la arista, y ahora ambos vértices están en la misma componente conexa.

Complejidad = O(E Log V)

Codigo

```
def mst_kruskal(grafo):
    conjuntos = UnionFind(grafo.obtener_vertices())
    aristas = sort(obtener_aristas(grafo))
    arbol = grafo_crear(grafo.obtener_vertices())
    for a in aristas:
        v, w, peso = a
        if conjuntos.find(v) == conjuntos.find(w):
            continue
        arbol.agregar_arista(v, w, peso)
        conjuntos.union(v, w)
    return arbol
```