Resumen AYED

Arboles binarios

- ***Camino***: desde n_1 hasta n_k, es una secuencia de nodos n_1, n_2, ….,n_k tal que n_i es el padre de n_{i+1}, para $1 \leq i < k$.
  - La longitud del camino es el número de aristas, es decir k-1.
  - Existe un camino de longitud cero desde cada nodo a sí mismo.
  - Existe un único camino desde la raíz a cada nodo.

- ***Profundidad***: de n_i es la longitud del único camino desde la raíz hasta n_i.
  - La raíz tiene profundidad cero.

- ***Grado*** de n_i es el número de hijos del nodo n_i .

- ***Altura*** de n_i es la longitud del camino más largo desde n_i hasta una hoja.
  - Las hojas tienen altura cero.
  - La altura de un árbol es la altura del nodo raíz.

- ***Ancestro/Descendiente***: si existe un camino desde n_1 a n_2, se dice que n_1 es ancestro de n_2 y n_2 es descendiente de n_1.

Árbol binario lleno: Dado un árbol binario T de altura h,
diremos que T es lleno si cada nodo interno tiene grado 2
y todas las hojas están en el mismo nivel (h).

- ***Cantidad de nodos en un árbol binario lleno***:

  Sea T un árbol binario lleno de altura ***h***, la cantidad de nodos N es $(2^{h+1} - 1)$

- ***Árbol binario completo***: Dado un árbol binario T de altura h, diremos que T es completo si es lleno de altura h-1 y el nivel h se completa de izquierda a derecha.

- ***Cantidad de nodos en un árbol binario completo***:
  Sea T un árbol binario completo de altura ***h***, la cantidad de nodos N varía entre (2^{h}) y $(2^{h+1} - 1)$

Un árbol binario lleno es a l vez un árbol completo, pero, un árbol completo puede no ser lleno

Recorridos

➢ Preorden
Se procesa primero la raíz y luego sus hijos, izquierdo y derecho.
➢ Inorden
Se procesa el hijo izquierdo, luego la raíz y último el hijo derecho
➢ Postorden
Se procesan primero los hijos, izquierdo y derecho, y luego la raíz
➢ Por niveles
Se procesan los nodos teniendo en cuenta sus niveles, primero la raíz, luego los hijos, los hijos de éstos, etc.

Árbol de expresión

Construcción de un árbol de expresión a partir de una expresión postfija

➢empezar de adelante para atrás
➢ Mientras sigan habiendo caracteres
  ➢Si es un operando lo apilo
  ➢Si es un operador lo junto con los 2 operandos anteriores

    si [3,z,*] → (3*z)

Construccion de un arbol de expresion a partir de una expresion prefija

➢empezar de atras para adelante
➢ Mientras sigan habiendo caracteres
➢Si es un operando lo apilo
➢Si es un operador lo junto con los 2 operandos anteriores pero de forma invertida
si [*,3,z] → (z*3)

Construccion de un arbol de expresion a partir de una expresion infija a prefija

Se empieza de atras para adelante

cuando hay un operando se pone en el resultado

cuando hay un parentesis de cierre “)” se apila

cuando hay un operador se apila

cuando hay un parentesis de apertura“(” se desapila todo hasta encontrar el parentesis de cierre y al resultado va el operando que quedó en medio

Construccion de un arbol de expresion a partir de una expresion infija a posfija

Se empieza de adelante para atras

cuando hay un operando se pone en el resultado

cuando hay un parentesis de apertura“(” se apila

cuando hay un operador se apila

cuando hay un parentesis de cierre“)” se desapila todo hasta encontrar el parentesis de apertura y al resultado va el operando que quedó en medio

Árboles generales

- ***Grado*** de n_i es el número de hijos del nodo n_i .
  - Grado del árbol es el grado del nodo con mayor grado.
- ***Altura*** de n_i es la longitud del camino más largo desde n_i hasta una hoja.
  - Las hojas tienen altura **cero**.
  - La altura de un árbol es la altura del nodo raíz.
- ***Profundidad / Nivel***: de n_i es la longitud del único camino desde la raíz hasta n_i.
  - La raíz tiene profundidad o nivel **cero**.
- ***Árbol lleno***: Dado un árbol T de grado ***k*** y altura ***h***, diremos que T es *lleno* si cada nodo interno tiene grado k y todas las hojas están en el mismo nivel (h).
- ***Árbol completo***: Dado un árbol T de grado ***k*** y altura ***h***, diremos que T es *completo* si es lleno de altura h-1 y el nivel h se completa de izquierda a derecha.
- ***Cantidad de nodos en un árbol lleno***:

  Sea T un árbol lleno de grado ***k*** y altura ***h***, la cantidad de nodos N es $\mathbf{(k^{h+1} - 1) / (k-1)}$ ya que:

- ***Cantidad de nodos en un árbol completo***:

Sea T un árbol completo de grado ***k*** y altura ***h***, la cantidad de nodos N varía entre $(k^h + k - 2) / (k-1)$ y $(k^{h+1} - 1) / (k-1)$ ya que …

Recorridos

➢ Preorden
Se procesa primero la raíz y luego los hijos
➢ Inorden
Se procesa el primer hijo, luego la raíz y por último los restantes hijos
➢ Postorden
Se procesan primero los hijos y luego la raíz
➢ Por niveles
Se procesan los nodos teniendo en cuenta sus niveles, primero la raíz, luego los hijos, los hijos de éstos, etc.

Heap

Es una implementación de colas de prioridad que no usa punteros y permite implementar ambas operaciones con O(log N) operaciones en el peor caso

Una heap es un árbol binario completo

✔️ En un árbol binario lleno de altura h, los nodos internos tienen exactamente 2 hijos y las hojas tienen la misma profundidad

✔️ Un árbol binario completo de altura h es un árbol binario lleno de altura h-1 y en el nivel h, los nodos se completan de izquierda a derecha

✓ El número de nodos n de un árbol binario completo de altura h, satisface:

$$2^h \leq n \leq (2^{h+1}-1)$$

Demostración:

- Si el árbol es lleno, $n= 2^{h+1}-1$
- Si no, el árbol es lleno en la altura $h-1$ y tiene por lo menos un nodo en el nivel h:

$n= 2^{h-1+1}-1+1=2^h$

La altura h del árbol es de **O(log n)**

Propiedad de orden

➢MinHeap
• El elemento mínimo está almacenado en la raíz
• El dato almacenado en cada nodo es menor o igual al de sus hijos

➢MaxHeap
• Se usa la propiedad inversa

Operacion: Insert

➢ El dato se inserta como último ítem en la heap

➢ La propiedad de la heap puede ser violada

➢ Se debe hacer un filtrado hacia arriba para restaurar la propiedad de orden

Operación: DeleteMin
➢ Guardo el dato de la raíz
➢ Elimino el último elemento y lo almaceno en la raíz
➢ Se debe hacer un filtrado hacia abajo para restaurar la propiedad de orden

Algoritmo BuildHeap

➢ primero se agregan los elementos como aparecen en el vector

➢Se empieza filtrando desde el elemento que está en la posición (tamaño/2) o se empieza desde la ultima raiz con al menos un hijo:

➢se filtran los nodos que tienen hijos

➢el resto de los nodos son hojas

Por ejemplo el ultimo elemento está en la posicion 10, entonces empezarias filtrando(segun el orden) desde la pos 5, y de ahi empezas a filtrar los nodos anteriores a 5(4,3,2,1) hasta llegar a la pos 1

Teorema:

En un árbol binario lleno de altura h que contiene $2^{h+1} - 1$ nodos, la suma de las alturas de los nodos es: $2^{h+1} - 1 - (h + 1)$

Algoritmo HeapSort

Si queres que el vector te quede de menor a mayor primero se construye una max-heap y luego se realiza el algoritmo

Si queres que el vector te quede de mayor a menor primero se construye una min-heap y luego se realiza el algoritmo

➢ Construir una Heap dependiendo el orden(creciente o decreciente) con los elementos que se desean ordenar,
intercambiar el último elemento con el primero, decrementar el tamaño de la heap y filtrar hacia abajo. Usa sólo el espacio de almacenamiento de la heap.

Grafos

Longitud de un camino: número de arcos(aristas) del camino

***Ciclo**: camino desde $v_1, v_2, \ldots, v_k$ tal que $v_1 = v_k$*

Ej: <2,5,4,2> es un ciclo de longitud 3.

Bucle: ciclo de longitud 1

Grafo acíclico: grafo sin ciclos

Un grafo ponderado, pesado o con costos: cada arco o arista tiene asociado un valor o etiqueta

Un grafo no dirigido es conexo si hay un camino entre cada par de vértices.

Un bosque es un grafo sin ciclos.
Un árbol libre es un bosque conexo.
Un árbol es un árbol libre en el que un nodo se ha designado como raíz.

v es alcanzable desde u, si existe un camino de u a v.
Un grafo dirigido se denomina fuertemente conexo si existe un camino desde cualquier vértice a cualquier otro vértice

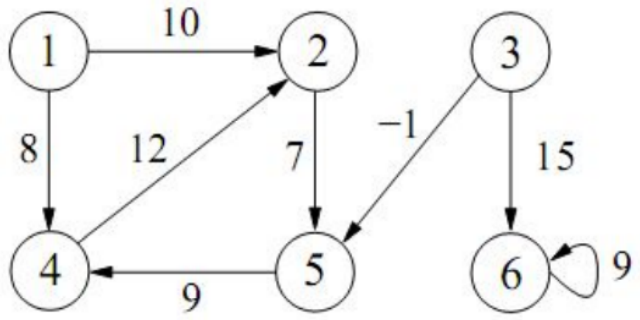
Si un grafo dirigido no es fuertemente conexo, pero el grafo subyacente (sin sentido en los arcos) es conexo, el grafo es débilmente conexo

En un grafo no dirigido, una componente conexa es un subgrafo conexo tal que no existe otra componente conexa que lo contenga.
Es un subgrafo conexo maximal.
Un grafo no dirigido es no conexo si está formado por varias componentes conexas.

En un grafo dirigido, una componente fuertemente conexa, es el máximo subgrafo fuertemente conexo.
Un grafo dirigido es no fuertemente conexo si está formado por varias componentes fuertemente conexas.

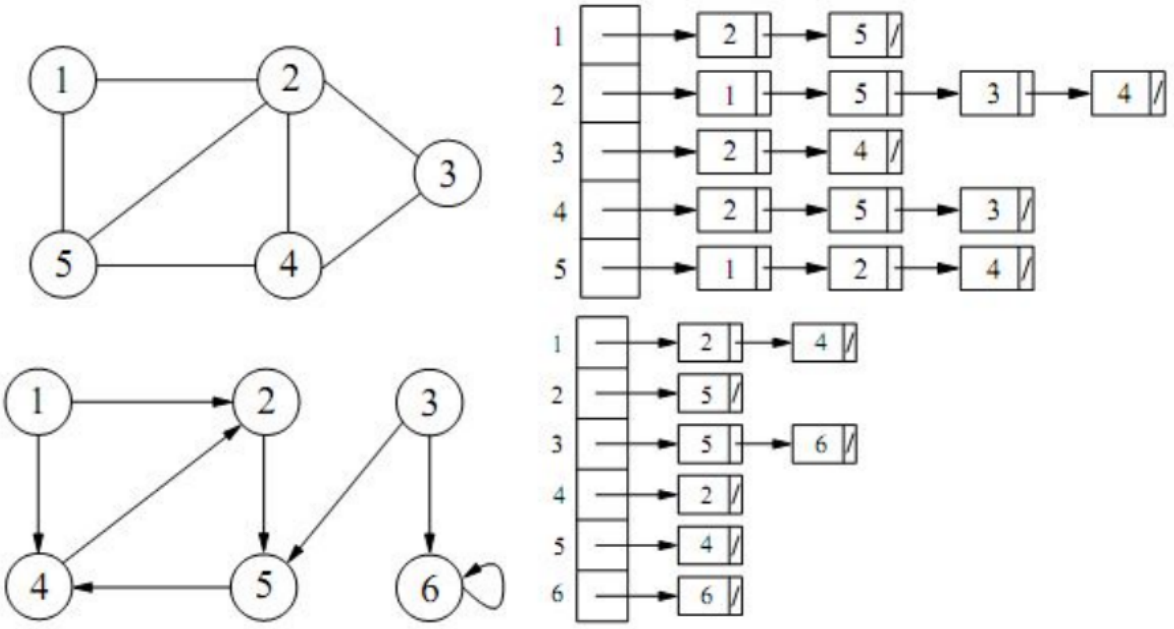
Representaciones

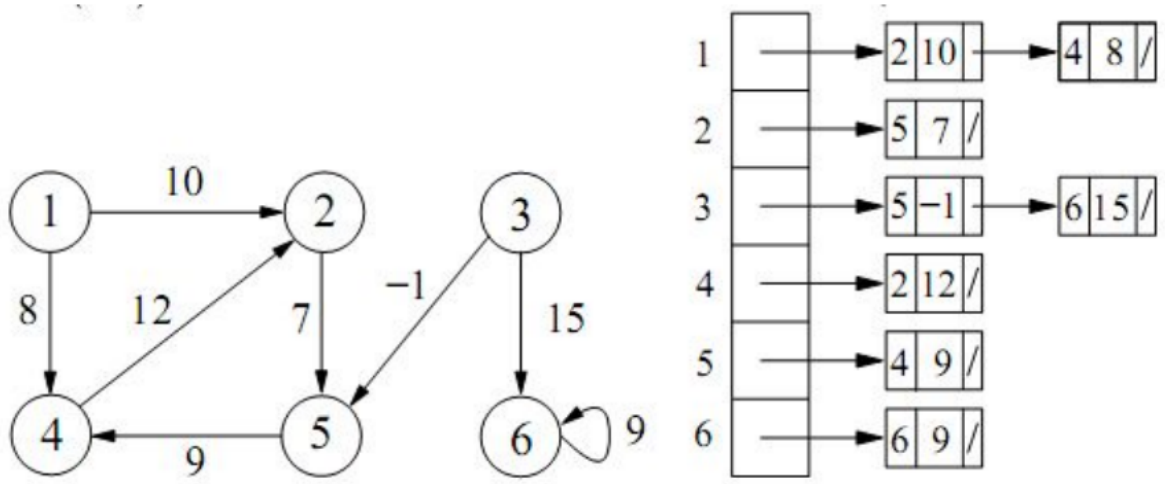
Matriz de adyacencias



	1	2	3	4	5	6
1	0	10	0	8	0	0
2	0	0	0	0	7	0
3	0	0	0	0	−1	15
4	0	12	0	0	0	0
5	0	0	0	9	0	0
6	0	0	0	0	0	9

Lista de Adyacencias





Recorridos

en profundidad: DFS (Depth First Search)

→ Generalización del recorrido preorden de un árbol.
Estrategia:
Partir de un vértice determinado v.
Cuando se visita un nuevo vértice, explorar cada camino que salga de él.
Hasta que no se haya finalizado de explorar uno de los caminos no se comienza con el siguiente.
Un camino deja de explorarse cuando se llega a un vértice ya visitado.
Si existían vértices no alcanzables desde v el recorrido queda incompleto; entonces, se debe seleccionar algún vértice como nuevo vértice de partida, y repetir el proceso.

en amplitud: BFS (Breath First Search)

→ Generalización del recorrido por niveles de un árbol.
Estrategia:
Partir de algún vértice u, visitar u y, después, visitar cada uno de los vértices adyacentes a u.
Repetir el proceso para cada nodo adyacente a u, siguiendo el orden en que fueron visitados.

Bosque de expansión del DFS

- El recorrido no es único: depende del nodo inicial y del orden de visita de los
  adyacentes.
- El orden de visita de unos nodos a partir de otros puede ser visto como un árbol: árbol de expansión (o abarcador) en profundidad asociado al grafo.
- Si aparecen varios árboles: bosque de expansión (o abarcador) en profundidad.

Clasificación de los arcos de un grafo dirigido en el
bosque de expansión de un DFS.
• Arcos tree (del árbol): son los arcos en el bosque depth-first-search, arcos que conducen a vértices no visitados durante la búsqueda.
• Arcos forward: son los arcos u→v que no están en el bosque, donde v es descendiente, pero no es hijo en el árbol.
• Arcos backward: son los arcos u→v, donde v es antecesor en el árbol. Un arco de un vértice a si mismo es considerado un arco back.
• Arcos cross: son todos los otros arcos u→v, donde v no es ni antecesor ni descendiente de u. Son arcos que pueden ir entre vértices del mismo árbol o entre vértices de diferentes árboles en el bosque depth-first-search

Encontrar las componentes fuertemente conexas de un grafo
dirigido: Algoritmo de Kosaraju
Pasos:

1. Aplicar DFS(G) rotulando los vértices de G en post-orden (apilar).
2. Construir el grafo reverso de G, es decir GR
   (invertir los arcos).
3. Aplicar DFS (G^R) comenzando por los vértices de mayor rótulo (tope de la pila).
4. Cada árbol de expansión resultante del paso 3 es una componente fuertemente conexa.
   Si resulta un único árbol entonces el digrafo es fuertemente conexo.

Ordenación topológica - (versión 1)
Pasos generales:

1. Seleccionar un vértice v con grado de entrada cero
2. Visitar v
3. “Eliminar” v, junto con sus aristas salientes
4. Repetir el paso 1 hasta seleccionar todos los vértices

Ordenación topológica - (versión 2)
En esta versión el algoritmo utiliza un arreglo Grado_in en el que se almacenan los grados de entradas de los vértices y una pila P (o una cola Q) en donde se almacenan los vértices con grados de entrada igual a cero.

Ordenación topológica - (versión 3)
→ En esta versión se aplica el recorrido en profundidad. Se realiza un recorrido DFS, marcando cada vértice en post-orden, es decir, una vez visitados todos los vértices a partir de uno dado, el marcado de los vértices en post-orden puede implementarse según una de las sig. opciones:
a) numerándolos antes de retroceder en el recorrido; luego se listan los vértices según sus números de post-orden de mayor a menor.
b) colocándolos en una pila P, luego se listan empezando por el top

Algoritmo de Dijkstra

Estrategia: Algoritmo de Dijkstra
Pasos:
Dado un vértice origen s, elegir el vértice v que esté a la menor distancia de s, dentro de los vértices no procesados
Marcar v como procesado
Actualizar la distancia de w adyacente a

- Para cada vértice v mantiene la siguiente información:
  - D_v : distancia mínima desde el origen (inicialmente ∞ para todos lo vértices excepto el origen con valor 0)
  - P_v : vértice por donde paso para llegar
  - Conocido : dato booleano que me indica si está procesado (inicialmente todos en 0)

VER EJERCICIOS HECHOS

Algoritmos de Caminos mínimos
Grafos con pesos positivos y negativoS

Estrategia: Encolar los vértices
Si el grafo tiene aristas negativas, el algoritmo de Dijkstra puede dar un resultado erróneo.

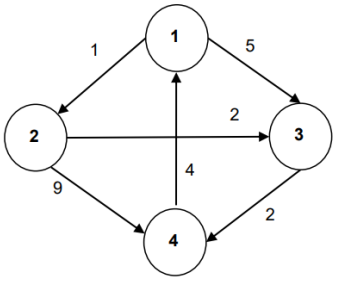
Pasos:
Encolar el vértice origen s.
Procesar la cola:
Desencolar un vértice.
Actualizar la distancia de los adyacentes Dw
siguiendo el mismo criterio de Dijkstra.
Si w no está en la cola, encolarlo.

Grafos	BFS O(V+E)	Dijkstra O(E log V)	Algoritmo modificado (encola vértices) O(V*E)	Optimización de Dijkstra (sort top) O(V+E)
No pesados	Óptimo	Correcto	Malo	Incorrecto si tiene ciclos
Pesados	Incorrecto	Óptimo	Malo	Incorrecto si tiene ciclos
Pesos negativos	Incorrecto	Incorrecto	Óptimo	Incorrecto si tiene ciclos
Grafos pesados acíclicos	Incorrecto	Correcto	Malo	Óptimo

Algoritmo de Floyd

Camino de costo mínimo entre cada par de vértices

➢ Ejemplo:



$D_{i,j}$	1	2	3	4
1	0	1	5	∞
2	∞	0	2	9
3	∞	∞	0	2
4	4	∞	∞	0

K=1

$D_{i,j}$	1	2	3	4
1	0	1	5	∞
2	∞	0	2	9
3	∞	∞	0	2
4	4	~~∞~~ **5**	~~∞~~ **9**	0

Poner otros ejemplos

D (4,2) > D(4,1) + D(1,2) → D (4,2) = D(4,1) + D(1,2)

∞ > 4 + 1 → D (4,2) = 5

Árbol de expansión mínimo

Dado un grafo G=(V, E) no dirigido y conexo
El árbol de expansión mínima es un árbol formado por las aristas de G que conectan todos los vértices con un costo total mínimo.

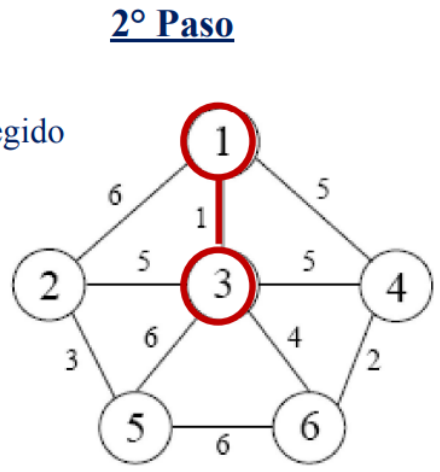
Algoritmo de Prim

Construye el árbol haciéndolo crecer por etapas
Se elige un vértice como raíz del árbol.
En las siguientes etapas:
a) se selecciona la arista (u,v) de mínimo costo que cumpla: u ∈ árbol y v ∉ árbol
b) se agrega al árbol la arista seleccionada en a) (es decir, ahora el vértice v ∈ árbol)
c) se repite a) y b) hasta que se hayan tomado todos los vértices del grafo

Para la implementación se usa una tabla (similar a la utilizada en la implementación del algoritmo de Dijkstra).
La dinámica del algoritmo consiste en, una vez seleccionado una arista (u,v) de costo mínimo tq u ∈ árbol y v ∉ árbol:
se agrega la arista seleccionada al árbol
se actualizan los costos a los adyacentes del vértice

V	Costo	W	Conoc.
1	0	0	1
2	6	1	0
3	1	1	1
4	5	1	0
5	∞	0	0
6	∞	0	0

Vértice elegido



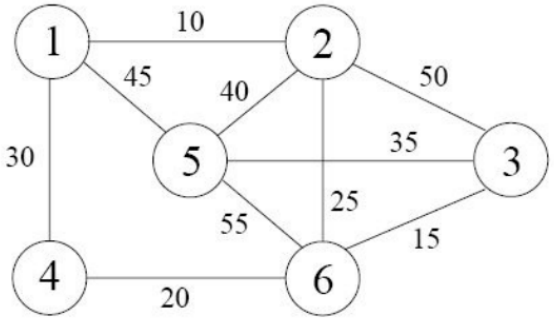
VER EJEMPLO EN LOS EJERCICIOS

Algoritmo de Kruskal

Selecciona las aristas en orden creciente según su peso y las acepta si no originan un ciclo.
El invariante que usa me indica que en cada punto del proceso, dos vértices pertenecen al mismo conjunto si y sólo sí están conectados.
Si dos vértices u y v están en el mismo conjunto, la arista (u,v) es rechazada porque al aceptarla forma un ciclo.

Inicialmente cada vértice pertenece a su propio conjunto
→ |V| conjuntos con un único elemento
Al aceptar una arista se realiza la Unión de dos conjuntos.
Las aristas se organizan en una heap, para ir recuperando la de mínimo costo en cada paso.

Ejemplo:

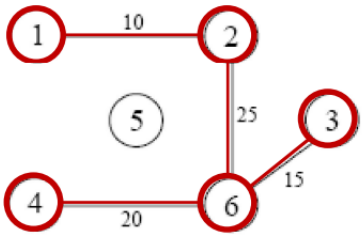


Aristas ordenadas por su costo de menor a mayor:

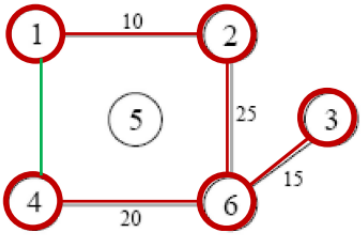
(1,2) □ 10
(3,6) □ 15
(4,6) □ 20
(2,6) □ 25
(1,4) □ 30
(5,3) □ 35
(5,2) □ 40
(1,5) □ 45
(2,3) □ 50
(5,6) □ 55

- Ordenar las aristas, usando un algoritmo de ordenación
- Construir una min-heap □ **más eficiente**

Se agrega la arista (2,6)



¿Se agrega la arista (1,4) con costo 30?



No, porque forma ciclo, ya que pertenece a la misma componente conexa

VER EJERCICIOS HECHOS