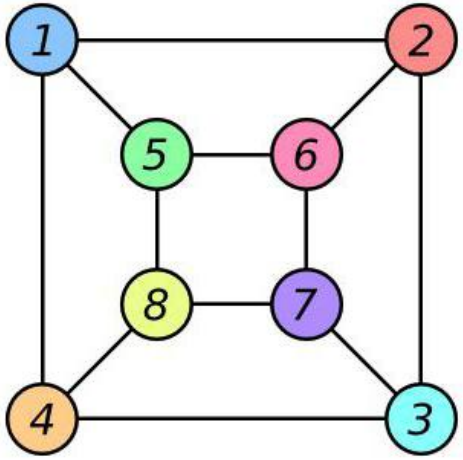
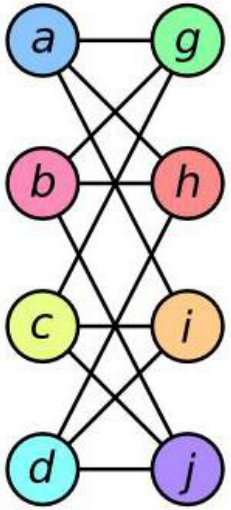
Algoritmos
Estructuras de Datos I

Facultad de Ciencias Exactas y Tecnología
Universidad Nacional de Tucumán
2024

Grafo(2)





Recorrido en un Grafo

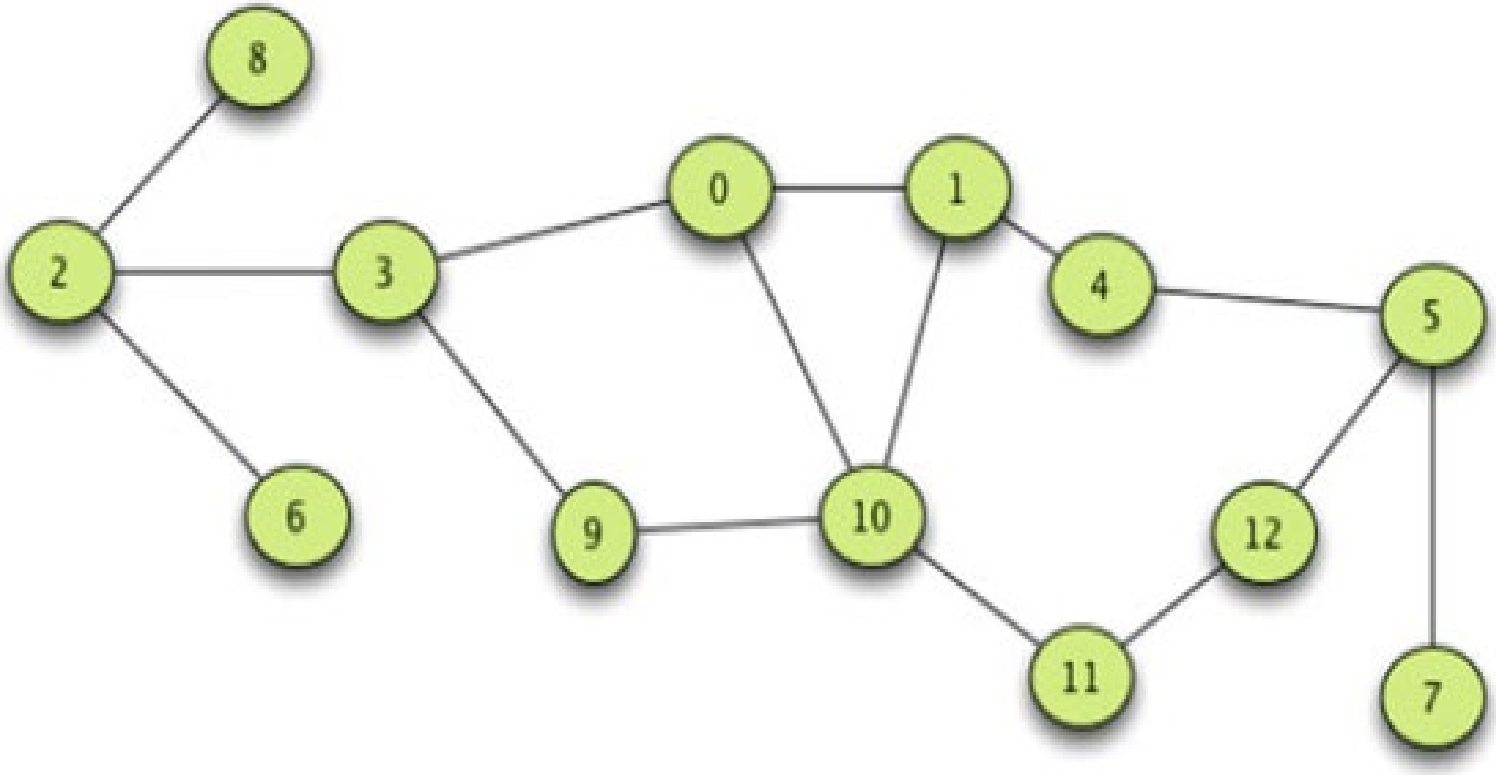
En muchas aplicaciones de grafos es necesario encontrar algoritmos que permitan *recorrerlos en forma sistemática*.

La definición de recorrido relacionado con la estructura de un grafo es mas compleja que para una lista o un árbol por tres razones:

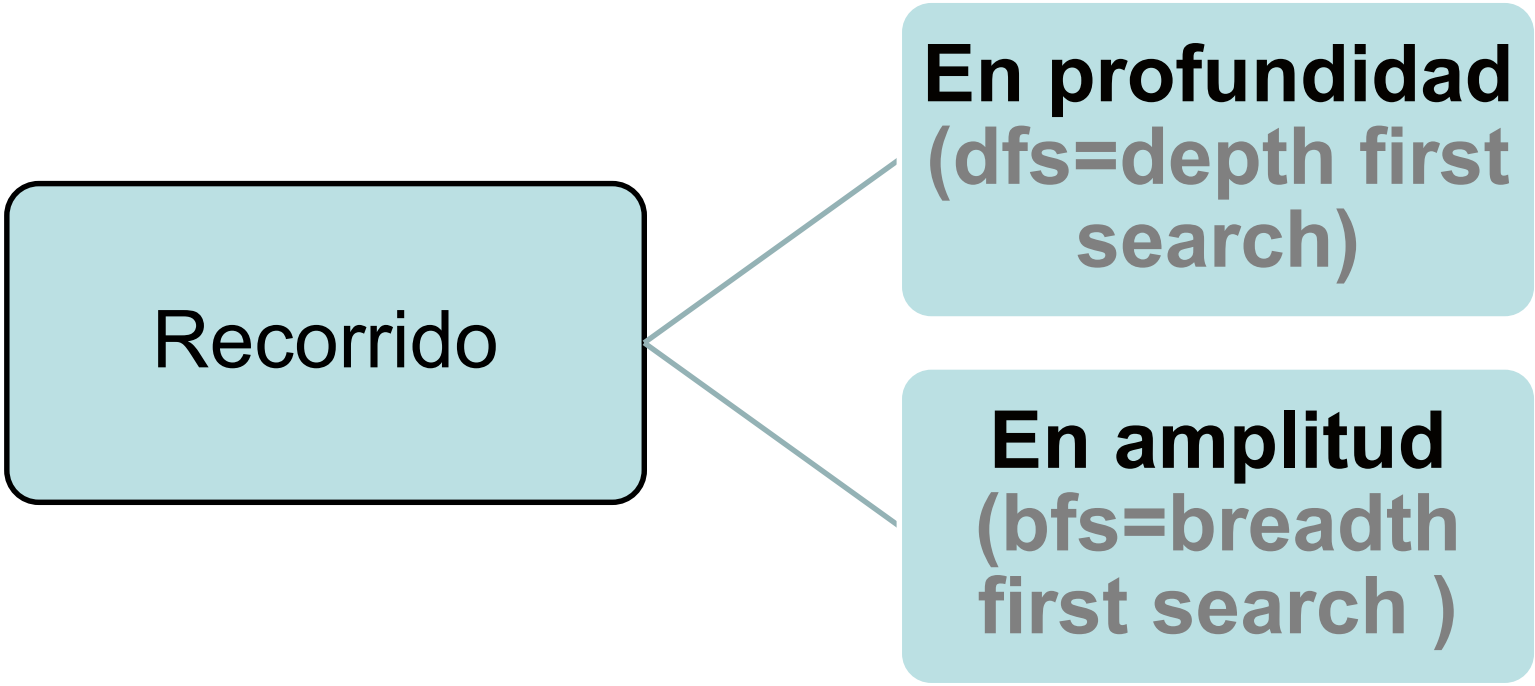
- No hay un primer nodo en el grafo a partir del cual empezar el recorrido. Además una vez determinado un nodo inicial y visitado todos los nodos que se pueden alcanzar a partir de ese nodo, pueden quedar nodos no visitados en el grafo.
- No hay un orden natural entre los sucesores de un nodo particular.
- Un nodo en un grafo puede tener mas de un predecesor. Así un nodo puede ser visitado antes que sus antecesores.

Recorrido en un Grafo

Ejemplo:



Recorrido de un Grafo



Recorrido en profundidad en un Grafo

El recorrido en profundidad de los vértices de un grafo se trata de una generalización del recorrido preorden en arboles.

- Se comienza con un vértice **v**, el cual se marca como visitado, a continuación se selecciona un vértice **w** adyacente de **v** que no haya sido visitado y se procede de la misma manera con él.
- Cuando un vértice **u** es tal que todos sus vértices adyacentes han sido visitados, se cierra ese vértice y se vuelve atrás al ultimo vértice visitado que todavía tenga un vértice adyacente no visitado.
- La búsqueda termina cuando no se encuentran vértices adyacentes no visitados desde un vértice visitado.
- Se mantiene la marca de vértices ya visitados en un vector de booleanos.

Recorrido en profundidad
(dfs=depth first search)

Algoritmo RecorridoEnProfundidad (G)
ENTRADA: G: GRAFO de vértices numerados
SALIDA: listado de vértices
Global: visitado: arreglo con dominio en vértices y valores bool

P1. PARA cada vértice v de G HACER
visitado (v) ← falso
P2. PARA cada vértice v de G HACER
SI NOT visitado (v) ENTONCES
dfs (v)
P3. FIN

Recorrido en profundidad (dfs=depth first search)

Algoritmo de recorrido en profundidad desde un vértice dado v:

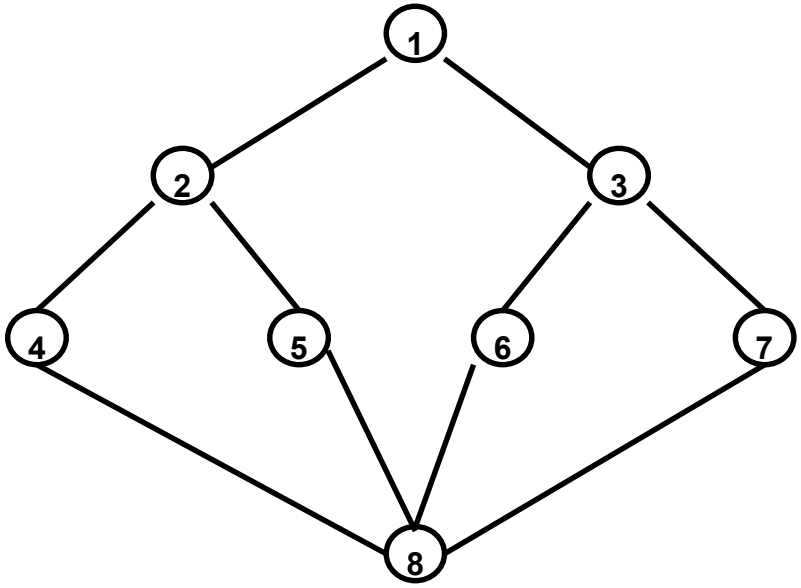
ALGORITMO dfs (v)
ENTRADA: vértice v no visitado
SALIDA: listado de vértices
Auxiliar : w ϵ vertices
P1. visitado (v) ← verdadero
P2. ESCRIBIR (v)
P3. PARA cada vértice w adyacente a v HACER
SI NOT visitado (w) ENTONCES
dfs (w)
P3. FIN

dfs ϵ O(a) con listas de adyacencia
dfs ϵ $O(n^2)$ con matriz de adyacencia

Recorrido en profundidad en un Grafo

Ej.
Lista de adyacencia:

1 → 2 → 3
2 → 1 → 4 → 5
3 → 1 → 6 → 7
4 → 2 → 8
5 → 2 → 8
6 → 3 → 8
7 → 3 → 8
8 → 4 → 5 → 6 → 7



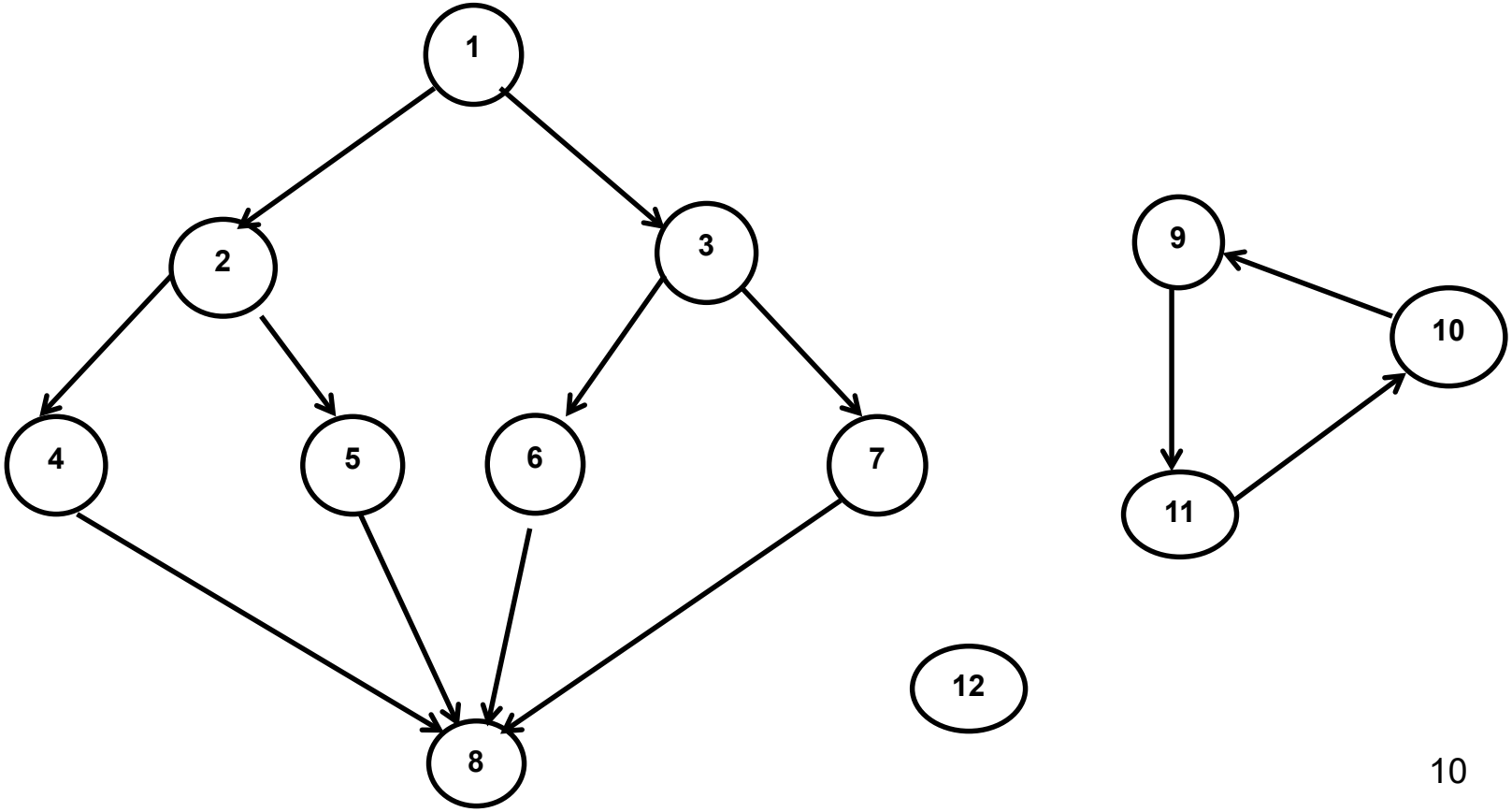
dfs(1)= < 1 2 4 8 5 6 3 7 >

Ejemplo:

ALGORITMO dfs (v)
P1. visitado (v) ← verdadero
P2. ESCRIBIR (v)
P3. PARA cada vértice w adyacente a v HACER
SI NOT visitado (w) ENTONCES
dfs (w)
P3. FIN

Recorrido en profundidad en un Grafo

Ejercitación: recorrer en profundidad el digrafo de la figura



Recorrido en amplitud en un Grafo

El recorrido en amplitud de un Grafo es la generalización del listado por niveles en un árbol:

- Comienza en un vértice **v** marcándolo como visitado, a partir de ese vértice todos los vértices adyacentes a **v** son ahora visitados.
- Recién después se visitan todos los vértices adyacentes a estos recién visitados y así.
- A diferencia del recorrido en profundidad que es recursivo, este algoritmo tiene una formalicen iterativa.
- Se mantiene la lista de vértices ya visitados en un vector de booleanos.

Recorrido en amplitud
(bfs=breadth first search)

Algoritmo RecorridoEnAmplitud (G)
ENTRADA: G: GRAFO de vértices numerados
SALIDA: listado de vértices
Global: visitado: arreglo con dominio en vértices y valores bool

P1. PARA cada vértice v de G HACER
visitado (v) ← falso
P2. PARA cada vértice v de G HACER
SI NOT visitado (v) ENTONCES
bfs (v)
P3. FIN

Recorrido en amplitud (bfs=breadth first search)

Algoritmo de recorrido en amplitud desde un vértice dado v:

ALGORITMO bfs (v)
ENTRADA: vértice v no visitado
SALIDA: listado de vértices
Auxiliar : q ∈ FILA(vertices), u,w ∈ vértices
P1. visitado (v) ← verdadero
P2. FILAVACIA (q)
P3. ENFILA(q,v)
P4. MIENTRAS NOT ESFILAVACIA(q) HACER
u ← FRENTE(q)
ESCRIBIR (u)
DEFILA(q)
PARA cada vértice w adyacente de u HACER
SI NOT visitado(w) ENTONCES
visitado(w) ← verdadero
ENFILA(q,w)
P5. FIN

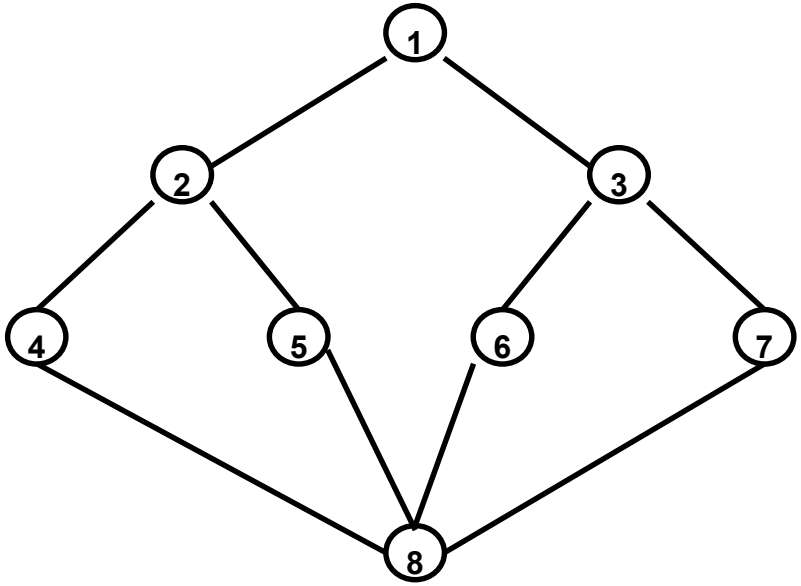
bfs ϵ O(a) con listas de adyacencia
bfs ϵ O(n^2) con matriz de adyacencia

Recorrido en amplitud en un Grafo

Ej.
Lista de adyacencia:

1 → 2 → 3
2 → 1 → 4 → 5
3 → 1 → 6 → 7
4 → 2 → 8
5 → 2 → 8
6 → 3 → 8
7 → 3 → 8
8 → 4 → 5 → 6 → 7

FILA
FV
1
2,3
3,4,5
4,5,6,7
5,6,7,8
6,7,8
7,8
8
FV

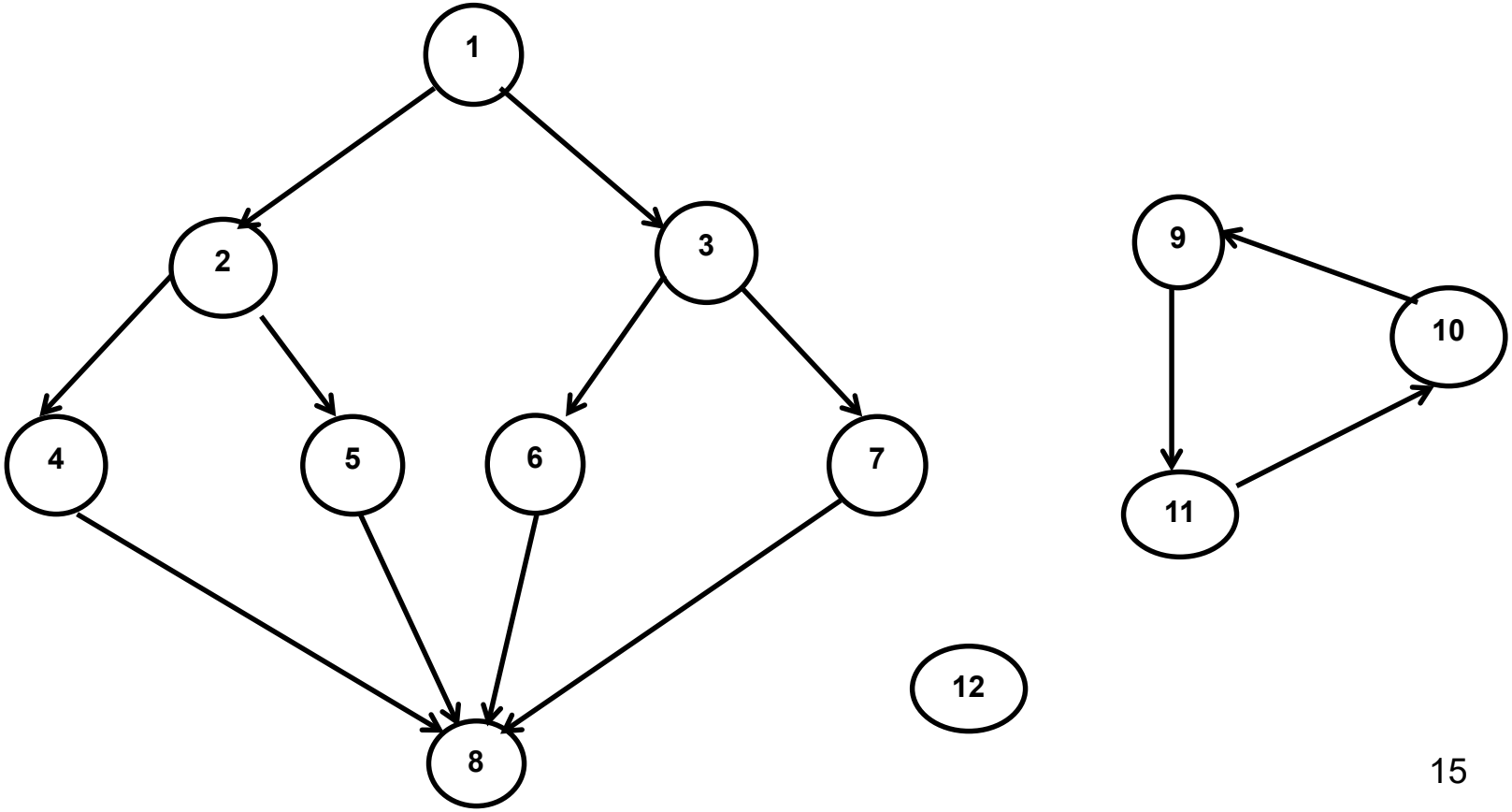


bfs(1)= < 1 2 3 4 5 6 7 8 **>**

Ejemplo:

Recorrido en amplitud en un Grafo

Ejercitación: recorrer en amplitud el digrafo de la figura



Árbol de recubrimiento mínimo
Minimun Spanning Tree (MST)

PROBLEMA: Dado un grafo conexo, ponderado y no dirigido $G = (V,E)$ con costos no negativos de aristas, encontrar el árbol de recubrimiento de G de costo mínimo.

- El ***árbol de recubrimiento*** que abarca todos los nodos del grafo también se denomina: *árbol generador, árbol de expansión* o *de extensión.*

- Un ***árbol de recubrimiento de un grafo*** es un subgrafo sin ciclos $S=(V,T)$, que contiene a todos sus vértices V, donde ***T*** es un subconjunto de *E.*

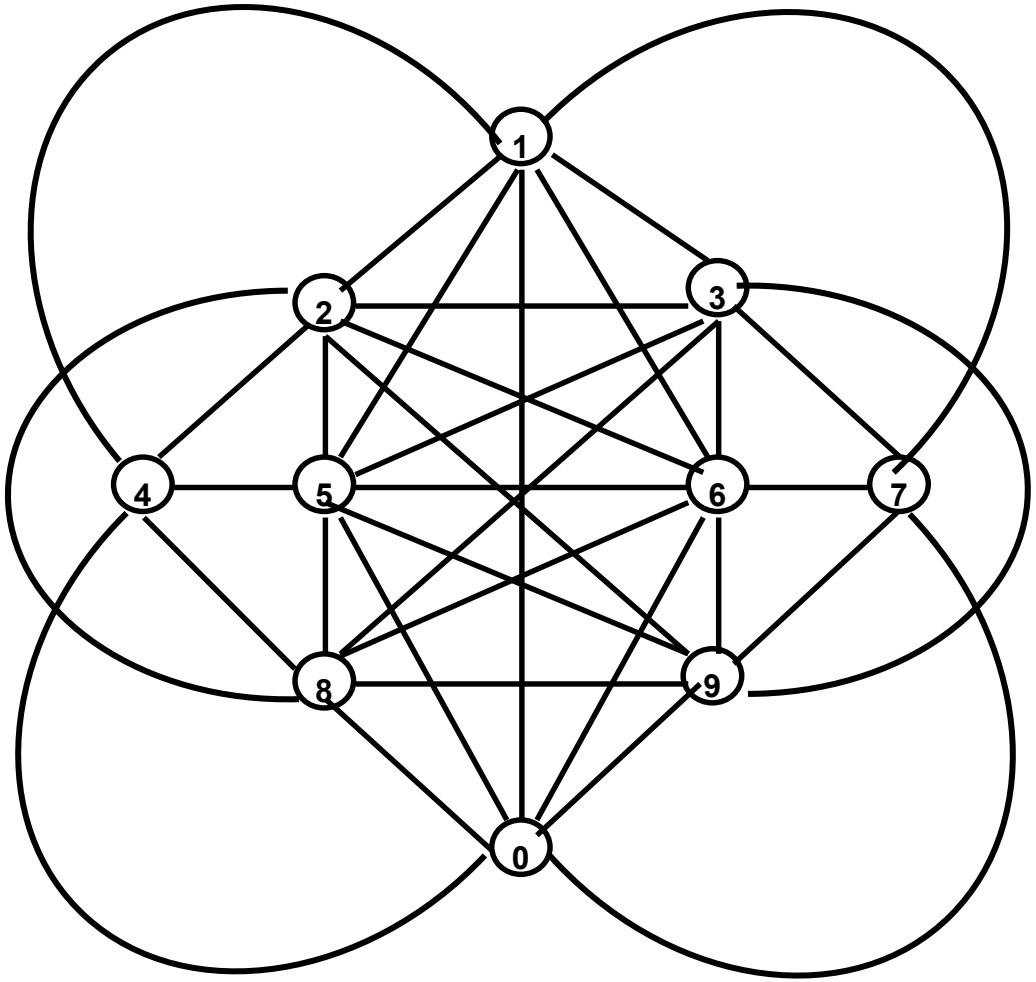
- El problema consiste en encontrar el conjunto de aristas ***T*** de modo que todos los nodos queden conectados y además la suma de las longitudes de las aristas de ***T*** debe ser tan pequeña como sea posible.

- Como G es conexo existe al menos una solución.

Árbol de recubrimiento mínimo

Ejemplo:

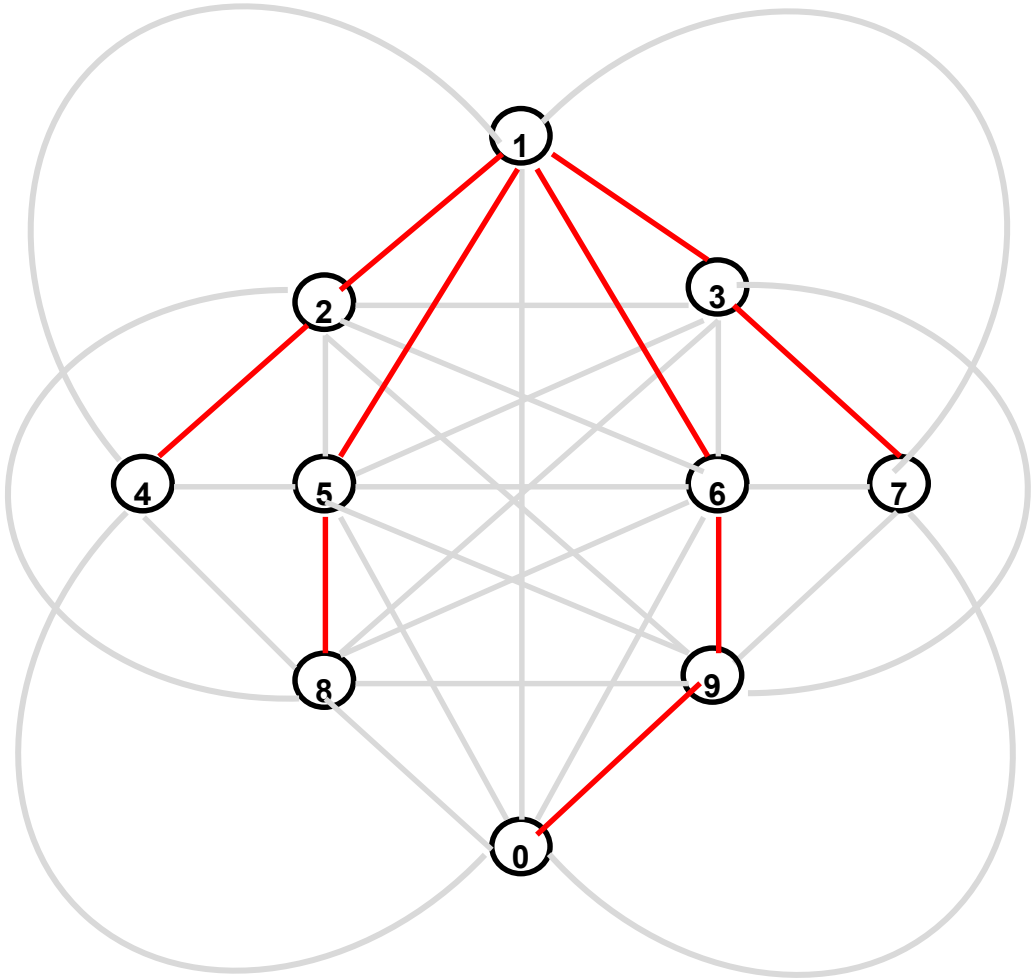
***G* = (*V*,*E*)**



Árbol de recubrimiento mínimo

Ejemplo:

***S* = (*V*,*T*)**



Árbol de recubrimiento mínimo

- Un grafo conexo con ***n*** nodos debe tener por lo menos ***n-1*** aristas.
- Un grafo conexo con ***n*** nodos y más de ***n-1*** aristas contiene al menos un ciclo.
- Como el subgrafo ***S*** tiene el mismo conjunto de vértices de ***G***, si en ***V*** hay ***n*** nodos, el subgrafo sin ciclos ***S*** debe tener exactamente ***n-1*** aristas en ***T***.
- Además como ***S*** es conexo entonces tiene que ser un árbol.

Árbol de recubrimiento mínimo

Existen dos algoritmos muy conocidos que resuelven este problema:

- Algoritmo de Prim (1957) , descubierto por Jarnik (1930) y redescubierto por Prim y Dijkstra.
- Algoritmo de Kruskal (1956).
- En ambos se aplica una estrategia greedy.
- Se va construyendo el árbol por etapas, y en cada una se agrega un arco. La forma en la que se realiza esa elección es la que distingue a ambos algoritmos.
- Algoritmos más sofisticados: Yao(1975), Cheriton y Tarjan (1976) y Tarjan (1983).

Árbol de recubrimiento mínimo

El algoritmo de Prim:

- Comienza por un vértice y elige en cada etapa el arco el menor peso que verifique que uno de sus vértices se encuentre en el conjunto de vértices ya seleccionados y el otro afuera del conjunto.
- Al incluir un nuevo arco a la solución, se agrega su otro extremo al conjunto de vértices seleccionados.

En el algoritmo de Kruskal:

- Se arma el árbol de expansión mínima agregando de a una las aristas usando en cada paso la arista más pequeña que no forme ciclo.
- Para ello en cada etapa se elige una arista por su valor en orden creciente y se decide qué hacer con cada una de ellas.
- Si la arista no forma un ciclo con las ya seleccionados se incluye en la solución; sino, se descarta.

Árbol de recubrimiento mínimo

- Se aplica el esquema general de los algoritmos greedy, en este caso:

- Los candidatos: el conjunto de aristas del grafo con sus costos: E

- Un conjunto de aristas es una solución si constituye un árbol de recubrimiento mínimo para los nodos de V.

- Un conjunto de aristas es factible si no contiene ningún ciclo.

- La función de selección se elegirá de acuerdo al algoritmo.

- La función objetivo es: minimizar el costo total de las aristas de la solución.

Algoritmo de Prim

- En el algoritmo de Kruskal, la función de selección elige las aristas por orden creciente de longitud, sin preocuparse por su conexión con las aristas seleccionadas anteriormente, sólo se tiene cuidado de no formar un ciclo con ellas.

- En el algoritmo de Prim el árbol de recubrimiento mínimo crece de forma natural, comenzando por una raíz arbitraria. En cada paso se agrega una rama al árbol ya construido y el algoritmo se detiene cuando se han alcanzado todos los nodos.

Algoritmo de Prim

El algoritmo de Prim encuentra *un árbol de recubrimiento mínimo del grafo.*

Dado el grafo **G = (V, E)**

- Sea B un conjunto de nodos y sea T un conjunto de aristas.
- Inicialmente B contiene un nodo arbitrario, y T está vacío.
- En cada paso, el algoritmo busca la arista **(u,v)** de menor costo tal que el vértice **u** esté en el conjunto B y el vértice **v** esté fuera de él. Entonces agrega ese arco a T y el vértice **v** a B.
- Entonces en cada paso las aristas de T forman un árbol de recubrimiento mínimo para los nodos de B.
- Se continua así hasta que B coincide con el conjunto de vértices.

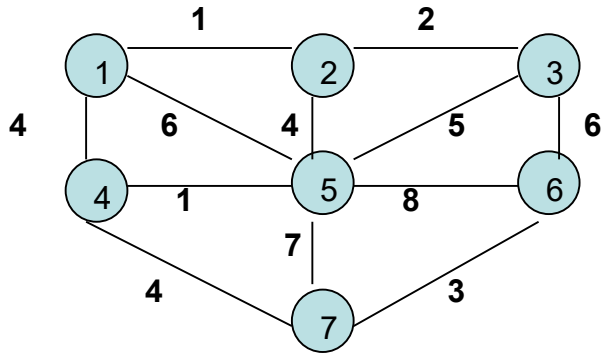
Algoritmo de Prim

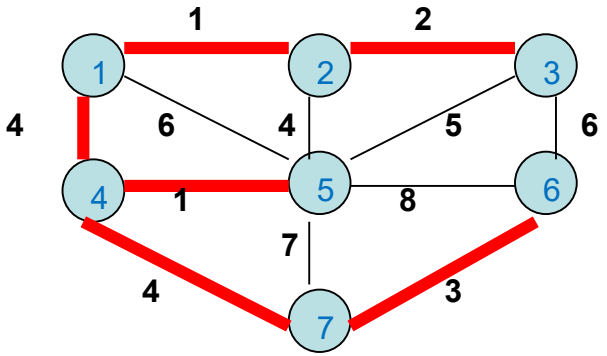
ALGORITMO PRIM (G): grafo→arbol
Obtiene el árbol de recubrimiento mínimo de un grafo

ENTRADA: V: conjunto de vértices
E: conjunto de arcos de *costo positivo*
SALIDA: T: conjunto de arcos
AUXILIARES: B: conjunto de vértices

P1. B ← { 1 } ; T=∅
P2. Mientras B≠V hacer
Elegir en E un arco e=(v1,v2) de costo mínimo
tal que v1∈B y v2∈V-B
T ← T U {e}
B ← B U {v2}
P3. Retorna T
P4. Fin.

Algoritmo de Prim
Ejemplo





Pasos del algoritmo:

Inicial:		**B={1}**	**T=∅**
it 1:	**e=(1,2)**	**B={1,2}**	**T={(1,2)}**
it 2:	**e=(2,3)**	**B={1,2,3}**	**T={(1,2),(2,3)}**
it 3:	**e=(1,4)**	**B={1,2,3,4}**	**T={(1,2),(2,3),(1,4)}**
it 4:	**e=(4,5)**	**B={1,2,3,4,5}**	**T={(1,2),(2,3),(1,4), (4,5)}**
it 5:	**e=(4,7)**	**B={1,2,3,4,5,7}**	**T={(1,2),(2,3),(1,4), (4,5),(4,7)}**
it 6:	**e=(7,6)**	**B={1,2,3,4,5,6,7}**	**T={(1,2),(2,3),(1,4), (4,5),(4,7),(7,6)}**

Algoritmo de Prim

- Para estimar el ***tiempo de ejecución*** se considera que el algoritmo tiene una iteración de n-1 veces, en el cuerpo de la iteración se requiere O(n). Por lo tanto se requiere $O(n^2)$.

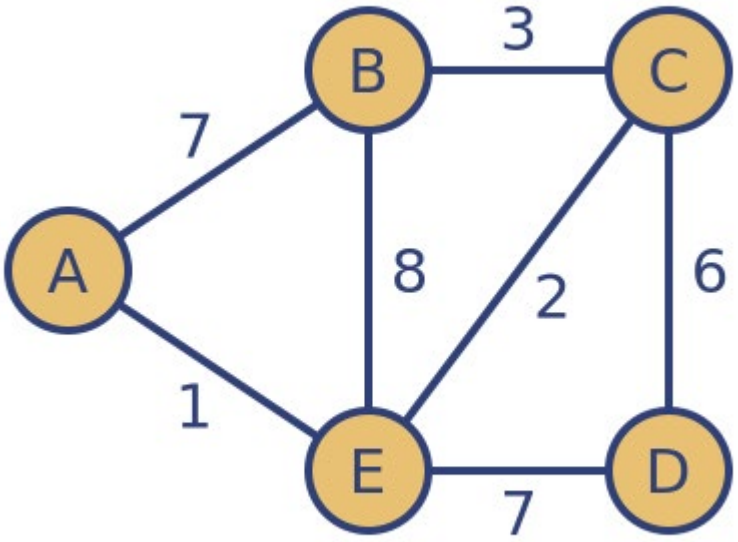
- El algoritmo de Prim es siempre de orden $O(n^2)$ mientras que el orden de complejidad del algoritmo de Kruskal $O(a \log n)$ no sólo depende del número de vértices, sino también del número de arcos.

- Para *grafos densos* el número de arcos *a* es cercano a: $n(n–1)/2$ por lo que el orden de complejidad del algoritmo de Kruskal es $O(n^2 \log n)$, peor que la complejidad $O(n^2)$ de Prim.

- Sin embargo, para *grafos dispersos* en los que *a* es próximo a *n*, el algoritmo de Kruskal es de orden $O(n \log n)$, comportándose probablemente de forma más eficiente que el de Prim.

Algoritmo de Prim

Ejercitación: aplique el algoritmo de Prim al siguiente grafo, tomando el nodo A como nodo inicial.



Algoritmo de Prim

Ejercitación: aplique el algoritmo de Prim al siguiente grafo, tomando el nodo A como nodo inicial.

Pasos del algoritmo:

Inicial: B={A}
T=∅
it 1: e=(A,E) B={A,E}
T={(A,E)}
it 2: e=(E,C) B={A,E,C}
T={(A,E),(E,C)}
it 3: e=(C,B) B={A,E,C,B}
T={(A,E),(E,C),(C,B)}
it 4: e=(C,D) B={A,E,C,B,D}
T={(A,E),(E,C),(C,B),(C,D)}