Programación 2

Tecnicatura Universitaria en Inteligencia Artificial

Profesores:

Damian
Andrea
Ariel
Joaquin
Ana

2C. 2024

Ejemplo

Problema: Queremos salir de la FCEIA y recorrer todas las facultades de la universidad nacional.

Nos sirve este grafo para el problema?

Otro mapa

Quisiera recorrer todas las ciudades que tienen Universidades nacionales, sin pasar dos veces por la misma ciudad.

Buscando la definición

Recordemos primeros algunas definiciones:

Definición

- Un camino se llamará *cerrado* si comienza y termina en el mismo vértice.
- Llamamos *recorrido* a un camino donde no se encuentran aristas repetidas. Se llama *circuito* a un recorrido cerrado.
- Denominamos *camino simple* a un camino que no repite vértices, y *ciclo* si es cerrado.

Definición

Un camino euleriano es un camino que pasa por cada arista una y sólo una vez. Un circuito euleriano es un camino cerrado que recorre cada arista exactamente una vez.

Cuáles de estas nos sirven?

Ciclo Hamiltoniano

Definición

Llamaremos ciclo Hamiltoniano a un ciclo(es decir un camino simple cerrado) que visite cada vértice del grafo

Qué quiere decir esto?

simple:

Ciclo Hamiltoniano

Definición

Llamaremos ciclo Hamiltoniano a un ciclo(es decir un camino simple cerrado) que visite cada vértice del grafo

Qué quiere decir esto?

simple:No repite vértices
cerrado:

Ciclo Hamiltoniano

Definición

Llamaremos ciclo Hamiltoniano a un ciclo(es decir un camino simple cerrado) que visite cada vértice del grafo

Qué quiere decir esto?

simple:No repite vértices
cerrado:Comienza y termina en el mismo vértice
Es decir que un ciclo Hamiltoniano es un camino que termina donde empieza y recorre cada vértice exactamente una vez.
Es posible hacer esto con cualquier grafo?

Ciclo Hamiltoniano

Definición

Llamaremos ciclo Hamiltoniano a un ciclo(es decir un camino simple cerrado) que visite cada vértice del grafo

Qué quiere decir esto?

simple:No repite vértices
cerrado:Comienza y termina en el mismo vértice
Es decir que un ciclo Hamiltoniano es un camino que termina donde empieza y recorre cada vértice exactamente una vez.
Es posible hacer esto con cualquier grafo?
Tiene alguna relación con circuitos eulerianos?

Ciclo Hamiltoniano

Definición

Llamaremos ciclo Hamiltoniano a un ciclo(es decir un camino simple cerrado) que visite cada vértice del grafo

Qué quiere decir esto?

simple:No repite vértices
cerrado:Comienza y termina en el mismo vértice
Es decir que un ciclo Hamiltoniano es un camino que termina donde empieza y recorre cada vértice exactamente una vez.
Es posible hacer esto con cualquier grafo?
Tiene alguna relación con circuitos eulerianos?
En el K_4 tengo circuito euleriano? Ciclo hamiltoniano?

Ciclo Hamiltoniano

Definición

Llamaremos ciclo Hamiltoniano a un ciclo(es decir un camino simple cerrado) que visite cada vértice del grafo

Qué quiere decir esto?

simple:No repite vértices
cerrado:Comienza y termina en el mismo vértice
Es decir que un ciclo Hamiltoniano es un camino que termina donde empieza y recorre cada vértice exactamente una vez.
Es posible hacer esto con cualquier grafo?
Tiene alguna relación con circuitos eulerianos?
En el K_4 tengo circuito euleriano? Ciclo hamiltoniano?
En el $K_{2,3}$ que tengo?

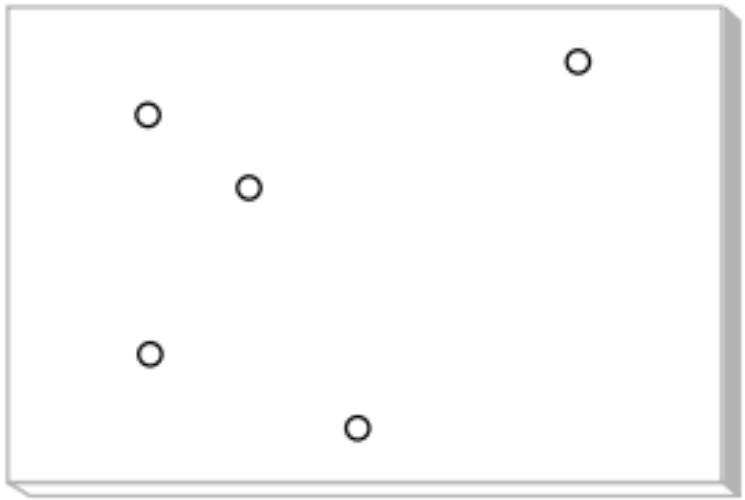
Características

Para circuitos eulerianos teníamos una caracterización(Así le decimos cuando tenemos un sii). Lamentablemente no tenemos una para ciclos Hamiltonianos. Lo siguiente nos puede ayudar a encontrarlo:

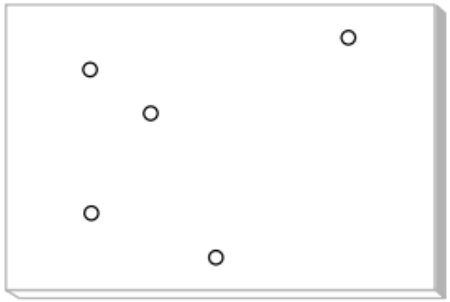
- Si G tiene ciclo hamiltoniano $gr(v) \geq 2$
- Si $gr(a) = 2$ para una arista a, las dos aristas incidentes deben aparecer en cualquier ciclo hamiltoniano
- Si $gr(a) > 2$ una vez que usamos dos aristas incidentes podemos desestimar las otras al buscar el ciclo hamiltoniano
- Un ciclo Hamiltoniano no puede tener un ciclo para un subrafo

Grafos Ponderados

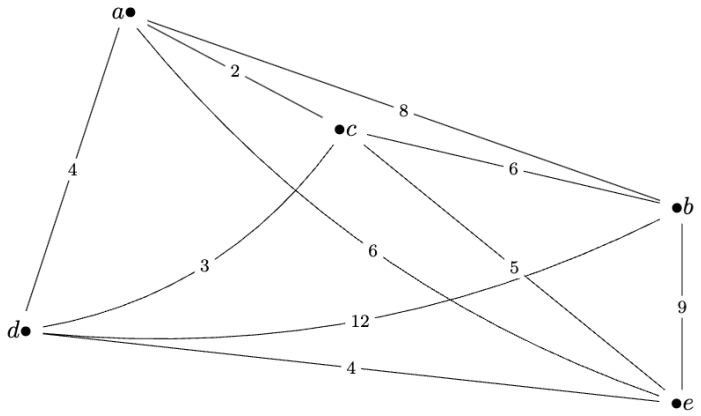
Ejemplo A menudo en la manufactura, es necesario hacer agujeros en hojas de metal. Luego se atornillan las componentes a estas hojas de metal. Los agujeros se perforan utilizando un taladro controlado por computadora. Para ahorra tiempo y dinero, el taladro debe moverse tan rápido como sea posible. La figura sirve para ilustrar el problema.



Grafos Ponderados



Se modelará la situación como un grafo. Los vértices del grafo corresponden a los agujeros. Cada par de vértices se conecta por una arista. En cada arista se escribe el tiempo para mover el taladro entre los hoyos correspondientes.



Grafos Ponderados

Este problema es muy distinto que los dos que vimos hasta ahora. Para empezar, ahora tenemos un costo en las aristas que hay que considerar. Además, no estamos interesado en un camino que recorra todas las aristas, si no en un camino que recorra todos los vértices exactamente una vez, además de hacerlo de la forma óptima.

Grafos Ponderados

La siguiente tabla muestra todas las rutas que visitan los nodos comenzando por *a* y terminando en *e*.

Camino	Costo
a, b, c, d, e	21
a, b, d, c, e	28
a, c, b, d, e	24
a, c, d, b, e	26
a, d, b, c, e	27
a, d, c, b, e	22

Grafos Ponderados

La siguiente tabla muestra todas las rutas que visitan los nodos comenzando por *a* y terminando en *e*.

Camino	Costo
a, b, c, d, e	21
a, b, d, c, e	28
a, c, b, d, e	24
a, c, d, b, e	26
a, d, b, c, e	27
a, d, c, b, e	22

Se ve que la ruta que visita los vértices a, b, c, d, e, en ese orden, tiene longitud mínima.

Grafos Ponderados

La siguiente tabla muestra todas las rutas que visitan los nodos comenzando por *a* y terminando en *e*.

Camino	Costo
a, b, c, d, e	21
a, b, d, c, e	28
a, c, b, d, e	24
a, c, d, b, e	26
a, d, b, c, e	27
a, d, c, b, e	22

Se ve que la ruta que visita los vértices a, b, c, d, e, en ese orden, tiene longitud mínima.
Numerar todas las trayectorias posibles es un método muy ineficiente. Por desgracia, nadie conoce un método que sea mucho más práctico para grafos arbitrarias.

Grafos Ponderados

Definición Llamamos **grafo ponderado** o **grafo con pesos** a un grafo $G = (V, E)$ con una función $w : E \to \mathbb{R}$, es decir, un grafo donde a cada arista se le asigna un peso. Notamos $w(i, j)$ al peso de la arista que va de i a j, si esta existe.

Grafos Ponderados

Definición Llamamos **grafo ponderado** o **grafo con pesos** a un grafo $G = (V, E)$ con una función $w : E \to \mathbb{R}$, es decir, un grafo donde a cada arista se le asigna un peso. Notamos $w(i, j)$ al peso de la arista que va de i a j, si esta existe.

Definición En un grafo con pesos, modificamos la definición de **longitud de un camino**. La longitud del camino en un grafo con pesos es la suma de los pesos de las aristas que recorre dicho camino.

Ojo que no sólo sirve para distancias

Edsger Dijkstra

Edsger W. Dijkstra (1930-2002) fue un científico de la computación holandés que ideo un algoritmo que resuelve el problema de la ruta más corta de forma óptima. Además, fue de los primeros en proponer la programación como una ciencia.
Ganador del premio Turing en 1972. El premio Turing es “el Nobel de la programación".
Poco después de su muerte recibió también el premio de la ACM.
El premio pasó a llamarse Premio Dijkstra el siguiente año en su honor.

El algoritmo de Dijkstra

Este algoritmo Dijkstra **encuentra la longitud de una ruta más corta** del vértice a al vértice z en un **grafo** $G = (V, E)$ **ponderado y conexo**. El peso de la arista (i, j) es $w(i, j) > 0$ (es decir, requerimos que los pesos sean siempre positivos).

Es importante verificar que estas condiciones se cumplen antes de aplicar el algoritmo.

El algoritmo de Dijkstra

El algoritmo funciona asignando **distancias** a los vértices, que siempre son relativas al vértice inicial. Notaremos a la distancia del vértice a al vértice v como $D(v)$.

El algoritmo de Dijkstra

El algoritmo funciona asignando **distancias** a los vértices, que siempre son relativas al vértice inicial. Notaremos a la distancia del vértice a al vértice v como $D(v)$.
Algunas distancias son temporales y otras son permanentes. Al ilustrar el algoritmo, por lo general se encierran en un círculo, o se pintan de un color distinto los vértices cuya distancia calculada ya es permanente.

El algoritmo de Dijkstra

El algoritmo funciona asignando **distancias** a los vértices, que siempre son relativas al vértice inicial. Notaremos a la distancia del vértice a al vértice v como $D(v)$.
Algunas distancias son temporales y otras son permanentes. Al ilustrar el algoritmo, por lo general se encierran en un círculo, o se pintan de un color distinto los vértices cuya distancia calculada ya es permanente.
Llamaremos T al conjunto de vértices que aún no fueron visitados (y, por lo tanto, tienen una distancia temporal). Esos son los vértices con lo que "tenemos que seguir trabajando". La idea es que, una vez $D(v)$ sea la etiqueta definitiva de un vértice v, $D(v)$ sea la distancia más corta desde a hasta v.

El algoritmo de Dijkstra

El algoritmo funciona asignando **distancias** a los vértices, que siempre son relativas al vértice inicial. Notaremos a la distancia del vértice a al vértice v como $D(v)$.
Algunas distancias son temporales y otras son permanentes. Al ilustrar el algoritmo, por lo general se encierran en un círculo, o se pintan de un color distinto los vértices cuya distancia calculada ya es permanente.
Llamaremos T al conjunto de vértices que aún no fueron visitados (y, por lo tanto, tienen una distancia temporal). Esos son los vértices con lo que "tenemos que seguir trabajando". La idea es que, una vez $D(v)$ sea la etiqueta definitiva de un vértice v, $D(v)$ sea la distancia más corta desde a hasta v.
Al inicio, todos los vértices estarán en la lista de no visitados. Cada iteración del algoritmo visita un nuevo nodo, determinando su distancia definitiva. El algoritmo termina cuando z es visitado. Cuando llega ese momento, $D(z)$ es el valor de la distancia del camino más corto desde a hasta z.

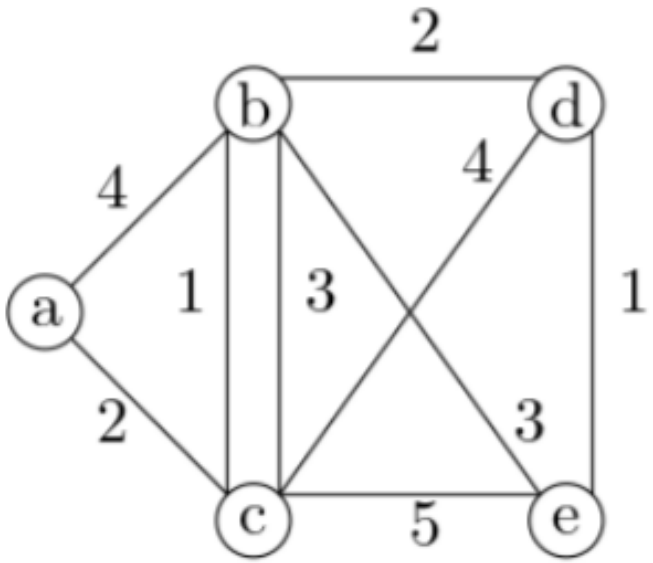
El algoritmo de Dijkstra

Los pasos detallados del **algoritmo de Dijkstra** son los siguientes:

1. Primero, notemos que la ruta más corta del vértice a al vértice a, es la ruta de cero aristas, que tiene peso 0. Así, inicializamos $D(a) = 0$.
2. No sabemos aún el valor de una ruta más corta de a a los otros vértices, entonces para cada vértice $v \neq a$, inicializamos $D(v) = \infty$.
3. Inicializamos el conjunto T como el conjunto de todos los vértices, i.e. $T = V$.
4. Seleccionamos un vértice $v \in T$ tal que $D(v)$ sea mínimo.
5. Quitamos el vértice v del conjunto T: $T = T - v$
6. Para cada $t \in T$ adyacente a v, actualizamos su etiqueta: $D(t) = \min\{D(t), D(v) + w(v,t)\}$
7. Si $z \in T$, repetimos desde el paso 4, si no, hemos terminado y $D(z)$ es el valor de la ruta mas corta entre a y z.

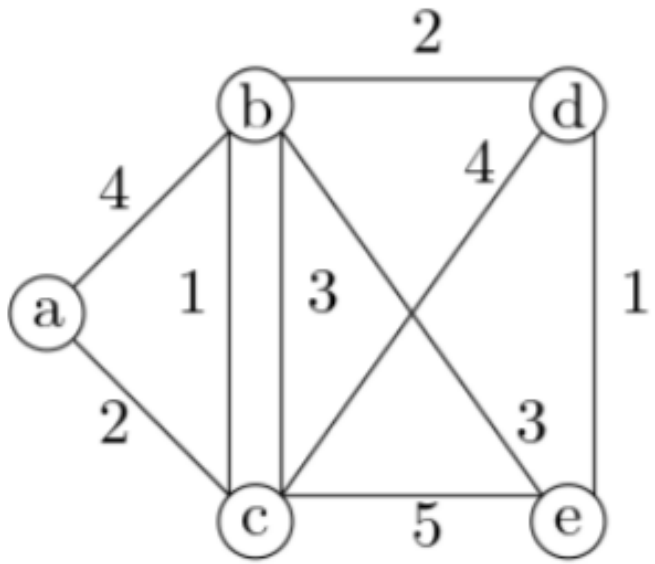
El algoritmo Dijkstra - ejemplo

Utilizaremos el algoritmo de Dijkstra para encontrar el tamaño de la ruta más corta de *a* a *e* en el siguiente grafo:



El algoritmo Dijkstra - ejemplo

Utilizaremos el algoritmo de Dijkstra para encontrar el tamaño de la ruta más corta de *a* a *e* en el siguiente grafo:



Distancias

$a = 0$

$b = 3$

$c = 2$

$d = 5$

$e = 6$

Grafos como TAD

Un Grafo es un conjunto de vértices (o nodos) y un conjunto de aristas (o arcos, o relaciones) entre ellos que admite la siguientes operaciones:

Grafos como TAD

Un Grafo es un conjunto de vértices (o nodos) y un conjunto de aristas (o arcos, o relaciones) entre ellos que admite la siguientes operaciones:
class Graph:

```
def __init__(self) -> None: .
            self.nodes: set[Any] = set()
            self.edges: set[tuple[Any, Any]] = set()
            self.adj: dict[Any, set[Any]] = {}
            self.degree: dict[Any, int] = {}
```

`def add_node(self, node: Any) -> None:` agrega un nodo.

Grafos como TAD

Un Grafo es un conjunto de vértices (o nodos) y un conjunto de aristas (o arcos, o relaciones) entre ellos que admite la siguientes operaciones:
class Graph:

`def __init__(self) -> None:` .

self.nodes: set[Any] = set()
self.edges: set[tuple[Any, Any]] = set()
self.adj: dict[Any, set[Any]] = {}
self.degree: dict[Any, int] = {}

`def add_node(self, node: Any) -> None:` agrega un nodo.

self.nodes.add(node)

`def add_edge(self, node1: Any, node2: Any) -> None:`

agrega la arista entre los nodos x e y

Grafos como TAD

Un Grafo es un conjunto de vértices (o nodos) y un conjunto de aristas (o arcos, o relaciones) entre ellos que admite la siguientes operaciones:
class Graph:

`def __init__(self) -> None:` .

self.nodes: set[Any] = set()
self.edges: set[tuple[Any, Any]] = set()
self.adj: dict[Any, set[Any]] = {}
self.degree: dict[Any, int] = {}

`def add_node(self, node: Any) -> None:` agrega un nodo.

self.nodes.add(node)

`def add_edge(self, node1: Any, node2: Any) -> None:`

agrega la arista entre los nodos x e y

...

Grafos como un TAD

Un Grafo Ponderado es un conjunto de vértices (o nodos) y un conjunto de aristas (o arcos, o relaciones) entre ellos que admite la siguientes operaciones:

Grafos como un TAD

Un Grafo Ponderado es un conjunto de vértices (o nodos) y un conjunto de aristas (o arcos, o relaciones) entre ellos que admite la siguientes operaciones:

`__init__()` Que crea un grafo ponderado vacío.

Grafos como un TAD

Un Grafo Ponderado es un conjunto de vértices (o nodos) y un conjunto de aristas (o arcos, o relaciones) entre ellos que admite la siguientes operaciones:

`__init__()` Que crea un grafo ponderado vacío.

`add_node(x)` Que agrega un nodo al grafo ponderado.

Grafos como un TAD

Un Grafo Ponderado es un conjunto de vértices (o nodos) y un conjunto de aristas (o arcos, o relaciones) entre ellos que admite la siguientes operaciones:

`__init__()` Que crea un grafo ponderado vacío.

`add_node(x)` Que agrega un nodo al grafo ponderado.

`remove_node(x)` Que remueve un nodo del grafo ponderado.

Grafos como un TAD

Un Grafo Ponderado es un conjunto de vértices (o nodos) y un conjunto de aristas (o arcos, o relaciones) entre ellos que admite la siguientes operaciones:

`__init__()` Que crea un grafo ponderado vacío.

`add_node(x)` Que agrega un nodo al grafo ponderado.

`remove_node(x)` Que remueve un nodo del grafo ponderado.

`add_edge(x,y, w)` Que agrega la arista entre los nodos x e y, con peso w, al grafo ponderado.

Grafos como un TAD

Un Grafo Ponderado es un conjunto de vértices (o nodos) y un conjunto de aristas (o arcos, o relaciones) entre ellos que admite la siguientes operaciones:

`__init__()` Que crea un grafo ponderado vacío.

`add_node(x)` Que agrega un nodo al grafo ponderado.

`remove_node(x)` Que remueve un nodo del grafo ponderado.

`add_edge(x,y, w)` Que agrega la arista entre los nodos x e y, con peso w, al grafo ponderado.

`remove_edge(x, y)` Que remueve la arista entre x e y del grafo ponderado.

Grafos como un TAD

Un Grafo Ponderado es un conjunto de vértices (o nodos) y un conjunto de aristas (o arcos, o relaciones) entre ellos que admite la siguientes operaciones:

`__init__()` Que crea un grafo ponderado vacío.

`add_node(x)` Que agrega un nodo al grafo ponderado.

`remove_node(x)` Que remueve un nodo del grafo ponderado.

`add_edge(x,y, w)` Que agrega la arista entre los nodos x e y, con peso w, al grafo ponderado.

`remove_edge(x, y)` Que remueve la arista entre x e y del grafo ponderado.

`are_adjacent(x, y)` Devuelve el peso de la arista (x,y) si x e y son adyacentes, imprime un error en caso contrario.

Grafos como un TAD

Un Grafo Ponderado es un conjunto de vértices (o nodos) y un conjunto de aristas (o arcos, o relaciones) entre ellos que admite la siguientes operaciones:

`__init__()` Que crea un grafo ponderado vacío.

`add_node(x)` Que agrega un nodo al grafo ponderado.

`remove_node(x)` Que remueve un nodo del grafo ponderado.

`add_edge(x,y, w)` Que agrega la arista entre los nodos x e y, con peso w, al grafo ponderado.

`remove_edge(x, y)` Que remueve la arista entre x e y del grafo ponderado.

`are_adjacent(x, y)` Devuelve el peso de la arista (x,y) si x e y son adyacentes, imprime un error en caso contrario.

`is_node(x)` Devuelve True si x es un nodo del grafo ponderado, False si no.

Grafos como un TAD

Un Grafo Ponderado es un conjunto de vértices (o nodos) y un conjunto de aristas (o arcos, o relaciones) entre ellos que admite la siguientes operaciones:

`__init__()` Que crea un grafo ponderado vacío.

`add_node(x)` Que agrega un nodo al grafo ponderado.

`remove_node(x)` Que remueve un nodo del grafo ponderado.

`add_edge(x,y, w)` Que agrega la arista entre los nodos x e y, con peso w, al grafo ponderado.

`remove_edge(x, y)` Que remueve la arista entre x e y del grafo ponderado.

`are_adjacent(x, y)` Devuelve el peso de la arista (x,y) si x e y son adyacentes, imprime un error en caso contrario.

`is_node(x)` Devuelve True si x es un nodo del grafo ponderado, False si no.

`get_nodes()` Devuelve una lista de nodos del grafo ponderado.

Grafos como un TAD

Un Grafo Ponderado es un conjunto de vértices (o nodos) y un conjunto de aristas (o arcos, o relaciones) entre ellos que admite la siguientes operaciones:

`__init__()` Que crea un grafo ponderado vacío.

`add_node(x)` Que agrega un nodo al grafo ponderado.

`remove_node(x)` Que remueve un nodo del grafo ponderado.

`add_edge(x,y, w)` Que agrega la arista entre los nodos x e y, con peso w, al grafo ponderado.

`remove_edge(x, y)` Que remueve la arista entre x e y del grafo ponderado.

`are_adjacent(x, y)` Devuelve el peso de la arista (x,y) si x e y son adyacentes, imprime un error en caso contrario.

`is_node(x)` Devuelve True si x es un nodo del grafo ponderado, False si no.

`get_nodes()` Devuelve una lista de nodos del grafo ponderado.

`get_adjacent(x)` Devuelve una lista de todos los nodos adyacentes al nodo x.

Arboles

Los **árboles** son un tipo especial de **grafo**. La definición que dimos en unidades anteriores se puede expresar ahora en términos de teoría de grafos.

Arboles

Los **árboles** son un tipo especial de **grafo**. La definición que dimos en unidades anteriores se puede expresar ahora en términos de teoría de grafos.

Definición. Un **árbol** (libre) T es una grafo simple que satisface lo siguiente: si v y w son vértices en T existe un camino único de v a w.

Arboles

Los **árboles** son un tipo especial de **grafo**. La definición que dimos en unidades anteriores se puede expresar ahora en términos de teoría de grafos.

Definición. Un **árbol** (libre) T es una grafo simple que satisface lo siguiente: si v y w son vértices en T existe un camino único de v a w.

Recordemos también que llamamos **árbol con raíz** a un árbol donde se ha designado un nodo especial, llamado raíz.

Arboles

Los **árboles** son un tipo especial de **grafo**. La definición que dimos en unidades anteriores se puede expresar ahora en términos de teoría de grafos.

Definición. Un **árbol** (libre) T es una grafo simple que satisface lo siguiente: si v y w son vértices en T existe un camino único de v a w.

Recordemos también que llamamos **árbol con raíz** a un árbol donde se ha designado un nodo especial, llamado raíz.

Ahora que sabemos teoría de grafos podemos, además, dar algunas propiedades adicionales de árboles.

Arboles

Los **árboles** son un tipo especial de **grafo**. La definición que dimos en unidades anteriores se puede expresar ahora en términos de teoría de grafos.

Definición. Un **árbol** (libre) T es una grafo simple que satisface lo siguiente: si v y w son vértices en T existe un camino único de v a w.

Recordemos también que llamamos **árbol con raíz** a un árbol donde se ha designado un nodo especial, llamado raíz.

Ahora que sabemos teoría de grafos podemos, además, dar algunas propiedades adicionales de árboles.

Propiedad Un grafo conexo y sin ciclos es un árbol. En efecto, como el grafo es conexo entre dos vértices cualesquiera siempre hay un camino.

Arboles

Los **árboles** son un tipo especial de **grafo**. La definición que dimos en unidades anteriores se puede expresar ahora en términos de teoría de grafos.

Definición. Un **árbol** (libre) T es una grafo simple que satisface lo siguiente: si v y w son vértices en T existe un camino único de v a w.

Recordemos también que llamamos **árbol con raíz** a un árbol donde se ha designado un nodo especial, llamado raíz.

Ahora que sabemos teoría de grafos podemos, además, dar algunas propiedades adicionales de árboles.

Propiedad Un grafo conexo y sin ciclos es un árbol. En efecto, como el grafo es conexo entre dos vértices cualesquiera siempre hay un camino.

Arboles

Los **árboles** son un tipo especial de **grafo**. La definición que dimos en unidades anteriores se puede expresar ahora en términos de teoría de grafos.

Definición. Un **árbol** (libre) T es una grafo simple que satisface lo siguiente: si v y w son vértices en T existe un camino único de v a w.

Recordemos también que llamamos **árbol con raíz** a un árbol donde se ha designado un nodo especial, llamado raíz.

Ahora que sabemos teoría de grafos podemos, además, dar algunas propiedades adicionales de árboles.

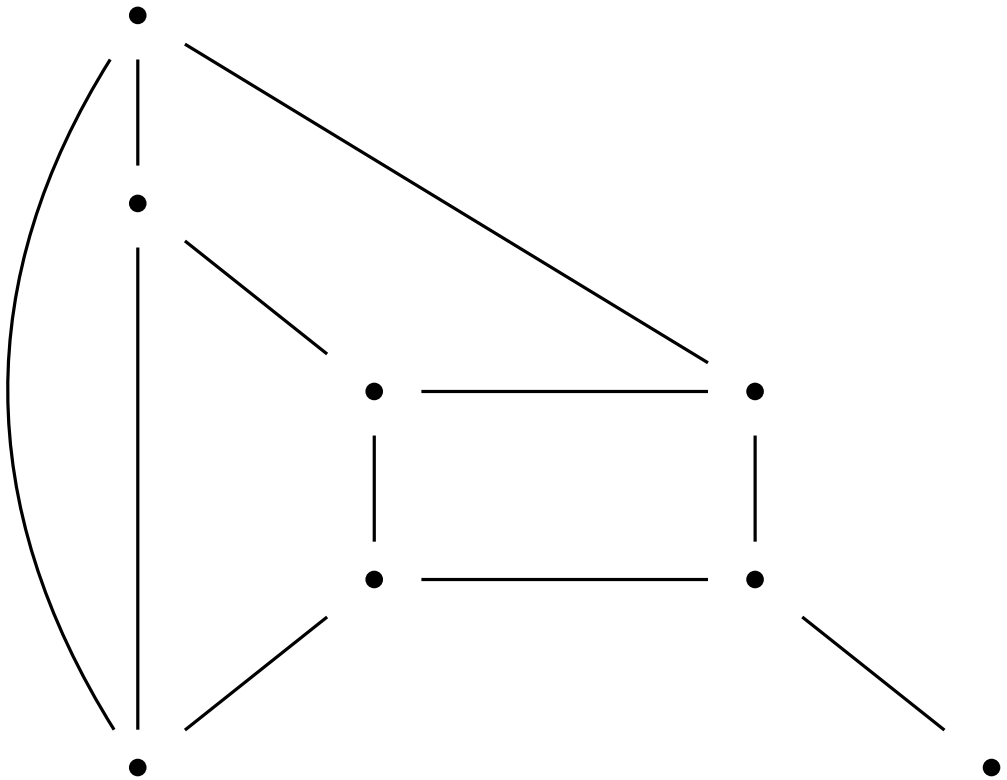
Propiedad Un grafo conexo y sin ciclos es un árbol. En efecto, como el grafo es conexo entre dos vértices cualesquiera siempre hay un camino.

Arboles

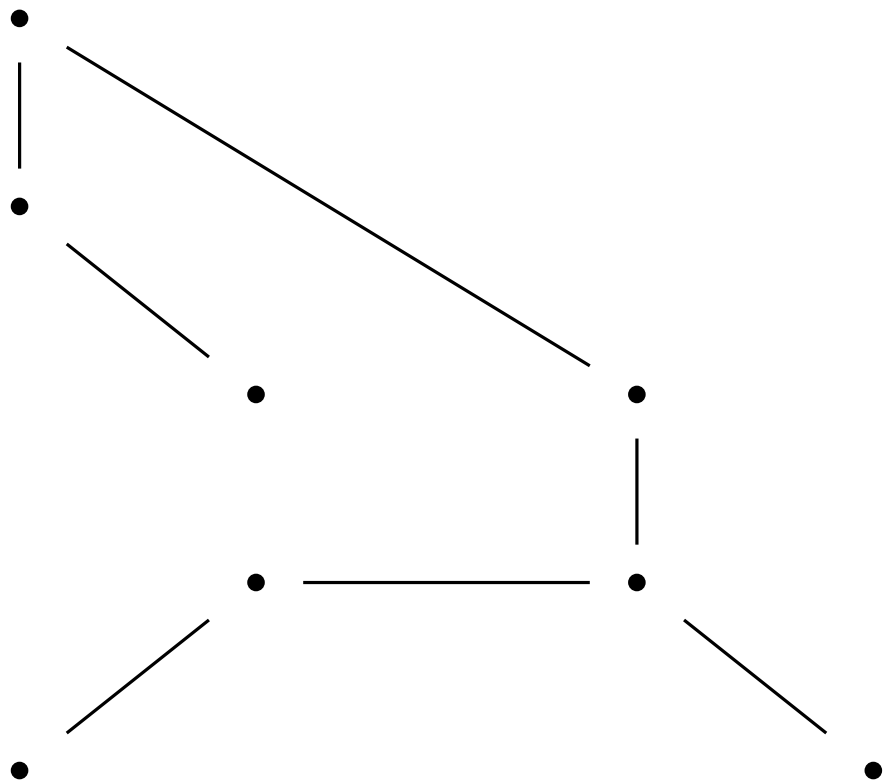
Se dice que árbol T es un **árbol de expansión** de un grafo G si:

1. T es un subgrafo de G.
2. T es un árbol.
3. T contiene todos los vértices de G.

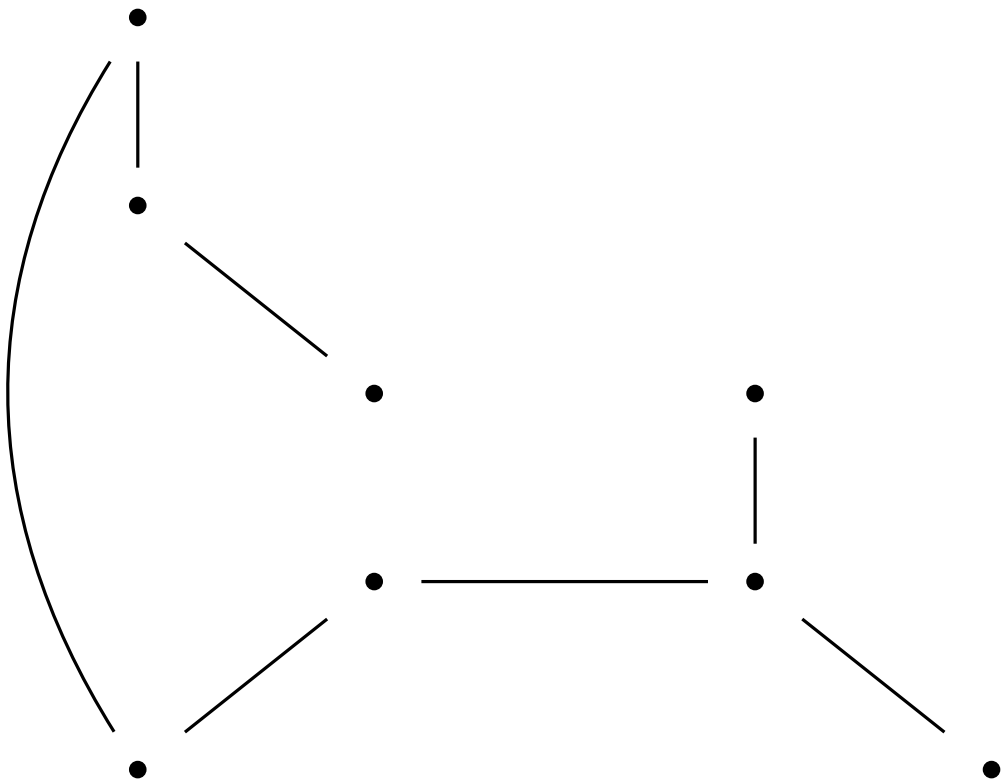
Por ejemplo, dado este grafo



Este es un árbol de expansión de ese grafo:



Nota El árbol de expansión de un grafo en general no es único, por ejemplo, acá mostramos otro árbol de expansión distinto del mismo grafo.



Un árbol de expansión para un grafo existe sí y sólo sí el grafo es conexo.

Propiedad

En efecto, todos los árboles pueden pensarse como grafos conexos sin ciclos. Si un grafo G tiene un árbol de expansión T, entonces entre dos vértices cualesquiera existe un camino en el árbol que los une (pues todos los árboles son conexos). Ahora bien, como T es subgrafo de G, necesariamente debe existir ese mismo camino en el grafo G.

El algoritmo de búsqueda en profundidad

El algoritmo de búsqueda en profundidad, normalmente llamado DFS por sus siglas en inglés *Depth First Search* permite encontrar el árbol de expansión de un grafo conexo.
La estrategia consiste en partir de un vértice determinado *v* y a partir de allí, cuando se visita un nuevo vértice, explorar cada camino que salga de él. Un camino deja de explorarse cuando se llega a un vértice ya visitado.
Si existen vértices no alcanzables, el recorrido queda incompleto; entonces, se debe "volver hacia atrás" hasta encontrar un vértice donde hayamos dejado un camino sin explorar, y repetir el proceso.

El algoritmo de búsqueda en profundidad

input: Un grafo G = (V, E)
output: Un árbol T = (V', E')
El algoritmo construye un árbol de expansión para el grafo G.
variables
T = Grafo() # Grafo sin vértices
P = Pila() # Una pila vacía
visitados = [] # Una lista de nodos visitados
elegimos origen (un vertice en V)
agregamos origen a S
mientras S no sea vacia **hacer**
v = S.pop()
visitados.append(v)
para cada w adyacente a v en G **hacer**
si w no esta en visitados **entonces**
visitados.append(w)
P.push(w)
fin mientras

El algoritmo de búsqueda en profundidad

¿Es un algoritmo?

El algoritmo de búsqueda en profundidad

¿Es un algoritmo?

No podemos dejar al azar el criterio con el que se elige el nodo inicial.

El algoritmo de búsqueda en profundidad

¿Es un algoritmo?

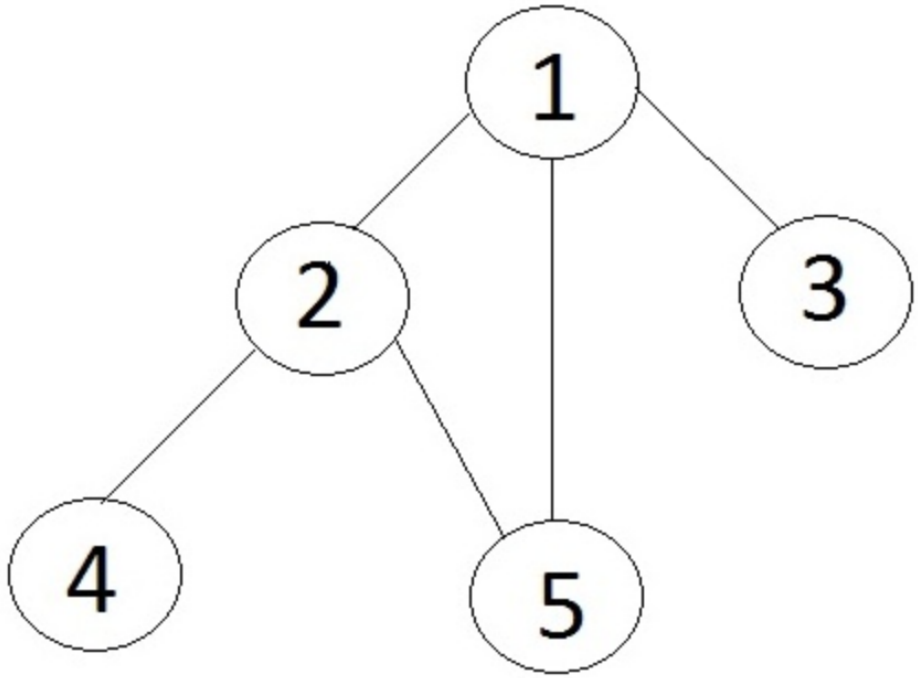
No podemos dejar al azar el criterio con el que se elige el nodo inicial.

Convención Si los nodos están etiquetados con texto, utilizaremos el orden lexicográfico para elegir el nodo inicial.

Convención Si los nodos están numerados, elegimos el orden natural de los números para elegir el nodo inicial.

El algoritmo de búsqueda en profundidad

Utilizaremos el algoritmo de búsqueda en profundidad para encontrar el árbol de expansión para el siguiente árbol:



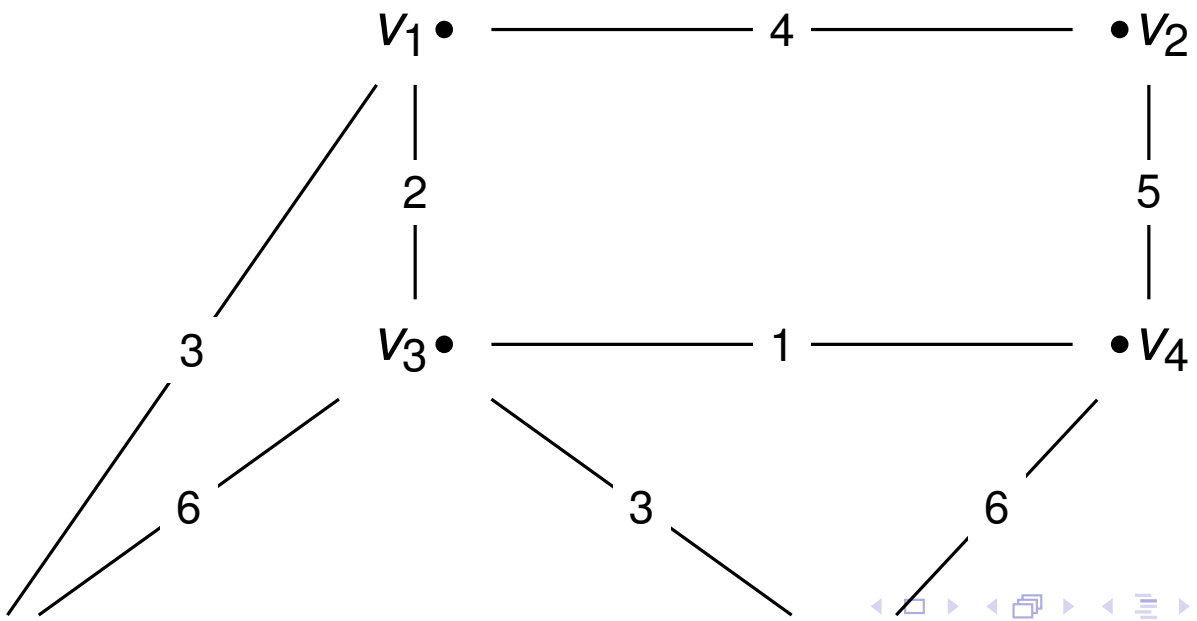
El algoritmo de búsqueda en profundidad

Aplicaciones

1. Decidir si un grafo es conexo.
2. Encontrar todas las componentes conexas de un grafo.
3. Encontrar todos los posibles caminos de un vértice a otro.

Árboles de expansión mínima

El grafo con pesos de la figura muestra seis ciudades y los costos de construir carreteras entre ellas. Se desea construir el sistema de carreteras de menor costo que conecte a las seis ciudades. La solución debe necesariamente ser un árbol de expansión ya que debe contener a todos los vértices y para ser de costo mínimo, seria redundante tener dos caminos entre ciudades. Entonces lo que necesitamos es el árbol de expansión del grafo que sea de peso mínimo.



Árboles de expansión mínima

Definición Sea G un grafo con pesos. Un árbol de expansión mínima de G es un árbol de expansión de G que tiene peso mínimo entre todos los posibles.

Nota El algoritmo DFS no asegura que el árbol encontrado sea de peso mínimo.

El algoritmo de Prim

El **algoritmo de Prim** permite encontrar un **árbol de expansión mínimo** para un grafo con pesos conexo de vértices $v_1, v_2, ...v_n$.

El algoritmo comienza con un único nodo, elegido mediante algún criterio, e incrementa continuamente el tamaño de un árbol agregando sucesivamente vértices cuya distancia a los anteriores es mínima.
Esto significa que en cada paso, las aristas a considerar son aquellas que inciden en vértices que ya pertenecen al árbol.
El árbol está completamente construido cuando no quedan más vértices por agregar.

El algoritmo de Prim

input: Un grafo G = (V, E)
output: Un árbol T = (V', E')
El algoritmo construye un árbol de expansión mínimo para el grafo G.
variables
T = Grafo() # El grafo vacio
U = [v] # v es el vértice inicial, elegido con algún criterio
mientras T != U **hacer**
elegimos (u,v) la arista más barata / u esta en U y v esta en V - U.
V' = V' + [v]
E' = E' + (u,v)
fin mientras

El algoritmo de Prim

¿Es un algoritmo?

El algoritmo de Prim

¿Es un algoritmo?

No podemos dejar al azar el criterio con el que se elige el nodo inicial.

El algoritmo de Prim

¿Es un algoritmo?

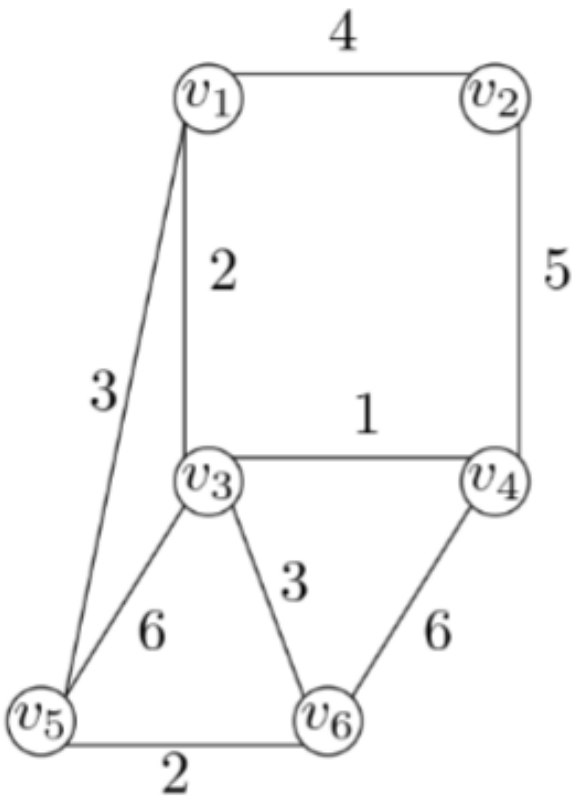
No podemos dejar al azar el criterio con el que se elige el nodo inicial.

Convención Si los nodos están etiquetados con texto, utilizaremos el orden lexicográfico para elegir el nodo inicial.

Convención Si los nodos están numerados, elegimos el orden natural de los números para elegir el nodo inicial.

El algoritmo de Prim

Utilizaremos el algoritmo de Prim para encontrar el árbol de expansión mínimo para el siguiente árbol:



El algoritmo de Kruskal

El **algoritmo de Kruskal** es otro algoritmo que permite encontrar el **árbol de expansión mínimo** para un grafo con pesos conexo de vértices $v_1, v_2, \ldots v_n$.

Este algoritmo funciona eligiendo siempre la arista de menor costo que no forme un ciclo, aunque el grafo quede desconexo.

Comienza con un grafo que contiene los mismos nodos que el grafo inicial, pero ninguna de las aristas. Luego, va a agregando aristas del menor costo posible, de forma tal que no se formen ciclos. El árbol está completamente construido cuando el grafo

resultante es conexo

El algoritmo de Kruskal

input: Un grafo G = (V, E)
output: Un árbol T = (V', E')
El algoritmo construye un árbol de expansión mínimo para el grafo G.
variables
T = Grafo(V) # El grafo con los mismos vértices, sin aristas.
mientras T no sea conexo **hacer**
elegimos (u,v) la arista más barata / T no tiene ciclos
E' = E' + (u,v)
fin mientras

El algoritmo de Kruskal

¿Es un algoritmo?

El algoritmo de Kruskal

¿Es un algoritmo?

¿Que pasa si hay un empate en varias aristas?
Convención Ordenamos las aristas utilizando los criterios ya vistos (numérico o alfabético).
Convención Si hay un empate, elegiremos la arista que aparezca antes en nuestro orden.

El algoritmo de Kruskal

Utilizaremos el algoritmo de Kruskal para encontrar el árbol de expansión mínimo para el siguiente árbol: