Grafos

Leopoldo Taravilse[1]

[1]Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Training Camp 2016

Contenidos

1. Definiciones básicas
   - Algoritmos
   - Complejidad algoritmica
   - Grafos
2. Formas de Recorrer un grafo y camino mínimo
   - BFS y DFS
   - Camino mínimo en grafos ponderados
3. Árbol Generador Mínimo
   - Algoritmo de Kruskal
   - Algoritmo de Prim
4. Componentes Fuertemente Conexas
   - Componentes Fuertemente Conexas
   - Algoritmo de Kosaraju
   - SAT-2

Contenidos

1. Definiciones básicas
   - Algoritmos
   - Complejidad algoritmica
   - Grafos
2. Formas de Recorrer un grafo y camino mínimo
   - BFS y DFS
   - Camino mínimo en grafos ponderados
3. Árbol Generador Mínimo
   - Algoritmo de Kruskal
   - Algoritmo de Prim
4. Componentes Fuertemente Conexas
   - Componentes Fuertemente Conexas
   - Algoritmo de Kosaraju
   - SAT-2

Qué es un algoritmo?

- “Un algoritmo es un conjunto preescrito de instrucciones o reglas bien definidas, ordenadas y finitas que permite realizar una actividad mediante pasos sucesivos que no generen dudas a quien deba realizar dicha actividad.” Wikipedia

Qué es un algoritmo?

- “Un algoritmo es un conjunto preescrito de instrucciones o reglas bien definidas, ordenadas y finitas que permite realizar una actividad mediante pasos sucesivos que no generen dudas a quien deba realizar dicha actividad.” Wikipedia
- “Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.” Real Academia Española

Qué es un algoritmo?

- “Un algoritmo es un conjunto preescrito de instrucciones o reglas bien definidas, ordenadas y finitas que permite realizar una actividad mediante pasos sucesivos que no generen dudas a quien deba realizar dicha actividad.” Wikipedia
- “Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.” Real Academia Española
- “Un algoritmo es una serie de pasos a seguir con el fin de obtener un resultado” Nuestra definición simplificada.

Qué es un algoritmo computacionalmente hablando?

A qué le llamamos algoritmo?

Como dijimos anteriormente, un algoritmo es una serie de pasos (o instrucciones) a seguir. En computación le decimos algoritmo a los pasos que sigue una computadora al ejecutar un programa.

Qué es un algoritmo computacionalmente hablando?

A qué le llamamos algoritmo?

Como dijimos anteriormente, un algoritmo es una serie de pasos (o instrucciones) a seguir. En computación le decimos algoritmo a los pasos que sigue una computadora al ejecutar un programa.

Le llamamos algoritmo a la idea que usamos para escribir el código y no al conjunto de instrucciones. Por ejemplo, un algoritmo para ver si un número es primo es dividirlo por los números menores o iguales a su raiz cuadrada y ver si el resto es cero, y no importa cómo lo implementemos es siempre el mismo algoritmo.

Contenidos

1. Definiciones básicas
   - Algoritmos
   - Complejidad algoritmica
   - Grafos
2. Formas de Recorrer un grafo y camino mínimo
   - BFS y DFS
   - Camino mínimo en grafos ponderados
3. Árbol Generador Mínimo
   - Algoritmo de Kruskal
   - Algoritmo de Prim
4. Componentes Fuertemente Conexas
   - Componentes Fuertemente Conexas
   - Algoritmo de Kosaraju
   - SAT-2

Complejidad de un algoritmo

Para decidir si un algoritmo es “bueno” o “malo” (podemos entender por bueno que sea eficiente) vamos a medir su complejidad.

Complejidad de un algoritmo

Para decidir si un algoritmo es “bueno” o “malo” (podemos entender por bueno que sea eficiente) vamos a medir su complejidad.
La complejidad de un algoritmo mide la utilización de los recursos disponibles. Los recursos más importantes con los que cuenta una computadora a la hora de ejecutar un programa son dos:

Complejidad de un algoritmo

Para decidir si un algoritmo es “bueno” o “malo” (podemos entender por bueno que sea eficiente) vamos a medir su complejidad.
La complejidad de un algoritmo mide la utilización de los recursos disponibles. Los recursos más importantes con los que cuenta una computadora a la hora de ejecutar un programa son dos:

- Memoria: La máxima cantidad de memoria que usa el programa al mismo tiempo. Si usa varias veces la misma memoria se cuenta una sóla vez.
- Tiempo: Cuánto tarda en correr el algoritmo. Se mide en cantidad de operaciones básicas (sumas, restas, asignaciones, etc)

Como medimos el uso de la memoria?

El uso de la memoria

El uso de la memoria lo medimos por la mayor cantidad de memoria que usa un programa al mismo tiempo. Por ejemplo, si el programa utiliza 200MB de memoria, libera 50MB y ocupa otros 150MB, la memoria usada es primero 200MB, después 150MB y por último 300MB. Por más que el programa utilize en un momento 200MB y luego ocupe otros 150MB, la memoria que utiliza el programa es 300MB y no 350MB ya que nunca utiliza 350MB al mismo tiempo.

Como medimos el uso de la memoria?

El uso de la memoria

El uso de la memoria lo medimos por la mayor cantidad de memoria que usa un programa al mismo tiempo. Por ejemplo, si el programa utiliza 200MB de memoria, libera 50MB y ocupa otros 150MB, la memoria usada es primero 200MB, después 150MB y por último 300MB. Por más que el programa utilize en un momento 200MB y luego ocupe otros 150MB, la memoria que utiliza el programa es 300MB y no 350MB ya que nunca utiliza 350MB al mismo tiempo.

En contexto de competencias depende del problema y la competencia pero por lo general el límite de memoria disponible es entre 1GB y 2GB.

Como medimos el uso del tiempo?

El uso del tiempo

El uso del tiempo lo medimos por la cantidad de operaciones básicas que ejecuta el programa, como pueden ser sumas, restas, asignaciones, etc. Es muy difícil calcular este número y por lo general nos interesa más tener una idea de cómo crece esta complejidad a medida que crece la entrada del problema, por eso hablamos de órdenes de complejidad.

Como medimos el uso del tiempo?

El uso del tiempo

El uso del tiempo lo medimos por la cantidad de operaciones básicas que ejecuta el programa, como pueden ser sumas, restas, asignaciones, etc. Es muy difícil calcular este número y por lo general nos interesa más tener una idea de cómo crece esta complejidad a medida que crece la entrada del problema, por eso hablamos de órdenes de complejidad.

Órdenes de complejidad

Decimos que una función tiene el órden de complejidad de otra función si se parecen suficiente luego de multiplicar por una constante. Existen tres formas de calcular órdenes de complejidad.

Órdenes de complejidad

Los órdenes de complejidad los podemos medir de tres maneras:

- $O(f(n))$
- $\Omega(f(n))$
- $\theta(f(n))$

Órdenes de complejidad

Los órdenes de complejidad los podemos medir de tres maneras:

- $O(f(n))$:$g(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 : n > n_0 \Rightarrow g(n) < cf(n)$
- $\Omega(f(n))$:$g(n) \in \Omega(f(n)) \Leftrightarrow \exists c, n_0 : n > n_0 \Rightarrow g(n) > cf(n)$
- $\theta(f(n))$:$g(n) \in \theta(f(n)) \Leftrightarrow g(n) \in O(f(n)) \wedge g(n) \in \Omega(f(n))$

Órdenes de complejidad

Los órdenes de complejidad los podemos medir de tres maneras:

- $O(f(n))$:$g(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 : n > n_0 \Rightarrow g(n) < cf(n)$
- $\Omega(f(n))$:$g(n) \in \Omega(f(n)) \Leftrightarrow \exists c, n_0 : n > n_0 \Rightarrow g(n) > cf(n)$
- $\theta(f(n))$:$g(n) \in \theta(f(n)) \Leftrightarrow g(n) \in O(f(n)) \wedge g(n) \in \Omega(f(n))$

La notación que solemos usar es la primera de estas tres, a la cuál le llamamos "O grande", ya que la notación Ω no es muy útil y la notación θ puede ser muy difícil de calcular y no aporta información útil.

Órdenes de complejidad

Los órdenes de complejidad los podemos medir de tres maneras:

- $O(f(n))$:$g(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 : n > n_0 \Rightarrow g(n) < cf(n)$
- $\Omega(f(n))$:$g(n) \in \Omega(f(n)) \Leftrightarrow \exists c, n_0 : n > n_0 \Rightarrow g(n) > cf(n)$
- $\theta(f(n))$:$g(n) \in \theta(f(n)) \Leftrightarrow g(n) \in O(f(n)) \wedge g(n) \in \Omega(f(n))$

La notación que solemos usar es la primera de estas tres, a la cuál le llamamos "O grande", ya que la notación Ω no es muy útil y la notación θ puede ser muy difícil de calcular y no aporta información útil.

Dado un algoritmo nos interesa conocer la complejidad del algoritmo en el peor caso, tanto en memoria como en tiempo.

Contenidos

1 Definiciones básicas
- Algoritmos
- Complejidad algoritmica
- Grafos

2 Formas de Recorrer un grafo y camino mínimo
- BFS y DFS
- Camino mínimo en grafos ponderados

3 Árbol Generador Mínimo
- Algoritmo de Kruskal
- Algoritmo de Prim

4 Componentes Fuertemente Conexas
- Componentes Fuertemente Conexas
- Algoritmo de Kosaraju
- SAT-2

Qué es un grafo?

"Un grafo es un conjunto, no vacío, de objetos llamados vértices (o nodos) y una selección de pares de vértices, llamados aristas (edges en inglés) que pueden ser orientados (dirigidos) o no." Wikipedia

Qué es un grafo?

“Un grafo es un conjunto, no vacío, de objetos llamados vértices (o nodos) y una selección de pares de vértices, llamados aristas (edges en inglés) que pueden ser orientados (dirigidos) o no.” Wikipedia
“Diagrama que representa mediante puntos y líneas las relaciones entre pares de elementos y que se usa para resolver problemas lógicos, topológicos y de cálculo combinatorio.” Real Academia Española

Qué es un grafo?

"Un grafo es un conjunto, no vacío, de objetos llamados vértices (o nodos) y una selección de pares de vértices, llamados aristas (edges en inglés) que pueden ser orientados (dirigidos) o no." Wikipedia

"Diagrama que representa mediante puntos y líneas las relaciones entre pares de elementos y que se usa para resolver problemas lógicos, topológicos y de cálculo combinatorio." Real Academia Española

"Un grafo es un conjunto de puntos y líneas que unen pares de esos puntitos" La posta

Definición formal de grafo

Grafo

Un grafo se define como un conjunto V cuyos elementos se denominan vértices o nodos, y un conjunto $E \subseteq V \times V$ cuyos elementos se llaman ejes o aristas. Un grafo puede ser dirigido, es decir, $(a, b) \in E$ no es lo mismo que $(b, a) \in E$ o no dirigido, cuando $(a, b) = (b, a)$.

Definición formal de grafo

Grafo

Un grafo se define como un conjunto V cuyos elementos se denominan vértices o nodos, y un conjunto $E \subseteq V \times V$ cuyos elementos se llaman ejes o aristas. Un grafo puede ser dirigido, es decir, $(a, b) \in E$ no es lo mismo que $(b, a) \in E$ o no dirigido, cuando $(a, b) = (b, a)$.

Ejemplo

Mediante un grafo podemos representar, por ejemplo, una ciudad. Las esquinas serían los vértices (elementos de V) y las conexiones por medio de una calle entre dos esquinas serían los ejes (elementos de E). Si las calles son mano y contramano el grafo es dirigido, si en cambio son doble mano, el grafo es no dirigido.

Contenidos

1. Definiciones básicas
   - Algoritmos
   - Complejidad algoritmica
   - Grafos
2. Formas de Recorrer un grafo y camino mínimo
   - BFS y DFS
   - Camino mínimo en grafos ponderados
3. Árbol Generador Mínimo
   - Algoritmo de Kruskal
   - Algoritmo de Prim
4. Componentes Fuertemente Conexas
   - Componentes Fuertemente Conexas
   - Algoritmo de Kosaraju
   - SAT-2

Distancia

Comenzaremos trabajando con grafos no dirigidos y daremos algunas definiciones que tienen sentido en grafos no dirigidos, pero que luego podremos adaptar a grafos dirigidos.

Distancia

Comenzaremos trabajando con grafos no dirigidos y daremos algunas definiciones que tienen sentido en grafos no dirigidos, pero que luego podremos adaptar a grafos dirigidos.

- Decimos que dos vértices v_1 y v_2 son adyacentes si $(v_1, v_2) \in E$. En este caso decimos que hay una arista entre v_1 y v_2.

Distancia

Comenzaremos trabajando con grafos no dirigidos y daremos algunas definiciones que tienen sentido en grafos no dirigidos, pero que luego podremos adaptar a grafos dirigidos.

- Decimos que dos vértices v_1 y v_2 son adyacentes si $(v_1, v_2) \in E$. En este caso decimos que hay una arista entre v_1 y v_2.
- Un camino de largo n entre v y w es una lista de $n + 1$ vértices $v = v_0, v_1, \dots, v_n = w$ tales que para $0 \leq i < n$ los vértices v_i y $v_i + 1$ son adyacentes. La distancia entre dos vértices v y w se define como el menor número n tal que existe un camino entre v y w de largo n. Si no existe ningún camino entre v y w decimos que la distancia entre v y w es ∞

Camino mínimo

- Si la distancia entre v y w es n, entonces un camino entre v y w de largo n se llama camino mínimo.

Camino mínimo

- Si la distancia entre v y w es n, entonces un camino entre v y w de largo n se llama camino mínimo.
- Un problema muy frecuente es tener que encontrar la distancia entre dos vértices, este problema se resuelve encontrando un camino mínimo entre los vértices.

Camino mínimo

- Si la distancia entre v y w es n, entonces un camino entre v y w de largo n se llama camino mínimo.
- Un problema muy frecuente es tener que encontrar la distancia entre dos vértices, este problema se resuelve encontrando un camino mínimo entre los vértices.
- Este problema es uno de los problemas más comunes de la teoría de grafos y se puede resolver de varias maneras, una de ellas es el algoritmo llamado BFS (Breadth First Search).

BFS

Breadth First Search

El BFS es un algoritmo que calcula las distancias de un nodo de un grafo a todos los demás. Para esto empieza en el nodo desde el cual queremos calcular la distancia a todos los demás y se mueve a todos sus vecinos, una vez que hizo esto, se mueve a los vecinos de los vecinos, y así hasta que recorrió todos los nodos del grafo a los que existe un camino desde el nodo inicial.

BFS

Breadth First Search

El BFS es un algoritmo que calcula las distancias de un nodo de un grafo a todos los demás. Para esto empieza en el nodo desde el cual queremos calcular la distancia a todos los demás y se mueve a todos sus vecinos, una vez que hizo esto, se mueve a los vecinos de los vecinos, y así hasta que recorrió todos los nodos del grafo a los que existe un camino desde el nodo inicial.

Representación del grafo

A la hora de implementar un algoritmo sobre un grafo es importante saber cómo vamos a representar el grafo. Hay más de una forma de representar un grafo, nosotros veremos dos de ellas.

Formas de representar un grafo

Lista de adyacencia

La lista de adyacencia es un vector de vectores de enteros, que en el i-ésimo vector tiene el número j si hay una arista entre los nodos i y j. Esta representación es la que usaremos para nuestra implementación del BFS

Formas de representar un grafo

Lista de adyacencia

La lista de adyacencia es un vector de vectores de enteros, que en el i-ésimo vector tiene el número j si hay una arista entre los nodos i y j. Esta representación es la que usaremos para nuestra implementación del BFS

Matriz de adyacencia

La matriz de adyacencia es una matriz de $n \times n$ donde n es la cantidad de nodos del grafo, que en la posición (i, j) tiene un 1 (o true) si hay una arista entre los nodos i y j y 0 (o false) sino.

Formas de representar un grafo

Lista de adyacencia

La lista de adyacencia es un vector de vectores de enteros, que en el i-ésimo vector tiene el número j si hay una arista entre los nodos i y j. Esta representación es la que usaremos para nuestra implementación del BFS

Matriz de adyacencia

La matriz de adyacencia es una matriz de $n \times n$ donde n es la cantidad de nodos del grafo, que en la posición (i, j) tiene un 1 (o true) si hay una arista entre los nodos i y j y 0 (o false) sino.

A partir de ahora en nuestras implementaciones n será la cantidad de nodos y m la cantidad de ejes.

Detalles de implementación del BFS

- Las distancias del nodo inicial a los demás nodos las inicializaremos en un virtual infinito (puede ser, por ejemplo, la cantidad de nodos del grafo, ya que es imposible que la distancia entre dos nodos del grafo sea mayor o igual a ese número.)

Detalles de implementación del BFS

- Las distancias del nodo inicial a los demás nodos las inicializaremos en un virtual infinito (puede ser, por ejemplo, la cantidad de nodos del grafo, ya que es imposible que la distancia entre dos nodos del grafo sea mayor o igual a ese número.)
- Usaremos una cola para ir agregando los nodos por visitar. Como agregamos primero todos los vecinos del nodo inicial, los primeros nodos en entrar a la cola son los de distancia 1, luego agregamos los vecinos de esos nodos, que son los de distancia 2, y así vamos recorriendo el grafo en orden de distancia al vértice inicial.

Detalles de implementación del BFS

- Las distancias del nodo inicial a los demás nodos las inicializaremos en un virtual infinito (puede ser, por ejemplo, la cantidad de nodos del grafo, ya que es imposible que la distancia entre dos nodos del grafo sea mayor o igual a ese número.)
- Usaremos una cola para ir agregando los nodos por visitar. Como agregamos primero todos los vecinos del nodo inicial, los primeros nodos en entrar a la cola son los de distancia 1, luego agregamos los vecinos de esos nodos, que son los de distancia 2, y así vamos recorriendo el grafo en orden de distancia al vértice inicial.
- Cuando visitamos un nodo, sabemos cuáles de sus vecinos agregar a la cola. Tenemos que visitar los vecinos que todavía no han sido visitados.

Implementación del BFS

```
1  vector<int> BFS(vector<vector<int> > &lista, int nodoInicial){
2      int n = lista.size(),t;
3      queue<int> cola;
4      vector<int> distancias(n,n);
5      cola.push(nodoInicial);
6      distancias[nodoInicial] = 0;
7      while(!cola.empty()){
8          t = cola.front();
9          cola.pop();
10         for(int i=0;i<lista[t].size();i++){
11             if(distancias[lista[t][i]]==n){
12                 distancias[lista[t][i]] = distancias[t]+1;
13                 cola.push(lista[t][i]);
14             }
15         }
16     }
17     return distancias;
18 }
```

Formas de recorrer un grafo

- Existen varias maneras de recorrer un grafo, cada una puede ser útil según el problema. Una forma de recorrer un grafo es el BFS, que recorre los nodos en orden de distancia a un nodo.

Formas de recorrer un grafo

- Existen varias maneras de recorrer un grafo, cada una puede ser útil según el problema. Una forma de recorrer un grafo es el BFS, que recorre los nodos en orden de distancia a un nodo.
- Otra forma de recorrer un grafo es un DFS (Depth First Search), que recorre el grafo en profundidad, es decir, empieza por el nodo inicial y en cada paso visita un nodo no visitado del nodo donde está parado, si no hay nodos por visitar vuelve para atrás.

Formas de recorrer un grafo

- Existen varias maneras de recorrer un grafo, cada una puede ser útil según el problema. Una forma de recorrer un grafo es el BFS, que recorre los nodos en orden de distancia a un nodo.
- Otra forma de recorrer un grafo es un DFS (Depth First Search), que recorre el grafo en profundidad, es decir, empieza por el nodo inicial y en cada paso visita un nodo no visitado del nodo donde está parado, si no hay nodos por visitar vuelve para atrás.
- Un problema que se puede resolver con un DFS es decidir si un grafo es un árbol.

Árboles

Definición

Un árbol es un grafo conexo acíclico. Un grafo es conexo si para todo par de vértices hay un camino que los une, y es acíclico si no contiene ciclos.

Árboles

Definición

Un árbol es un grafo conexo acíclico. Un grafo es conexo si para todo par de vértices hay un camino que los une, y es acíclico si no contiene ciclos.

- Un DFS empieza en un nodo cualquiera de un grafo, y se expande en cada paso a un vecino del nodo actual, si ya se expandio a todos los vecinos vuelve para atrás.

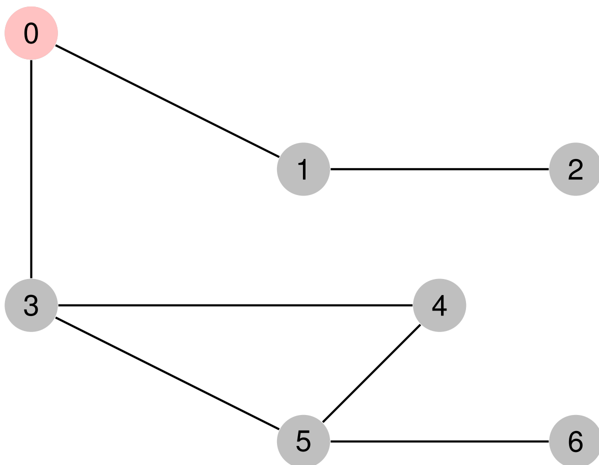
Árboles

Definición

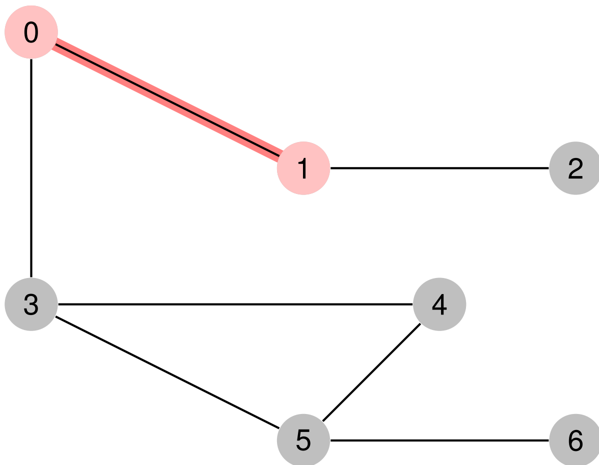
Un árbol es un grafo conexo acíclico. Un grafo es conexo si para todo par de vértices hay un camino que los une, y es acíclico si no contiene ciclos.

- Un DFS empieza en un nodo cualquiera de un grafo, y se expande en cada paso a un vecino del nodo actual, si ya se expandio a todos los vecinos vuelve para atrás.
- Si el DFS se quiere expandir a un nodo ya visitado desde otro nodo, esto quiere decir que hay un ciclo.

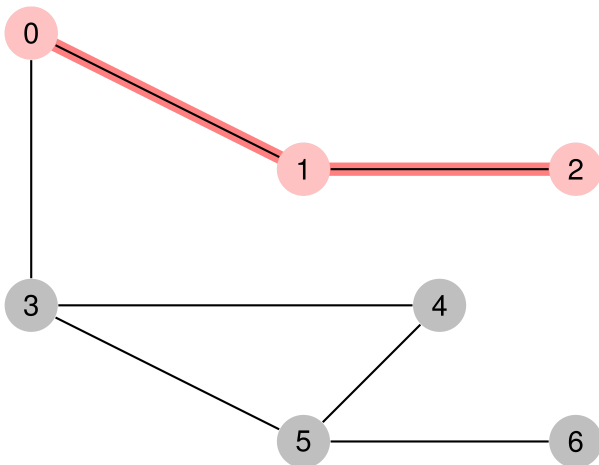
Ejemplo



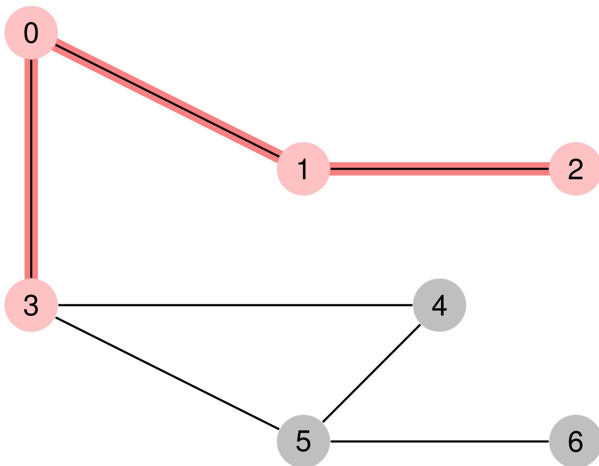
Ejemplo



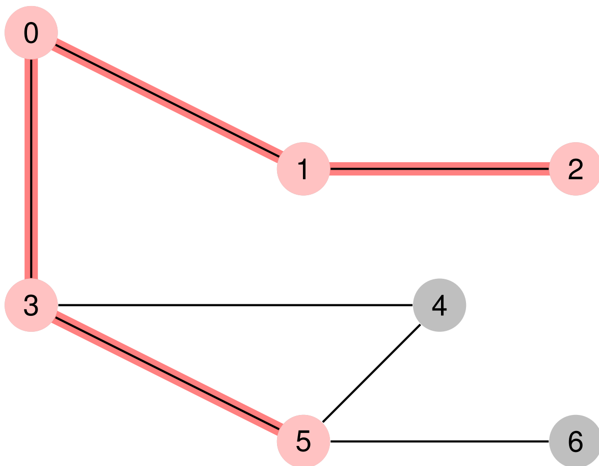
Ejemplo



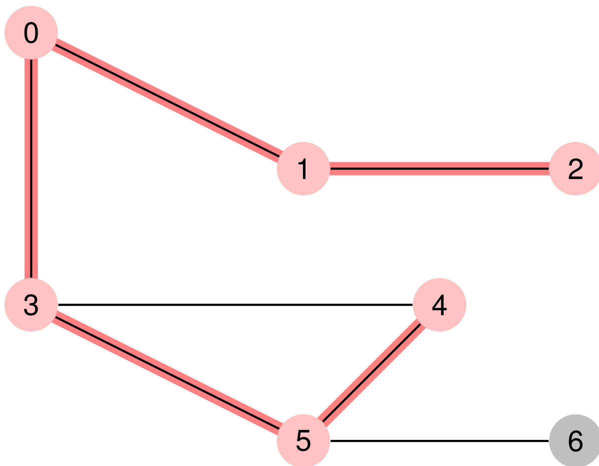
Ejemplo



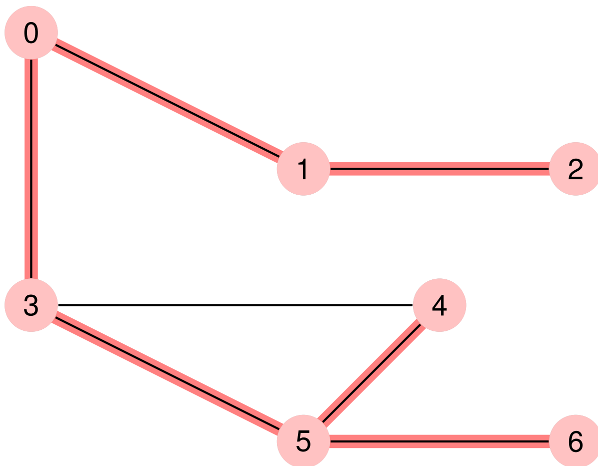
Ejemplo



Ejemplo



Ejemplo



Ejemplo

- En el ejemplo anterior vimos cómo recorre el grafo un DFS.

Ejemplo

- En el ejemplo anterior vimos cómo recorre el grafo un DFS.
- Cuando llega a un nodo que ya fue visitado se da cuenta de que pudo acceder a ese nodo por dos caminos, eso quiere decir que si recorremos uno de los caminos en un sentido y el otro camino en sentido opuesto formamos un ciclo.

Ejemplo

- En el ejemplo anterior vimos cómo recorre el grafo un DFS.
- Cuando llega a un nodo que ya fue visitado se da cuenta de que pudo acceder a ese nodo por dos caminos, eso quiere decir que si recorremos uno de los caminos en un sentido y el otro camino en sentido opuesto formamos un ciclo.
- Así como el BFS se implementa con una cola, el DFS se implementa con una pila.

Implementación del DFS

```
1  bool esArbol(vector<vector<int> > &lista, int t, vector<bool> &toc, int
       padre)
2  {
3      toc[t] = true;
4      for(int i=0;i<lista[t].size();i++)
5      {
6          if((toc[lista[t][i]]==true&&lista[t][i]!=padre))
7              return false;
8          if(toc[lista[t][i]]==false)
9          if(esArbol(lista,lista[t][i],toc,t)==false))
10             return false;
11     }
12     return true;
13 }
```

Análisis del DFS

- En ningún momento usamos una pila explícita, pero la pila está implícita en la recursión.

Análisis del DFS

- En ningún momento usamos una pila explícita, pero la pila está implícita en la recursión.
- Guardamos el padre del nodo, es decir, el nodo desde el cuál fuimos a parar al nodo actual, para no confundir un ciclo con ir y volver por la misma arista.

Análisis del DFS

- En ningún momento usamos una pila explícita, pero la pila está implícita en la recursión.
- Guardamos el padre del nodo, es decir, el nodo desde el cuál fuimos a parar al nodo actual, para no confundir un ciclo con ir y volver por la misma arista.
- Si el nodo ya lo visitamos y no es el nodo desde el cual venimos quiere decir que desde algún nodo llegamos por dos caminos, o sea que hay un ciclo.

Análisis del DFS

- En ningún momento usamos una pila explícita, pero la pila está implícita en la recursión.
- Guardamos el padre del nodo, es decir, el nodo desde el cuál fuimos a parar al nodo actual, para no confundir un ciclo con ir y volver por la misma arista.
- Si el nodo ya lo visitamos y no es el nodo desde el cual venimos quiere decir que desde algún nodo llegamos por dos caminos, o sea que hay un ciclo.
- Si el nodo no lo visitamos, pero desde uno de sus vecinos podemos llegar a un ciclo, entonces es porque hay un ciclo en el grafo y por lo tanto no es un árbol.

Análisis del DFS

- En ningún momento usamos una pila explícita, pero la pila está implícita en la recursión.
- Guardamos el padre del nodo, es decir, el nodo desde el cuál fuimos a parar al nodo actual, para no confundir un ciclo con ir y volver por la misma arista.
- Si el nodo ya lo visitamos y no es el nodo desde el cual venimos quiere decir que desde algún nodo llegamos por dos caminos, o sea que hay un ciclo.
- Si el nodo no lo visitamos, pero desde uno de sus vecinos podemos llegar a un ciclo, entonces es porque hay un ciclo en el grafo y por lo tanto no es un árbol.
- Faltan tener en cuenta algunas consideraciones. ¿Se dan cuenta cuáles?

Análisis del DFS

- t es el nodo en el que estamos parados, *toc* guarda los nodos que ya tocamos, y *padre* es el nodo desde el que venimos.

Análisis del DFS

- *t* es el nodo en el que estamos parados, *toc* guarda los nodos que ya tocamos, y *padre* es el nodo desde el que venimos.
- *toc* empieza inicializado en false y el tamaño de *toc* es el mismo que el de *lista*.

Análisis del DFS

- *t* es el nodo en el que estamos parados, *toc* guarda los nodos que ya tocamos, y *padre* es el nodo desde el que venimos.
- *toc* empieza inicializado en false y el tamaño de *toc* es el mismo que el de *lista*.
- Estamos asumiendo que el grafo es conexo, ya que si no lo es la función *esArbol* puede devolver true sin ser un árbol

Análisis del DFS

- *t* es el nodo en el que estamos parados, *toc* guarda los nodos que ya tocamos, y *padre* es el nodo desde el que venimos.
- *toc* empieza inicializado en false y el tamaño de *toc* es el mismo que el de *lista*.
- Estamos asumiendo que el grafo es conexo, ya que si no lo es la función *esArbol* puede devolver true sin ser un árbol
- La forma de chequear si el grafo es conexo es chequear que hayamos tocado todos los nodos, es decir, que todas las posiciones de *toc* terminen en true.

Complejidades

- Para saber si un algoritmo es bueno en cuanto a su tiempo de ejecución es muy útil conocer su complejidad temporal, es decir, una función evaluada en el tamaño del problema que nos de una cota de la cantidad de operaciones del algoritmo.

Complejidades

- Para saber si un algoritmo es bueno en cuanto a su tiempo de ejecución es muy útil conocer su complejidad temporal, es decir, una función evaluada en el tamaño del problema que nos de una cota de la cantidad de operaciones del algoritmo.
- Por ejemplo, si el problema tiene un tamaño de entrada n la complejidad podría ser $O(< n^2)$, esto quiere decir que existe una constante c tal que el algoritmo tarda menos de cn^2 para todos los casos posibles.

Complejidades

- Para saber si un algoritmo es bueno en cuanto a su tiempo de ejecución es muy útil conocer su complejidad temporal, es decir, una función evaluada en el tamaño del problema que nos de una cota de la cantidad de operaciones del algoritmo.
- Por ejemplo, si el problema tiene un tamaño de entrada n la complejidad podría ser $O(< n^2)$, esto quiere decir que existe una constante c tal que el algoritmo tarda menos de cn^2 para todos los casos posibles.
- El cálculo de complejidades es un análisis teórico y no tiene en cuenta la constante c que puede ser muy chica o muy grande, sin embargo es por lo general una buena referencia para saber si un algoritmo es bueno o malo.

Complejidades

- Para saber si un algoritmo es bueno en cuanto a su tiempo de ejecución es muy útil conocer su complejidad temporal, es decir, una función evaluada en el tamaño del problema que nos de una cota de la cantidad de operaciones del algoritmo.
- Por ejemplo, si el problema tiene un tamaño de entrada n la complejidad podría ser $O(< n^2)$, esto quiere decir que existe una constante c tal que el algoritmo tarda menos de cn^2 para todos los casos posibles.
- El cálculo de complejidades es un análisis teórico y no tiene en cuenta la constante c que puede ser muy chica o muy grande, sin embargo es por lo general una buena referencia para saber si un algoritmo es bueno o malo.
- La complejidad del BFS y del DFS es $O(n + m) = O(m)$

Contenidos

1 Definiciones básicas
- Algoritmos
- Complejidad algoritmica
- Grafos

2 Formas de Recorrer un grafo y camino mínimo
- BFS y DFS
- Camino mínimo en grafos ponderados

3 Árbol Generador Mínimo
- Algoritmo de Kruskal
- Algoritmo de Prim

4 Componentes Fuertemente Conexas
- Componentes Fuertemente Conexas
- Algoritmo de Kosaraju
- SAT-2

Ejes con peso

- Hasta ahora en los grafos que vimos la distancia entre dos nodos vecinos siempre fue 1.

Ejes con peso

- Hasta ahora en los grafos que vimos la distancia entre dos nodos vecinos siempre fue 1.
- Un grafo ponderado es un grafo en el cuál los ejes tienen peso. Los pesos de los ejes pueden ser números en $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, etc.

Ejes con peso

- Hasta ahora en los grafos que vimos la distancia entre dos nodos vecinos siempre fue 1.
- Un grafo ponderado es un grafo en el cuál los ejes tienen peso. Los pesos de los ejes pueden ser números en $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, etc.
- La distancia entre dos nodos v y w es el menor d tal que existen $v = v_0, v_1, \ldots, v_n = w$ tales que si p_i es el peso del eje que une v_i y v_{i+1} entonces $d = \sum_{i=0}^{n-1} p_i$.

Algoritmo de Dijkstra

- El BFS ya no sirve para calcular el camino mínimo entre dos nodos de un grafo ponderado.

Algoritmo de Dijkstra

- El BFS ya no sirve para calcular el camino mínimo entre dos nodos de un grafo ponderado.
- Para solucionar ese problema, existe, entre otros, el algoritmo de Dijkstra.

Algoritmo de Dijkstra

- El BFS ya no sirve para calcular el camino mínimo entre dos nodos de un grafo ponderado.
- Para solucionar ese problema, existe, entre otros, el algoritmo de Dijkstra.
- El algoritmo de Dijkstra calcula dado un vértice la distancia mínima a todos los demás vértices desde ese vértice en un grafo ponderado.

Qué hace el algoritmo de Dijkstra?

Algoritmo de Dijkstra

El algoritmo de Dijkstra empieza en el vértice inicial, que empieza con distancia cero, y todos los demás vértices empiezan en distancia infinito, en cada paso el algoritmo actualiza las distancias de los vecinos del nodo actual cambiándolas, si las hace más chicas, por la distancia al nodo actual más el peso del eje que los une. Luego elige el vértice aún no visitado de distancia más chica y se mueve a ese vértice.

Hay varias versiones del algoritmo de Dijkstra de las cuales nosotros veremos dos, con y sin cola de prioridad, la diferencia está en cómo se elige a qué nodo moverse.

Distintas versiones de Dijkstra

Dijkstra sin cola de prioridad

Una vez que actualiza las distancias para elegir a qué nodo moverse revisa todos los nodos del grafo uno por uno y se queda con el más cercano aún no visitado.

Dijkstra con cola de prioridad

En cada paso guarda en una cola de prioridad los nodos aún no visitados ordenados según su distancia. La cola de prioridad es una estructura en la que se puede insertar y borrar en tiempo logarítmico en función de la cantidad de elementos de la cola y consultar el menor en tiempo constante. Así, siempre sabe en tiempo constante qué nodo es el próximo a visitar.

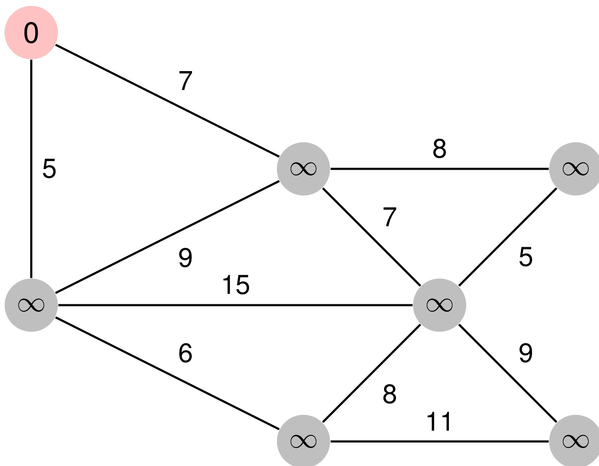
Distintas versiones de Dijkstra

- Nosotros veremos sólo el algoritmo sin cola de prioridad porque es más fácil de implementar el algoritmo sin la cola.

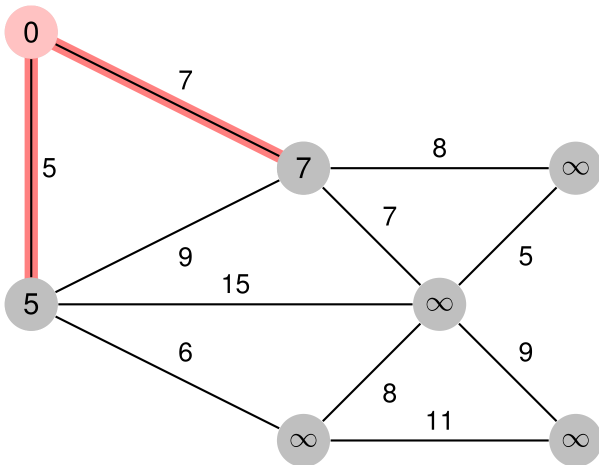
Distintas versiones de Dijkstra

- Nosotros veremos sólo el algoritmo sin cola de prioridad porque es más fácil de implementar el algoritmo sin la cola.
- La cola de prioridad es una estructura difícil de implementar pero el set que viene en la STL de C++ funciona como una cola de prioridad.

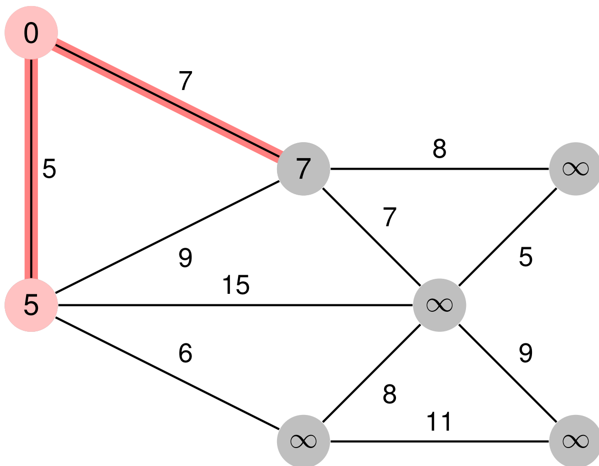
Ejemplo



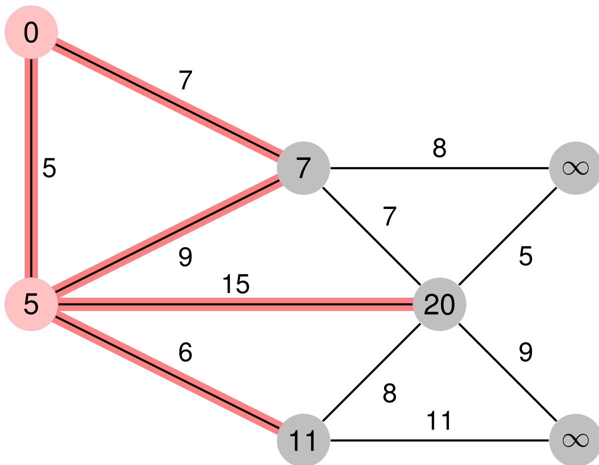
Ejemplo



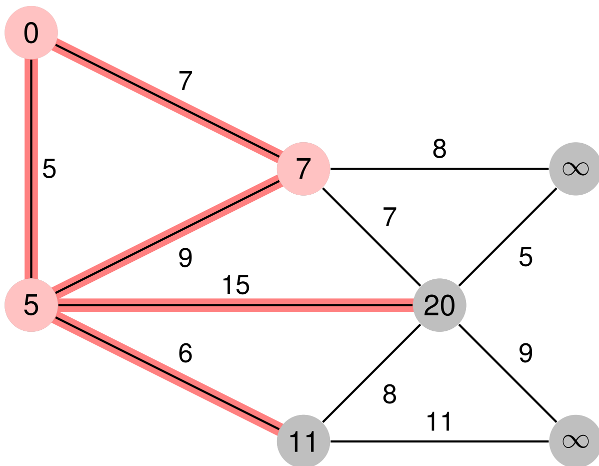
Ejemplo



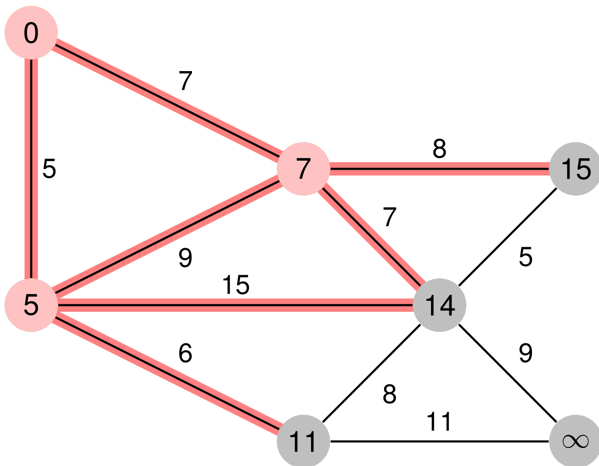
Ejemplo



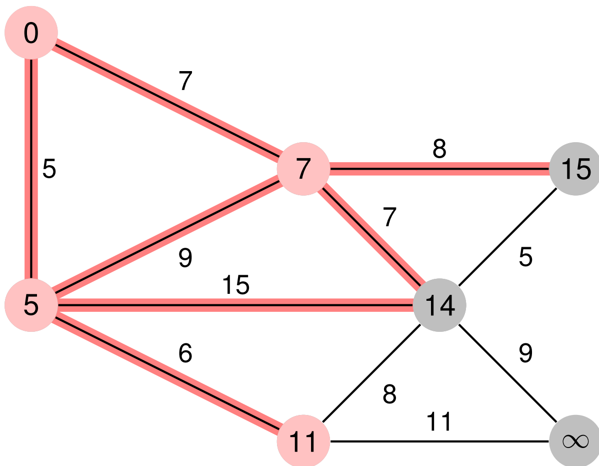
Ejemplo



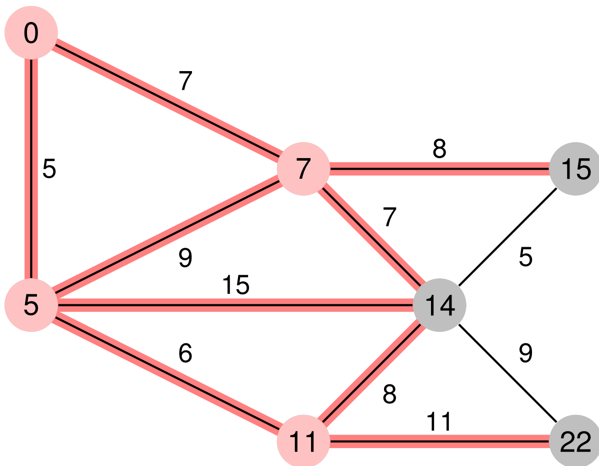
Ejemplo



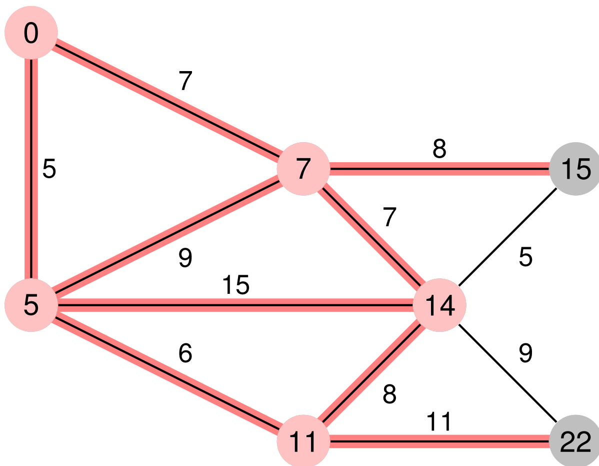
Ejemplo



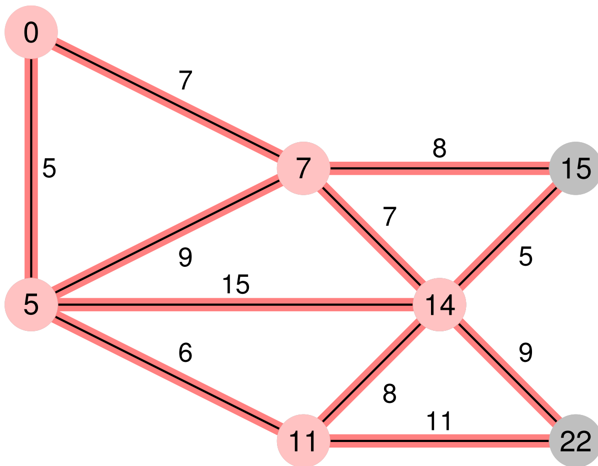
Ejemplo



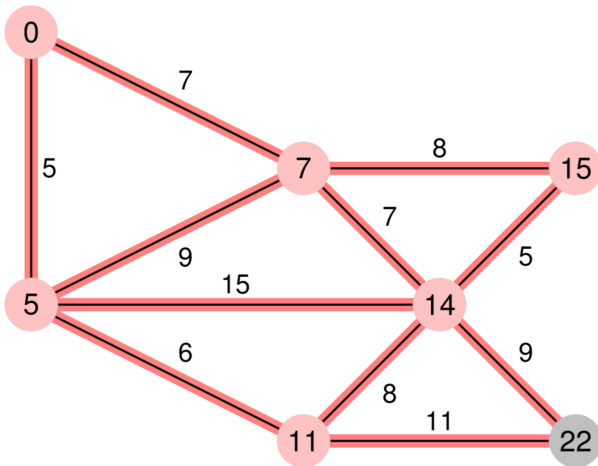
Ejemplo



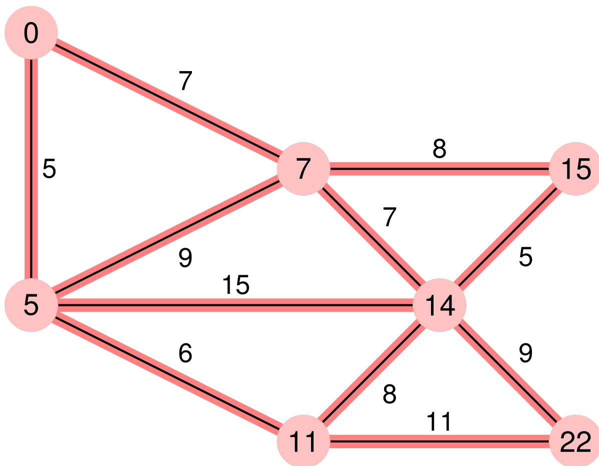
Ejemplo



Ejemplo



Ejemplo



Implementación de Dijkstra

```
1  vector<int> dijkstra(vector<vector<pair<int,int> > > &lista,int nodoInicial)
2  {
3      int n = lista.size();
4      vector<int> dist(n,INF);
5      vector<bool> toc(n,false);
6      dist[nodoInicial] = 0;
7      int t = nodoInicial;
8      for(int i=0;i<n;i++){
9          toc[t] = true;
10         for(int i=0;i<lista[t].size();i++)
11             dist[lista[t][i].first] = min(dist[lista[t][i].first],dist[t]+
                   lista[t][i].second);
12         for(int i=0;i<n;i++)
13         if(toc[t]==true||(toc[i]==false&&dist[i]<dist[t]))
14             t = i;
15     }
16     return dist;
17 }
```

Análisis de Dijkstra

- Cuando usamos el valor INF es un valor previamente definido como un virtual infinito, puede ser, por ejemplo, la suma de todos los pesos de los ejes más uno.

Análisis de Dijkstra

- Cuando usamos el valor INF es un valor previamente definido como un virtual infinito, puede ser, por ejemplo, la suma de todos los pesos de los ejes más uno.
- La complejidad del algoritmo de Dijkstra es $O(n^2 + m)$ y como $E = O(n^2)$ entonces la complejidad es $O(n^2)$. La versión con cola de prioridad tiene complejidad $O((m + n) \log n)$

Análisis de Dijkstra

- Cuando usamos el valor INF es un valor previamente definido como un virtual infinito, puede ser, por ejemplo, la suma de todos los pesos de los ejes más uno.
- La complejidad del algoritmo de Dijkstra es $O(n^2 + m)$ y como $E = O(n^2)$ entonces la complejidad es $O(n^2)$. La versión con cola de prioridad tiene complejidad $O((m + n) \log n)$
- El algoritmo de Dijkstra funciona sólo con pesos no negativos. Para calcular camino mínimo en un grafo con pesos negativos existe el algoritmo de Bellman-Ford, que además de calcular camino mínimo detecta ciclos negativos.

Grafos dirigidos con ciclos negativos

- En un grafo no dirigido conexo, si hay un ciclo negativo podemos movernos por ese ciclo y llegar a cualquier nodo con un costo arbitrariamente bajo.

Grafos dirigidos con ciclos negativos

- En un grafo no dirigido conexo, si hay un ciclo negativo podemos movernos por ese ciclo y llegar a cualquier nodo con un costo arbitrariamente bajo.
- Es por eso que tiene sentido ver si en un grafo hay ciclos negativos cuando el grafo es dirigido.

Grafos dirigidos con ciclos negativos

- En un grafo no dirigido conexo, si hay un ciclo negativo podemos movernos por ese ciclo y llegar a cualquier nodo con un costo arbitrariamente bajo.
- Es por eso que tiene sentido ver si en un grafo hay ciclos negativos cuando el grafo es dirigido.
- Para esto vamos a usar el algoritmo de Bellman Ford, y para este algoritmo veremos una nueva forma de representar un grafo que es la lista de incidencia.

Lista de incidencia

Definición

La lista de incidencia de un grafo es una lista de ejes de un grafo, en donde cada eje se representa por sus nodos incidentes como pares (ordenados si el grafo es dirigido), más el costo del eje si el grafo es ponderado.

Lista de incidencia

Definición

La lista de incidencia de un grafo es una lista de ejes de un grafo, en donde cada eje se representa por sus nodos incidentes como pares (ordenados si el grafo es dirigido), más el costo del eje si el grafo es ponderado.

Ventajas de la lista de incidencia

Cuando sólo tenemos que iterar sobre los ejes sin importar si lo hacemos en un orden particular según qué ejes inciden en qué nodos, suele ser más eficiente que revisar listas de adyacencias con muchos nodos sin ejes que son iteraciones que consumen tiempo y no hacen nada.

Bellman-Ford

- El algoritmo de Bellman Ford itera hasta $n - 1$ veces sobre cada eje. Si moverse por ese eje disminuye la distancia desde el nodo inicial hasta el nodo hacia el cuál nos movemos por ese eje, entonces actualiza la distancia a la que obtenemos moviendonos por ese eje.

Bellman-Ford

- El algoritmo de Bellman Ford itera hasta $n - 1$ veces sobre cada eje. Si moverse por ese eje disminuye la distancia desde el nodo inicial hasta el nodo hacia el cuál nos movemos por ese eje, entonces actualiza la distancia a la que obtenemos moviendonos por ese eje.
- Esto funciona porque un camino mínimo tiene como máximo $n - 1$ ejes, entonces el i-ésimo paso actualizamos la distancia al $(i + 1)$-ésimo nodo del camino mínimo.

Bellman-Ford

- El algoritmo de Bellman Ford itera hasta $n-1$ veces sobre cada eje. Si moverse por ese eje disminuye la distancia desde el nodo inicial hasta el nodo hacia el cuál nos movemos por ese eje, entonces actualiza la distancia a la que obtenemos moviendonos por ese eje.
- Esto funciona porque un camino mínimo tiene como máximo $n-1$ ejes, entonces el i-ésimo paso actualizamos la distancia al $(i+1)$-ésimo nodo del camino mínimo.
- Si luego de los $n-1$ pasos podemos seguir disminuyendo la distancia a un nodo quiere decir que hay un camino de longitud al menos n que asigna una menor distancia que todos los caminos más chicos, pero este camino contiene un ciclo, luego hay un ciclo negativo.

Bellman-Ford

- Si hay un ciclo negativo entonces el camino mínimo a los nodos a los que se puede llegar desde un nodo del ciclo no existe porque hay caminos arbitrariamente chicos.

Bellman-Ford

- Si hay un ciclo negativo entonces el camino mínimo a los nodos a los que se puede llegar desde un nodo del ciclo no existe porque hay caminos arbitrariamente chicos.
- Vamos a ver una versión del algoritmo que sólo nos dice si hay o no ciclos negativos

Bellman-Ford

- Si hay un ciclo negativo entonces el camino mínimo a los nodos a los que se puede llegar desde un nodo del ciclo no existe porque hay caminos arbitrariamente chicos.
- Vamos a ver una versión del algoritmo que sólo nos dice si hay o no ciclos negativos
- Cuando guardamos los padres de cada nodo, lo hacemos para poder luego modificar el algoritmo para que nos diga para qué nodos está definida la distancia desde el nodo inicial y para esos nodos podemos obtener la distancia.

Implementación de Bellman-Ford

```
int padresBF[1000]; int distBF[1000];
bool bFord(vector<pair<double,pair<int,int> > > lista, int n, int source){
    int m = lista.size();
    for(int i=0;i<n;i++){
        distBF[i] = INF; padresBF[i] = i;
    distBF[source] = 0;
    for(int i=0;i<n;i++)for(int j=0;j<m;j++)
    if(distBF[lista[j].second.second] > distBF[lista[j].second.first]+ lista
        [j].first){
        distBF[lista[j].second.second] = distBF[lista[j].second.first]+
            lista[j].first;
padresBF[lista[j].second.second] = lista[j].second.first;
    for(int j=0;j<n;j++)
    if(distBF[lista[j].second.second] > distBF[lista[j].second.first]+ lista
        [j].first)
        return false;
    return true;
}
```

Algoritmo de Floyd-Warshall

- Hasta ahora vimos cómo calcular el camino mínimo desde un nodo hasta todos los demás.

Algoritmo de Floyd-Warshall

- Hasta ahora vimos cómo calcular el camino mínimo desde un nodo hasta todos los demás.
- ¿Qué pasa si queremos calcular la distancia de todos los nodos a todos los nodos?

Algoritmo de Floyd-Warshall

- Hasta ahora vimos cómo calcular el camino mínimo desde un nodo hasta todos los demás.
- ¿Qué pasa si queremos calcular la distancia de todos los nodos a todos los nodos?
- Si el grafo es no ponderado, podemos hacer un BFS para cada nodo, esto funciona tanto con grafos dirigidos como con grafos no dirigidos, si en cambio el grafo es ponderado, el algoritmo de Bellman Ford funciona en ambos casos pero es muy costoso, para esto existe el algoritmo de Floyd Warshal, que incluso funciona con ejes de peso negativo.

Algoritmo de Floyd-Warshall

- El algoritmo de Floyd-Warshall (también conocido como algoritmo de Floyd) va compactando aristas. En el i-ésimo paso compacta las que unen los vértices v con el i-ésimo y el i-ésimo con w generando, si la distancia es más chica que la que hay hasta el momento, una arista entre v y w.

Algoritmo de Floyd-Warshall

- El algoritmo de Floyd-Warshall (también conocido como algoritmo de Floyd) va compactando aristas. En el i-ésimo paso compacta las que unen los vértices v con el i-ésimo y el i-ésimo con w generando, si la distancia es más chica que la que hay hasta el momento, una arista entre v y w.
- Su complejidad es $O(n^3)$ y se implementa con una matriz de adyacencia. Su implementación es mucho más sencila que la de los algoritmos que vimos anteriormente.

Algoritmo de Floyd-Warshall

- El algoritmo de Floyd-Warshall (también conocido como algoritmo de Floyd) va compactando aristas. En el i-ésimo paso compacta las que unen los vértices v con el i-ésimo y el i-ésimo con w generando, si la distancia es más chica que la que hay hasta el momento, una arista entre v y w.
- Su complejidad es $O(n^3)$ y se implementa con una matriz de adyacencia. Su implementación es mucho más sencila que la de los algoritmos que vimos anteriormente.
- Los nodos i tales que el camino mínimo de i a i tienen peso negativo son los que participan de un ciclo negativo.

Implementación del algoritmo de Floyd

```
1 void floyd(vector<vector<int> > &matriz){
2     for(int k=0;k<n;k++)
3     for(int i=0;i<n;i++)
4     for(int j=0;j<n;j++)
5         matriz[i][j] = min(matriz[i][j],matriz[i][k]+matriz[k][j]);
6 }
```

Implementación del algoritmo de Floyd

```
1  void floyd(vector<vector<int> > &matriz){
2      for(int k=0;k<n;k++)
3      for(int i=0;i<n;i++)
4      for(int j=0;j<n;j++)
5          matriz[i][j] = min(matriz[i][j],matriz[i][k]+matriz[k][j]);
6  }
```

- La matriz empieza inicializada con las distancias dadas por los ejes en donde hay ejes y con infinito en todas las demás posiciones, además, la diagonal principal empieza inicializada en cero.

Contenidos

1 Definiciones básicas
- Algoritmos
- Complejidad algoritmica
- Grafos

2 Formas de Recorrer un grafo y camino mínimo
- BFS y DFS
- Camino mínimo en grafos ponderados

3 Árbol Generador Mínimo
- Algoritmo de Kruskal
- Algoritmo de Prim

4 Componentes Fuertemente Conexas
- Componentes Fuertemente Conexas
- Algoritmo de Kosaraju
- SAT-2

Árbol generador mínimo

Definición

Sea $G = (V, E)$ un grafo ponderado conexo. Un árbol generador mínimo de G es un árbol $G' = (V, E')$ tal que la suma de los costos de las aristas de G' es mínima.

Árbol generador mínimo

Definición

Sea $G = (V, E)$ un grafo ponderado conexo. Un árbol generador mínimo de G es un árbol $G' = (V, E')$ tal que la suma de los costos de las aristas de G' es mínima.

- Existen dos algoritmos conocidos que resuelven este problema. Uno de ello es el algoritmo de Kruskal, y el otro el algoritmo de Prim.

Árbol generador mínimo

Definición

Sea $G = (V, E)$ un grafo ponderado conexo. Un árbol generador mínimo de G es un árbol $G' = (V, E')$ tal que la suma de los costos de las aristas de G' es mínima.

- Existen dos algoritmos conocidos que resuelven este problema. Uno de ello es el algoritmo de Kruskal, y el otro el algoritmo de Prim.
- Vamos a ver el algoritmo de Kruskal en detalle y contaremos cómo funciona el algoritmo de Prim sin entrar en detalles ni dar el código.

Union-Find

- Antes de ver cómo funciona y saber qué hace el algoritmo de Kruskal hay que conocer una estructura de datos llamada Union-Find.

Union-Find

- Antes de ver cómo funciona y saber qué hace el algoritmo de Kruskal hay que conocer una estructura de datos llamada Union-Find.
- Esta estructura se utiliza para trabajar con componentes conexas, comienza con todos los nodos del grafo como componentes conexas separadas y en cada paso de unión junta dos componentes en una sóla componente. Las consultas consisten en dar dos nodos y preguntar si están en una misma componente.

Union-Find

- Antes de ver cómo funciona y saber qué hace el algoritmo de Kruskal hay que conocer una estructura de datos llamada Union-Find.
- Esta estructura se utiliza para trabajar con componentes conexas, comienza con todos los nodos del grafo como componentes conexas separadas y en cada paso de unión junta dos componentes en una sóla componente. Las consultas consisten en dar dos nodos y preguntar si están en una misma componente.
- Una implementación eficiente como la que daremos a continuación tiene como complejidad, y no lo vamos a dar ni a demostrar porque es muy difícil, una función que crece muy lento, y es casi lineal.

Implementación de Union-Find

```
int padres[1000];
int prof[1000];
void initUF(int n){
    for(int i=0;i<n;i++){ padres[i] = i; prof[i] = 0;}
}
int find(int x){
    if(padres[x] == x) return x;
    padres[x] = find(padres[x]);
    return padres[x];
}
void join(int x, int y){
    int xRaiz = find(x), yRaiz = find(y);
    if(prof[xRaiz]<prof[yRaiz]) padres[xRaiz] = yRaiz;
    else if(prof[xRaiz]>prof[yRaiz]) padres[yRaiz] = xRaiz;
    else{
        padres[yRaiz] = xRaiz;
        prof[xRaiz]++;
    }
}
```

Algoritmo de Kruskal

- El algoritmo de Kruskal ordena las aristas por peso y va agregando desde la arista de menor peso hasta la de mayor peso, evitando agregar las aristas que forman ciclos.

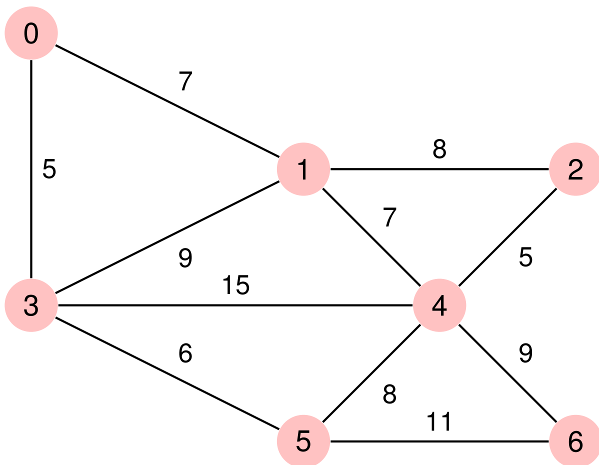
Algoritmo de Kruskal

- El algoritmo de Kruskal ordena las aristas por peso y va agregando desde la arista de menor peso hasta la de mayor peso, evitando agregar las aristas que forman ciclos.
- El árbol generador mínimo no es único y Kruskal puede encontrar todos los AGM según como se ordenen las aristas de igual peso.

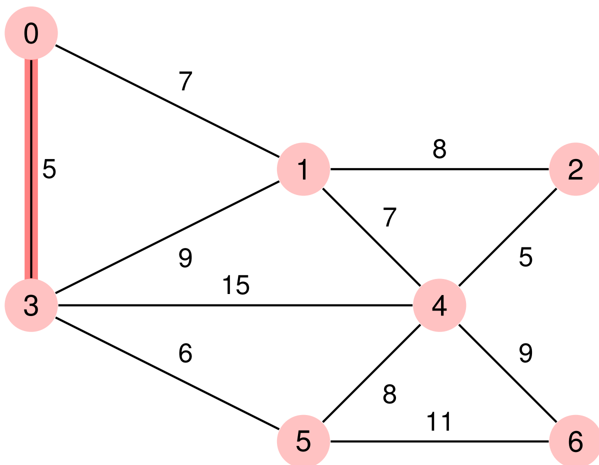
Algoritmo de Kruskal

- El algoritmo de Kruskal ordena las aristas por peso y va agregando desde la arista de menor peso hasta la de mayor peso, evitando agregar las aristas que forman ciclos.
- El árbol generador mínimo no es único y Kruskal puede encontrar todos los AGM según como se ordenen las aristas de igual peso.
- Ahora es fácil ver que la definición de AGM es equivalente a decir que la arista de mayor peso entre todos los árboles generadores tenga peso mínimo.

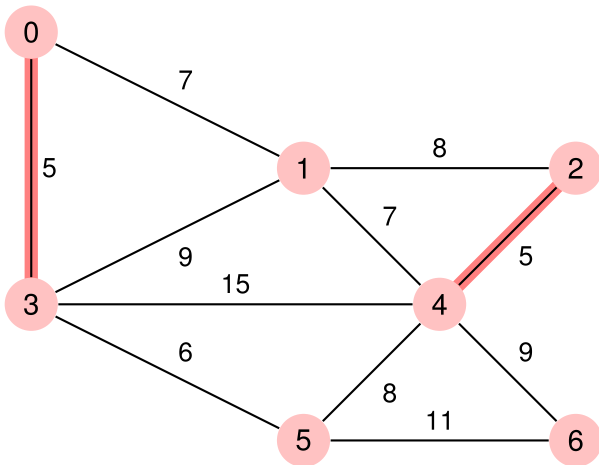
Ejemplo



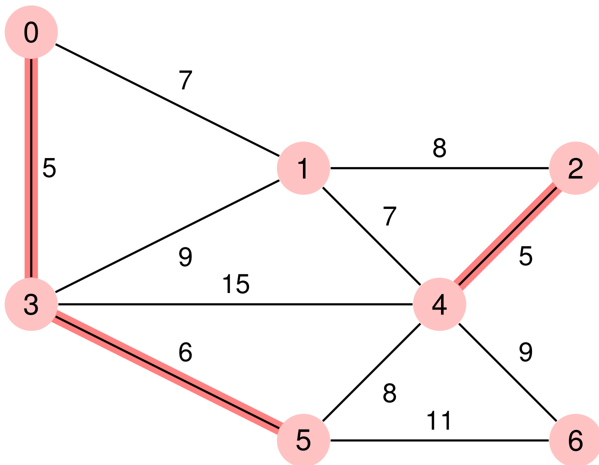
Ejemplo



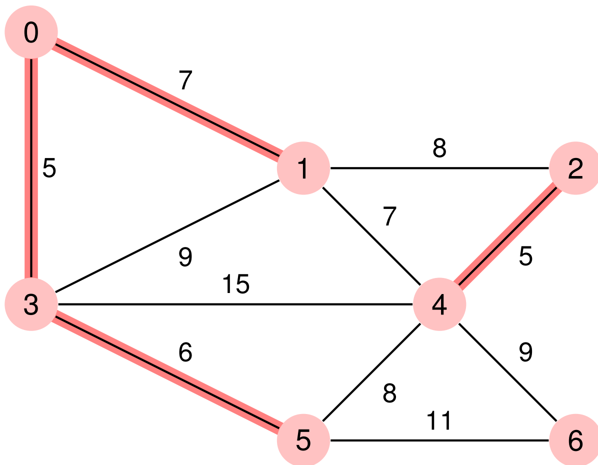
Ejemplo



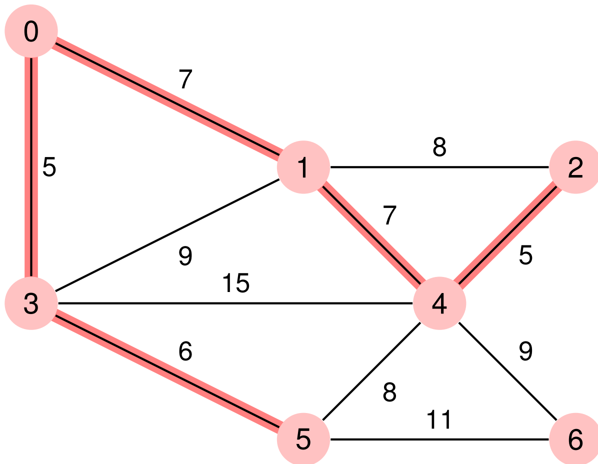
Ejemplo



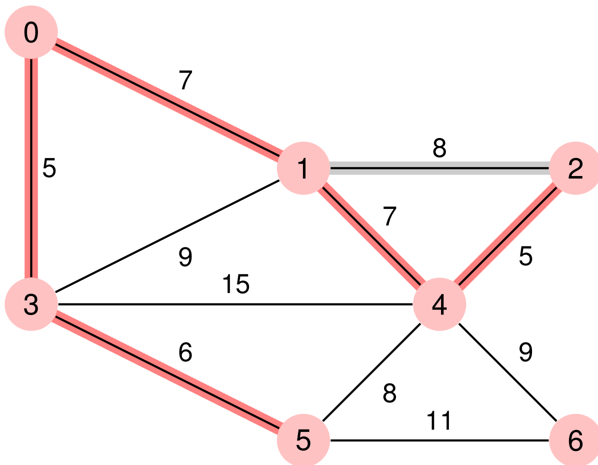
Ejemplo



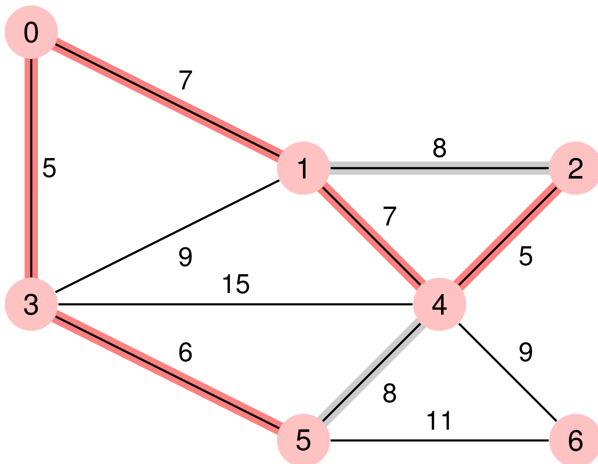
Ejemplo



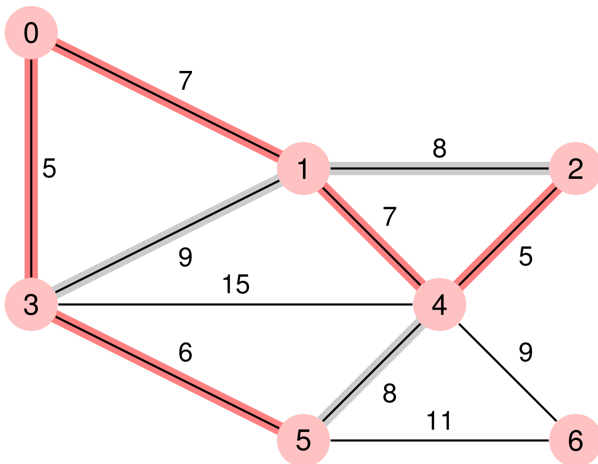
Ejemplo



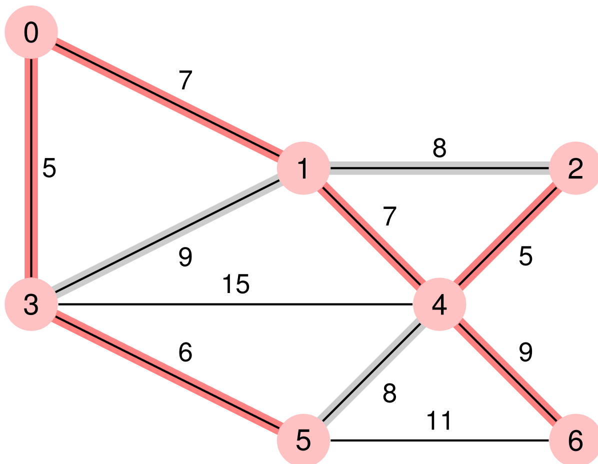
Ejemplo



Ejemplo



Ejemplo



Implementación de Kruskal

```
1  int kruskal(vector<pair<int,pair<int,int> > > ejes, int n)
2  {
3      sort(ejes.begin(),ejes.end());
4      initUF(n);
5      int u = 0;
6      long long t = 0;
7      for(int i=0;i<ejes.size();i++)
8      if(find(ejes[i].second.first)!=find(ejes[i].second.second)){
9          u++;
10         t+=ejes[i].first;
11         if(u==n-1)
12             return t;
13         join(ejes[i].second.first,ejes[i].second.second);
14     }
15     return -1;
16 }
```

Análisis de Kruskal

- En ejes recibimos los ejes como pares $(p, (v_1, v_2))$ que representa un eje de peso p entre los nodos v_1 y v_2, al ordenarlos se ordenan por peso.

Análisis de Kruskal

- En ejes recibimos los ejes como pares $(p, (v_1, v_2))$ que representa un eje de peso p entre los nodos v_1 y v_2, al ordenarlos se ordenan por peso.
- Ahora empezamos a recorrer todos los ejes, cada vez que podemos agregar un eje sumamos su peso, si recorrimos todos los ejes y no conectamos el grafo es porque este no era conexo y devolvemos -1.

Análisis de Kruskal

- En ejes recibimos los ejes como pares $(p,(v_1,v_2))$ que representa un eje de peso p entre los nodos v_1 y v_2, al ordenarlos se ordenan por peso.
- Ahora empezamos a recorrer todos los ejes, cada vez que podemos agregar un eje sumamos su peso, si recorrimos todos los ejes y no conectamos el grafo es porque este no era conexo y devolvemos -1.
- Si llegamos a $n-1$ aristas donde $n=\#V$ entonces ya tenemos el árbol y devolvemos la suma de sus pesos.

Análisis de Kruskal

- En ejes recibimos los ejes como pares $(p, (v_1, v_2))$ que representa un eje de peso p entre los nodos v_1 y v_2, al ordenarlos se ordenan por peso.
- Ahora empezamos a recorrer todos los ejes, cada vez que podemos agregar un eje sumamos su peso, si recorrimos todos los ejes y no conectamos el grafo es porque este no era conexo y devolvemos -1.
- Si llegamos a $n - 1$ aristas donde $n = \#V$ entonces ya tenemos el árbol y devolvemos la suma de sus pesos.
- Podemos modificar levemente el algoritmo para que devuelva los ejes en lugar de la suma de sus pesos pero buscar la suma de los pesos del AGM suele ser un problema bastante frecuente y por eso dimos este algoritmo.

Contenidos

1. Definiciones básicas
   - Algoritmos
   - Complejidad algoritmica
   - Grafos
2. Formas de Recorrer un grafo y camino mínimo
   - BFS y DFS
   - Camino mínimo en grafos ponderados
3. Árbol Generador Mínimo
   - Algoritmo de Kruskal
   - Algoritmo de Prim
4. Componentes Fuertemente Conexas
   - Componentes Fuertemente Conexas
   - Algoritmo de Kosaraju
   - SAT-2

Algoritmo de Prim

Descripción del Algoritmo

El algoritmo de Prim es parecido al algoritmo de Dijkstra. Se empieza por un nodo como AGM y se le va agregando siempre el vértice más cercano al AGM actual, hasta que el AGM tiene los n vértices. Para esto se guardan las distancias al AGM de todos los vértices aún no agregados al mismo.

Algoritmo de Prim

Descripción del Algoritmo

El algoritmo de Prim es parecido al algoritmo de Dijkstra. Se empieza por un nodo como AGM y se le va agregando siempre el vértice más cercano al AGM actual, hasta que el AGM tiene los n vértices. Para esto se guardan las distancias al AGM de todos los vértices aún no agregados al mismo.

- Al igual que Dijkstra, Prim se puede implementar con o sin cola de prioridad, y tiene las mismas complejidades que Dijkstra. Implementandolo con cola de prioridad con un set de C++ la complejidad es $O((m+n)\log n)$.

Comparaciones Kruskal vs. Prim

Implementación

Por lo general es conveniente implementar uno u otro algorimo según cómo venga la entrada, ya que Kruskal toma lista de ejes y Prim, por lo general, lista de adyacencia. También, hay variantes del problema, que pueden ser resueltos por uno de los algoritmos y no por el otro.

Comparaciones Kruskal vs. Prim

Implementación

Por lo general es conveniente implementar uno u otro algorimo según cómo venga la entrada, ya que Kruskal toma lista de ejes y Prim, por lo general, lista de adyacencia. También, hay variantes del problema, que pueden ser resueltos por uno de los algoritmos y no por el otro.

Complejidades

La complejdiad de Kruskal es $O(m \log m)$ mientras que la de Prim es $O(n^2)$ u $O((m+n)\log n)$ según como se implemente. Para grafos con pocos ejes es conveniente Kruskal ya que la complejidad depende de m que es muy parecido a n, para grafos con muchos ejes suele ser más conveniente Prim.

Contenidos

1. Definiciones básicas
   - Algoritmos
   - Complejidad algoritmica
   - Grafos
2. Formas de Recorrer un grafo y camino mínimo
   - BFS y DFS
   - Camino mínimo en grafos ponderados
3. Árbol Generador Mínimo
   - Algoritmo de Kruskal
   - Algoritmo de Prim
4. Componentes Fuertemente Conexas
   - Componentes Fuertemente Conexas
   - Algoritmo de Kosaraju
   - SAT-2

Definición

Componente Fuertemente Conexa

Dado un grafo no dirigido una componente conexa es un conjunto maximal de vértices tales que dados dos de ellos hay un camino que los une. Si el grafo es dirigido, una componente fuertemente conexa es un conjunto de vértices tales que dados dos de ellos hay un camino que va de uno al otro, y un camino que vuelve (es decir que hay un ciclo que pasa por ambos vértices).

Definición

Componente Fuertemente Conexa

Dado un grafo no dirigido una componente conexa es un conjunto maximal de vértices tales que dados dos de ellos hay un camino que los une. Si el grafo es dirigido, una componente fuertemente conexa es un conjunto de vértices tales que dados dos de ellos hay un camino que va de uno al otro, y un camino que vuelve (es decir que hay un ciclo que pasa por ambos vértices).

No es muy difícil ver que las componentes fuertemente conexas definen clases de equivalencia ya que son por definición reflexivas (un nodo está en su componente), simétricas, y sólo queda probar la transitividad que resulta de unir los caminos.

Propiedades de las componentes fuertemente conexas

- Una componente fuertemente conexa es maximal, es decir, si se agrega otro vértice del grafo con todos los ejes que lo unen a un vértice de la componente esta deja de ser fuertemente conexa.

Propiedades de las componentes fuertemente conexas

- Una componente fuertemente conexa es maximal, es decir, si se agrega otro vértice del grafo con todos los ejes que lo unen a un vértice de la componente esta deja de ser fuertemente conexa.

Contenidos

1. Definiciones básicas
   - Algoritmos
   - Complejidad algoritmica
   - Grafos
2. Formas de Recorrer un grafo y camino mínimo
   - BFS y DFS
   - Camino mínimo en grafos ponderados
3. Árbol Generador Mínimo
   - Algoritmo de Kruskal
   - Algoritmo de Prim
4. Componentes Fuertemente Conexas
   - Componentes Fuertemente Conexas
   - Algoritmo de Kosaraju
   - SAT-2

Algoritmos para calcular componentes fuertemente conexas

Existen dos algoritmos conocidos con complejidad $o(n+m)$ para calcular componentes fuertementes conexas. Uno de ellos es el algoritmo de Kosaraju y el otro es el algoritmo de Tarjan. El algoritmo de Kosaraju es un poco más simple de implementar y veremos sólo este algoritmo.

Algoritmos para calcular componentes fuertemente conexas

Existen dos algoritmos conocidos con complejidad $o(n+m)$ para calcular componentes fuertementes conexas. Uno de ellos es el algoritmo de Kosaraju y el otro es el algoritmo de Tarjan. El algoritmo de Kosaraju es un poco más simple de implementar y veremos sólo este algoritmo.
Vamos a dar los detalles de la implementación y la demostración de correctitud pero no el código.

Algoritmo de Kosaraju

- Empezamos con una pila vacía y recorremos todos los vértices del grafo. Para cada vértice que no está en la pila hacemos un DFS.

Algoritmo de Kosaraju

- Empezamos con una pila vacía y recorremos todos los vértices del grafo. Para cada vértice que no está en la pila hacemos un DFS.
- Cada vez que en un DFS llegamos a un vértice del cuál no nos podemos seguir expandiendo a nuevos vértices lo agregamos a la pila.

Algoritmo de Kosaraju

- Empezamos con una pila vacía y recorremos todos los vértices del grafo. Para cada vértice que no está en la pila hacemos un DFS.
- Cada vez que en un DFS llegamos a un vértice del cuál no nos podemos seguir expandiendo a nuevos vértices lo agregamos a la pila.
- Luego de agregar todos los vértices a la pila invertimos la dirección de todos los ejes.

Algoritmo de Kosaraju

- Empezamos con una pila vacía y recorremos todos los vértices del grafo. Para cada vértice que no está en la pila hacemos un DFS.
- Cada vez que en un DFS llegamos a un vértice del cuál no nos podemos seguir expandiendo a nuevos vértices lo agregamos a la pila.
- Luego de agregar todos los vértices a la pila invertimos la dirección de todos los ejes.
- El próximo paso es ir desapilando vértices de la pila, si no están hasta el momento en ninguna componente hacemos un DFS desde ese nodo con los vértices que no están en ninguna componente. Los vértices a los que llegamos con ese DFS determinan una componente fuertemente conexa.

Correctitud del Algoritmo de Kosaraju

Si en uno de los DFS llegamos a un vértice, este vértice pertenece a la misma componente conexa que el vértice inicial:

Correctitud del Algoritmo de Kosaraju

Si en uno de los DFS llegamos a un vértice, este vértice pertenece a la misma componente conexa que el vértice inicial:

- Como llegamos al nodo en el DFS con ejes invertidos podemos ir de ese nodo al inicial en el grafo original.

Correctitud del Algoritmo de Kosaraju

Si en uno de los DFS llegamos a un vértice, este vértice pertenece a la misma componente conexa que el vértice inicial:

- Como llegamos al nodo en el DFS con ejes invertidos podemos ir de ese nodo al inicial en el grafo original.
- Si el nodo inicial del DFS lo apilamos después en la pila (y por eso lo desapilamos antes) entonces quiere decir que desde el nodo al cuál llegamos no nos podíamos seguir expandiendo hasta el nodo inicial, pero como había un camino quiere decir que ya lo habíamos visitado en el DFS, luego habíamos llegado desde el nodo inicial.

Correctitud del Algoritmo de Kosaraju

Si en uno de los DFS llegamos a un vértice, este vértice pertenece a la misma componente conexa que el vértice inicial:

- Como llegamos al nodo en el DFS con ejes invertidos podemos ir de ese nodo al inicial en el grafo original.
- Si el nodo inicial del DFS lo apilamos después en la pila (y por eso lo desapilamos antes) entonces quiere decir que desde el nodo al cuál llegamos no nos podíamos seguir expandiendo hasta el nodo inicial, pero como había un camino quiere decir que ya lo habíamos visitado en el DFS, luego habíamos llegado desde el nodo inicial.
- Esto prueba que hay un camino de ida y uno de vuelta entre los dos nodos.

Correctitud del algoritmo de Kosaraju

- Si dos nodos están en una misma componente, los tenemos que encontrar con este método ya que desde el primero que desapilamos vamos a llegar con el DFS al otro.

Correctitud del algoritmo de Kosaraju

- Si dos nodos están en una misma componente, los tenemos que encontrar con este método ya que desde el primero que desapilamos vamos a llegar con el DFS al otro.
- Esto prueba que el algoritmo de Kosaraju es correcto y su complejidad es $O(n + m)$ ya que hacemos dos DFS y recorremos todos los ejes una vez para crear el grafo con ejes invertidos.

Correctitud del algoritmo de Kosaraju

- Si dos nodos están en una misma componente, los tenemos que encontrar con este método ya que desde el primero que desapilamos vamos a llegar con el DFS al otro.
- Esto prueba que el algoritmo de Kosaraju es correcto y su complejidad es O(n + m) ya que hacemos dos DFS y recorremos todos los ejes una vez para crear el grafo con ejes invertidos.
- Es importante notar que la complejidad es $o(n+m)$ y no $o(m)$ ya que el grafo puede no ser conexo y luego no hay cotas para m en función de n.

Contenidos

1. Definiciones básicas
   - Algoritmos
   - Complejidad algoritmica
   - Grafos
2. Formas de Recorrer un grafo y camino mínimo
   - BFS y DFS
   - Camino mínimo en grafos ponderados
3. Árbol Generador Mínimo
   - Algoritmo de Kruskal
   - Algoritmo de Prim
4. Componentes Fuertemente Conexas
   - Componentes Fuertemente Conexas
   - Algoritmo de Kosaraju
   - SAT-2

Satisfability Problem

Satisfability Problem

El problema de satisficibilidad (en inglés, Satisfability Problem) consiste en encontrar, dadas variables booleanas y una fórmula, si esta es satisfacible. Toda fórmula puede ser reescrita como una o más fórmulas, que son OR de una o varias variables o negaciones de variables, tales que la fórmula original es satisfacible sii todas las nuevas fórmulas lo son

Satisfability Problem

Satisfability Problem

El problema de satisficibilidad (en inglés, Satisfability Problem) consiste en encontrar, dadas variables booleanas y una fórmula, si esta es satisfacible. Toda fórmula puede ser reescrita como una o más fórmulas, que son OR de una o varias variables o negaciones de variables, tales que la fórmula original es satisfacible sii todas las nuevas fórmulas lo son

$$F = F_1 \wedge F_2 \wedge \ldots \wedge F_n$$

$$F_i = p_{i,1} \vee p_{i,2} \vee \ldots \vee p_{i,m_i}$$

$$p_{i,j} = v_t | \neg v_t \text{ siendo } v_t \text{ una variable proposicional.}$$

Satisfability Problem

Satisfability Problem

El problema de satisficibilidad (en inglés, Satisfability Problem) consiste en encontrar, dadas variables booleanas y una fórmula, si esta es satisfacible. Toda fórmula puede ser reescrita como una o más fórmulas, que son OR de una o varias variables o negaciones de variables, tales que la fórmula original es satisfacible sii todas las nuevas fórmulas lo son

$$F = F_1 \wedge F_2 \wedge ... \wedge F_n$$

$$F_i = p_{i,1} \vee p_{i,2} \vee ... \vee p_{i,m_i}$$

$$p_{i,j} = v_t | \neg v_t \text{ siendo } v_t \text{ una variable proposicional.}$$

Este problema pertenece a una clase de problemas que se denominan NP-completos y para los cuáles no se conoce solución polinomial.

SAT-2

Qué es SAT-2?

SAT-2 es un caso particular de SAT, en el que usando la notación de la definición anterior todos los m_i son 1 o 2.

SAT-2

Qué es SAT-2?

SAT-2 es un caso particular de SAT, en el que usando la notación de la definición anterior todos los m_i son 1 o 2.

- Para este problema se conoce una solución lineal en la cantidad de variables más la cantidad de fórmulas (las F_i).

SAT-2

Qué es SAT-2?

SAT-2 es un caso particular de SAT, en el que usando la notación de la definición anterior todos los m_i son 1 o 2.

- Para este problema se conoce una solución lineal en la cantidad de variables más la cantidad de fórmulas (las F_i).
- Para resolver este problema se genera un grafo al cuál se le calculan las componentes fuertemente conexas.

Implementación de SAT-2

- Generamos un grafo en el que los nodos son las variables y las negaciones de las variables, y hay una arista de p_i a p_j si $p_i \Rightarrow p_j$.

Implementación de SAT-2

- Generamos un grafo en el que los nodos son las variables y las negaciones de las variables, y hay una arista de p_i a p_j si $p_i \Rightarrow p_j$.
- Esto lo hacemos porque $(p_i \vee p_j)$ se puede reescribir como $(\neg p_i \Rightarrow p_j) \wedge (\neg p_j \Rightarrow p_i)$

Implementación de SAT-2

- Generamos un grafo en el que los nodos son las variables y las negaciones de las variables, y hay una arista de p_i a p_j si $p_i \Rightarrow p_j$.
- Esto lo hacemos porque $(p_i \vee p_j)$ se puede reescribir como $(\neg p_i \Rightarrow p_j) \wedge (\neg p_j \Rightarrow p_i)$
- Haciendo uso de la transitividad del operador $\Rightarrow$ calculamos las componentes fuertemente conexas de este grafo. Si p_i y $\neg p_i$ pertenecen a la misma componente para algún i entonces la fórmula es insatifacible, de lo contrario, podemos satisfacer la fórmula asignandole para cada variable v_i, el valor verdadero si no hay camino de v_i a $\neg v_i$ y falso en caso contrario., ya que no hay un camino de $\neg v_i$ a v_i.

¿Preguntas?