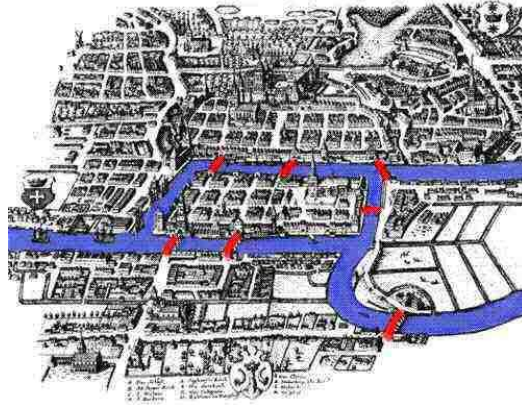


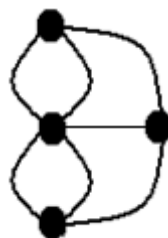
Grafos

El origen del concepto de grafo se remonta a una historia sucedida en la ciudad de Königsberg, en Prusia (actualmente, Kaliningrado, en Rusia). Se cuenta que había, en épocas del matemático Euler, dos pequeñas islas entre las orillas del río Pregel, con siete puentes alrededor, tal como se muestra en la siguiente ilustración:



La gente del lugar a menudo se deasfiaba a hacer un recorrido que permitiera atravesar cada puente una sola vez regresando al punto de partida.

Este problema, que tenía gran dificultad ya que nadie superaba el desafío, atrajo la atención de Euler, quien lo analizó empleando una técnica de graficado por la cual redujo a puntos la representación de las islas y las orillas, y a arcos los siete puentes. El grafico resulto del siguiente modo. A los puntos se los denomino 'nodos' o 'vértices'. Los enlaces son 'aristas' o 'arcos'.



El problema original se volvió entonces ahora es análogo al de intentar dibujar el grafo anterior partiendo de un vértice, sin levantar el lápiz y sin pasar dos veces por el mismo arco. (A un recorrido con estas características actualmente se lo denomina camino euleriano).

Euler observó que para que eso fuera posible, para todos los nodos, exceptuando a lo sumo el nodo de inicio y el de finalización, debería cumplirse, que en cada uno de ellos (nodos 'intermedios') debía incidir un número par de aristas, ya que por una de ellas se 'entraba' en el nodo y por otra se 'salía' del mismo.

Entonces, se debe verificar, para poder realizar un dibujo sin levantar el lápiz y sin pasar dos veces por el mismo arco, que el número de nodos con un número impar de aristas incidentes en él debe ser 2 (problema del "camino euleriano").

En el caso particular del problema planteado en Königsberg, como el vértice de inicio y el de finalización coinciden, sólo podría haber un vértice con un número impar de aristas incidentes en él (lo cual no puede cumplirse, problema del “ciclo euleriano”).

Euler demostró que el problema de los puentes de Königsberg no tenía solución y dio con su desarrollo, origen a la teoría de Grafos.

Los grafos constituyen una muy útil herramienta matemática para modelizar situaciones referidas a cuestiones tan diferentes como mapas de interrelación de datos, carreteras, cañerías, circuitos eléctricos, diagrama de dependencia, etc.

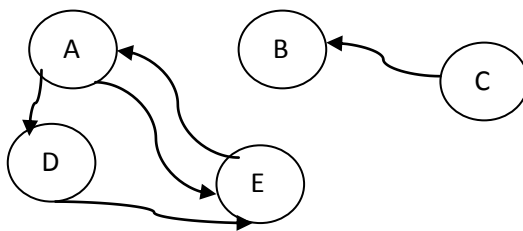
Definición matemática de Grafo

Un grafo dirigido o digrafo o grafo orientado consiste en una dupla formada por un conjunto V de vértices o nodos del grafo, y un conjunto de pares ordenados A (aristas orientadas) pertenecientes a $V \times V$. (La relación establecida entre los vértices es antisimétrica)

En símbolos el grafo dirigido G es

$G = (V ; A)$ donde A es un subconjunto de $\square V \times V$ (Aristas orientadas o dirigidas)

Ejemplo:



$V = \{A, B, C, D, E\}$

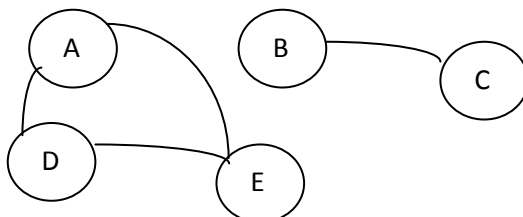
$A = \{ (A,D), (A,E), (E,A), (D,E), (C,B) \}$

Un grafo no dirigido (o no orientado) es una dupla formada por un conjunto V de vértices o nodos del grafo, y un conjunto de pares NO ordenados A (aristas no orientadas) pertenecientes a $V \times V$. (La relación establecida entre los vértices es simétrica).

En símbolos el grafo no dirigido G es

$G = (V ; A)$ donde A es un conjunto de \square pares no ordenados de $V \times V$ (Aristas NO orientadas o NO dirigidas)

Ejemplo:



$V = \{A, B, C, D, E\}$

$A = \{ (A,D), (A,E), (D,E), (C,B) \}$ (pares NO ordenados)

En ambos casos, si (v,w) pertenece a A se dice que w es adyacente a v

Algunas definiciones más:

Camino: serie alternada de vértices y aristas que inicia y finaliza con vértices y donde cada arista conecta el vértice que le precede con el que le sucede

Longitud de Camino: Cantidad de Aristas del camino

Camino Abierto: Camino donde el vértice inicial y final difieren.

Camino Cerrado: Camino donde el vértice inicial y final coinciden.

Recorrido: Camino que no repite aristas

Recorrido Euleriano: Recorrido que contiene todas las aristas del grafo \Leftrightarrow Tiene 2 vértices de grado impar

Circuito: Recorrido Cerrado

Circuito Euleriano: Recorrido Euleriano Cerrado \Leftrightarrow Todos los vértices de grado par

Camino Simple: Camino que no repite vértices (salvo inicial y final).

Camino de Hamilton: Camino Simple que contiene todos los vértices del grafo

Ciclo: Camino Simple Cerrado (o bien , camino que conteniendo al menos tres vértices distintos tales que el primer vértice es adyacente al ultimo)

Ciclo de Hamilton: Camino Hamiltoniano Cerrado

Sub Grafo G' de $G(V,A)$: es el grafo $G' = (V', A')$ donde V' es un subconjunto de $\square V$ y A' es un subconjunto de $\square A$

Grafo no dirigido conexo: un grafo no orientado es conexo sii para todo vértice del grafo hay un camino que lo conecte con otro vértice cualquiera del grafo.

Árbol libre: es un grafo no dirigido conexo sin ciclos.

Grafo dirigido fuertemente conexo: un grafo dirigido es fuertemente conexo sii entre cualquier par de vértices hay un camino que los une.

Grafo subyacente de un grafo: es el grafo no dirigido que se obtiene reemplazando cada arista (orientada) del mismo, por una arista no orientada.

Un grafo dirigido débilmente conexo: es aquel grafo dirigido que no es fuertemente conexo y cuyo grafo subyacente es conexo.

Grado de un vértice: es el número de aristas incidentes en el.

Grado de entrada de un vértice: es el número de aristas del cual el vértice dado es destino.

Grado de salida de un vértice: es el número de aristas de las cuales el vértice dado es origen.

Vértice fuente: vértice cuyo Grado de Salida es 0 y no es aislado (esto ultimo significa que es primera o segunda componente de al menos un par del conjunto de aristas)

Vértice sumidero: vértice cuyo Grado de Entrada es 0 y no es aislado (esto ultimo significa que es primera o segunda componente de al menos un par del conjunto de aristas)

El TDA Grafo:

Es un TDA contenedor de un conjunto de datos llamados nodos y de un conjunto de aristas cada una de las cuales se determina mediante un par de nodos.

Este es un listado de posibles primitivas para el TDA Grafo:

Crear grafo: esta primitiva genera un grafo vacío.

Precondición: -----

Poscondición: grafo generado vacío

Destruir grafo: esta primitiva destruye el grafo.

Precondición: que el grafo exista previamente.

Poscondición: -----

Insertar nodo: esta primitiva inserta un nodo nuevo, recibido como argumento, en el grafo

Precondición: que el grafo exista previamente y que el nodo no esté previamente

Poscondición: el grafo queda modificado por el agregado del nuevo nodo

Insertar arista: esta primitiva inserta una arista nueva, recibida como argumento, en el grafo

Precondición: que el grafo exista previamente, que la arista no esté previamente y que existan en el grafo los nodos origen y destino de la arista

Poscondición: el grafo queda modificado por el agregado de la nueva arista

Eliminar nodo: esta primitiva elimina un nodo, recibido como argumento, del grafo

Precondición: que el grafo exista previamente y que el nodo a eliminar esté en él y no tenga aristas incidentes en él.

Poscondición: el grafo queda modificado por la eliminación del nodo

Eliminar arista: esta primitiva elimina una arista, recibida como argumento, del grafo

Precondición: que el grafo exista previamente y la arista estén en él.

Poscondición: el grafo queda modificado por la eliminación de la arista

Recorrer grafo: esta primitiva permite visitar cada uno de los nodos del grafo

Precondición: que el grafo exista

Poscondición: -----

Existe arista: esta primitiva recibe una arista y retorna un valor logico indicando si la arista existe en el grafo

Precondición: que el grafo exista

Poscondición: -----

Existe nodo: esta primitiva recibe una arista y retorna un valor logico indicando si el nodo existe en el grafo.

Precondición: que el grafo exista

Poscondición: -----

Estructuras de datos utilizadas para implementar grafos

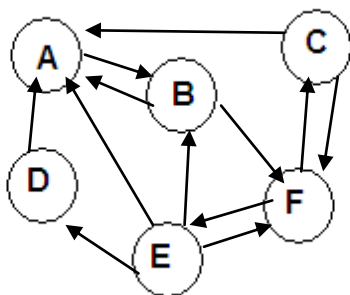
Se puede utilizar una Matriz de Adyacencia, la cual, para N nodos es de $N \times N$.

Si M es la matriz de adyacencia del; grafo G entonces verifica que $M[i][j]$ es true (o 1) si la arista (i,j) pertenece al grafo, o false (o bien 0) si no pertenece.

Otra posibilidad es usar listas de adyacencia. En este caso, la representación es mediante una lista de listas.

De este modo, se tiene una lista con nodos; cada uno de ellos conteniendo la información sobre un vértice y un puntero a una lista ligada a los vértices adyacentes al vértice indicado.

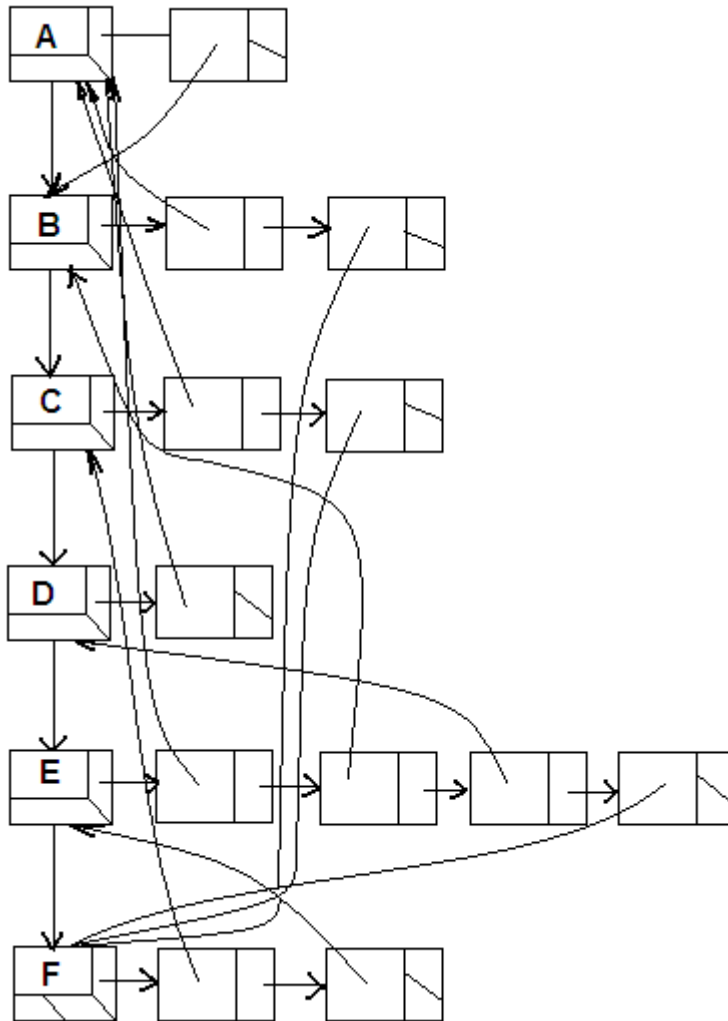
Ejemplo: para este grafo



La representación de la Matriz de Adyacencia es:

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

La representación con listas de adyacencia se muestra a continuación. Observar que cada lista de adyacencia (las listas horizontales apuntadas desde el nodo que contiene los datos de un vértice) esta formada por nodos que tienen información sobre la dirección de un nodo adyacente, y el puntero al siguiente nodo de la lista.



Recorridos de un grafo dirigido:

Los dos recorridos básicos en los grafos dirigidos son en profundidad y en anchura. Cada uno de ellos agotara todos los vértices del grafo, visitando una y solo una vez a cada uno.

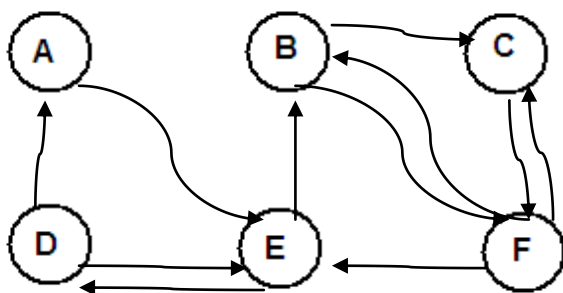
Además, si se verifica que el grafo sea acíclico, se pueden llevar a cabo los recorridos topológicos.

Algoritmo de recorrido en profundidad (o de 'busqueda en profundidad', *depth first search*):

Consiste en, para cada vértice del grafo que no ha sido visitado previamente, visitarlo y luego pasar al primer adyacente no visitado, luego al primer adyacente del adyacente que no haya sido visitado, y así hasta llegar a un nodo que no tenga adyacentes no visitados. Entonces se retrocede, para pasar al siguiente adyacente del anterior vértice y realizar el mismo proceso hasta agotar los adyacentes no visitados (se retrocede cuando estos se agotan).

Para implementar el algoritmo suele usarse una pila, o bien un algoritmo recursivo.

Para el siguiente grafo,



el recorrido en profundidad es: A, E, B, C, F, D

Informalmente se puede enunciar el algoritmo del siguiente modo:

Version recursiva del recorrido en profundidad

```

{
Desmarcar todos los vértices (es decir, para cada uno indicar que no ha sido visitado todavía)
Para cada vértice v del grafo que no haya sido visitado, aplicar dfs(v).
}
  
```

El algoritmo recursivo dfs es así:

dfs(v)

```

{
  Marcar v.
  Para cada vértice w adyacente a v que no haya sido marcado aplicar dfs(w).
}
  
```

Versión no recursiva del recorrido en profundidad:

Se muestra a continuación la versión no recursiva de recorrido en profundidad, usa una pila.

Algoritmo dfs

```

{
Desmarcar todos los vértices
Para cada vértice v del grafo que no haya sido visitado hacer
{
  Apilar v
  Mientras la pila no esté vacía hacer
  {
    Desapilar en x;
    Visitar y marcar x
    Para cada vértice w adyacente a x y no visitado hacer
      apilar w
  }
}
}

```

Recorrido en amplitud (o 'busqueda en amplitud', *breath first search*):

Consiste en considerar cada uno de los vértices no visitados previamente y, para cada uno, recorrer todos los adyacentes no visitados, luego todos los adyacentes no visitados del primer adyacente, luego todos los adyacentes no visitados del segundo adyacente, y así sucesivamente.

Para el grafo representado antes el recorrido en amplitud es:

A, E, B, D, C, F

Esta es una posible implementación informal del bfs:

Algoritmo bfs

```

{
Desmarcar todos los vértices
Para cada vértice v del grafo que no haya sido visitado hacer
{
  Acolar v
  Mientras la cola no esté vacía hacer
  {
    Desacolar en x;
    Visitar y marcar x
    Para cada vértice w adyacente a x y no visitado hacer
      acolar w
  }
}
}

```

Grafos dirigidos acíclicos: son aquellos que no tienen ciclos .

Prueba de aciclidad

Para establecer si un grafo dirigido $G = (V, A)$ es acíclico, se puede realizar un recorrido en profundidad del mismo; si en transcurso del recorrido se encuentra una

arista 'de retroceso' entonces el grafo tiene un ciclo, (ya que todo ciclo tiene una arista de retroceso).

Para determinar la existencia de arcos con retroceso hay que utilizar una marca extra para cada nodo. Esa marca se inicializa para cada vértice de partida del ciclo externo del algoritmo de recorrido en profundidad (el ciclo que considera todo vértice del grafo). Esta marca estará 'activa' durante las visitas a los vértices descendientes, de esta forma se puede saber si un vértice ya visitado es o no ancestro suyo.

Recorridos topológicos:

Estos recorridos solo se aplican a grafos acíclicos, y permiten linealizar un grafo, es decir, listar la secuencia de los vértices del grafo de tal modo que se respete la precedencia de los mismos.

Un grafo dirigido acíclico se puede recorrer en profundidad o en anchura.

Por ejemplo, el esquema de correlatividades de una carrera es un grafo dirigido acíclico; cualquier recorrido topológico permite listar las materias de la carrera respetando las correlatividades.

Recorrido topológico en profundidad

El algoritmo de recorrido topológico en profundidad se puede enunciar informalmente de este modo:

(El algoritmo devuelve una lista en la cual quedan insertados los nodos según fueron visitados en el recorrido)

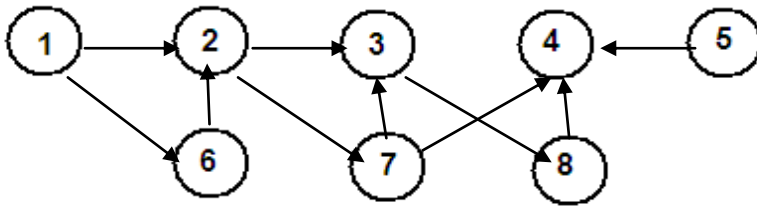
```
{
Para cada vértice del grafo, marcarlo como no visitado
Para cada vértice v del grafo
    Si v no fue visitado entonces ejecutar la función rec para v
}
```

Funcion rec aplicada a un vertice v (genera una lista)

```
{
Marcar v como visitado
Para cada vértice w adyacente a v hacer
    Si w no visitado entonces ejecutar función rec para w
Insertar v al frente en una lista
}
```

Ejemplo

Consideremos el siguiente grafo dirigido y acíclico:



Según el algoritmo anterior,

```

{
rec(1)
Adyacentes(1):2, 6
{
  rec(2)
  Adyacentes(2):3, 7
  {
    rec(3)
    Adyacentes(3):8
    {
      rec(8)
      Adyacentes(8):4
      {
        rec(4)
        Adyacentes(4):-
        Insertar 4 al frente en la lista de salida
      }
      Insertar 8 al frente en la lista de salida
    }
    Insertar 3 al frente en la lista de salida
  }
  {
    rec(7)
    Adyacentes(7): 3, 4 (ambos ya visitados)
    Insertar 7 al frente en la lista de salida
  }
  Insertar 2 al frente en la lista de salida
}
{
  rec(6)
  Adyacentes(6): 2 (ya visitado)
  Insertar 6 al frente en la lista de salida
}
Insertar 1 al frente en la lista de salida
}
{
  rec(5)
  Adyacentes(5): 4 (ya visitado)
  Insertar 5 al frente en la lista de salida
}

```

De este modo, la lista generada, que linealiza el grafo respetando las precedencias entre los nodos con un recorrido en profundidad es:

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

Recorrido topológico en anchura

El algoritmo del recorrido topológico en anchura, consiste en lo siguiente:

Se construye una tabla en la cual se anota el grado de entrada de cada vértice del grafo. (En esta tabla siempre hay al menos un vértice de grado 0, ya que, de otro modo, el grafo no sería acíclico).

Los vértices de grado de entrada 0 se acolan. Luego, por etapas se va desacolando. El nodo desacolado va a la salida (o a ser procesado). Se disminuye en 1 el grado de los vértices adyacentes, lo cual equivale a retirar el nodo desacolado del grafo, con el consiguiente decremento de grado de los adyacentes.

Si un nodo llega a 0 al ser decrementado, se acola.

Al quedar la cola vacía se termina el procesamiento.

De manera no formal, el algoritmo queda así:

```
{
Inicializar grados de entrada en tabla de grados
Se acolan los vértices de grado de entrada 0
Mientras la cola no este vacía hacer
{
    Desacolar y procesar un nodo, enviándolo al final en una lista
    Decrementar en 1 el grado de los vértices adyacentes al mismo
    Acolar los nodos que a causa del decremento en el grado hayan llegado a 0
}
}
```

Ejemplo:

Para el grafo anterior, la tabla de grados de entrada es:

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	2	2	3	0	1	1	1

Se acolan el 1 y el 5

Cola : 1 – 5

Se desacola y procesa el 1

Se decrementan los adyacentes a 1, que son 2 y 6.

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	1	2	3	0	0	1	1

6 ahora vale 0 y se acola.

Cola: 5 – 6

Se desacola y procesa 5.

Se decrementan los adyacentes a 5, que es 4.

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	1	2	2	0	0	1	1

Cola: 6

Se desacola y procesa 6.

Se decrementa el adyacente a 6, que es 2.

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	0	2	2	0	0	1	1

2 vale ahora 0 y se acola.

Cola: 2

Se desacola y procesa 2.

Se decrementan los adyacentes a 2, que son 3 y 7.

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	0	1	2	0	0	0	1

7 llega a 0, se acola.

Cola: 7

Se desacola y procesa 7

Se decrementan los adyacentes a 7, que son 3 y 4

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	0	0	1	0	0	0	1

3 llega a 0, se acola

Cola: 3

Desacolar 3

Decrementar el adyacente a 3, que es 8

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	0	0	1	0	0	0	0

8 vale 0, se acola

Cola: 8

Desacolar 8

Decrementar el adyacente a 8, que es 4

vértice	1	2	3	4	5	6	7	8
Grado entrada	0	0	0	0	0	0	0	0

4 se acola porque llegó a 0.

Cola: 4

Desacolar y procesar 4.

4 no tiene adyacentes.

La linealización del grafo da por resultado esta lista (se han indicado las precedencias con arcos)

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

Problemas sobre caminos en un grafo:

Hay un gran número de problemas sobre caminos en grafos dirigidos y en grafos no dirigidos. Algunos plantean determinar si dado un par de vértices, o todos los pares de vértices, hay o no camino entre ellos.

Otros trabajan sobre grafos con aristas o con vértices ponderados. Un ponderación es un valor asociado a una arista o a un vértice, o a ambos.

En esta sección trataremos algunos de los problemas más comunes sobre caminos en grafos con aristas ponderadas.

Problema de los caminos más cortos con un solo origen.

Dado un grafo dirigido $G = (V, A)$, en el cual cada arco tiene asociado un costo no negativo, y donde un vértice se considera como origen, el problema de los "caminos más cortos con un solo origen" consiste en determinar el costo del camino más corto desde el vértice considerado origen a todos los otros vértices de V .

Este es uno de los problemas más comunes que se plantean para los grafos dirigidos con aristas ponderadas (es decir con peso en las aristas).

El peso de las aristas debe no negativo. El grafo puede no ser conexo, en este caso la solución indicara que no hay camino que permita alcanzar algunos de los vértices.

El la 'longitud' o 'costo' de un camino es la sumatoria de los pesos de las aristas que lo conforman.

El modelo de grafo dirigido con aristas ponderadas no negativas puede, por ejemplo, corresponder a un mapa de vuelos en el cual cada vértice represente una ciudad, y cada arista (v,w) una ruta aérea de la ciudad v a w .

El peso de la arista (v,w) puede ser el tiempo que se requiere para volar de v a w (y para justificar que eventualmente el peso de (v,w) sea diferente del de (w,v) se puede considerar que eso se debe a la influencia de los vientos. En este caso el problema trataría de encontrar los tiempos mas cortos para, partiendo de una ciudad en particular, alcanzar cada una de las otras.

Otra posibilidad es considerar que los pesos corresponden a las tarifas de esos tramos, y que en ocasiones, el precio del tramo (v,w) es diferente del (w,v) . Para esta situación, el problema plantea la búsqueda de los costos mas bajos a pagar para partiendo de una ciudad en particular, llegar a cada una de las otras.

Algoritmo de Dijkstra para el calculo de los caminos mínimos

Este algoritmo se caracteriza por usar una estrategia *greedy*, voraz (o ávida).

En esta estrategia se trata de optimizar (alcanzar el máximo o el mínimo) de una función objetivo, la cual depende de ciertas variables, cada una de las cuales tiene un determinado dominio y está sujeta a restricciones.

En cada etapa se toma una decisión que no tiene vuelta atrás, y que involucra elegir el elemento más promisorio (es decir, el que parece ofrecer más posibilidades de mejorar la función objetivo) de un conjunto y se analizan ciertas restricciones asociadas a cada variable de la función objetivo para ver si se verifican mejoras en ella.

La solución puede expresarse como una sucesión de decisiones, y en cada etapa de la estrategia se elegirá la mejor opción de las disponibles (óptimo local), analizándose luego si esta elección verifica las restricciones, constituyendo una solución factible.

La estrategia greedy, muy interesante en sí, no sirve para cualquier problema: se debe tratar de un problema de optimización de una función; pero aún en este caso, no asegura que se llegue al óptimo global (podría llevarnos a un óptimo local) ni tampoco, en algunos casos, llegar a una solución factible: el problema en cuestión debe verificar ciertas condiciones para que quede garantizado que la estrategia greedy funcione.

Si bien no es estrictamente un tema a desarrollar aquí, cabe aclarar entonces, que sólo el problema cumple ciertas condiciones (las cuales se pueden expresar así: la estructura de las soluciones factibles debe ser un *matroide*), está demostrado que la estrategia 'greedy' permite optimizar la función objetivo.

¿Qué dice el algoritmo desarrollado por Dijkstra para el problema de los caminos mínimos con un origen determinado en un grafo orientado ponderado de forma no negativa?

Propone partir de la situación inicial, la cual si el vértice de partida es el 1, corresponde a los datos de la fila 1 de la matriz de Pesos.

Luego se desarrollan una serie de pasos, en cada uno de los cuales se intenta mejorar la situación del paso anterior considerando una ciudad no visitada antes como 'escala previa' para bajar el tiempo. Cada uno de estos pasos corresponde a la elección del elemento más promisorio para tratar de mejorar los valores asociados a cada camino.

Esto significa que, con la interpretación que hemos hecho antes de las ciudades y el costo de ir de una a otra, de las ciudades que no han sido visitadas aún se elegirá aquella a la cual el tiempo del viaje desde la ciudad origen sea el más bajo.

Entonces: partimos de un conjunto **S** de vértices cuya distancia más corta desde el origen ya es conocida. Inicialmente **S** contiene solo al vértice de origen.

Paso a paso se agrega algún vértice v , perteneciente a $V - S$, al conjunto S , cuya distancia desde el origen es la más corta posible.

Dado que todos los arcos tienen costos no negativos, será posible siempre encontrar un camino más corto desde el origen hasta cada uno de los otros vértices, que se llama 'especial'.

En cada paso del algoritmo se usa un vector D para anotar la longitud del 'camino especial' más corto a cada vértice.

Cuando S incluya a todos los vértices, todos los caminos serán especiales, y D contendrá la distancia más corta del origen a cada uno de los otros vértices del grafo.

En cada paso, en S se inserta el vértice v que verifique que su distancia $D[v]$ desde el vértice origen hasta el origen a él, es la mínima de todas las $D[i]$ (para las i que pertenecen a $V - S$).

Por eso cualquier otro camino para llegar desde el origen a v , y que considerara el paso por algún otro vértice, sería más costoso, dado que la distancia desde el origen a ese otro vértice sería mayor que la del origen al v y que esta guardada en $D[v]$.

A continuación se muestra el algoritmo de Dijkstra.

Se tiene, entonces, un grafo dirigido $G = (V, A)$

El conjunto de vértices es $V = \{1, 2, \dots, n\}$

Se considerara que el vértice de partida es 1.

La matriz de pesos es C .

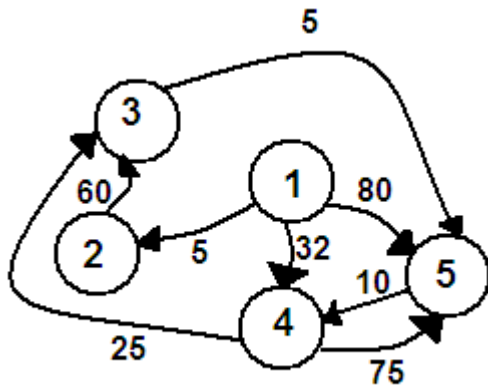
$C[i, j]$ es el tiempo de ir del vértice i al j de forma 'directa'.

Cuando la arista correspondiente no exista, consideraremos que el valor de ese elemento del arreglo será un valor mucho mayor que cualquier costo real, es decir 'infinito'.

D es el array en el cual se van almacenando la sucesivas mejoras de tiempo obtenidas por el algoritmo y donde finalmente queden almacenados los tiempos mas bajos.

Ejemplo:

Consideraremos un grafo y los valores asociados a los arcos los presentaremos en una tabla; pero hay que recordar que la implementación del grafo podría tener estos datos en una matriz o en listas de adyacencia.



Si usamos una matriz de pesos M , es:

	1	2	3	4	5
1	0	5	∞	32	80
2	∞	0	60	∞	∞
3	∞	∞	0	∞	5
4	∞	∞	25	0	75
5	∞	∞	∞	10	0

Se ha colocado el símbolo ∞ en aquellas celdas que corresponden a celdas para las cuales no hay aristas asociadas.

Consideraremos además el vector de vértices visitados, en el cual ya hemos marcado el vértice 1 por ser de salida.

1	2	3	4	5
+	-	-	-	-

Además, tenemos el vector D en el cual vamos a trabajar para obtener los tiempos mínimos.

Inicialmente el vector D es:

2	3	4	5
5	∞	32	80

Etapas 1:

Se selecciona, de los vértices no visitados, aquel cuyo tiempo es el menor, evaluando los tiempos de D . El vértice con menor tiempo es 2.

Se marca entonces 2 como visitado en el vector de marcas.

Ahora se analiza si se mejoran los tiempos pasando por 2 como vértice intermedio.

El vértice 2 obviamente no mejorará su tiempo, así que pasamos al vértice 3. Comparamos el valor de $D[3]$, que es ∞ , con $D[2] + M[2][3]$, que es $5+60=65$

$D[3] > D[2] + M[2][3]$ por lo tanto, reemplazamos en $D[2]$, ∞ por 65.

Para el vértice 4 se observa $D[4]=32$ y $D[2]+M[2][4]=5+\infty=\infty$, lo que indica que no se obtiene una mejora considerando al vértice 2 y dejamos el 32 en $D[4]$.

Para el vértice 5, es $D[5]=80$ y $D[2]+M[2][5]=5+\infty=\infty$, el vértice 2 no representa ninguna mejora y el valor de $D[5]$ se deja igual.

Entonces, finalizada esta etapa, se tiene:

Vector de vértices visitados:

1	2	3	4	5
+	+	-	-	-

Vector D:

2	3	4	5
5	65	∞	∞

Etapa 2:

Seleccionamos el vértice no visitado de menor tiempo. Es el 3, con un tiempo de 65.

Analizamos posibles mejoras en los tiempos pasando por 3.

Como se observa, no se producirá ninguna mejora en 2, ni obviamente en 3.

Analizamos la situación para el vértice 4.

$D[4]$ es ∞ , y $D[3]+M[3][4]=65+\infty=\infty$, por lo que no se mejora la situación del 4.

Análisis para el vértice 5:

$D[5]=\infty$ y $D[3]+M[3][5]=65+5=70$, el pasaje por 3 implica una mejora en el tiempo para 5.

Finalizada esta etapa, se tiene:

Vector de vértices visitados:

1	2	3	4	5
+	+	+	-	-

Vector D:

2	3	4	5
5	65	∞	70

Etapla 3 :

De los vértices no visitados, el de menor tiempo es 5

Las eventuales mejoras solo pueden darse en el caso del vértice 4, que es el que analizaremos.

$D[4] = \infty$ y $D[5] + M[5][4] = 70 + 10 = 80$, esto representa una mejora; se almacena en $D[4]$, 80

Quedan entonces,

Vector de vértices visitados:

1	2	3	4	5
+	+	+	-	+

Vector D:

2	3	4	5
5	65	80	70

Como se observa, el último vértice que queda sin visitar no nos llevaría a ninguna mejora en los tiempos obtenidos, por lo cual no es necesario considerarlo.

El algoritmo de Dijkstra se puede expresar de modo informal de la siguiente forma:

V es el conjunto de vértices del grafo.

S es el conjunto de vértices visitados.

M es la matriz de pesos.

D es el vector de tiempos mínimos, que se actualiza en cada etapa.

En el siguiente algoritmo se considera que el vértice de partida es el 1, y los vértices están numerados de 1 hasta N

```
{
  Inicializar S = { 1 }
  Para cada i desde 2 hasta N hacer
    Asignar M[1, i] a D[i]
  Para cada i entre 1 y n - 1 hacer
    {
      Elegir un vértice w en V – S tal que D[w] sea el mínimo de todos los D[i].
      Agregar w a S.
      Para cada vértice v adyacente a w en V – S hacer:
        D[v]=mín ( D[v], D[w] + M[w,v] )
    }
}
```

Si la implementación del grafo se lleva a cabo mediante una matriz de pesos y se usan arrays para D y la tabla de visitados, el costo temporal del algoritmo es de $O(N^2)$.

Si el número de aristas a es muy inferior a N^2 es más eficiente utilizar una representación con listas de adyacencia para el grafo dirigido, y usar una cola con prioridad implementada en un heap para almacenar los vértices del conjunto $V-S$ según la prioridad correspondiente a su tiempo ($D[v]$).

En este caso, se debería hacer lo siguiente:

- Cargar en un heap de mínimo la información sobre los nodos no visitados y el costo de alcanzar cada uno de ellos desde el vértice origen.

- Mientras el heap tenga elemento,

- remover la raíz y restaurar el heap

- para cada nodo adyacente al removido (revisar la lista de adyacencia de dicho nodo) determinar si hay mejora. Si la hay, y el nodo no fue visitado, actualizar los valores del heap, asegurando que quede restaurado.

Como ya hemos visto, el costo de extraer la raíz de un heap y restaurarlo es $O(\log N)$.

El ciclo de actualización de los costos se puede llevar a cabo recorriendo solamente la lista de adyacencia para w , ya que solo los adyacentes a w pueden mejorar sus tiempos.

Con cada nuevos valores se actualiza el heap a un costo $O(\log N)$.

¿cuántas actualizaciones se hacen sobre el heap?

El número de actualizaciones está dado por el número de aristas del grafo (para cada w se actualizarán los vértices indicados por su lista de adyacencia, entonces, al considerar todos los w posibles, el total de actualizaciones es igual al total de aristas del grafo)

Cada actualización implica en el peor caso una actualización del heap a un costo logarítmico, entonces el costo temporal total es $O(a \log N)$ siendo a el número de aristas del grafo.

En este caso, el algoritmo puede formularse como sigue:

```
{
Inicializar todas las posiciones de D a infinito
Inicializar S en vacío
Meter los vértices v a una cola de prioridad Q de acuerdo a D [v]
  Mientras Q no este vacía hacer
    {
      Extraer el mínimo de Q en u
      S = S unión {u}
      Para cada v adyacente a u hacer
        D[v]=mínimo(D[v] , D[u] + M(u,v) )
    }
}
```

Problema de los caminos más cortos entre todos los pares de vértices.

Se trata de determinar los caminos mínimos que unen cada vértice del grafo con todos los otros.

Si bien lo más común es que se aplique a grafos que no tengan aristas negativas, la restricción necesaria es que no existan ciclos con costos negativos.

Este problema no verifica las condiciones que aseguran que un algoritmo “greedy” funcione, por lo cual no puede aplicarse esa estrategia. Sin embargo, verifica las condiciones que permiten aplicar un algoritmo de los llamados de “programación dinámica”

Una breve digresión sobre la Programación Dinámica

La programación dinámica suele aplicarse a problemas de optimización. También considera que a la solución se llega a través de una secuencia de decisiones, las cuales deben verificarla condición de que “en una secuencia de decisiones óptima toda subsecuencia ha de ser también óptima” (lo cual, dicho sea de paso, se verificaba también en la estrategia “greedy”).

En estos problemas se realiza una división de problemas en otros problemas menores, pero, a diferencia de lo que ocurría en “Divide y Vencerás”, estos problemas resultado de la división no son independientes entre si, sino que tienen subproblemas en común, hay un ‘solapamiento’ de problemas.

La técnica consiste en resolver y almacenar las soluciones de las zonas solapadas para no volver a realizar los mismos cálculos.

En general en esta estrategia se llega a la solución realizando comparaciones y actualizaciones en datos que han sido tabulados (las soluciones de los subproblemas).

Algoritmo de Floyd

Este algoritmo se basa en una técnica de programación dinámica que almacena en cada iteración el mejor camino entre el que pasa por el nodo intermedio k y el que va directamente del nodo i al nodo j .

La programación dinámica, como se dijo, se basa en dividir el problema en subproblemas, los cuales deben tener ‘zonas comunes’ que constituyen a su vez nuevos subproblemas con las mismas características. El método procede resolviendo estos subproblemas comunes en primer lugar, almacenando los resultados y utilizándolos en la resolución de los problemas mayores. La estrategia de PD es aplicable a este problema por cumplirse el principio de optimalidad para las soluciones factibles.

Para determinar cuál es el mejor camino considera mínimo $(D[v], D[u] + M(u,v))$.

Inicialmente se carga una matriz de $N \times N$ con los pesos correspondientes a las aristas, asignándose 0 a $M[i][i]$ para todo i .

El método lleva a cabo N iteraciones del proceso en cada una de las cuales se evalúa la eventual mejora de los tiempos por la consideración del vértice i (que variara entre 1 y N a lo largo del proceso) como intermedio.

El algoritmo de Floyd usa una matriz **A** de $N \times N$ en la que se calculan las longitudes de los caminos más cortos.

Inicialmente se hace **A**[i , j] = **C**[i , j] para toda $i \leftrightarrow j$.

Si no existe arco que vaya de i a j , se supone que **C**[i , j] = infinito

Los elementos de la diagonal principal son 0.

Luego se realizan N iteraciones en la matriz **A**.

Al terminar la k -ésima iteración, **A**[i , j] contiene la menor longitud de cualquier camino que vaya desde el vértice i hasta el vértice j y que no pase por un vértice de número mayor que k .

Esta es la formula a aplicar en la k -ésima iteración para calcular **A**.

$$A_k[i, j] = \min (A_{k-1}[i, j], A_{k-1}[i, k] + A_{k-1}[k, j])$$

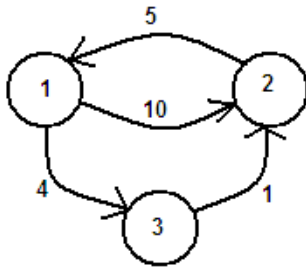
Este es el algoritmo de Floyd expresado de manera informal: (N es el número de vértices)

```
{
Inicializar A con ceros en la diagonal principal y los pesos indicados por el grafo para
cada uno de los restantes elementos. Si una arista no existe, se coloca peso infinito.
Para k desde 1 hasta N
  Para i desde 1 hasta N
    Para j desde 1 hasta N
      A[i][j] = mínimo (A[i][j], A[i][k] + A[k][j])
}
```

El coste temporal de este algoritmo es, obviamente $O(N^3)$.

Ejemplo:

Consideremos el siguiente grafo



La matriz A será

	1	2	3
1	0	10	4
2	5	0	∞
3	∞	1	0

Pasando por el vértice 1, luego de modificada, A queda así:

	1	2	3
1	0	10	4
2	5	0	7
3	∞	1	0

Pasando por el vértice 2 la matriz mejorada queda así:

	1	2	3
1	0	10	4
2	5	0	7
3	6	1	0

Pasando por el vértice 3, A queda finalmente así:

	1	2	3
1	0	5	4
2	5	0	7
3	6	1	0

Algunos problemas clásicos sobre Grafos no dirigidos.

Como ya se ha definido, un árbol libre es un grafo no dirigido conexo sin ciclos.
 Se verifica que en todo árbol libre con N vértices ($N > 1$), el árbol contiene $N-1$ aristas.
 Si se agrega una arista a un árbol libre, aparece un ciclo.

Definición de Árbol abarcador:

Dado $G = (V, A)$ grafo no dirigido conexo, un árbol abarcador para G , es un árbol libre que conecta todos los vértices de V .

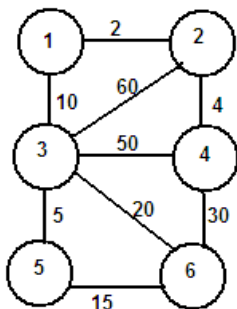
El concepto de árbol abarcador se asocia a grafos conexos.

Para un mismo grafo se pueden obtener diferentes árboles abarcadores.

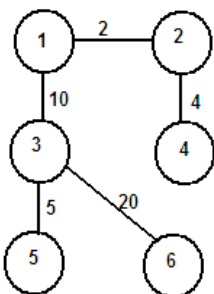
Arbol Abarcador de Coste Mínimo:

Dado un grafo $G=(V,A)$ no dirigido con aristas ponderadas (es decir, que cada arista (v, w) de A tiene un costo asociado, $C(v, w)$), se llama árbol abarcador de coste mínimo al árbol abarcador para G que verifica que la sumatoria de los pesos de las aristas es mínima. (El árbol abarcador de costo mínimo, puede no ser único).

Ejemplo: Para este grafo



El árbol abarcador de coste mínimo es;



Cuyo coste es $2+10+4+5+20=41$

Algoritmos que permiten obtener el árbol de expansión de coste mínimo para un grafo dado

Existen varios algoritmos que permiten obtener el árbol de expansión de coste mínimo. Se detallaran dos de ellos: el algoritmo de Prim y el de Kruskal.

En ambos casos se aplica la estrategia 'greedy' o voraz para alcanzar el óptimo.

Algoritmo de Prim.

Consideremos un grafo $G(V,A)$ tal que $V = \{ 1, 2, 3, \dots, n \}$.

El algoritmo habilita un conjunto U al que se le asigna inicialmente el valor 1,

$$U = \{1\}.$$

En dicho conjunto crece un árbol abarcador de costo mínimo, arista por arista.

En cada paso, localiza la arista de menor costo (u, v) (con u en U y v en $V - U$) que conecta a U y $V - U$.

Una vez localizada la arista agrega el vértice v al conjunto U , creciendo de esta forma el árbol, porque la arista en cuestión es de costo mínimo

Esto se repite hasta que $U = V$ (condición de parada).

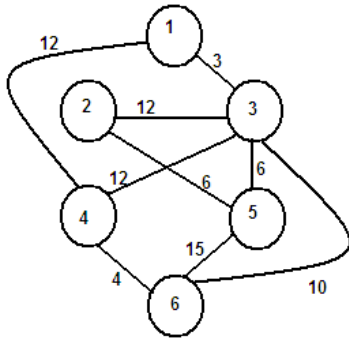
Llamando G al grafo y A al conjunto de aristas del árbol abarcador de coste mínimo, es

```
{
  A se inicializa en vacío
  U = {1} //se almacena vértice inicial
  Mientras U distinto de V hacer:
  {
    Determinar la arista de costo mínimo, (u,v) tal que u ∈ U y v ∈ (V-U)
    Agregar a A la arista (u, v)
    Agregar a U el vértice v
  }
}
```

La complejidad temporal de este algoritmo puede ser $O(n^3)$, dado que en cada iteración se deben analizar todas las aristas de cada vértice en U cuyos vértices estén en $(V-U)$ para detectar la menor de todas.

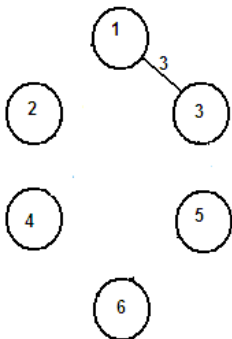
Gráficamente:

Consideremos partir del siguiente grafo:



La construcción del árbol abarcador de coste mínimo según el algoritmo de Prim es así:

Paso 1: se selecciona la menor arista que conecte el vértice 1 con alguno de los otros

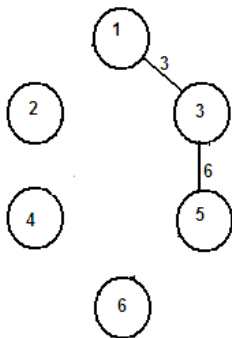


Las aristas disponibles para la conexión tienen pesos 3 y 12.

La de menor peso vale 3.

$U=\{1, 3\}$

Paso 2: se elige la arista de menor costo entre las que ligan alguno de los vértices del conjunto $\{1,3\}$ con otro de los vértices de G

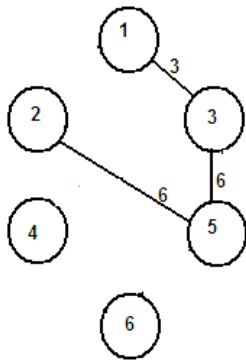


Las aristas disponibles para la conexión tienen pesos 12 (hay tres) y 6.

La de menor peso vale 6.

$U=\{1,3,5\}$

Paso 3: se selecciona una arista de peso mínimo que enlace un vértice de $\{1,3,5\}$ con alguno de los otros nodos.

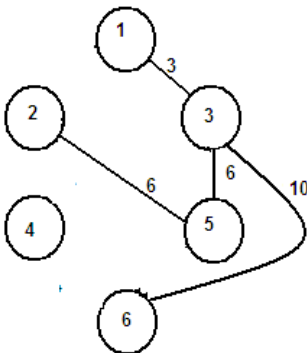


Las aristas disponibles para la conexión tienen pesos 12 (hay tres), 6 y 15

La de menor peso vale 6.

$U=\{1, 2, 3, 5\}$

Paso 4: se selecciona una arista de peso mínimo que enlace un vértice de $\{1, 2, 3, 5\}$ con alguno de los otros nodos.

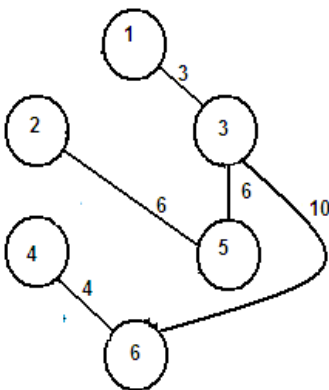


Las aristas disponibles para la conexión tienen pesos 12 (hay dos), 10 y 15

La de menor peso vale 10.

$U=\{1,2, 3, 5, 6\}$

Paso 5: Paso 4: se selecciona una arista de peso mínimo que enlace un vértice de $\{1, 2, 3, 5, 6\}$ con alguno de los otros nodos.



Las aristas disponibles para la conexión tienen pesos 12 (hay dos), y 4

La de menor peso vale 4.

$U=\{1,2, 3, 4, 5, 6\}$

Se ha obtenido el árbol de expansión de coste mínimo, con un costo de $3+6+6+4+10=29$

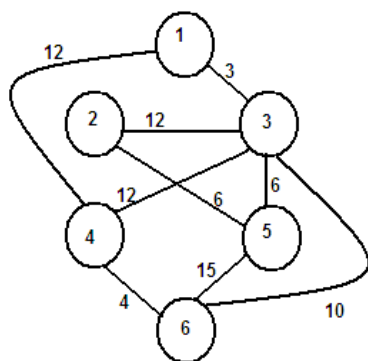
La secuencia de Prim, con esta variante, para el ejemplo anterior sería así:

Algoritmo de Kruskal.

Dado un grafo conexo $G = (V, A)$, con $V = \{1, 2, \dots, n\}$ y una función de costo C definida en las aristas de A , el algoritmo de Kruskal propone empezar con un grafo que contenga a todos los vértices de V pero no a sus aristas (es decir se comienza con un grafo con N subgrafos conexos, cada uno de los cuales corresponde a un vértice), llamémoslo T

La construcción del árbol abarcador de coste mínimo se hace de este modo: se itera sobre el conjunto de las aristas de G , ordenadas por peso. En cada iteración se halla una arista de costo mínimo que conecte a dos componentes conexas diferentes y se le añade dicha arista a T . De este modo se forma una nueva componente conexas a partir de otras dos. Cuando todos los vértices pertenezcan a una misma componente, el algoritmo finaliza, ya que esta componente única será un árbol abarcador de costo mínimo para el grafo G dado.

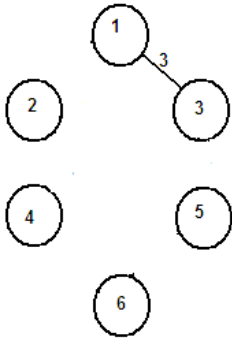
Para este grafo:



La lista de aristas en orden creciente de pesos es:

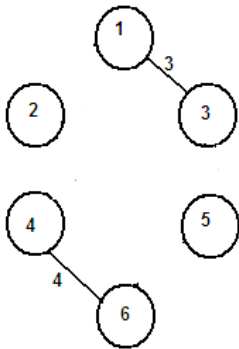
Costo	Aristas
3	(1, 3)
4	(4, 6)
6	(2, 5) , (3,5)
10	(3, 6)
12	(1, 4), (2, 3), (3, 4)
15	(5, 6)

Ahora, veamos cuáles son las componentes conexas que se forman en cada paso del algoritmo:



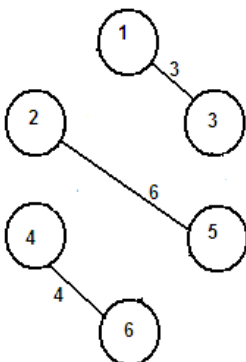
Se elige la arista de peso mas bajo,
que vale 3

Ahora hay 5 componentes conexas



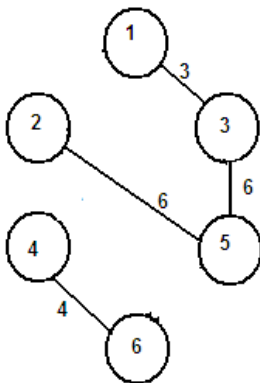
La arista de peso siguiente vale 4 y no
forma ciclo. Se agrega.

Ahora hay 4 componentes conexas



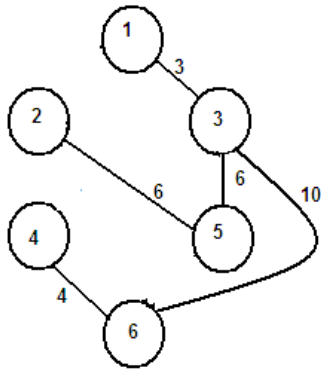
Las aristas de peso siguiente valen
6. Ninguna de ellas forma ciclo. Se elige
una de ellas y se agrega.

Ahora hay 3 componentes conexas



La siguiente arista también vale 6 y no
forma ciclo. Se agrega.

Ahora hay 2 componentes conexas



Las siguiente arista vale 10 y no forma ciclo. Se agrega.

Ahora hay 1 sola componente conexa.

Al considerar la complejidad de este algoritmo se debe evaluar el método usado para para ordenar las aristas por peso. Si hay A aristas, un buen método será $O(A \log A)$.

El ciclo de selección de la arista se lleva a cabo a lo sumo A veces (cuando se deban inspeccionar todas las aristas); dentro del ciclo, las operaciones son $O(\log V)$, entonces el ciclo es para el peor caso, de una complejidad que corresponde al máximo entre $A \log A$ y $A \log V$.

Como el árbol libre tiene $V - 1$ aristas, y $A > V - 1$ siempre que se trate de un grafo conexo, resulta que $\log A > \log (V - 1)$ y por tanto $A \log A > A \log V$.

Entonces, el algoritmo de Kruskal es $O(A \log A)$.