Programación III

Ricardo Wehbe

UADE

10 de septiembre de 2021

Programa

1. Repaso de la clase anterior
2. Grafos
   - Caminos más cortos en grafos: Dijkstra
   - Árboles de recubrimiento mínimo
   - El algoritmo de Prim
   - El algoritmo de Kruskal
3. Ejercicios propuestos

1. Repaso de la clase anterior

2. Grafos
   - Caminos más cortos en grafos: Dijkstra
   - Árboles de recubrimiento mínimo
   - El algoritmo de Prim
   - El algoritmo de Kruskal

3. Ejercicios propuestos

Algoritmos *greedy*

Algoritmos *greedy*

- Es una técnica de diseño de algoritmos que también se conocen como algoritmos *voraces* o *glotones* por su traducción desde el inglés.

Algoritmos *greedy*

- Es una técnica de diseño de algoritmos que también se conocen como algoritmos *voraces* o *glotones* por su traducción desde el inglés.
- Un algoritmo *greedy* construye la solución a partir de decisiones parciales basadas en la información disponible en el momento. No considera los efectos de sus decisiones en el futuro y nunca reconsidera una decisión ya tomada.

Algoritmos *greedy*

- Es una técnica de diseño de algoritmos que también se conocen como algoritmos *voraces* o *glotones* por su traducción desde el inglés.
- Un algoritmo *greedy* construye la solución a partir de decisiones parciales basadas en la información disponible en el momento. No considera los efectos de sus decisiones en el futuro y nunca reconsidera una decisión ya tomada.
- Se suelen utilizar para resolver problemas de optimización. Son muy eficientes, pero hay que demostrar formalmente su corrección.

Algoritmos *greedy*

- Es una técnica de diseño de algoritmos que también se conocen como algoritmos *voraces* o *glotones* por su traducción desde el inglés.
- Un algoritmo *greedy* construye la solución a partir de decisiones parciales basadas en la información disponible en el momento. No considera los efectos de sus decisiones en el futuro y nunca reconsidera una decisión ya tomada.
- Se suelen utilizar para resolver problemas de optimización. Son muy eficientes, pero hay que demostrar formalmente su corrección.
- Como el nombre lo sugiere, son "cortos de vista."

Algoritmos *greedy*. Candidatos y criterios.

Algoritmos *greedy*. Candidatos y criterios.

- Un algoritmo *greedy* selecciona en cada momento el mejor candidato para ser parte de una solución según un criterio determinado.

Algoritmos *greedy*. Candidatos y criterios.

- Un algoritmo *greedy* selecciona en cada momento el mejor candidato para ser parte de una solución según un criterio determinado.
- En otras palabras, en cada caso se evalúa un candidato de la lista de que disponemos y, dependiendo de esa evaluación, se lo incluye en la solución o se lo deja de lado.

Elementos de una solución *greedy*

Elementos de una solución *greedy*

- **Conjunto de candidatos**: el conjunto de objetos disponibles para incluir en la solución.

Elementos de una solución *greedy*

- **Conjunto de candidatos**: el conjunto de objetos disponibles para incluir en la solución.
- **Función de selección**: el criterio para seleccionar el mejor candidato.

Elementos de una solución *greedy*

- **Conjunto de candidatos**: el conjunto de objetos disponibles para incluir en la solución.
- **Función de selección**: el criterio para seleccionar el mejor candidato.
- **Función de factibilidad**: el criterio para determinar si un candidato puede ser incluido en la solución.

Elementos de una solución *greedy*

- **Conjunto de candidatos**: el conjunto de objetos disponibles para incluir en la solución.
- **Función de selección**: el criterio para seleccionar el mejor candidato.
- **Función de factibilidad**: el criterio para determinar si un candidato puede ser incluido en la solución.
- **Función solución**: el criterio para determinar si un conjunto solución resuelve efectivamente el problema.

Elementos de una solución *greedy*

- **Conjunto de candidatos**: el conjunto de objetos disponibles para incluir en la solución.
- **Función de selección**: el criterio para seleccionar el mejor candidato.
- **Función de factibilidad**: el criterio para determinar si un candidato puede ser incluido en la solución.
- **Función solución**: el criterio para determinar si un conjunto solución resuelve efectivamente el problema.
- **Objetivo**: la magnitud que se quiere maximizar o minimizar.

La teoría de grafos, Euler y los puentes de Königsberg

La teoría de grafos, Euler y los puentes de Königsberg

- La ciudad de Königsberg (hoy Kaliningrad) está atravesada por el río Pregel. Entre la rivera norte y la rivera sur hay dos islas, Kneiphof and Lomse.

La teoría de grafos, Euler y los puentes de Königsberg

- La ciudad de Königsberg (hoy Kaliningrad) está atravesada por el río Pregel. Entre la rivera norte y la rivera sur hay dos islas, Kneiphof and Lomse.
- En el s. XVIII, cuando Euler residía en esta ciudad, siete puentes conectaban las diferentes partes de la ciudad, como se muestra en el siguiente slide. Dos unían Kneiphof con a rivera sur y dos con la rivera norte. Había además tres puentes que unían a Lomse con la rivera norte, con la rivera sur y con la isla de Kneiphof.

La teoría de grafos, Euler y los puentes de Königsberg

- La ciudad de Königsberg (hoy Kaliningrad) está atravesada por el río Pregel. Entre la rivera norte y la rivera sur hay dos islas, Kneiphof and Lomse.
- En el s. XVIII, cuando Euler residía en esta ciudad, siete puentes conectaban las diferentes partes de la ciudad, como se muestra en el siguiente slide. Dos unían Kneiphof con a rivera sur y dos con la rivera norte. Había además tres puentes que unían a Lomse con la rivera norte, con la rivera sur y con la isla de Kneiphof.
- Una pregunta folklórica en la ciudad era si era posible, comenzando desde cualquier parte de la ciudad, cruzar todos los puentes exactamente una vez y terminar en el punto de origen.

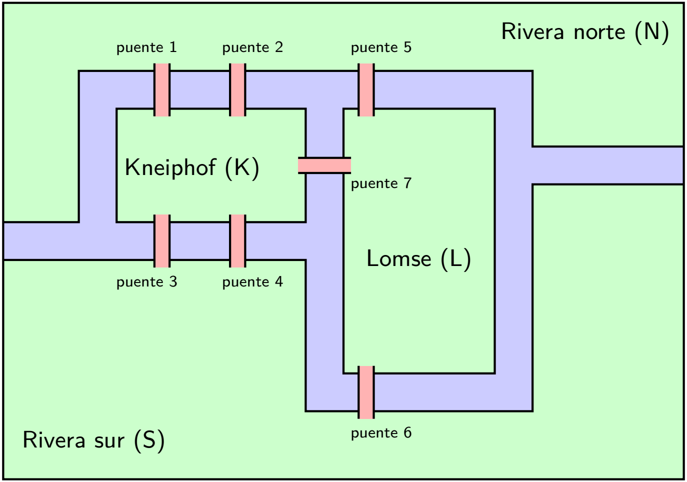
La teoría de grafos, Euler y los puentes de Königsberg

- La ciudad de Königsberg (hoy Kaliningrad) está atravesada por el río Pregel. Entre la rivera norte y la rivera sur hay dos islas, Kneiphof and Lomse.
- En el s. XVIII, cuando Euler residía en esta ciudad, siete puentes conectaban las diferentes partes de la ciudad, como se muestra en el siguiente slide. Dos unían Kneiphof con a rivera sur y dos con la rivera norte. Había además tres puentes que unían a Lomse con la rivera norte, con la rivera sur y con la isla de Kneiphof.
- Una pregunta folklórica en la ciudad era si era posible, comenzando desde cualquier parte de la ciudad, cruzar todos los puentes exactamente una vez y terminar en el punto de origen.
- Euler acabó con este pasatiempo demostrndo la imposibilidad de tal solución. Esto se conoce ahora como un *círculo Euleriano*.

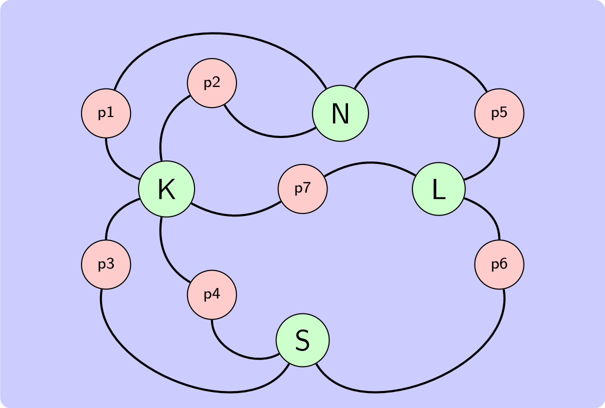
La teoría de grafos, Euler y los puentes de Königsberg

- La ciudad de Königsberg (hoy Kaliningrad) está atravesada por el río Pregel. Entre la rivera norte y la rivera sur hay dos islas, Kneiphof and Lomse.
- En el s. XVIII, cuando Euler residía en esta ciudad, siete puentes conectaban las diferentes partes de la ciudad, como se muestra en el siguiente slide. Dos unían Kneiphof con a rivera sur y dos con la rivera norte. Había además tres puentes que unían a Lomse con la rivera norte, con la rivera sur y con la isla de Kneiphof.
- Una pregunta folklórica en la ciudad era si era posible, comenzando desde cualquier parte de la ciudad, cruzar todos los puentes exactamente una vez y terminar en el punto de origen.
- Euler acabó con este pasatiempo demostrndo la imposibilidad de tal solución. Esto se conoce ahora como un *círculo Euleriano*.
- Una versión más débil del problema es atravesar todos los puentes exactamente una vez pero sin terminar en el punto de partida. Esto se conoce como *camino Euleriano* y en este caso es también imposible.

Los puentes de Königsberg



Los puentes de Königsberg



La imposibilidad de una solución

La imposibilidad de una solución

- ¿Por qué es imposible una solución?

La imposibilidad de una solución

- ¿Por qué es imposible una solución?
- Puesto que queremos describir un ciclo, cada vértice debe tener un número par de puentes incidentes.

La imposibilidad de una solución

- ¿Por qué es imposible una solución?
- Puesto que queremos describir un ciclo, cada vértice debe tener un número par de puentes incidentes.
- Por cada puente de llegada debe existir uno diferente de salida.

La imposibilidad de una solución

- ¿Por qué es imposible una solución?
- Puesto que queremos describir un ciclo, cada vértice debe tener un número par de puentes incidentes.
- Por cada puente de llegada debe existir uno diferente de salida.
- Pero todos los nodos tienen aquí un número impar de puentes incidentes; por lo tanto, un ciclo Euleriano es imposible.

La imposibilidad de una solución

- ¿Por qué es imposible una solución?
- Puesto que queremos describir un ciclo, cada vértice debe tener un número par de puentes incidentes.
- Por cada puente de llegada debe existir uno diferente de salida.
- Pero todos los nodos tienen aquí un número impar de puentes incidentes; por lo tanto, un ciclo Euleriano es imposible.
- Análogamente, todos los puntos intermedios de un Euleriano deben estar conectados a un número par de puentes y los puntos inicial y final (si no son el mismo)deben estar conectados a un número impar de puentes.

La imposibilidad de una solución

- ¿Por qué es imposible una solución?
- Puesto que queremos describir un ciclo, cada vértice debe tener un número par de puentes incidentes.
- Por cada puente de llegada debe existir uno diferente de salida.
- Pero todos los nodos tienen aquí un número impar de puentes incidentes; por lo tanto, un ciclo Euleriano es imposible.
- Análogamente, todos los puntos intermedios de un Euleriano deben estar conectados a un número par de puentes y los puntos inicial y final (si no son el mismo)deben estar conectados a un número impar de puentes.
- Esto es imposible, pues todos los nodos están conectados a un número impar de puentes.

Euler y los puentes de Königsberg. Colofón

Euler y los puentes de Königsberg. Colofón

- Königsberg fue tomada en 1945 por los soviéticos y rebautizada Kaliningrad.

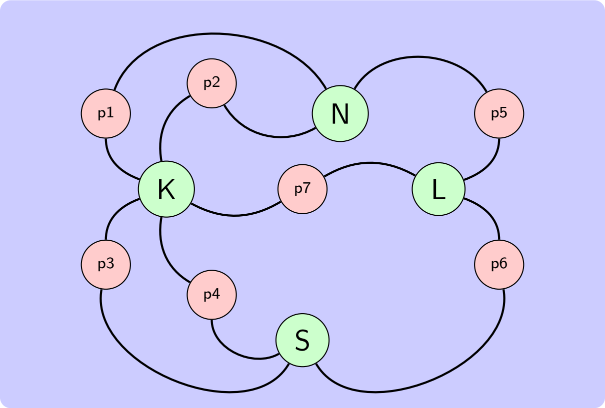
Euler y los puentes de Königsberg. Colofón

- Königsberg fue tomada en 1945 por los soviéticos y rebautizada Kaliningrad.
- Los bombardeos destruyeron dos de los puentes. Actualmente sólo hay un puente entre la rivera norte y Kneiphof y otro entre la rivera sur y Kneiphof. Los tres puentes de Lomse permanecen.

Euler y los puentes de Königsberg. Colofón

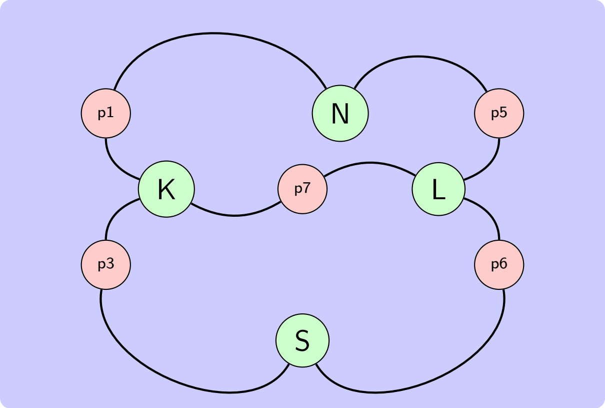
- Königsberg fue tomada en 1945 por los soviéticos y rebautizada Kaliningrad.
- Los bombardeos destruyeron dos de los puentes. Actualmente sólo hay un puente entre la rivera norte y Kneiphof y otro entre la rivera sur y Kneiphof. Los tres puentes de Lomse permanecen.
- Actualmente un camino Euleriano es posible. Un círculo Euleriano sigue siendo imposible.

Los puentes de Königsberg. La situación actual



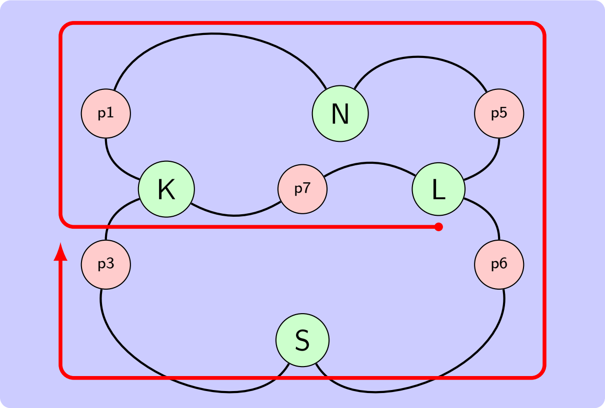
Antes de la guerra

Los puentes de Königsberg. La situación actual



Después de la guerra

Los puentes de Königsberg. La situación actual



Un camino Euleriano

Grafos. Un souvenir

Grafos. Un souvenir

- Recordemos que un *grafo* es un par (V, A) en el que V es un conjunto de *vértices* y $A \subseteq V \times V$ es un conjunto de *aristas*.

Grafos. Un souvenir

- Recordemos que un *grafo* es un par (V, A) en el que V es un conjunto de *vértices* y $A \subseteq V \times V$ es un conjunto de *aristas*.
- Un grafo puede ser *dirigido* o *no dirigido*. En el último caso, tenemos la equivalencia $(x, y) \equiv (y, x)$ para todo $(x, y) \in A$.

Algunas definiciones sobre grafos

Algunas definiciones sobre grafos

- Un *camino* en un grafo (V, A) es una secuencia finita $v_1, \ldots, v_n$ de vértices de V tal que $(v_i, v_{i+1}) \in A$ para todo $i \in \{1, \ldots, n-1\}$.

Algunas definiciones sobre grafos

- Un *camino* en un grafo (V, A) es una secuencia finita $v_1, \ldots, v_n$ de vértices de V tal que $(v_i, v_{i+1}) \in A$ para todo $i \in \{1, \ldots, n-1\}$.
- Un grafo (V, A) es *conectado* o *conexo* si para cualquier par de nodos $x, y \in V$ hay un camino de x a y.

Algunas definiciones sobre grafos

- Un *camino* en un grafo (V, A) es una secuencia finita $v_1, \ldots, v_n$ de vértices de V tal que $(v_i, v_{i+1}) \in A$ para todo $i \in \{1, \ldots, n-1\}$.
- Un grafo (V, A) es *conectado* o *conexo* si para cualquier par de nodos $x, y \in V$ hay un camino de x a y.
- Un grafo (V, A) es *fuertemente conexo* si para cualquier par de nodos $x, y \in V$, $(x, y) \in A$.

Algunas definiciones sobre grafos

- Un *camino* en un grafo (V, A) es una secuencia finita $v_1, \ldots, v_n$ de vértices de V tal que $(v_i, v_{i+1}) \in A$ para todo $i \in \{1, \ldots, n-1\}$.
- Un grafo (V, A) es *conectado* o *conexo* si para cualquier par de nodos $x, y \in V$ hay un camino de x a y.
- Un grafo (V, A) es *fuertemente conexo* si para cualquier par de nodos $x, y \in V$, $(x, y) \in A$.
- Se pueden asociar distancias o costos a las aristas. Para ello, basta modificar ligeramente la definición de grafo.

Algunas definiciones sobre grafos

- Un *camino* en un grafo (V, A) es una secuencia finita $v_1, \ldots, v_n$ de vértices de V tal que $(v_i, v_{i+1}) \in A$ para todo $i \in \{1, \ldots, n-1\}$.
- Un grafo (V, A) es *conectado* o *conexo* si para cualquier par de nodos $x, y \in V$ hay un camino de x a y.
- Un grafo (V, A) es *fuertemente conexo* si para cualquier par de nodos $x, y \in V$, $(x, y) \in A$.
- Se pueden asociar distancias o costos a las aristas. Para ello, basta modificar ligeramente la definición de grafo.
- En un grafo (V, A) con costos, tenemos $A \subseteq \mathbb{N} \times V \times V$.

Caminos mínimos en un grafo. El algoritmo de Dijkstra

Caminos mínimos en un grafo. El algoritmo de Dijkstra

- El problema es el siguiente: dado un grafo dirigido con costos (V, A) y un vértice $x \in V$, encontrar el camino de mínimo costo desde x a cualquier otro vértice de V.

Caminos mínimos en un grafo. El algoritmo de Dijkstra

- El problema es el siguiente: dado un grafo dirigido con costos (V, A) y un vértice $x \in V$, encontrar el camino de mínimo costo desde x a cualquier otro vértice de V.
- El resultado del proceso es un grafo con los mismos vértices pero que sólo tiene aristas desde el vértice x hacia los demás con el costo encontrado.

Caminos mínimos en un grafo. El algoritmo de Dijkstra

- El problema es el siguiente: dado un grafo dirigido con costos (V, A) y un vértice $x \in V$, encontrar el camino de mínimo costo desde x a cualquier otro vértice de V.
- El resultado del proceso es un grafo con los mismos vértices pero que sólo tiene aristas desde el vértice x hacia los demás con el costo encontrado.
- El algoritmo de Dijkstra es un algoritmo *greedy* que resuelve este problema.

El algoritmo de Dijkstra. Estrategia

El algoritmo de Dijkstra. Estrategia

- El conjunto de candidatos es el conjunto de vértices del grafo sin el vértice de origen.

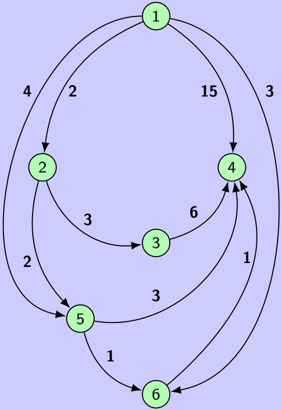
El algoritmo de Dijkstra. Estrategia

- El conjunto de candidatos es el conjunto de vértices del grafo sin el vértice de origen.
- En cada iteración, el método de selección elige el candidato que tenga el camino de mínimo costo desde el origen.

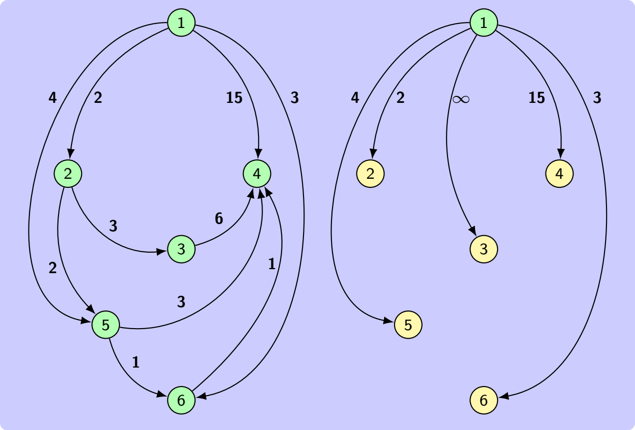
El algoritmo de Dijkstra. Estrategia

- El conjunto de candidatos es el conjunto de vértices del grafo sin el vértice de origen.
- En cada iteración, el método de selección elige el candidato que tenga el camino de mínimo costo desde el origen.
- La función de factibilidad compara el costo del camino directo con el costo del camino que pasa por el vértice seleccionado, en caso de que este camino exista.

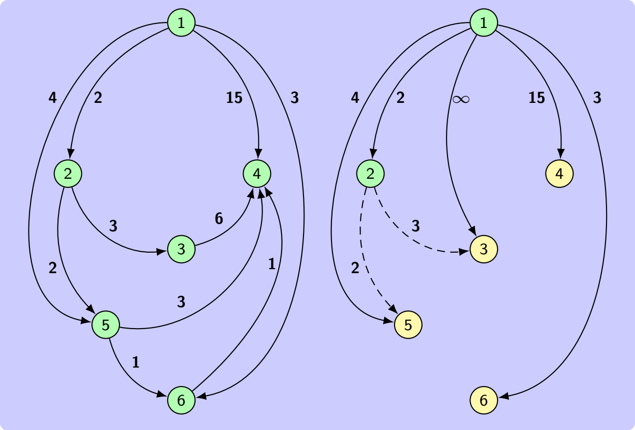
El algoritmo de Dijkstra. Un ejemplo



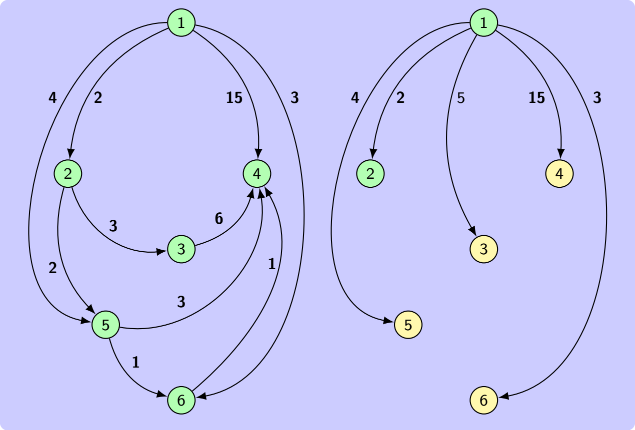
El algoritmo de Dijkstra. Un ejemplo



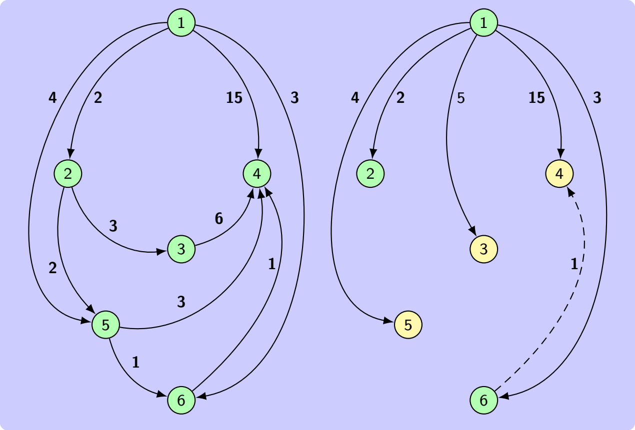
El algoritmo de Dijkstra. Un ejemplo



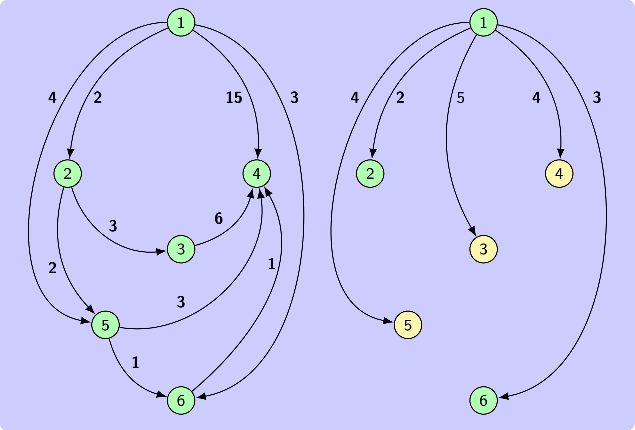
El algoritmo de Dijkstra. Un ejemplo



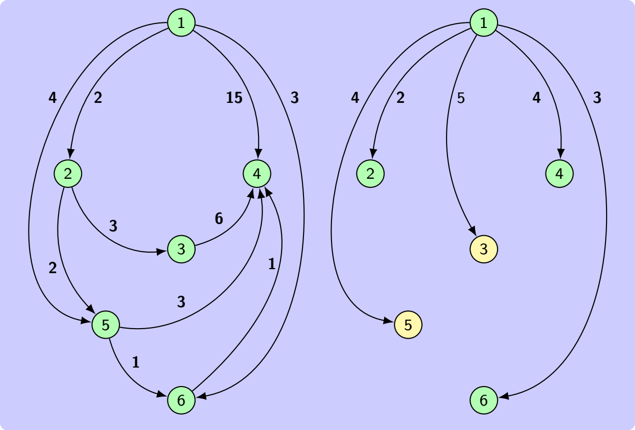
El algoritmo de Dijkstra. Un ejemplo



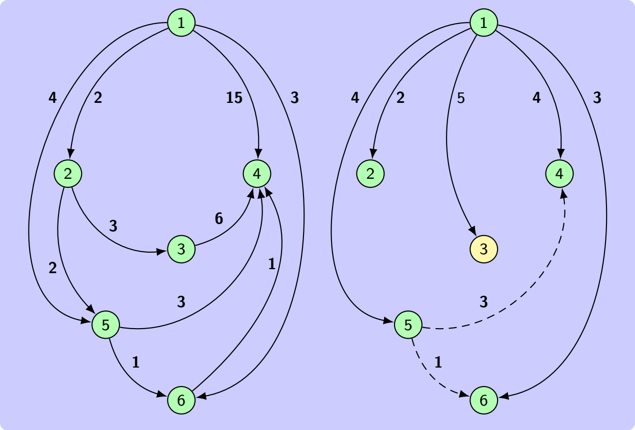
El algoritmo de Dijkstra. Un ejemplo



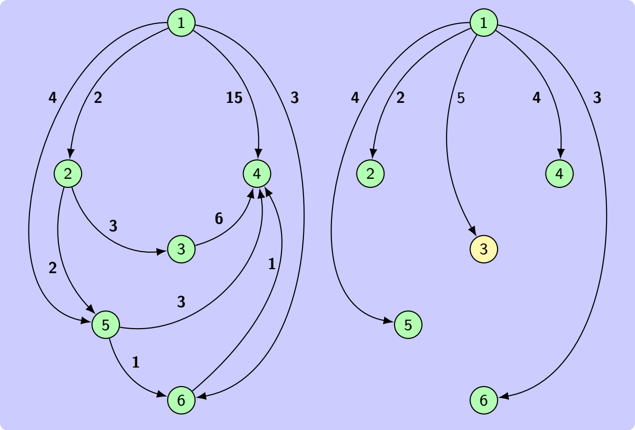
El algoritmo de Dijkstra. Un ejemplo



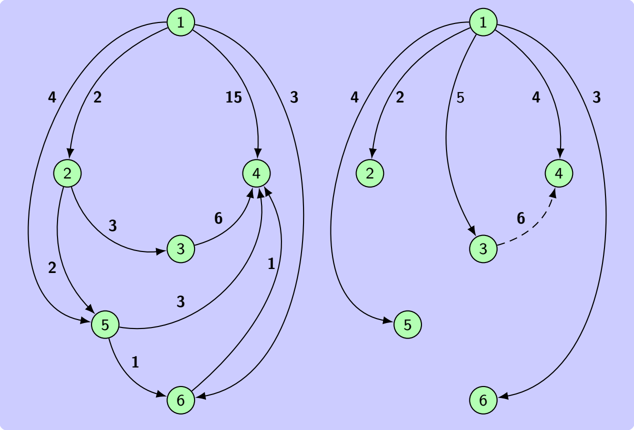
El algoritmo de Dijkstra. Un ejemplo



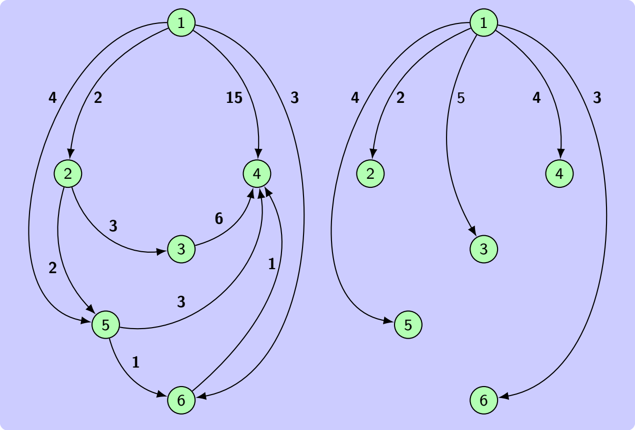
El algoritmo de Dijkstra. Un ejemplo



El algoritmo de Dijkstra. Un ejemplo



El algoritmo de Dijkstra. Un ejemplo



El algoritmo de Dijkstra. Algunas consideraciones

El algoritmo de Dijkstra. Algunas consideraciones

- Asumimos que tenemos una clase *grafo* con los métodos *InicializarGrafo*, que crea un grafo vacío, *AgregarVertice* y *AgregarArista*, que hacen exactamente lo que anuncian.

El algoritmo de Dijkstra. Algunas consideraciones

- Asumimos que tenemos una clase *grafo* con los métodos *InicializarGrafo*, que crea un grafo vacío, *AgregarVertice* y *AgregarArista*, que hacen exactamente lo que anuncian.
- Asumimos además un método *Vecindario* que, para un nodo v, devuelve el conjunto de todos los nodos adyacentes de v en G. Es decir, los nodos w tales que existe una arista de v a w (los *vecinos* de v.) Observe que aquí nos estamos tomando libertades con el tipo de datos abstracto que habíamos visto en Programación II.

El algoritmo de Dijkstra. Algunas consideraciones

- Asumimos que tenemos una clase *grafo* con los métodos *InicializarGrafo*, que crea un grafo vacío, *AgregarVertice* y *AgregarArista*, que hacen exactamente lo que anuncian.
- Asumimos además un método *Vecindario* que, para un nodo v, devuelve el conjunto de todos los nodos adyacentes de v en G. Es decir, los nodos w tales que existe una arista de v a w (los *vecinos* de v.) Observe que aquí nos estamos tomando libertades con el tipo de datos abstracto que habíamos visto en Programación II.
- Tenemos además los métodos del tipo de datos abstracto `Conjunto` tal como se lo vio en Programación II ¿Recuerdan?.

El algoritmo de Dijkstra. Algunas consideraciones

- Asumimos que tenemos una clase *grafo* con los métodos *InicializarGrafo*, que crea un grafo vacío, *AgregarVertice* y *AgregarArista*, que hacen exactamente lo que anuncian.
- Asumimos además un método *Vecindario* que, para un nodo v, devuelve el conjunto de todos los nodos adyacentes de v en G. Es decir, los nodos w tales que existe una arista de v a w (los *vecinos* de v.) Observe que aquí nos estamos tomando libertades con el tipo de datos abstracto que habíamos visto en Programación II.
- Tenemos además los métodos del tipo de datos abstracto `Conjunto` tal como se lo vio en Programación II ¿Recuerdan?.
- Finalmente, asumimos la existencia de una operación de resta de conjuntos con el operador $\setminus$ y *CopiarConjunto*.

El algoritmo de Dijkstra. Partes uno y dos

```
algoritmo Dijkstra (G : GrafoTDA, v : int)
    Visitados = {v}
    Dijkstra.InicializarGrafo()                      // El grafo solución
    foreach w ∈ G.vertices {
        Dijkstra.AgregarVertice(w)                   // Los nodos de G
    }
    foreach w ∈ G.Vecindario(v) {
        Dijkstra.AgregarArista(v, w, G.Peso(v, w))
    }
    Candidatos.InicializarConjunto()                 // Nodos candidatos
    Candidatos = G.vertices \ Visitados
```

El algoritmo de Dijkstra. Parte tres

```
while !Candidatos.ConjuntoVacio() {                    // El proceso
    int min = ∞
    foreach u ∈ Candidatos {
        if (Dijkstra.ExisteArista(v, u) && Dijkstra.peso(v, u) < min) {
            min = Dijkstra.Peso(v, u)
            w = u
        }
    }
    Visitados.AgregarVertice(w)
    Candidatos.Sacar(w)
    auxCandidatos = Copiar(Candidatos)
    while !auxCandidatos.ConjuntoVacio {
        p = auxCandidatos.Elegir()
        auxCandidatos.Sacar(p)
        if (G.ExisteArista(w, p)) {
            if (Dijkstra.ExisteArista(v, p)) {
                if (Dijkstra.Peso(v, w) + G.Peso(w, p) < Dijkstra.Peso(v, p)) {
                    Dijkstra.AgregarArista(v, p, Dijkstra.Peso(v, w) + G.Peso(w, p))
                }
            } else {
                Dijkstra.AgregarArista(v, p, A.Peso(v, w) + G.peso(w, p))
            }
        }
    }
}
return Dijkstra
```

El algoritmo de Dijkstra. Complejidad, partes uno y dos

```
algoritmo Dijkstra (G : GrafoTDA, v : int)
    Visitados = {v}
    Dijkstra.InicializarGrafo()
    foreach w ∈ G.vertices {
        Dijkstra.AgregarVertice(w)
    }
    foreach w ∈ G.Vecindario(v) {
        Dijkstra.AgregarArista(v, w, G.Peso(v, w))
    }
    Candidatos.InicializarConjunto()
    Candidatos = G.vertices \ Visitados
```

El algoritmo de Dijkstra. Complejidad, partes uno y dos

$\Theta(n)$

```
algoritmo Dijkstra (G : GrafoTDA, v : int)
    Visitados = {v}
    Dijkstra.InicializarGrafo()
    foreach w ∈ G.vertices {
        Dijkstra.AgregarVertice(w)
    }
    foreach w ∈ G.Vecindario(v) {
        Dijkstra.AgregarArista(v, w, G.Peso(v, w))
    }
    Candidatos.InicializarConjunto()
    Candidatos = G.vertices\Visitados
```

$\Theta(n)$

$\Theta(n)$

El algoritmo de Dijkstra. Complejidad, parte tres

```
while !Candidatos.ConjuntoVacio() {
    int min = ∞
    foreach u ∈ Candidatos {
        if (Dijkstra.ExisteArista(v, u) && Dijkstra.peso(v, u) < min) {
            min = Dijkstra.Peso(v, u)
            w = u
        }
    }
    Visitados.AgregarVertice(w)
    Candidatos.Sacar(w)
    auxCandidatos = Copiar(Candidatos)
    while !auxCandidatos.ConjuntoVacio {
        p = auxCandidatos.Elegir()
        auxCandidatos.Sacar(p)
        if (G.ExisteArista(w, p)) {
            if (Dijkstra.ExisteArista(v, p)) {
                if (Dijkstra.Peso(v, w) + G.Peso(w, p) < Dijkstra.Peso(v, p)) {
                    Dijkstra.AgregarArista(v, p, Dijkstra.Peso(v, w) + G.Peso(w, p))
                }
            } else {
                Dijkstra.AgregarArista(v, p, A.Peso(v, w) + G.peso(w, p))
            }
        }
    }
}
return Dijkstra
```

El algoritmo de Dijkstra. Complejidad, parte tres

$\Theta(n)$

$\Theta(n)$

$\Theta(n)$

```
while !Candidatos.ConjuntoVacio() {
    int min = ∞
    foreach u ∈ Candidatos {
        if (Dijkstra.ExisteArista(v, u) && Dijkstra.peso(v, u) < min) {
            min = Dijkstra.Peso(v, u)
            w = u
        }
    }
    Visitados.AgregarVertice(w)
    Candidatos.Sacar(w)
    auxCandidatos = Copiar(Candidatos)
    while !auxCandidatos.ConjuntoVacio {
        p = auxCandidatos.Elegir()
        auxCandidatos.Sacar(p)
        if (G.ExisteArista(w, p)) {
            if (Dijkstra.ExisteArista(v, p)) {
                if (Dijkstra.Peso(v, w) + G.Peso(w, p) < Dijkstra.Peso(v, p)) {
                    Dijkstra.AgregarArista(v, p, Dijkstra.Peso(v, w) + G.Peso(w, p))
                }
            } else {
                Dijkstra.AgregarArista(v, p, A.Peso(v, w) + G.peso(w, p))
            }
        }
    }
}
return Dijkstra
```

$\Theta(n^2)$

Árbol de recubrimiento mínimo (*minimum spanning tree*)

Árbol de recubrimiento mínimo (*minimum spanning tree*)

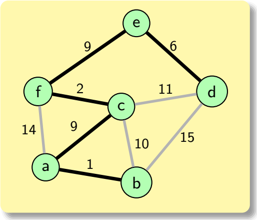
- Dado un grafo no dirigido y conectado con costos $G = (V, A)$, un *árbol de recubrimiento mínimo* para G es un grafo $G' = (V, A')$ con $A' \subseteq A$ que tiene estructura de árbol y que tiene costo mínimo.

Árbol de recubrimiento mínimo (*minimum spanning tree*)

- Dado un grafo no dirigido y conectado con costos $G = (V, A)$, un *árbol de recubrimiento mínimo* para G es un grafo $G' = (V, A')$ con $A' \subseteq A$ que tiene estructura de árbol y que tiene costo mínimo.
- Pueden existir múltiples árboles de recubrimiento mínimo para un mismo grafo.

Árbol de recubrimiento mínimo (*minimum spanning tree*)

- Dado un grafo no dirigido y conectado con costos $G = (V, A)$, un *árbol de recubrimiento mínimo* para G es un grafo $G' = (V, A')$ con $A' \subseteq A$ que tiene estructura de árbol y que tiene costo mínimo.
- Pueden existir múltiples árboles de recubrimiento mínimo para un mismo grafo.
- Por ejemplo:



Algoritmos *greedy* para árboles de recubrimiento mínimos

Algoritmos *greedy* para árboles de recubrimiento mínimos

- Existen dos algoritmos *greedy* para obtener un árbol de recubrimiento mínimo: el algoritmo de Prim y el de Kruskal.

Algoritmos *greedy* para árboles de recubrimiento mínimos

- Existen dos algoritmos *greedy* para obtener un árbol de recubrimiento mínimo: el algoritmo de Prim y el de Kruskal.
- Si el árbol de recubrimiento es único, ambos algoritmos arrojan el mismo resultado; si hay más de uno, es posible que los resultados difieran por la diferente estrategia que sigue cada uno de los algoritmos.

Algoritmo de Prim: estrategia

Algoritmo de Prim: estrategia

- Los candidatos son los vértices aún no incluidos. Se comienza con un vértice cualquiera.

Algoritmo de Prim: estrategia

- Los candidatos son los vértices aún no incluidos. Se comienza con un vértice cualquiera.
- La función de selección elige entre los candidatos al que tiene una arista de costo mínimo al conjunto de los vértices ya elegidos.

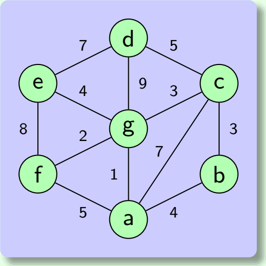
Algoritmo de Prim: estrategia

- Los candidatos son los vértices aún no incluidos. Se comienza con un vértice cualquiera.
- La función de selección elige entre los candidatos al que tiene una arista de costo mínimo al conjunto de los vértices ya elegidos.
- Dependiendo de la implementación, la función de factibilidad puede decidir no utilizar la arista encontrada.

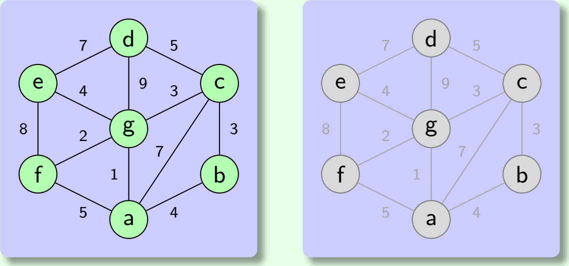
Algoritmo de Prim: estrategia

- Los candidatos son los vértices aún no incluidos. Se comienza con un vértice cualquiera.
- La función de selección elige entre los candidatos al que tiene una arista de costo mínimo al conjunto de los vértices ya elegidos.
- Dependiendo de la implementación, la función de factibilidad puede decidir no utilizar la arista encontrada.
- La función solución verifica que todos los vértices se hayan utilizado.

El algoritmo de Prim. Un ejemplo

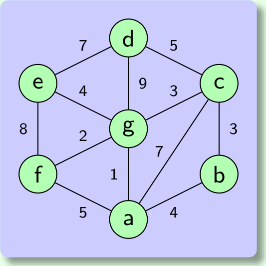


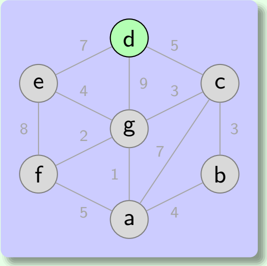
El algoritmo de Prim. Un ejemplo



visitados	candidatos
$\emptyset$	$\{a, b, c, d, e, f, g\}$

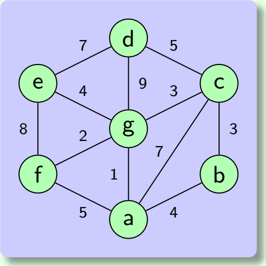
El algoritmo de Prim. Un ejemplo

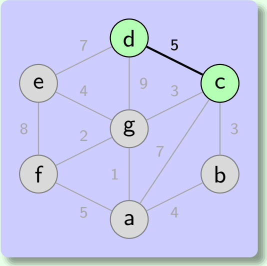




visitados	candidatos
$\{d\}$	$\{a,b,c,e,f,g\}$

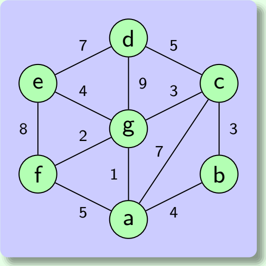
El algoritmo de Prim. Un ejemplo

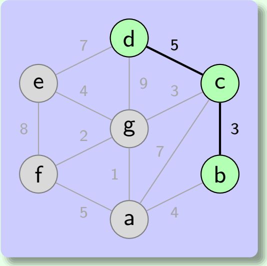




visitados	candidatos
$\{c,d\}$	$\{a,b,e,f,g\}$

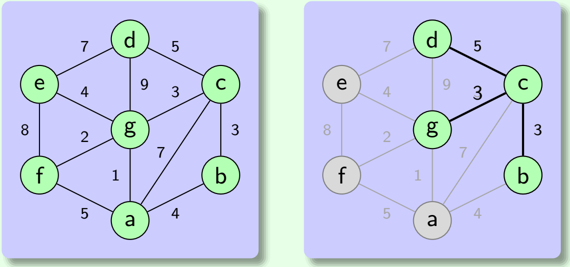
El algoritmo de Prim. Un ejemplo





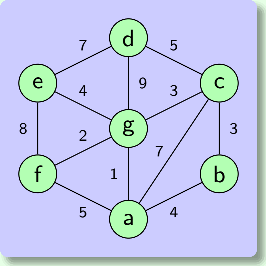
visitados	**candidatos**
$\{b,c,d\}$	$\{a,e,f,g\}$

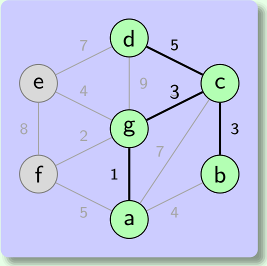
El algoritmo de Prim. Un ejemplo



visitados	**candidatos**
$\{b,c,d,g\}$	$\{a,e,f\}$

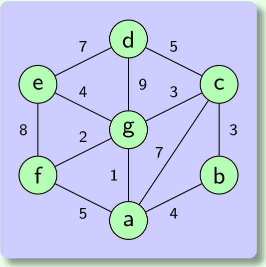
El algoritmo de Prim. Un ejemplo

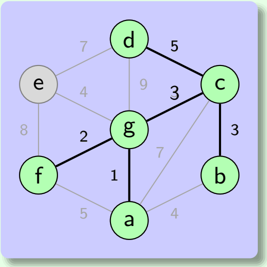




visitados	candidatos
$\{a,b,c,d,g\}$	$\{e,f\}$

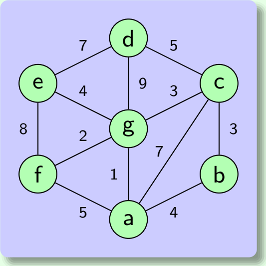
El algoritmo de Prim. Un ejemplo

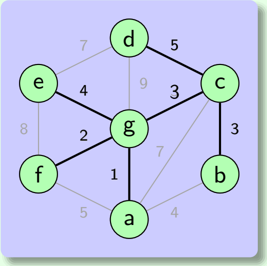




visitados	**candidatos**
$\{a, b, c, d, f, g\}$	$\{e\}$

El algoritmo de Prim. Un ejemplo





visitados	candidatos
$\{a,b,c,d,e,f,g\}$	$\{\ \}$

El algoritmo de Prim: algunas consideraciones

El algoritmo de Prim: algunas consideraciones

- El grafo se representa con una matriz de adyacencia L en la que $L[i, i] = -1$.

El algoritmo de Prim: algunas consideraciones

- El grafo se representa con una matriz de adyacencia L en la que $L[i, i] = -1$.
- Se utilizarán dos vectores. El valor $másCercano[i]$ da el vértice del árbol que está más cercano del vértice i. El valor $minDist[i]$ da la mínima distancia del vértice i al nodo $másCercano[i]$.

El algoritmo de Prim: algunas consideraciones

- El grafo se representa con una matriz de adyacencia L en la que $L[i, i] = -1$.
- Se utilizarán dos vectores. El valor $másCercano[i]$ da el vértice del árbol que está más cercano del vértice i. El valor $minDist[i]$ da la mínima distancia del vértice i al nodo $másCercano[i]$.
- Si un vértice i es incluido en el árbol, $minDist[i] = -1$.

El algoritmo de Prim: algunas consideraciones

- El grafo se representa con una matriz de adyacencia L en la que $L[i,i] = -1$.
- Se utilizarán dos vectores. El valor $másCercano[i]$ da el vértice del árbol que está más cercano del vértice i. El valor $minDist[i]$ da la mínima distancia del vértice i al nodo $másCercano[i]$.
- Si un vértice i es incluido en el árbol, $minDist[i] = -1$.
- El árbol se inicializa arbitrariamente al vértice 1. Los valores $másCercano[1]$ y $minDist[1]$ nunca se usan.

El algoritmo de Prim: algunas consideraciones

- El grafo se representa con una matriz de adyacencia L en la que $L[i, i] = -1$.
- Se utilizarán dos vectores. El valor $más Cercano[i]$ da el vértice del árbol que está más cercano del vértice i. El valor $minDist[i]$ da la mínima distancia del vértice i al nodo $más Cercano[i]$.
- Si un vértice i es incluido en el árbol, $minDist[i] = -1$.
- El árbol se inicializa arbitrariamente al vértice 1. Los valores $más Cercano[1]$ y $minDist[1]$ nunca se usan.
- Asumimos para no complicar el pseudo-código que podemos almacenar pares ordenados en un conjunto. Si no habría que crear la correspondiente clase.

El algoritmo de Prim

```
Algoritmo Prim (int [n][n] L)
set T = ∅                                // T contendrá el árbol
for (i = 0; i < n; i++) {
    masCercano[i] = 0
    minDist[i] = L[0, i]
}
repeat n − 1 times {                     // el ciclo greedy
    min = ∞
    for (j = 0; j < n; j++) {            // Buscamos el vértice más próximo
        if (0 ≤ minDist[j] < min)
            min = minDist[j]
            k = j
        }
    }
    T = T ∪ {(masCercano[k], k)}
    minDist[k]) = −1                     // Se agrega k al árbol
    for (j = 0; j < n; j++) {            // Se recalculan las distancias
        if (L[j, k] < minDist[j]) {
            minDist[j] = L[j, k]
            masCercano[j] = k
        }
    }
}
return T
```

Complejidad del algoritmo de Prim

```
Algoritmo Prim (int [n][n] L)
set T = ∅
for (i = 1; i < n; i++) {
    masCercano[i] = 0
    minDist[i] = L[0, i]
}
repeat n − 1 times {
    min = ∞
    for (j = 1; j < n; j++) {
        if (0 ≤ minDist[j] < min)
            min = minDist[j]
            k ← j
        }
    }
    T = T ∪ {(masCercano[k], k)}
    minDist[k]) = −1
    for (j = 1; j < n; j++) {
        if (L[j, k] < minDist[j]) {
            minDist[j] = L[j, k]
            masCercano[j] = k
        }
    }
}
return T
```

Complejidad del algoritmo de Prim

$\Theta(n^2)$

$\Theta(n)$

```
Algoritmo Prim (int [n][n] L)
set T = ∅
for (i = 1; i < n; i++) {
    masCercano[i] = 0
    minDist[i] = L[0, i]
}
repeat n − 1 times {
    min = ∞
    for (j = 1; j < n; j++) {
        if (0 ≤ minDist[j] < min)
            min = minDist[j]
            k ← j
        }
    }
    T = T ∪ {(masCercano[k], k)}
    minDist[k]) = −1
    for (j = 1; j < n; j++) {
        if (L[j, k] < minDist[j]) {
            minDist[j] = L[j, k]
            masCercano[j] = k
        }
    }
}
return T
```

$\Theta(n)$

$\Theta(n)$

$\Theta(n)$

Algoritmo de Kruskal: estrategia

Algoritmo de Kruskal: estrategia

- El algoritmo de Kruskal se basa en comenzar con árboles triviales (de un vértice) e ir uniéndolos hasta formar el árbol de recubrimiento mínimo.

Algoritmo de Kruskal: estrategia

- El algoritmo de Kruskal se basa en comenzar con árboles triviales (de un vértice) e ir uniéndolos hasta formar el árbol de recubrimiento mínimo.
- Los candidatos son todas las aristas del grafo.

Algoritmo de Kruskal: estrategia

- El algoritmo de Kruskal se basa en comenzar con árboles triviales (de un vértice) e ir uniéndolos hasta formar el árbol de recubrimiento mínimo.
- Los candidatos son todas las aristas del grafo.
- La función de selección elige la arista de menos peso entre las disponibles.

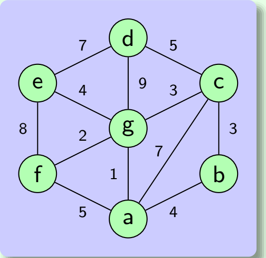
Algoritmo de Kruskal: estrategia

- El algoritmo de Kruskal se basa en comenzar con árboles triviales (de un vértice) e ir uniéndolos hasta formar el árbol de recubrimiento mínimo.
- Los candidatos son todas las aristas del grafo.
- La función de selección elige la arista de menos peso entre las disponibles.
- La función de factibilidad verifica que la arista seleccionada tenga sus vértices en diferentes árboles.

Algoritmo de Kruskal: estrategia

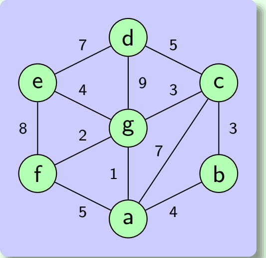
- El algoritmo de Kruskal se basa en comenzar con árboles triviales (de un vértice) e ir uniéndolos hasta formar el árbol de recubrimiento mínimo.
- Los candidatos son todas las aristas del grafo.
- La función de selección elige la arista de menos peso entre las disponibles.
- La función de factibilidad verifica que la arista seleccionada tenga sus vértices en diferentes árboles.
- La función solución verifica que haya quedado un único árbol.

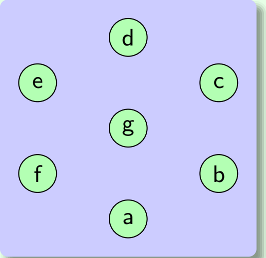
El algoritmo de Kruskal. Un ejemplo



El algoritmo de Kruskal. Un ejemplo

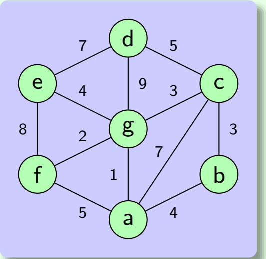
(a,g):1
(f,g):2
(b,c):3
(c,g):3
(a,b):4
(e,g):4
(a,f):5
(c,d):5
(a,c):7
(d,e):7
(e,f):8
(d,g):9

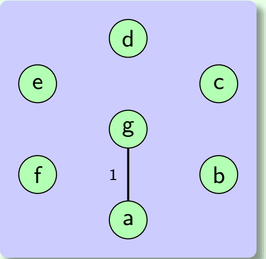




El algoritmo de Kruskal. Un ejemplo

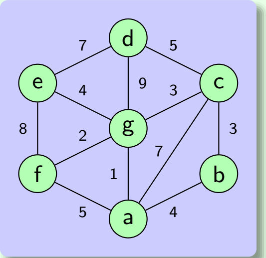
- ✓ (a,g):1
- (f,g):2
- (b,c):3
- (c,g):3
- (a,b):4
- (e,g):4
- (a,f):5
- (c,d):5
- (a,c):7
- (d,e):7
- (e,f):8
- (d,g):9

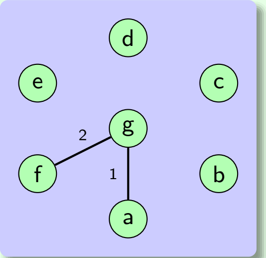




El algoritmo de Kruskal. Un ejemplo

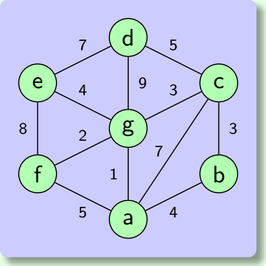
✓ (a,g):1
✓ (f,g):2
(b,c):3
(c,g):3
(a,b):4
(e,g):4
(a,f):5
(c,d):5
(a,c):7
(d,e):7
(e,f):8
(d,g):9

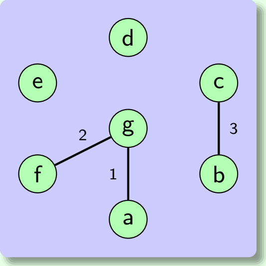




El algoritmo de Kruskal. Un ejemplo

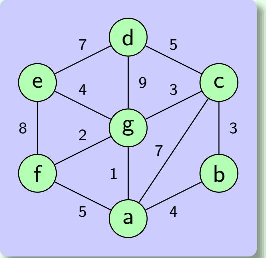
- ✓ (a,g):1
- ✓ (f,g):2
- ✓ (b,c):3
- (c,g):3
- (a,b):4
- (e,g):4
- (a,f):5
- (c,d):5
- (a,c):7
- (d,e):7
- (e,f):8
- (d,g):9

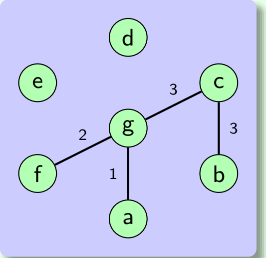




El algoritmo de Kruskal. Un ejemplo

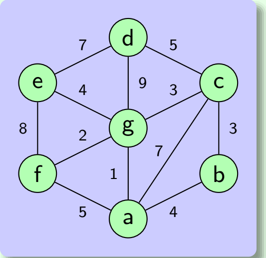
- ✓ (a,g):1
- ✓ (f,g):2
- ✓ (b,c):3
- ✓ (c,g):3
- (a,b):4
- (e,g):4
- (a,f):5
- (c,d):5
- (a,c):7
- (d,e):7
- (e,f):8
- (d,g):9

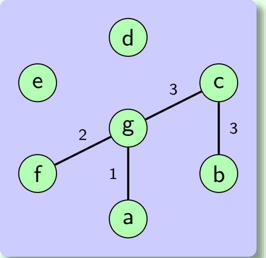




El algoritmo de Kruskal. Un ejemplo

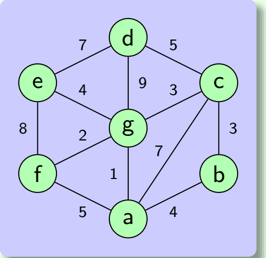
- ✓ (a,g):1
- ✓ (f,g):2
- ✓ (b,c):3
- ✓ (c,g):3
- ✗ (a,b):4
- (e,g):4
- (a,f):5
- (c,d):5
- (a,c):7
- (d,e):7
- (e,f):8
- (d,g):9

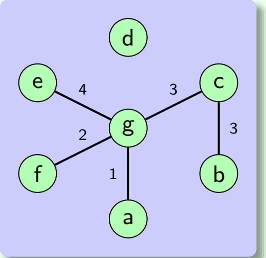




El algoritmo de Kruskal. Un ejemplo

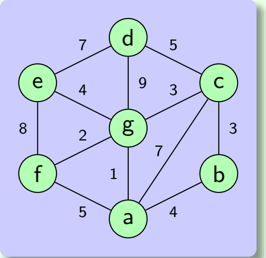
- ✓ (a,g):1
- ✓ (f,g):2
- ✓ (b,c):3
- ✓ (c,g):3
- ✗ (a,b):4
- ✓ (e,g):4
- (a,f):5
- (c,d):5
- (a,c):7
- (d,e):7
- (e,f):8
- (d,g):9

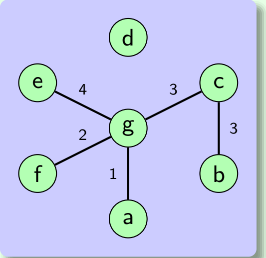




El algoritmo de Kruskal. Un ejemplo

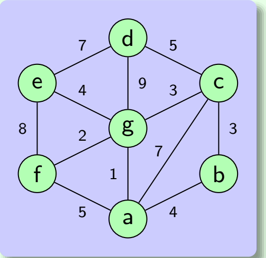
✓ (a,g):1
✓ (f,g):2
✓ (b,c):3
✓ (c,g):3
✗ (a,b):4
✓ (e,g):4
✗ (a,f):5
(c,d):5
(a,c):7
(d,e):7
(e,f):8
(d,g):9

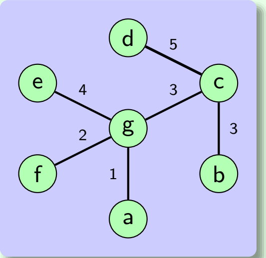




El algoritmo de Kruskal. Un ejemplo

- ✓ (a,g):1
- ✓ (f,g):2
- ✓ (b,c):3
- ✓ (c,g):3
- ✗ (a,b):4
- ✓ (e,g):4
- ✗ (a,f):5
- ✓ (c,d):5
- (a,c):7
- (d,e):7
- (e,f):8
- (d,g):9





El algoritmo de Kruskal: algunas consideraciones

El algoritmo de Kruskal: algunas consideraciones

- Los diferentes árboles se implementarán como conjuntos de vértices por un lado y guardando las aristas por el otro. Cada conjunto de vértices tendrá algún identificador.

El algoritmo de Kruskal: algunas consideraciones

- Los diferentes árboles se implementarán como conjuntos de vértices por un lado y guardando las aristas por el otro. Cada conjunto de vértices tendrá algún identificador.
- Asumiremos que tenemos implementadas las siguientes operaciones: *buscar*, que recibe un vértice y nos devuelve el identificador correspondiente y *combinar* que recibe dos identificadores y los combina.

El algoritmo de Kruskal: algunas consideraciones

- Los diferentes árboles se implementarán como conjuntos de vértices por un lado y guardando las aristas por el otro. Cada conjunto de vértices tendrá algún identificador.
- Asumiremos que tenemos implementadas las siguientes operaciones: *buscar*, que recibe un vértice y nos devuelve el identificador correspondiente y *combinar* que recibe dos identificadores y los combina.
- Por ejemplo, si tenemos tres conjuntos con un vértice cada uno, digamos $\{A\}$, $\{B\}$ y $\{C\}$, tendremos tres identificadores, digamos $x_1 : \{A\}$, $x_2 : \{B\}$, $x_3 : \{C\}$. La función $buscar(A)$ nos devolverá x_1.

El algoritmo de Kruskal: algunas consideraciones

- Los diferentes árboles se implementarán como conjuntos de vértices por un lado y guardando las aristas por el otro. Cada conjunto de vértices tendrá algún identificador.
- Asumiremos que tenemos implementadas las siguientes operaciones: *buscar*, que recibe un vértice y nos devuelve el identificador correspondiente y *combinar* que recibe dos identificadores y los combina.
- Por ejemplo, si tenemos tres conjuntos con un vértice cada uno, digamos $\{A\}$, $\{B\}$ y $\{C\}$, tendremos tres identificadores, digamos $x_1 : \{A\}$, $x_2 : \{B\}$, $x_3 : \{C\}$. La función $buscar(A)$ nos devolverá x_1.
- La función $combinar(x_1, x_3)$ aplicada al ejemplo anterior producirá la unión $\{A\} \cup \{C\} = \{A, C\}$, posiblemente asociada a un nuevo identificador o a alguno de los que ya existían, por ejemplo $x_1 : \{A, C\}$, $x_3 : \{A, C\}$ o $x_4 : \{A, C\}$.

El algoritmo de Kruskal: algunas consideraciones

- Los diferentes árboles se implementarán como conjuntos de vértices por un lado y guardando las aristas por el otro. Cada conjunto de vértices tendrá algún identificador.
- Asumiremos que tenemos implementadas las siguientes operaciones: *buscar*, que recibe un vértice y nos devuelve el identificador correspondiente y *combinar* que recibe dos identificadores y los combina.
- Por ejemplo, si tenemos tres conjuntos con un vértice cada uno, digamos $\{A\}$, $\{B\}$ y $\{C\}$, tendremos tres identificadores, digamos $x_1 : \{A\}$, $x_2 : \{B\}$, $x_3 : \{C\}$. La función $buscar(A)$ nos devolverá x_1.
- La función $combinar(x_1, x_3)$ aplicada al ejemplo anterior producirá la unión $\{A\} \cup \{C\} = \{A, C\}$, posiblemente asociada a un nuevo identificador o a alguno de los que ya existían, por ejemplo $x_1 : \{A, C\}$, $x_3 : \{A, C\}$ o $x_4 : \{A, C\}$.
- Volvemos a tomarnos algunas libertades con los tipos abstractos de datos de Programación II: por ejemplo, cargamos una cola de prioridad "en masa."

El algoritmo de Kruskal

```
Algoritmo Kruskal
    input: grafo G = (V, E)
    output T: Conjunto                    Devuelve el conjunto de aristas del árbol
    T.InicializarConjunto()
    Q.InicializarCola()                   // Cola de prioridad para las aristas
    Q = E                                 // Las claves son los pesos; O(|E|e log |E|)
    n = V.length
    foreach v ∈ V {
        inicializar un Conjunto {v}       // Los árboles de un elemento
    }
    repeat until (T.length = n − 1)       // El ciclo greedy
        (u, v) = Q.Primero()              // El candidato de menor costo
        uset = buscar(u)                  // ID de u
        vset = buscar(v)                  // ID de v
        if (uset ≠ vset)                  // La arista une árboles separados
            combinar(uset, vset)
            T = T ∪ {(u, v)}
        }
    }
    return T
```

Complejidad del algoritmo de Kruskal

```
Algoritmo Kruskal
    input: grafo G = (V, E)
    output T: Conjunto
    T.InicializarConjunto()
    Q.InicializarCola()
    Q = E
    n = V.length
    foreach v ∈ V {
        inicializar un Conjunto {v}
    }
    repeat until (T.length = n − 1)
        (u, v) = Q.Primero()
        uset = buscar(u)
        vset = buscar(v)
        if (uset ≠ vset)
            combinar(uset, vset)
            T = T ∪ {(u, v)}
        }
    }
    return T
```

Complejidad del algoritmo de Kruskal

```
Algoritmo Kruskal
    input: grafo G = (V, E)
    output T: Conjunto
    T.InicializarConjunto()
    Q.InicializarCola()
    Q = E                                   // Θ(|E| log |E|)
    n = V.length
    foreach v ∈ V {                         // Θ(n)
        inicializar un Conjunto {v}
    }
    repeat until (T.length = n − 1)         // Θ(n)
        (u, v) = Q.Primero()
        uset = buscar(u)
        vset = buscar(v)
        if (uset ≠ vset)
            combinar(uset, vset)
            T = T ∪ {(u, v)}
        }
    }
    return T
```

Complejidad del algoritmo de Kruskal

```
                   Algoritmo Kruskal
                       input: grafo G = (V, E)
                       output T: Conjunto
                       T.InicializarConjunto()
                       Q.InicializarCola()
Θ(|E| log |E|)         Q = E
                       n = V.length
Θ(n)                   foreach v ∈ V {
                           inicializar un Conjunto {v}
                       }
Θ(n)                   repeat until (T.length = n − 1)
                           (u, v) = Q.Primero()
                           uset = buscar(u)
                           vset = buscar(v)
                           if (uset ≠ vset)
                               combinar(uset, vset)
                               T = T ∪ {(u, v)}
                           }
                       }
                       return T
```

$$\Theta(e) = \left\{ \begin{array}{lcr} \Theta(|V|^2)(\text{conexión fuerte}) & \Rightarrow & \Theta(\log|E|) = 2\Theta(\log|V|) = \Theta(\log|V|) \\ \Theta(|V|)(\text{árboles}) & \Rightarrow & \Theta(\log|E|) = \Theta(\log n) \end{array} \right\} \Theta(|E| \log|V|)$$

Complejidad del algoritmo de Kruskal

$\Theta(|E|\log|V|)$

```
Algoritmo Kruskal
    input: grafo G = (V, E)
    output T: Conjunto
    T.InicializarConjunto()
    Q.InicializarCola()
    Q = E
    n = V.length
    foreach v ∈ V {
        inicializar un Conjunto {v}
    }
    repeat until (T.length = n − 1)
        (u, v) = Q.Primero()
        uset = buscar(u)
        vset = buscar(v)
        if (uset ≠ vset)
            combinar(uset, vset)
            T = T ∪ {(u, v)}
        }
    }
    return T
```

$\Theta(|E|\log|E|)$ — `Q = E`

$\Theta(n)$ — `foreach v ∈ V { inicializar un Conjunto {v} }`

$\Theta(n)$ — `repeat until (T.length = n − 1) … }`

$$\Theta(e) = \left\{ \begin{array}{lcr} \Theta(|V|^2)(\text{conexión fuerte}) & \Rightarrow & \Theta(\log|E|) = 2\Theta(\log|V|) = \Theta(\log|V|) \\ \Theta(|V|)(\text{árboles}) & \Rightarrow & \Theta(\log|E|) = \Theta(\log n) \end{array} \right\} \Theta(|E|\log|V|)$$

Prim vs. Kruskal

Prim vs. Kruskal

- Tenemos una complejidad $\mathcal{O}(n^2)$ para Prim y una complejidad $\mathcal{O}(a \log n)$ para Kruskal, donde a es el número de aristas.

Prim vs. Kruskal

- Tenemos una complejidad $\mathcal{O}(n^2)$ para Prim y una complejidad $\mathcal{O}(a \log n)$ para Kruskal, donde a es el número de aristas.
- Un grafo conectado oscila entre $n-1$ aristas (si es un árbol) y $n(n-1)/2$ aristas (si se trata de un grafo fuertemente conectado).

Prim vs. Kruskal

- Tenemos una complejidad $\mathcal{O}(n^2)$ para Prim y una complejidad $\mathcal{O}(a \log n)$ para Kruskal, donde a es el número de aristas.
- Un grafo conectado oscila entre $n - 1$ aristas (si es un árbol) y $n(n-1)/2$ aristas (si se trata de un grafo fuertemente conectado).
- Tenemos entonces dos casos límite para Kruskal: $\mathcal{O}(n \log n)$ (baja conectividad) y $\mathcal{O}(n^2 \log n)$ (alta conectividad).

Prim vs. Kruskal

- Tenemos una complejidad $\mathcal{O}(n^2)$ para Prim y una complejidad $\mathcal{O}(a \log n)$ para Kruskal, donde a es el número de aristas.
- Un grafo conectado oscila entre $n-1$ aristas (si es un árbol) y $n(n-1)/2$ aristas (si se trata de un grafo fuertemente conectado).
- Tenemos entonces dos casos límite para Kruskal: $\mathcal{O}(n \log n)$ (baja conectividad) y $\mathcal{O}(n^2 \log n)$ (alta conectividad).
- Dependiendo del tipo de grafo, uno de los dos algoritmos será entonces más adecuado.

Ejercicios propuestos 1

1. **Navegación en el río Ctalamochita**. Usted planea navegar en canoa aguas abajo por el río Ctalamochita entre las ciudades de Morrison y Monte Leña. Hay n puestos de canoas a lo largo de este trayecto. Antes de comenzar su excursión, usted consigue para cada $1 \leq i < j \leq n$, el precio $f_{i,j}$ para alquilar una canoa desde el puesto i hasta el puesto j. Estos precios son arbitrarios. Por ejemplo, es posible que sea $f_{1,3} = 10$ y $f_{1,4} = 5$. Usted comienza en el puesto 1 y debe terminar en el puesto n (usando canoas alquiladas). El objetivo es minimizar el costo.

   Muestre cómo podría aplicarse el algoritmo de Dijkstra para resolver este problema.

2. Suponga que tiene un grafo no dirigido con pesos que pueden ser positivos o negativos. ¿Producen los algoritmos de Prim y Kruskal un árbol de recubrimiento mínimo para ese grafo?

Ejercicios propuestos 2

3. **La propiedad de ciclo**: Suponga que G es un grafo no dirigido con pesos. Suponga que el grafo tiene por lo menos un ciclo y elija alguno. Para ese ciclo, supongamos que hay una arista e cuyo peso es estrictamente mayor que el de todas las otras aristas del ciclo. Observe que una arista así no necesariamente existe, pero supondremos que en este caso sí. Muestre que e no aparece en ningún árbol de recubrimiento mínimo de G.
4. Pruebe que si un grafo no dirigido con pesos es tal que los pesos son todos diferentes, entonces su árbol de recubrimiento mínimo es único.

   Ayuda: la propiedad de corte dice que, si una arista es la de menor costo entre dos particiones de los vértices de un grafo, entonces esta arista debe estar en todos los árboles de recubrimieno mínimo del grafo.
5. Considere el problema de calcular un árbol de recubrimiento *máximo*, es decir, un árbol de recubrimiento que maximice la suma de los costos de las aristas. ¿Pueden los algoritmos de Prim y Kruskal ayudarnos a solucionar este problema (asumiendo por supuesto que elegimos la arista de cruce de costo máximo)?

Ejercicios propuestos 3

6. Queremos encontrar el costo mínimo entre dos vértices a y b de un grafo G y nos dan el árbol de recubrimiento mínimo de este grafo. Entonces tomamos este árbol derecubrimiento mínimo y seguimos el camino desde a hasta b sumando los costos de cada arista.

   ¿Es la respuesta que da este procedimiento correcta?

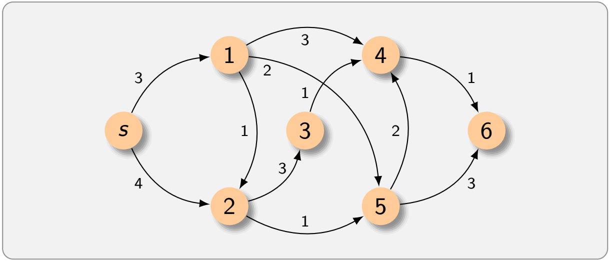
7. Usted trabaja para una compañía aérea que conecta varias ciudades de la provincia de Córdoba. Cada aeropuerto cobra una tarifa para su uso; esta tarifa debe pagarse cuando el avión llega y cuando parte. Usted calculó las conexiones de costo mínimo utilizando el algoritmo de Dijkstra. Sucede que ahora, ante la proximidad de las elecciones, los intendentes de las ciudades tienen una gran necesidad de dinero. Por lo tanto, proponen incrementar las tarifas de los aeropuertos un 20 %.

   La CEO de su compañía le dice que, dado que todas las tarifas se incrementan en el mismo porcentaje, es innecesario recalcular el algoritmo de Dijkstra; los trayectos de costo mínimo van a seguir siendo los mismos, según ella.

   ¿Está la CEO en lo cierto?

Ejercicios propuestos 4

6. Considere el grafo siguiente.



1. Calcule el camino más corto desde s a todos los otros vértices usando el algoritmo de Dijkstra. Determine el árbol de costos mínimos.
2. ¿Es el árbol de costos mínimos único?
3. Ahora cambie el peso de la arista $(3, 4)$ a -2. Muestre que el algoritmo de Dijkstra no funciona en este caso.