Árboles Generadores

Miguel Raggi

Redes
Escuela Nacional de Estudios Superiores
UNAM

11 de febrero de 2018

Índice:

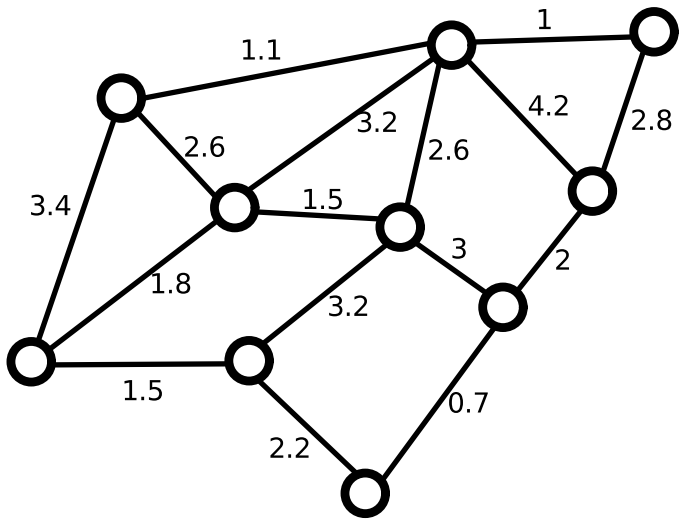
Índice:

Árbol Generador Mínimo

- Sea G una gráfica simple y conexa con costos en las aristas (positivos o negativos, no importa).

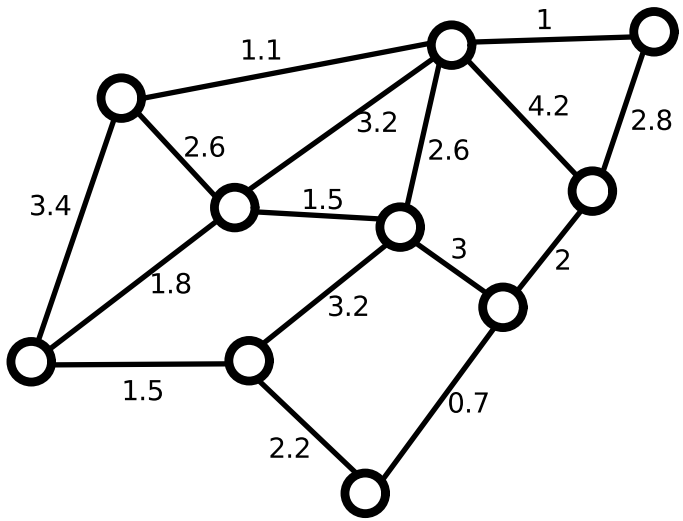
Árbol Generador Mínimo

- Sea G una gráfica simple y conexa con costos en las aristas (positivos o negativos, no importa).
- Queremos encontrar el árbol con menos costo.



Árbol Generador Mínimo

- Sea G una gráfica simple y conexa con costos en las aristas (positivos o negativos, no importa).
- Queremos encontrar el árbol con menos costo.



- Puedo sumar una constante a todos los pesos, o multiplicar por una constante positiva todos los pesos y no afecto el problema.

Aplicaciones

Aplicaciones:

- Conectar ciudades (circuitos, casas, etc) con calles (cables, tubos, etc) de manera que quede todo conectado.

Aplicaciones

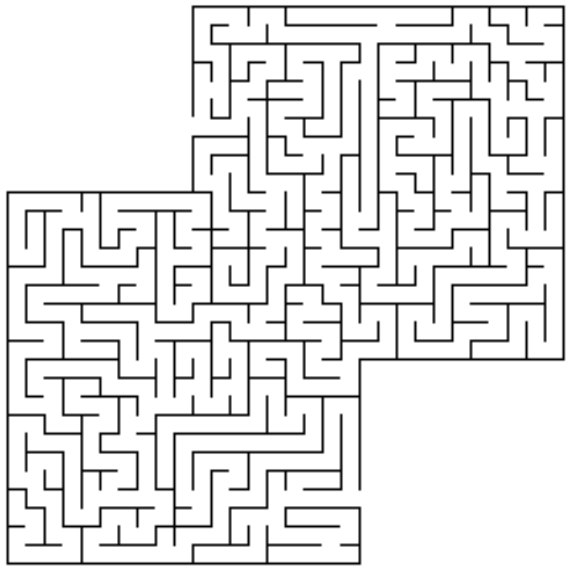
Aplicaciones:

- Conectar ciudades (circuitos, casas, etc) con calles (cables, tubos, etc) de manera que quede todo conectado.
- "k-Clustering": Dado un conjunto de puntos (en el espacio, por ejemplo), partir en k pedazos de manera que se minimice la suma de distancias entre objetos en el mismo pedazo (o más generalmente, con "propiedades" similares de alguna manera). Hay algoritmos que utilizan árboles generadores mínimos, pero no los veremos aquí.

Aplicaciones

Aplicaciones:

- Conectar ciudades (circuitos, casas, etc) con calles (cables, tubos, etc) de manera que quede todo conectado.
- "k-Clustering": Dado un conjunto de puntos (en el espacio, por ejemplo), partir en k pedazos de manera que se minimice la suma de distancias entre objetos en el mismo pedazo (o más generalmente, con "propiedades" similares de alguna manera). Hay algoritmos que utilizan árboles generadores mínimos, pero no los veremos aquí.
- ¡Generar un laberinto! (cuadrícula con pesos al azar)



Índice:

Algoritmo de Prim: Árbol Generador Mínimo

Aquí está el algoritmo:

Algoritmo de Prim: Árbol Generador Mínimo

Aquí está el algoritmo:

- Empieza en un vértice cualquiera escogido al azar y márcalo como "explorado".

Algoritmo de Prim: Árbol Generador Mínimo

Aquí está el algoritmo:

- Empieza en un vértice cualquiera escogido al azar y márcalo como "explorado".
- En cada paso del algoritmo, repite lo siguiente:

Algoritmo de Prim: Árbol Generador Mínimo

Aquí está el algoritmo:

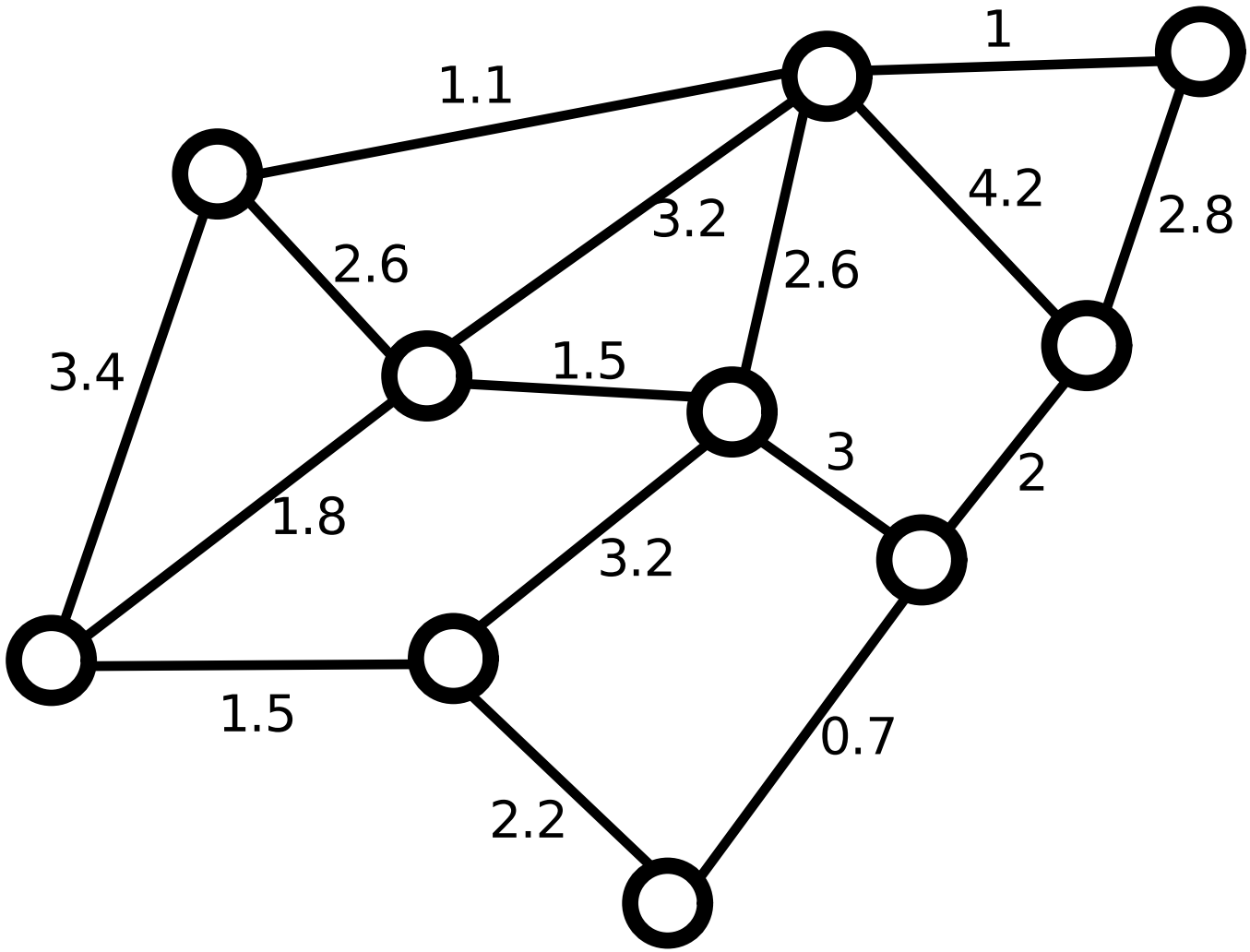
- Empieza en un vértice cualquiera escogido al azar y márcalo como "explorado".
- En cada paso del algoritmo, repite lo siguiente:
- Escoge la arista de menor costo que salga de alguno de los vértices explorados, pero que no vaya a un vértice explorado.

Algoritmo de Prim: Árbol Generador Mínimo

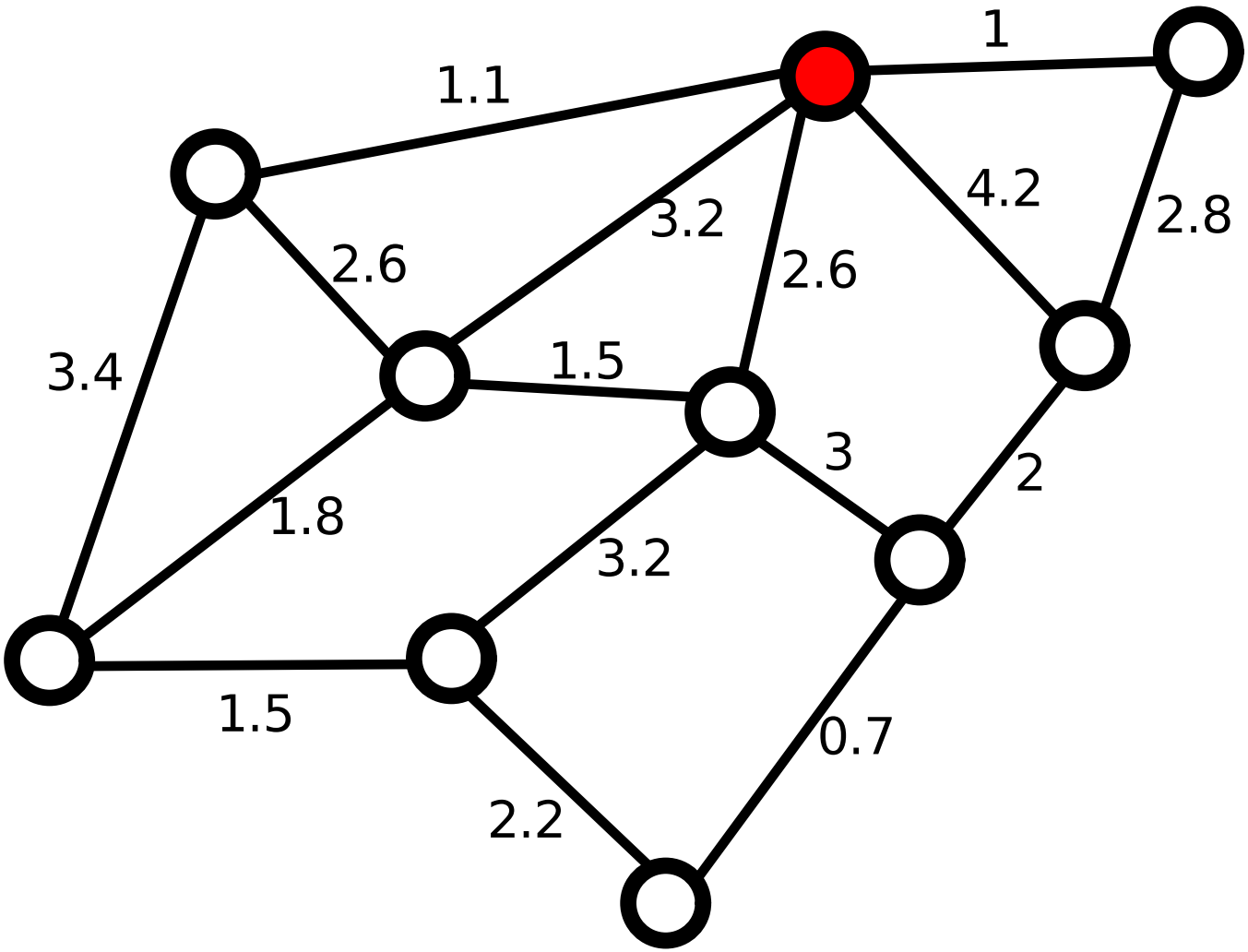
Aquí está el algoritmo:

- Empieza en un vértice cualquiera escogido al azar y márcalo como "explorado".
- En cada paso del algoritmo, repite lo siguiente:
- Escoge la arista de menor costo que salga de alguno de los vértices explorados, pero que no vaya a un vértice explorado.
- Repite, sólo fijándote en las aristas que no tienen del otro lado a un vértice "explorado".

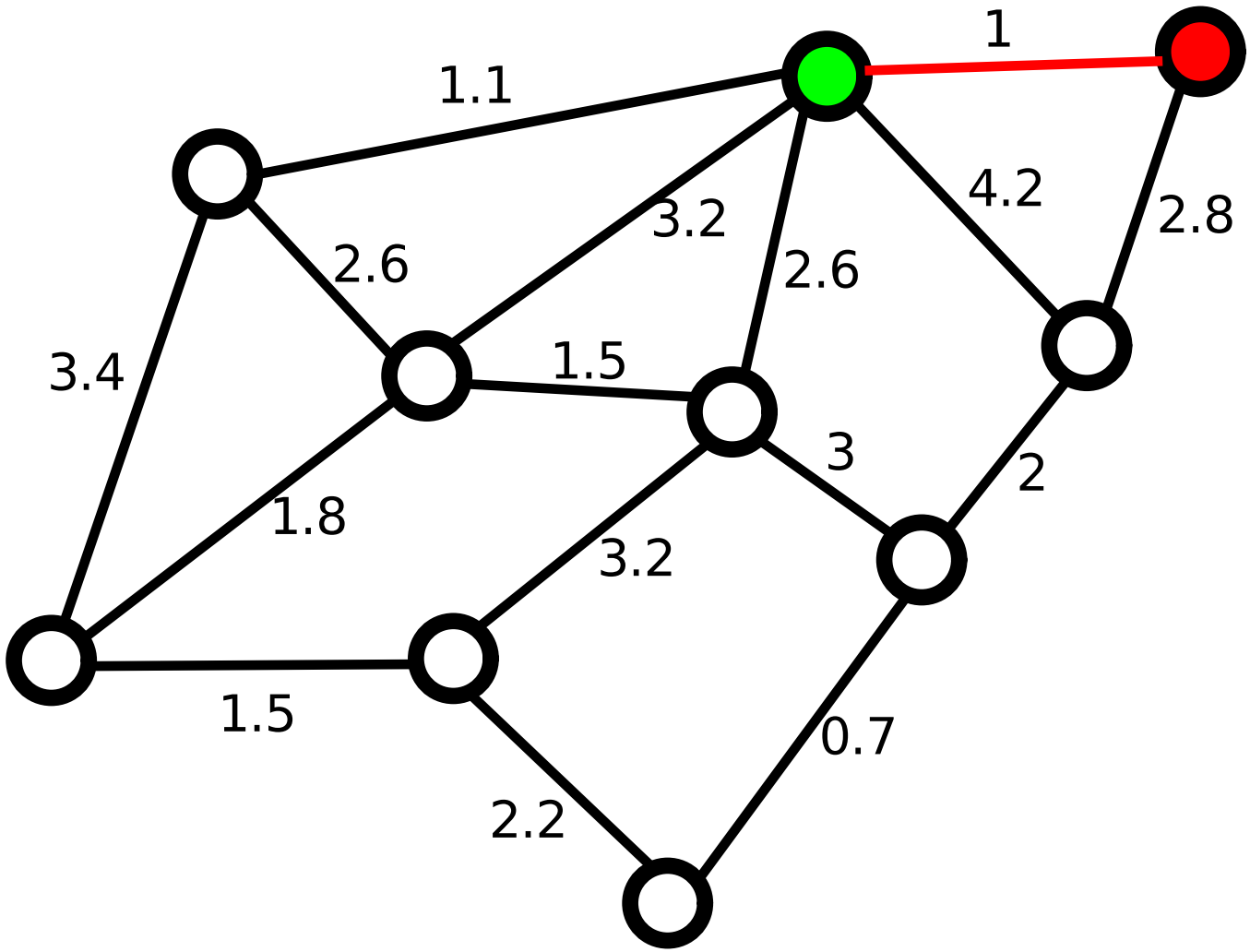
Algoritmo de Prim



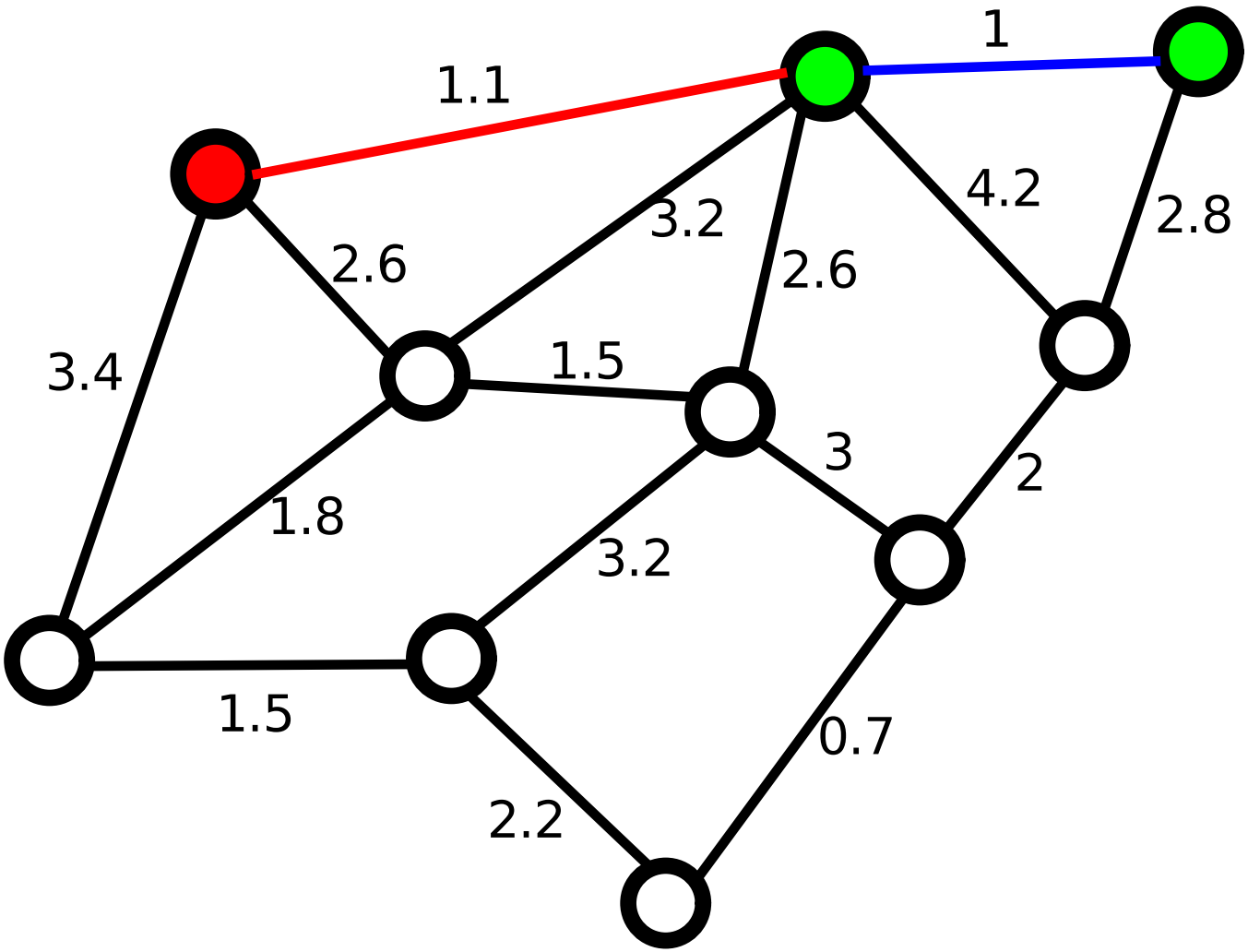
Algoritmo de Prim



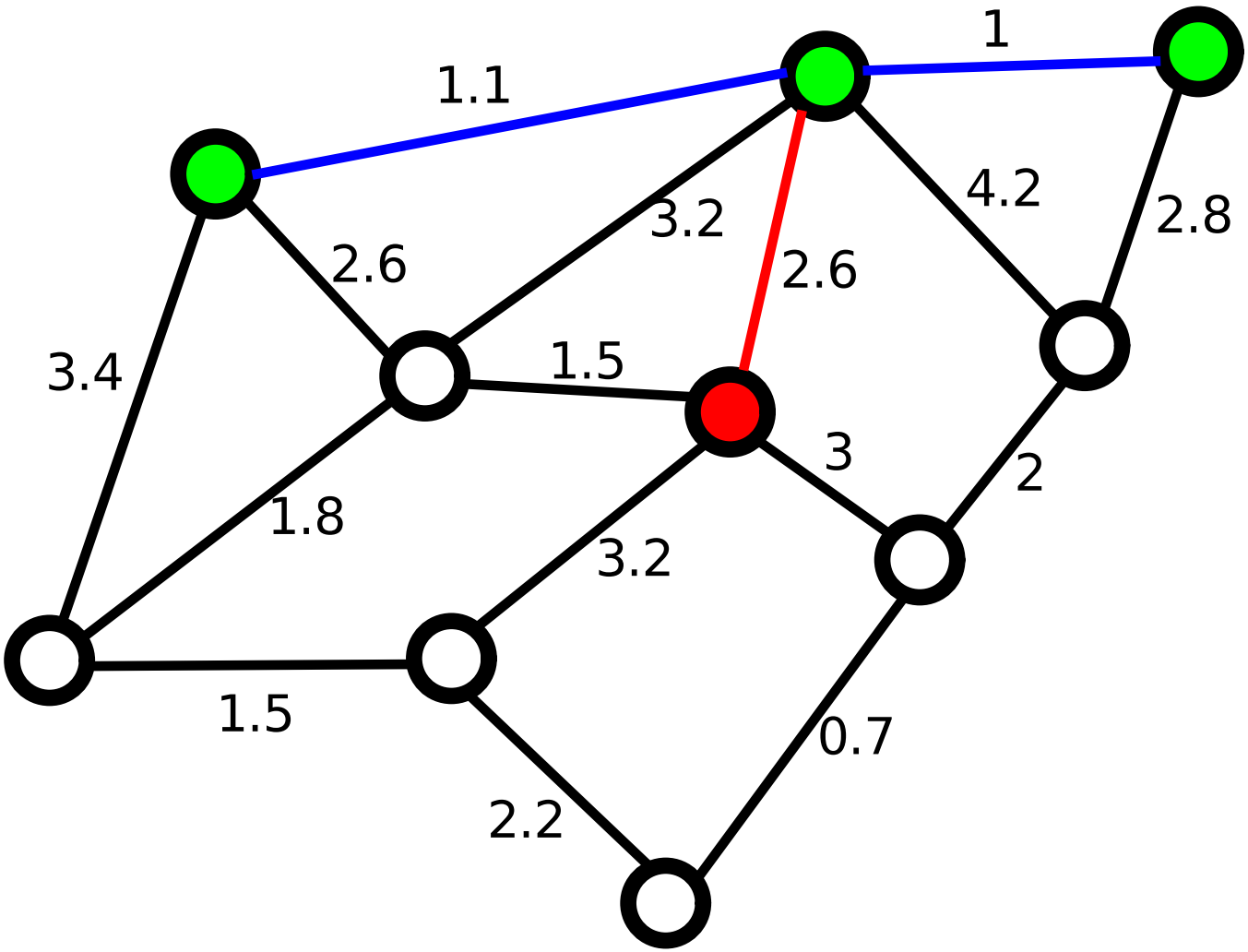
Algoritmo de Prim



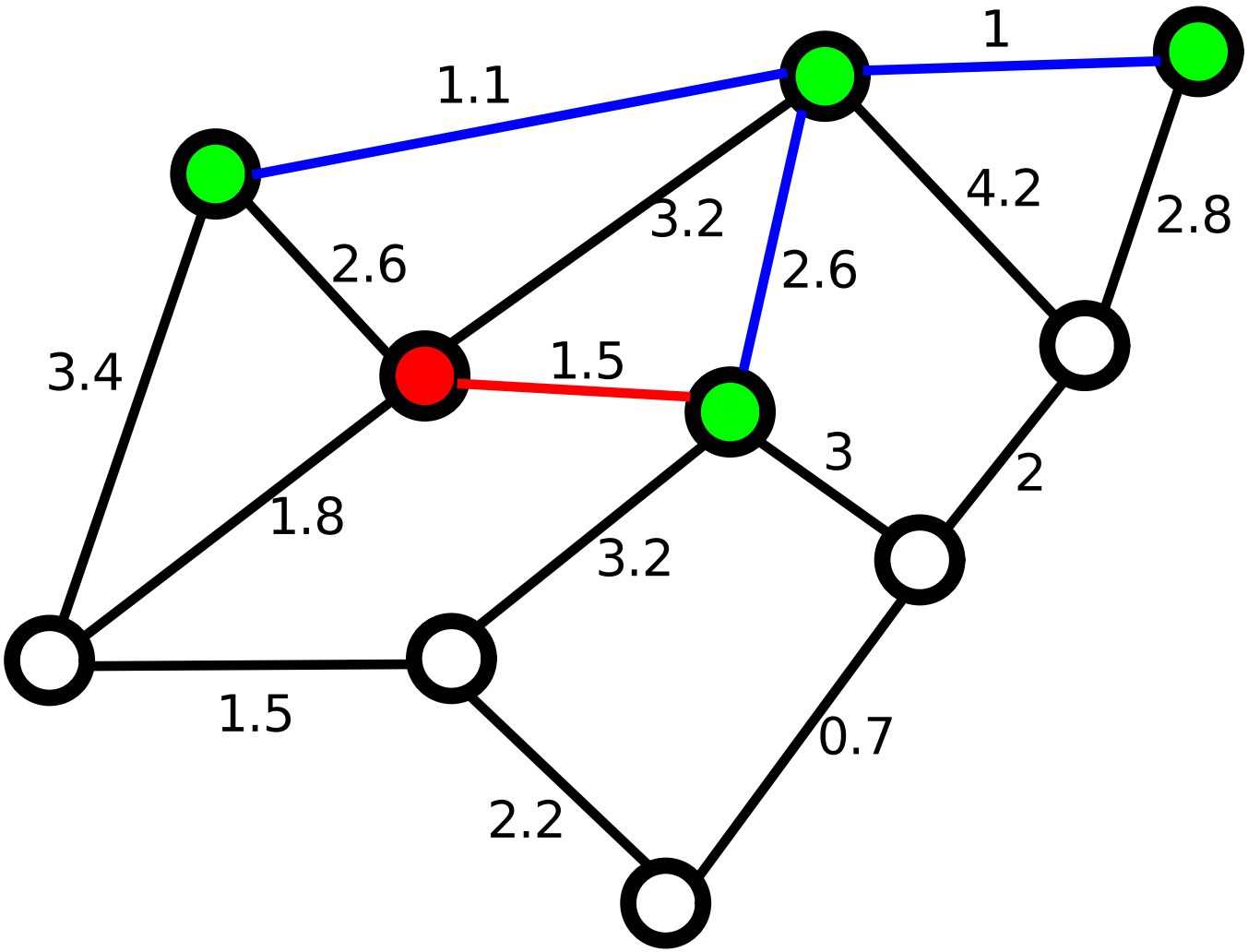
Algoritmo de Prim



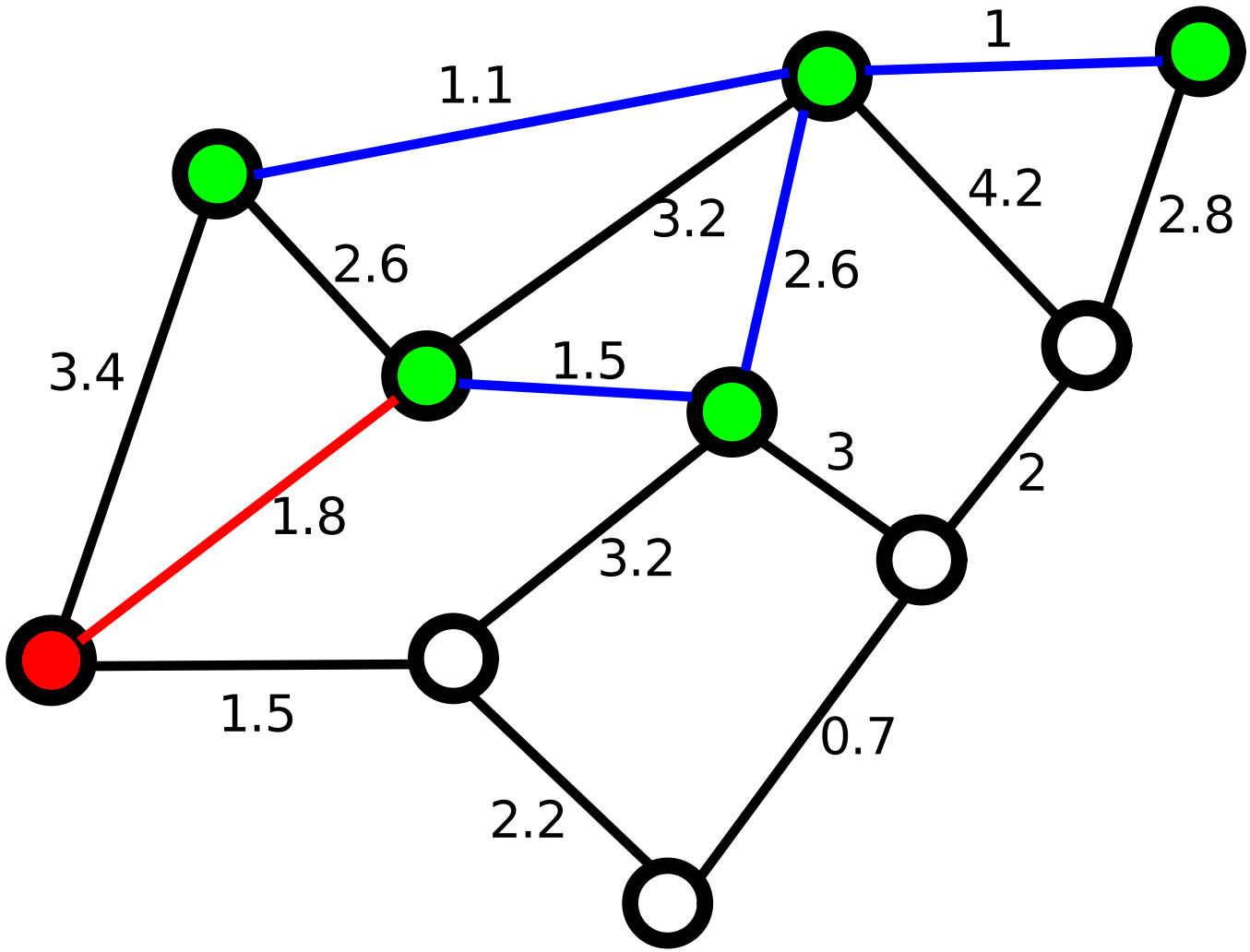
Algoritmo de Prim



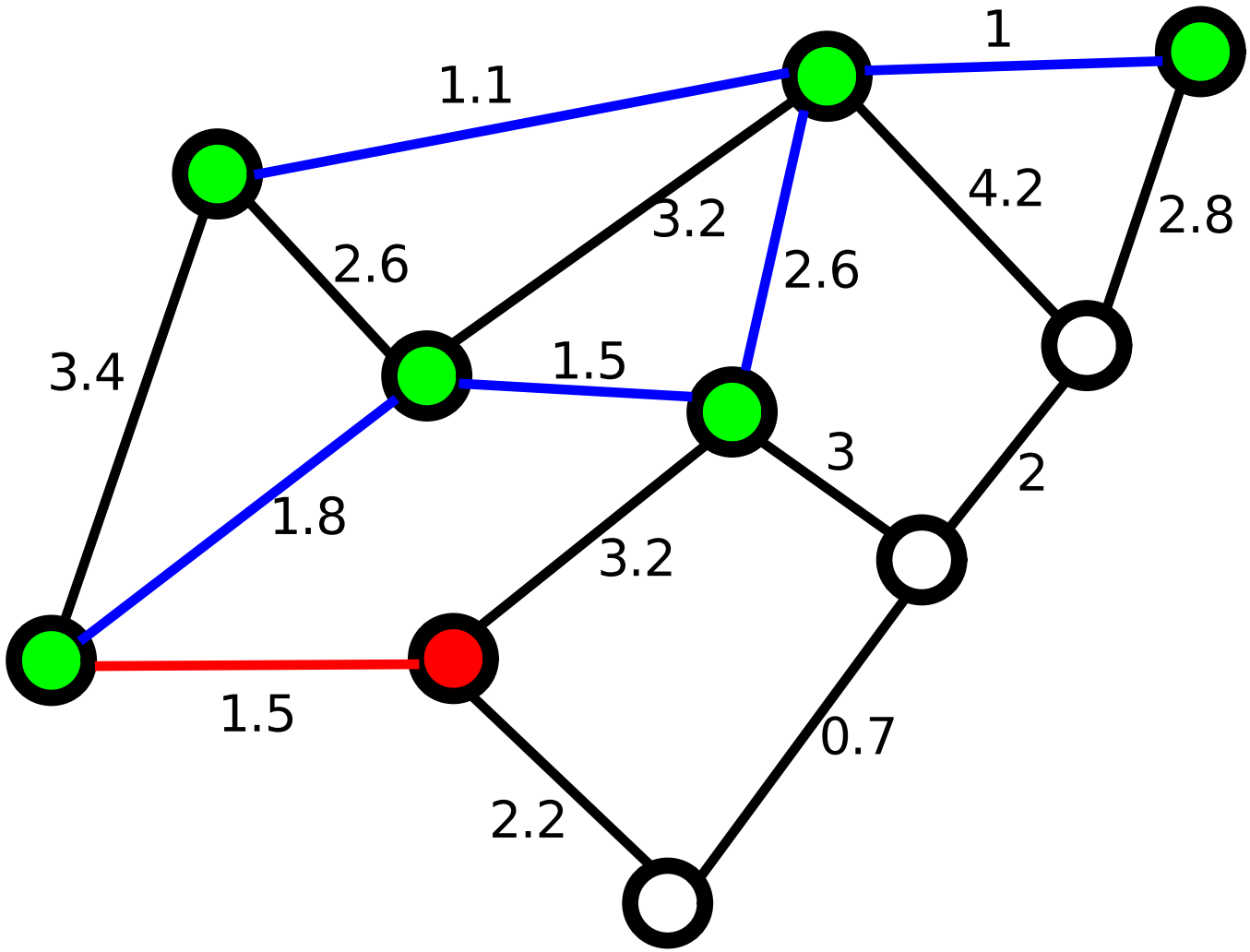
Algoritmo de Prim



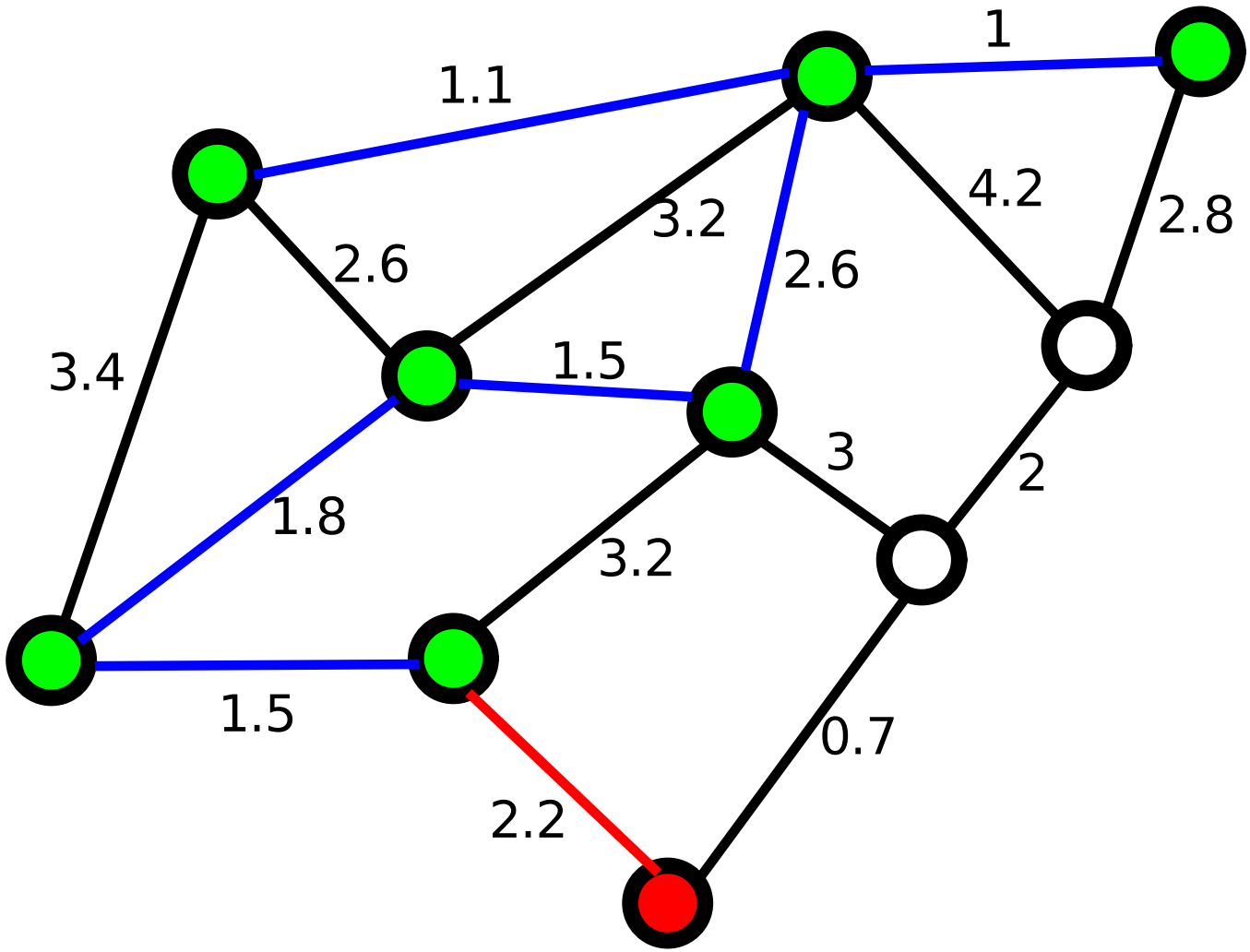
Algoritmo de Prim



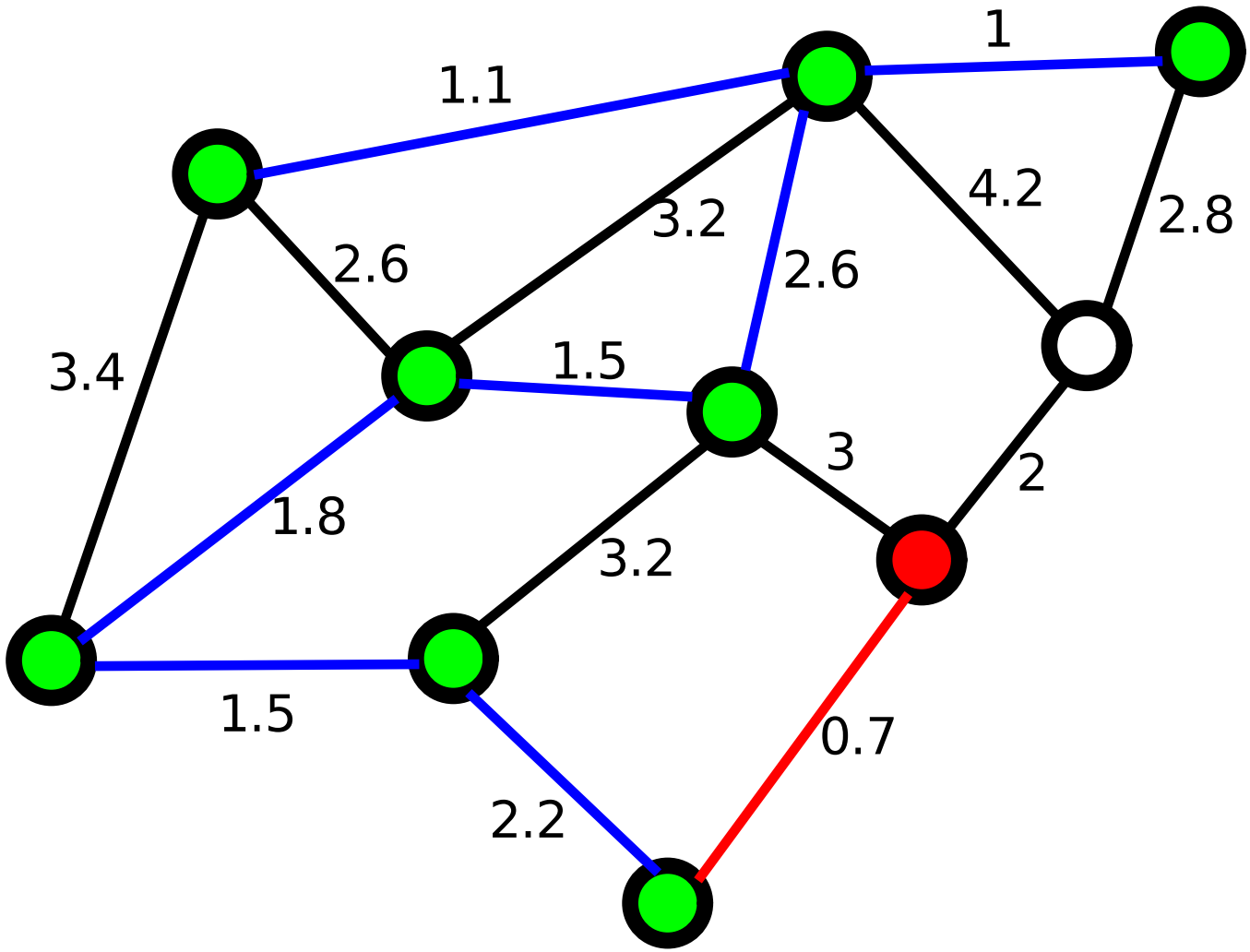
Algoritmo de Prim



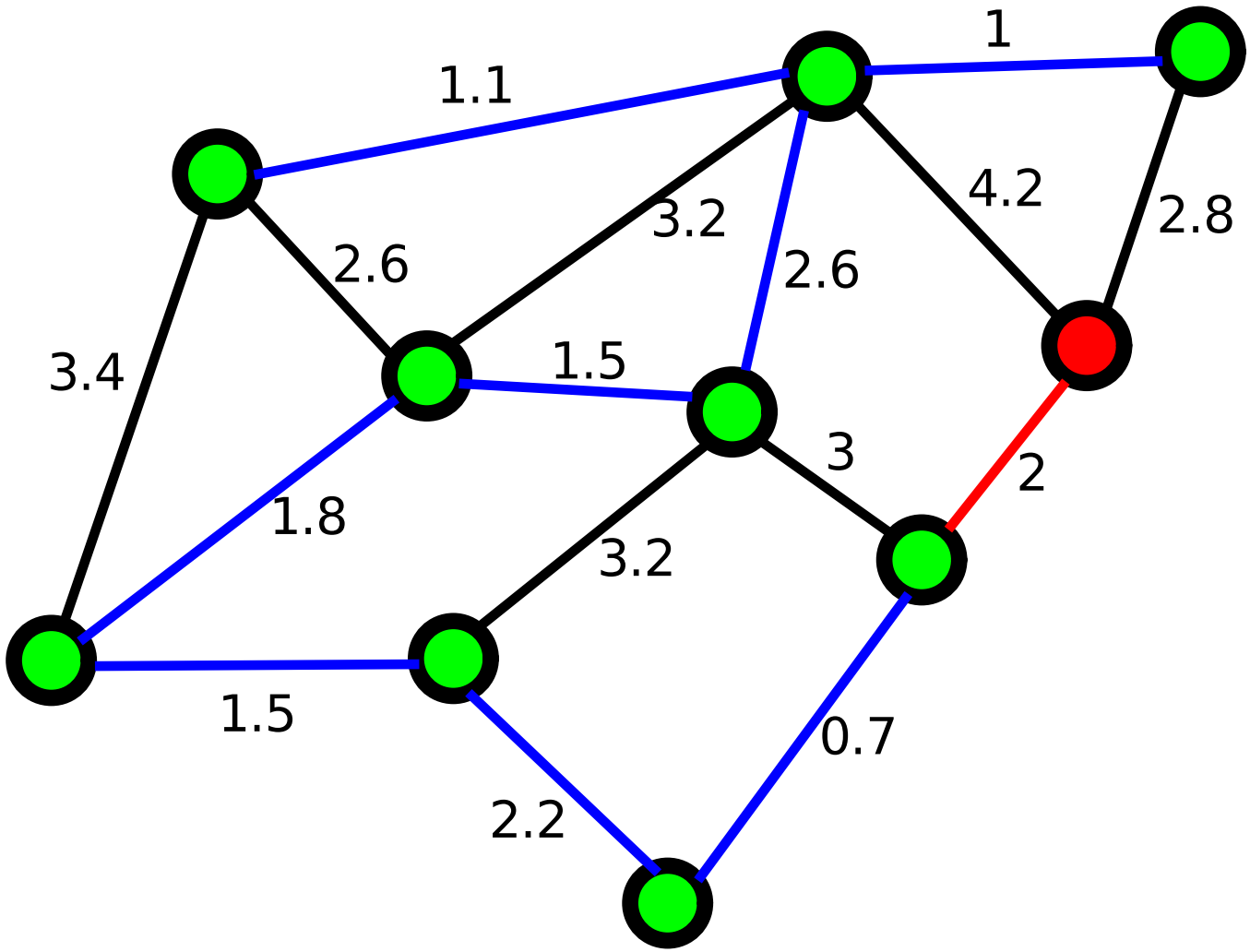
Algoritmo de Prim



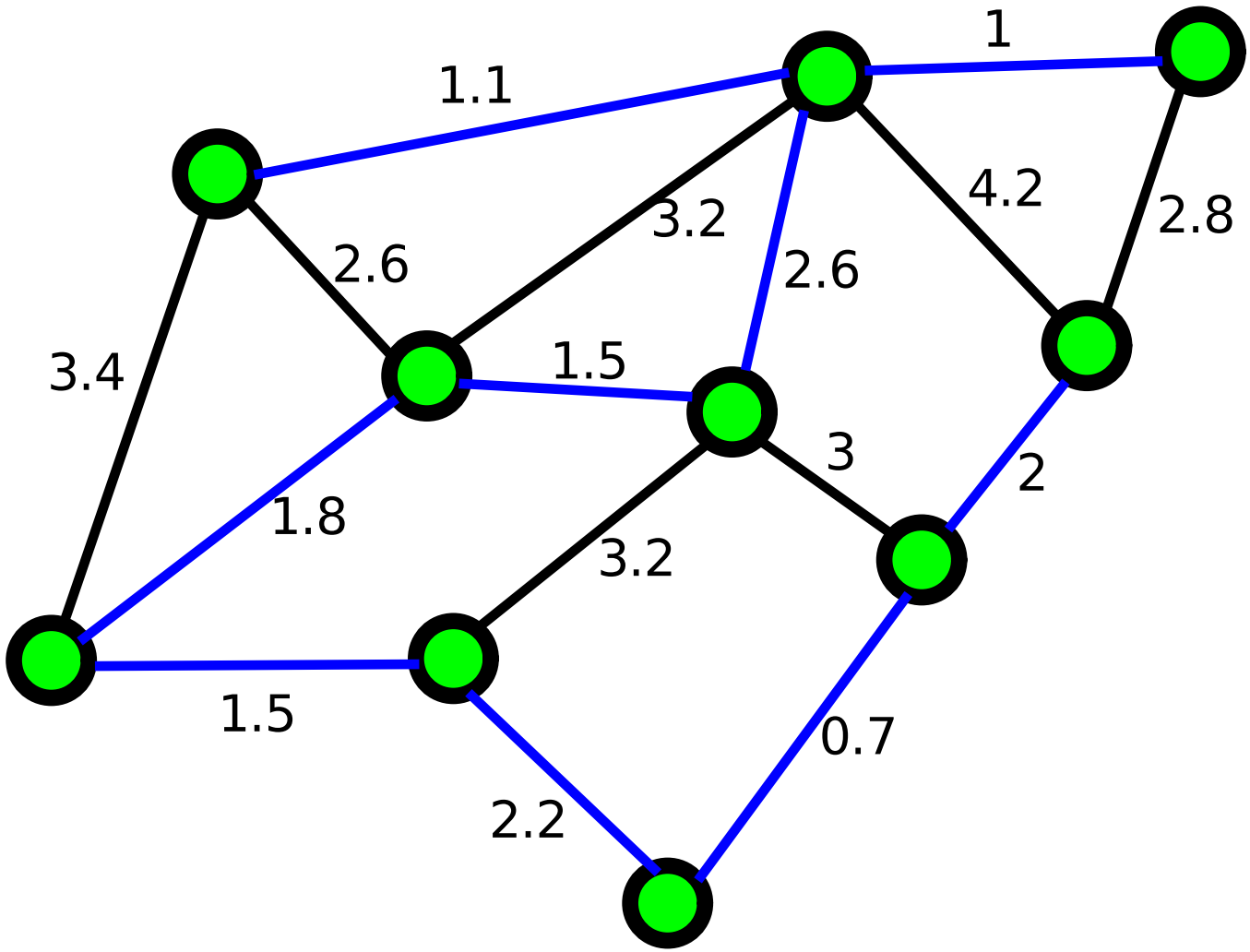
Algoritmo de Prim



Algoritmo de Prim



Algoritmo de Prim



Pseudo-código

VerticesExplorados $=$ {algún vértice al azar}
AristasPorExplorar $=$ aristas del vértice escogido antes
Arbol $= \emptyset$
while AristasPorExplorar $\neq \emptyset$:

- Selecciona arista s de menor peso de AristasPorExplorar y quítala.
- Sean u y v los dos vértices de s.
- if $v \notin$ VerticesExplorados
  - Añade s a Arbol y v a VerticesExplorados
  - Añade todas las demás aristas (v, w) a AristasPorExplorar, **si w no está en VerticesExplorados**.

return Arbol

¿Por qué funciona?

- Sea G una gráfica conexa con pesos y sea P el árbol generado por Prim.

¿Por qué funciona?

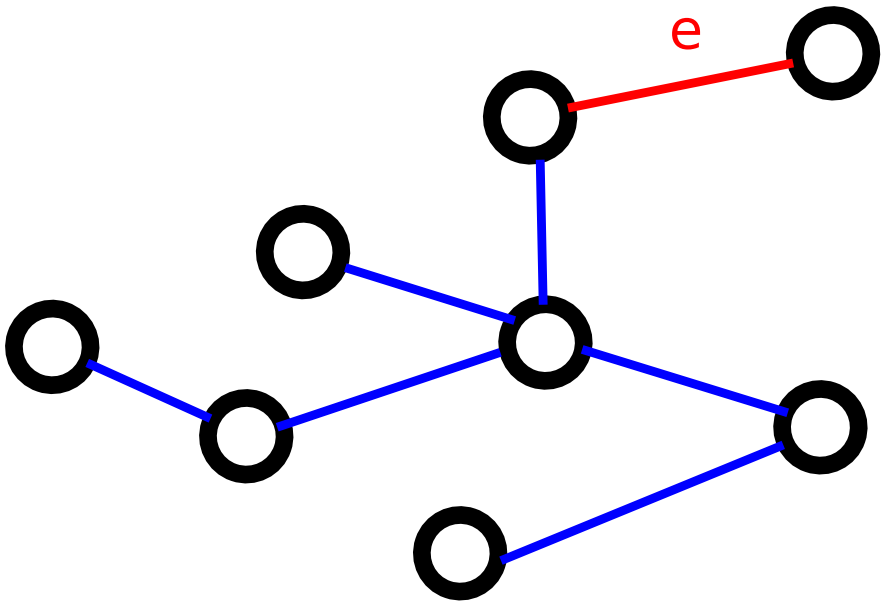
- Sea G una gráfica conexa con pesos y sea P el árbol generado por Prim.
- Sea A un árbol mínimo (luego le ponemos más condiciones).

¿Por qué funciona?

- Sea G una gráfica conexa con pesos y sea P el árbol generado por Prim.
- Sea A un árbol mínimo (luego le ponemos más condiciones).
- Si P fue generado por Prim, veamos el orden en que fuimos agregando aristas.

Prueba

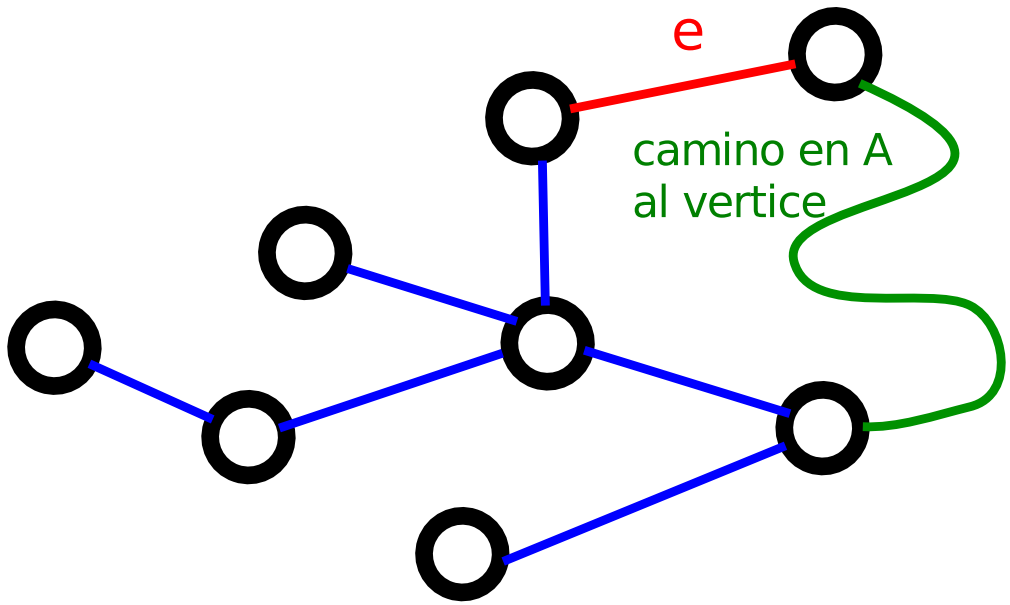
Sea e la primera arista que fue agregada por Prim que NO estaba en A.



e = primer nodo en P que no está en A

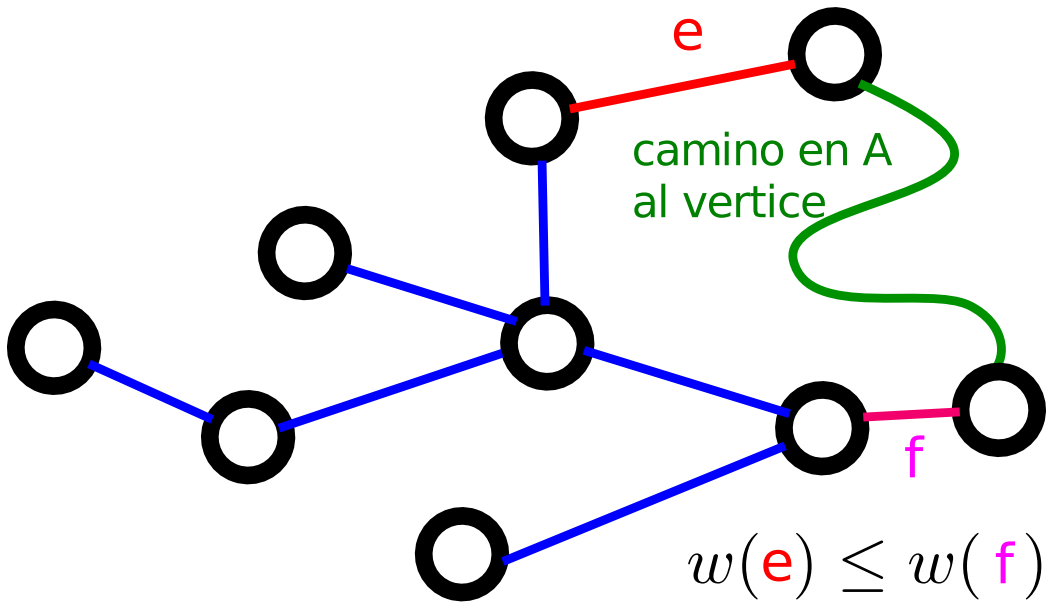
Prueba

Por ser A conexa, existe camino de los vértices que ya estaban al nuevo vértice:



Prueba

Sea f la primera arista del camino verde:



Prueba

Si agregamos e a A y quitamos f, construimos un árbol mínimo con más aristas en común con P que A.

Estructuras de datos

- ¿Qué estructura de datos utilizo para guardar las aristas “por explorar”? Debo:

Estructuras de datos

- ¿Qué estructura de datos utilizo para guardar las aristas “por explorar”? Debo:
  - Seleccionar la de peso más chiquito y quitarla.

Estructuras de datos

- ¿Qué estructura de datos utilizo para guardar las aristas "por explorar"? Debo:
  - Seleccionar la de peso más chiquito y quitarla.
  - Insertar varias nuevas aristas.

Estructuras de datos

- ¿Qué estructura de datos utilizo para guardar las aristas "por explorar"? Debo:
  - Seleccionar la de peso más chiquito y quitarla.
  - Insertar varias nuevas aristas.
- ¡Priority Queue!

Estructuras de datos

- ¿Qué estructura de datos utilizo para guardar las aristas "por explorar"? Debo:
  - Seleccionar la de peso más chiquito y quitarla.
  - Insertar varias nuevas aristas.
- ¡Priority Queue!
- ¿Qué estructura de datos utilizo para guardar los vértices? Las operaciones que haré son:

Estructuras de datos

- ¿Qué estructura de datos utilizo para guardar las aristas "por explorar"? Debo:
  - Seleccionar la de peso más chiquito y quitarla.
  - Insertar varias nuevas aristas.
- ¡Priority Queue!
- ¿Qué estructura de datos utilizo para guardar los vértices? Las operaciones que haré son:
  - Guardar nuevos vértices.

Estructuras de datos

- ¿Qué estructura de datos utilizo para guardar las aristas "por explorar"? Debo:
  - Seleccionar la de peso más chiquito y quitarla.
  - Insertar varias nuevas aristas.
- ¡Priority Queue!
- ¿Qué estructura de datos utilizo para guardar los vértices? Las operaciones que haré son:
  - Guardar nuevos vértices.
  - Revisar si uno ya fue guardado o no.

Estructuras de datos

- ¿Qué estructura de datos utilizo para guardar las aristas "por explorar"? Debo:
  - Seleccionar la de peso más chiquito y quitarla.
  - Insertar varias nuevas aristas.
- ¡Priority Queue!
- ¿Qué estructura de datos utilizo para guardar los vértices? Las operaciones que haré son:
  - Guardar nuevos vértices.
  - Revisar si uno ya fue guardado o no.
- Lo que más conviene es un arreglo L de 0's y 1's, en donde

$$L[n] = 1 \iff n \text{ ya fue explorado.}$$

Consideraciones

- Si quiero el árbol de máximo peso en vez del árbol del mínimo peso, multiplico por -1, o equivalentemente, siempre tomo el “mayor” en vez de el “menor” que salga de lo que llevo.

Consideraciones

- Si quiero el árbol de máximo peso en vez del árbol del mínimo peso, multiplico por -1, o equivalentemente, siempre tomo el "mayor" en vez de el "menor" que salga de lo que llevo.
- El algoritmo, usando heaps como describimos anteriormente, corre en tiempo $O((E+V)\log V) \approx O(E \log V)$.

Consideraciones

- Si quiero el árbol de máximo peso en vez del árbol del mínimo peso, multiplico por -1, o equivalentemente, siempre tomo el "mayor" en vez de el "menor" que salga de lo que llevo.
- El algoritmo, usando heaps como describimos anteriormente, corre en tiempo $O((E+V)\log V) \approx O(E \log V)$.
- Si en vez de heaps se utiliza algo llamado "Fibonacci heap" puede ser disminuido a $O(E + V \log V)$.

Consideraciones

- Si quiero el árbol de máximo peso en vez del árbol del mínimo peso, multiplico por -1, o equivalentemente, siempre tomo el "mayor" en vez de el "menor" que salga de lo que llevo.
- El algoritmo, usando heaps como describimos anteriormente, corre en tiempo $O((E+V)\log V) \approx O(E \log V)$.
- Si en vez de heaps se utiliza algo llamado "Fibonacci heap" puede ser disminuido a $O(E + V \log V)$.
- Hay otros algoritmos que funcionan, unos incluso teóricamente más rápidos, pero en la práctica este es el más utilizado.

A programar!

- MinSpanningTree.hpp
- `std::vector<Edge> minimum_spanning_tree(const Graph& G);`
- MinSpanningTree.cpp
- Considerar:

```
struct by_reverse_weight
{
    bool operator()(const Edge& a, const Edge& b)
    {
        return a.weight() > b.weight();
    }
};
```

- `std::priority_queue<Edge, std::vector<Edge>, by_reverse_weight>`

Índice:

Algoritmo de Kruskal

- Se parece a Prim y también es un algoritmo avaricioso:

Algoritmo de Kruskal

- Se parece a Prim y también es un algoritmo avaricioso:
  - Empieza ordenando el conjunto total de aristas por peso.

Algoritmo de Kruskal

- Se parece a Prim y también es un algoritmo avaricioso:
  - Empieza ordenando el conjunto total de aristas por peso.
  - En cada iteración, toma la arista de menor peso que no forme ciclo con las que tienes.

Algoritmo de Kruskal

- Se parece a Prim y también es un algoritmo avaricioso:
  - Empieza ordenando el conjunto total de aristas por peso.
  - En cada iteración, toma la arista de menor peso que no forme ciclo con las que tienes.
- Revisar si forma ciclo con las que tienes se puede implementar igual que Prim, con vértices explorados. Si ambos extremos de la arista estaban explorados, se formará un ciclo.

Algoritmo de Kruskal

- Se parece a Prim y también es un algoritmo avaricioso:
  - Empieza ordenando el conjunto total de aristas por peso.
  - En cada iteración, toma la arista de menor peso que no forme ciclo con las que tienes.
- Revisar si forma ciclo con las que tienes se puede implementar igual que Prim, con vértices explorados. Si ambos extremos de la arista estaban explorados, se formará un ciclo.
- El algoritmo de Kruskal es más rápido que el de Prim cuando la gráfica tiene pocas aristas, pero bastante más lento cuando tiene muchas.

Muestra!

Kruskal: ¿cómo sabemos si hay ciclo?

- Lo difícil de Kruskal es que para cada arista hay que revisar si podemos agregarla o no.
- Necesitamos una estructura de datos que haga sencilla (y eficiente) esta tarea:
-

Algoritmo Reversa-Borra

El algoritmo es el contrario del Kruskal.

- Empieza con todas las aristas.

Algoritmo Reversa-Borra

El algoritmo es el contrario del Kruskal.

- Empieza con todas las aristas.
- En cada iteración, borra la arista de mayor peso que no desconecte nada.

Algoritmo de Borůvka

- Empieza con todos los vértices en su propia componente conexa.

Algoritmo de Borůvka

- Empieza con todos los vértices en su propia componente conexa.
- En cada iteración, escoge una componente conexa y pégala a la que te cueste menos pegarla.

Algoritmo de Borůvka

- Empieza con todos los vértices en su propia componente conexa.
- En cada iteración, escoge una componente conexa y pégala a la que te cueste menos pegarla.

El algoritmo de Borůvka (y en especial una variante conocida como el algoritmo de Chazelle) es más rápido asintóticamente que los otros, pero generalmente no se utilizan mucho.