CS 211

Computers and Programming

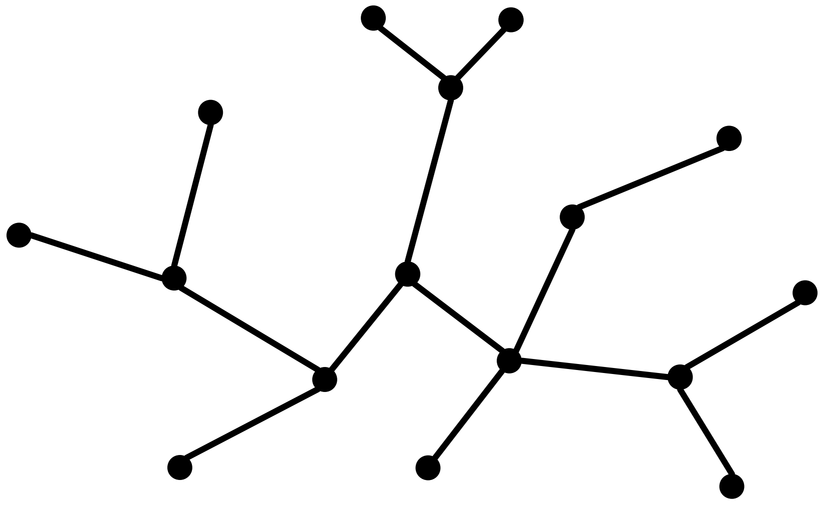
http://www.cs.cornell.edu/courses/cs211/2005su

Lecture 17: Spanning Trees for Graphs; Exceptions

Spanning Trees

Undirected Trees

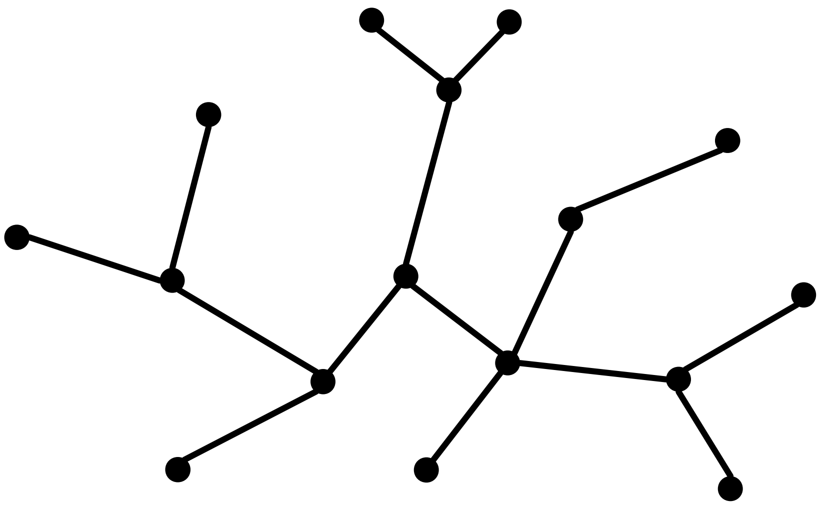
- An undirected graph is a *tree* if there is exactly one simple path between any pair of vertices



Facts About Trees

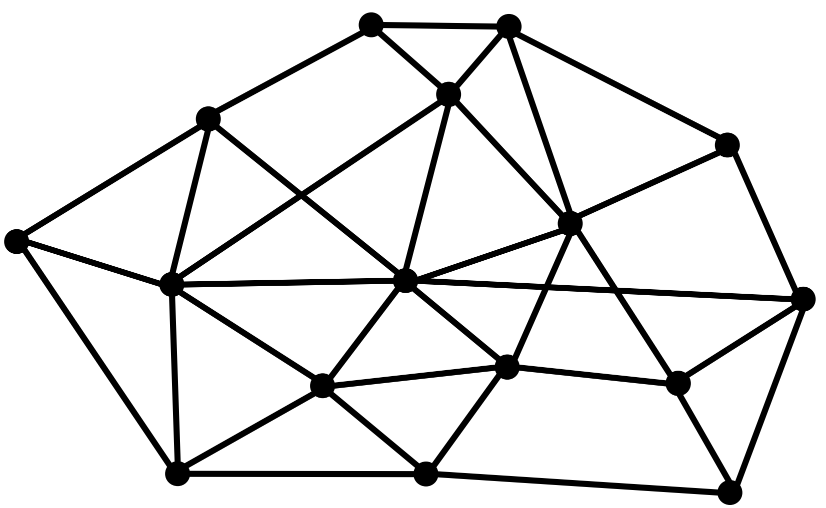
- $|E| = |V| - 1$
- connected
- no cycles

In fact, any two of these properties imply the third, and imply that the graph is a tree



Spanning Trees

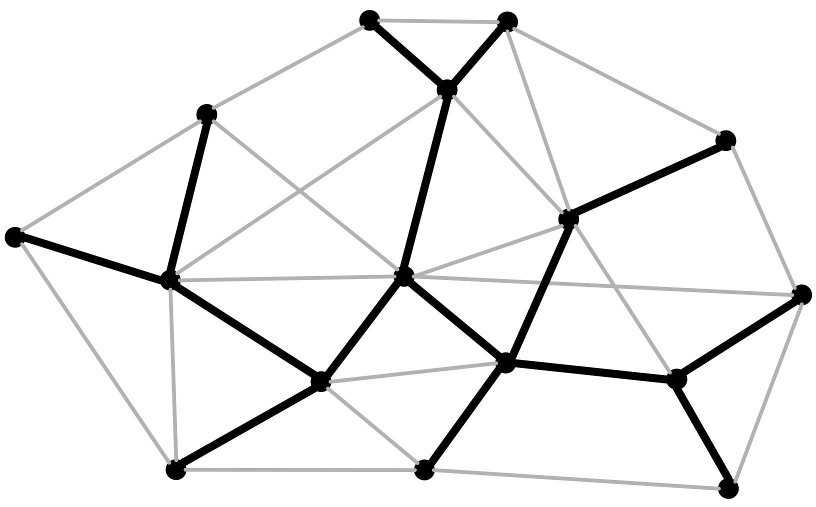
A *spanning tree* of a connected undirected graph (V,E) is a subgraph (V,E') that is a tree



Spanning Trees

A *spanning tree* of a connected undirected graph (V,E) is a subgraph (V,E') that is a tree

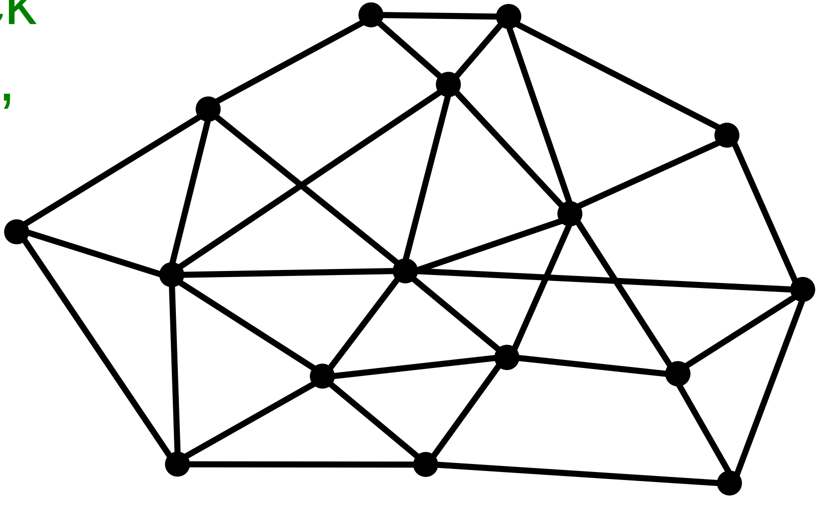
- Same set of vertices V
- $E' \subseteq E$
- (V,E') is a tree



Finding a Spanning Tree

A subtractive method

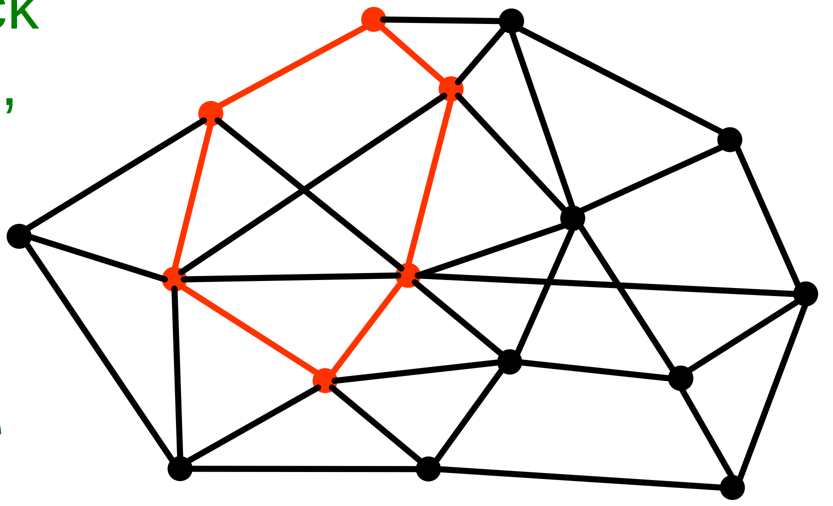
- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected (why?)
- Repeat until no more cycles



Finding a Spanning Tree

A subtractive method

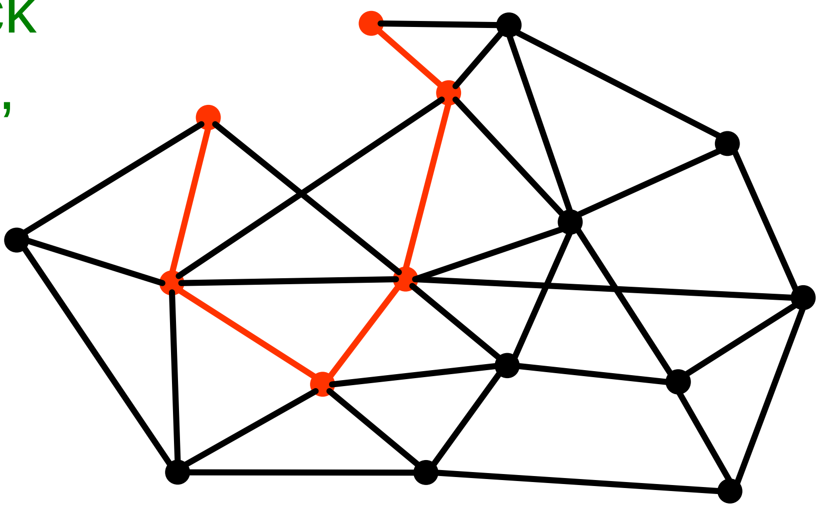
- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected (why?)
- Repeat until no more cycles



Finding a Spanning Tree

A subtractive method

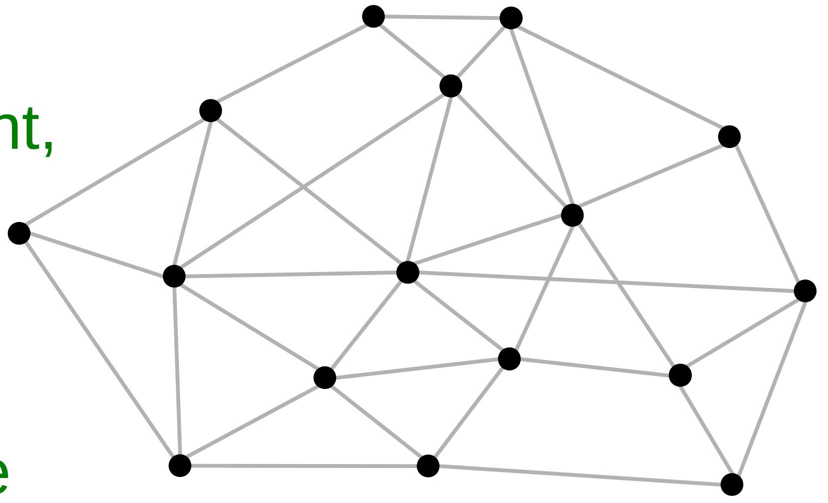
- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected (why?)
- Repeat until no more cycles



Finding a Spanning Tree

An additive method

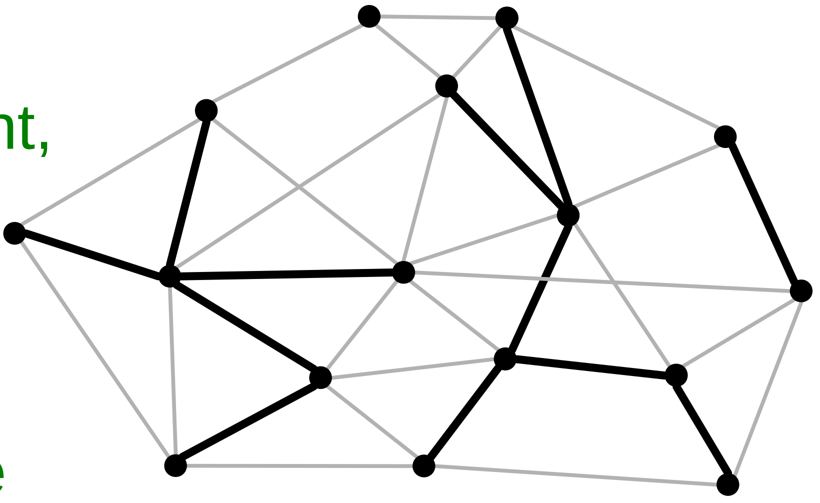
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component



Finding a Spanning Tree

An additive method

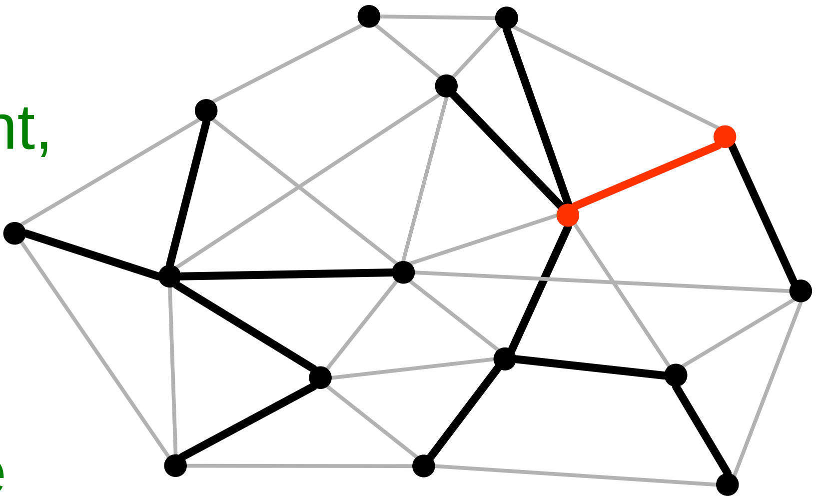
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component



Finding a Spanning Tree

An additive method

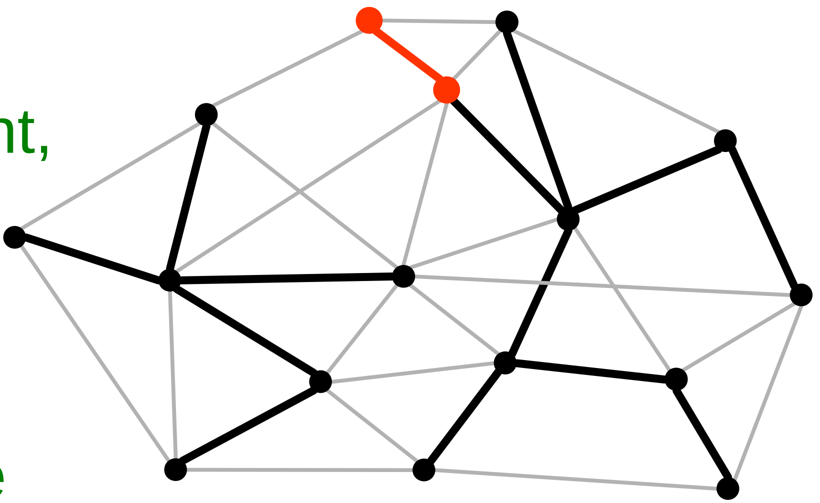
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component



Finding a Spanning Tree

An additive method

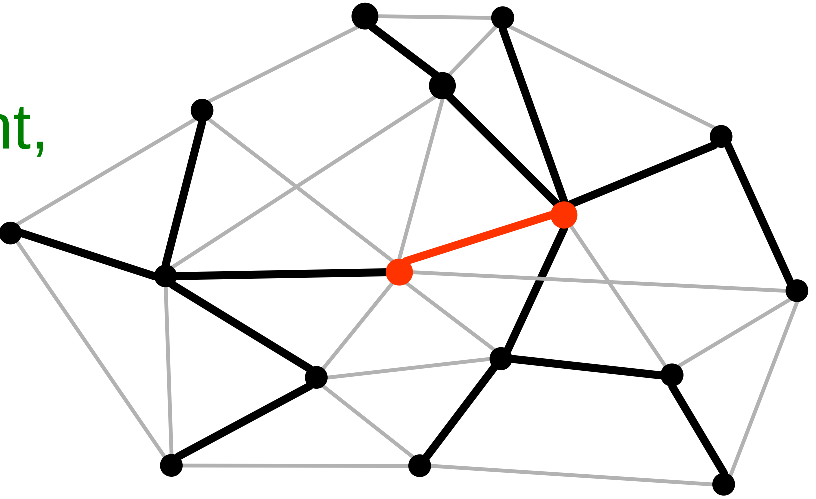
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component



Finding a Spanning Tree

An additive method

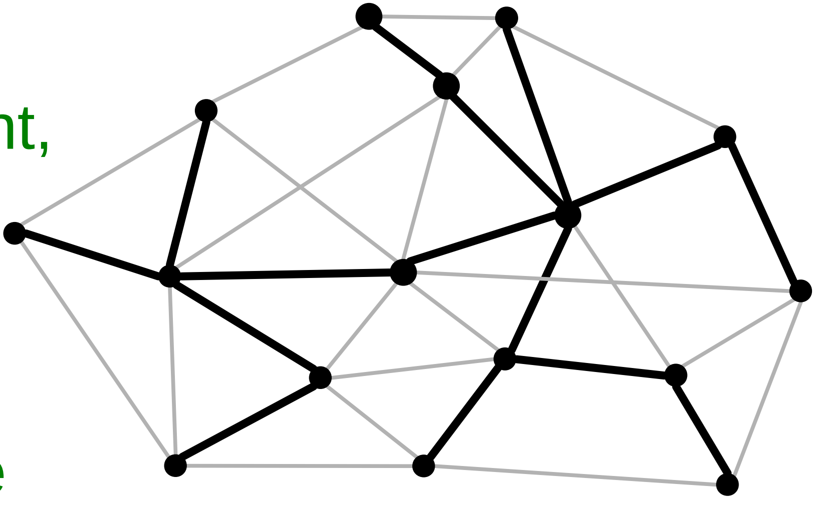
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component



Finding a Spanning Tree

An additive method

- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)

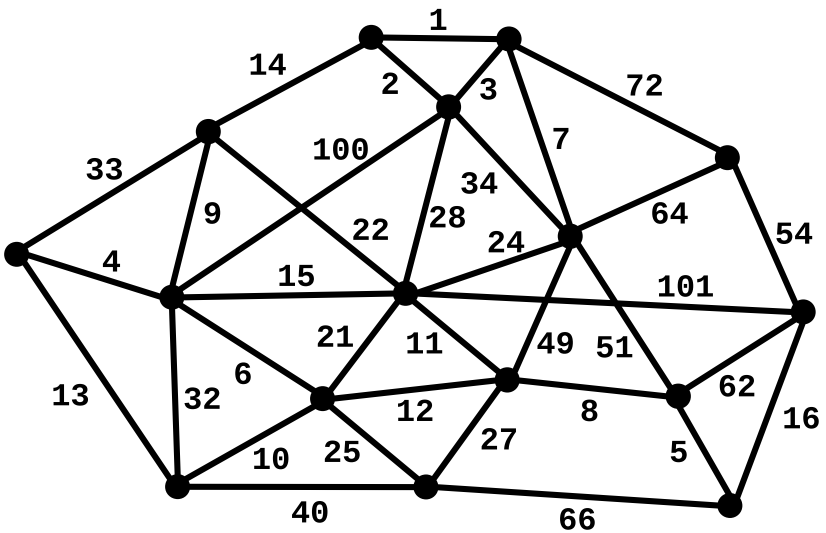


- Repeat until only one component

Minimum Spanning Trees

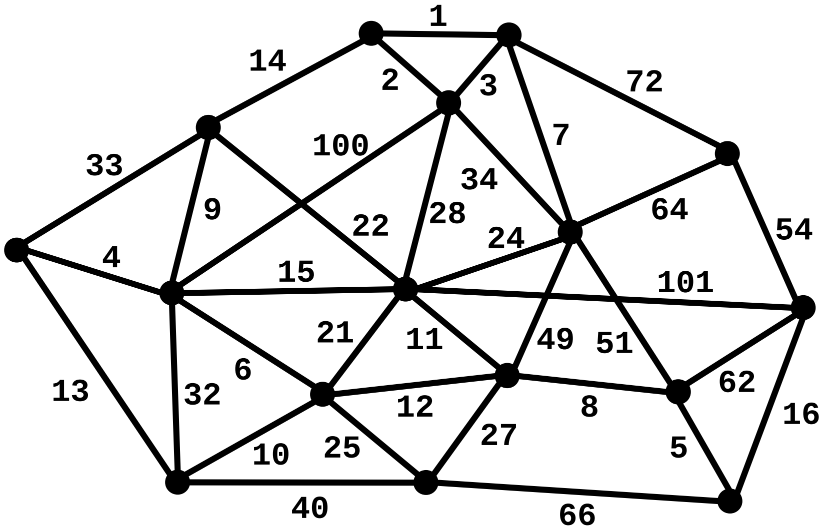
- Suppose edges are weighted, and we want a spanning tree of *minimum cost* (sum of edge weights)

- Useful in network routing & other applications



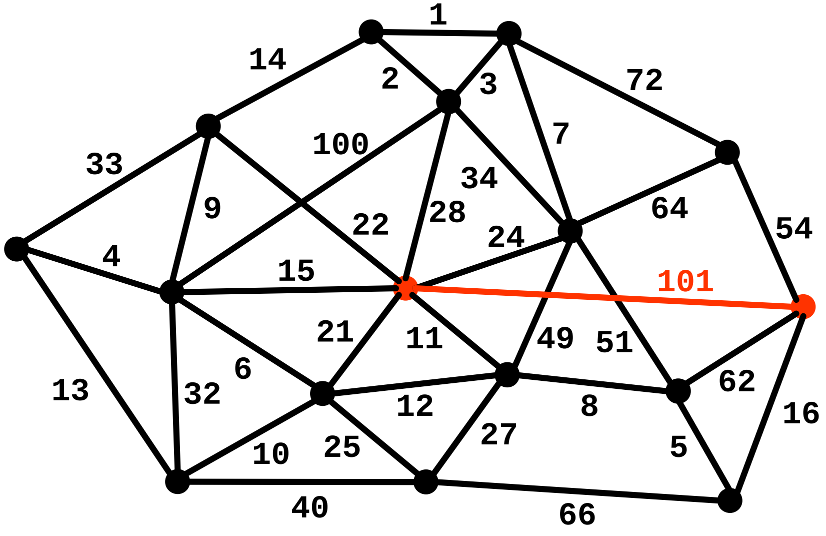
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



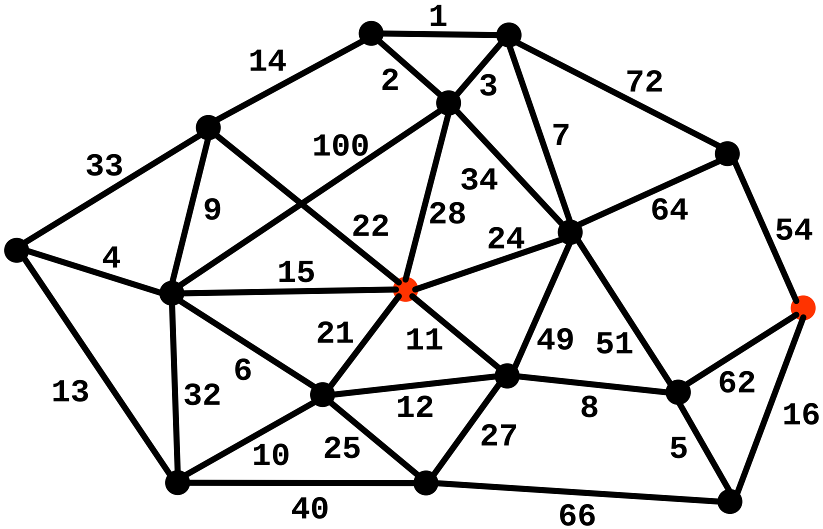
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



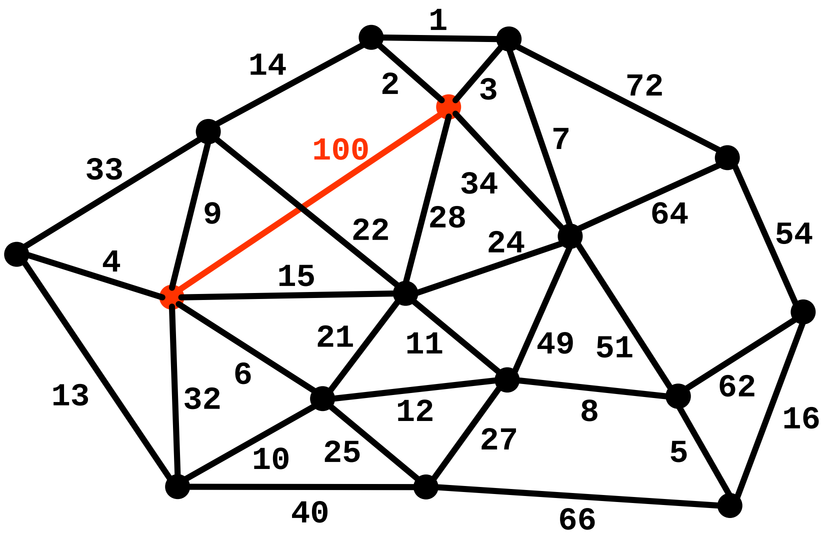
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



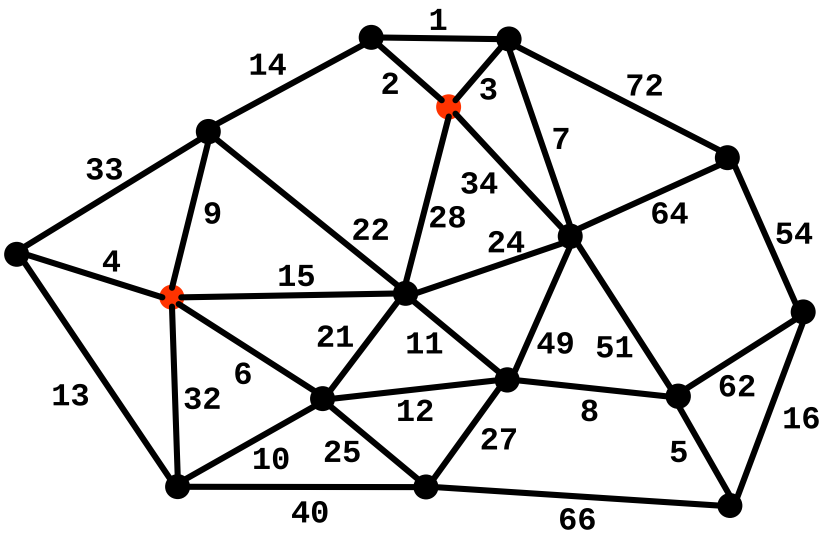
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



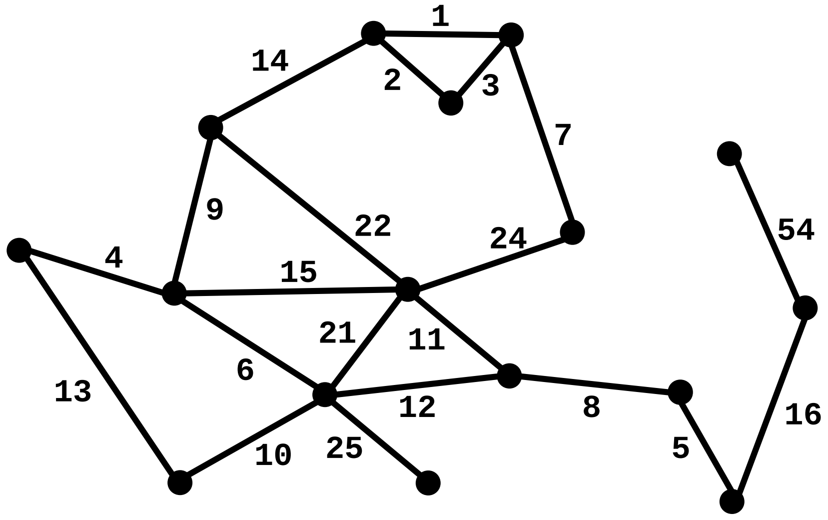
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



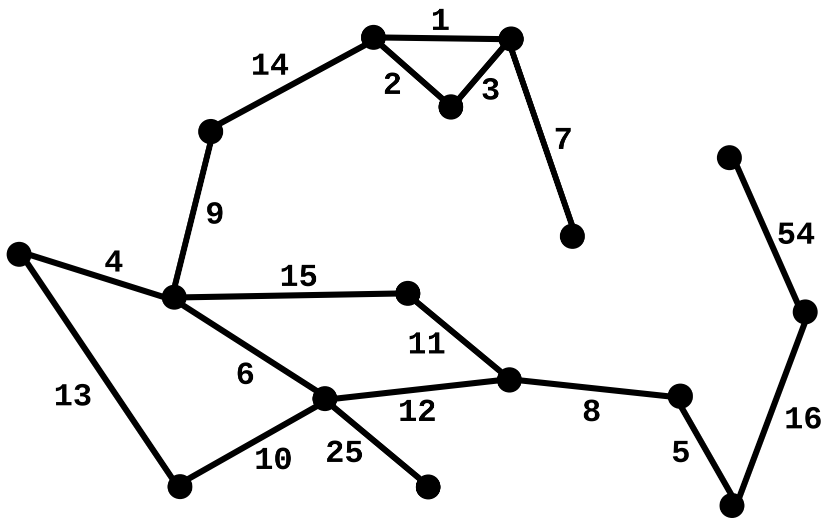
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



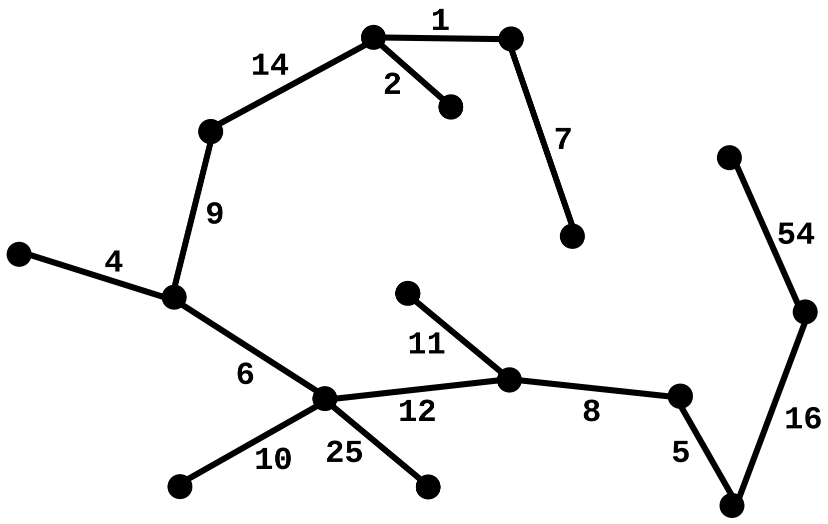
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



3 Greedy Algorithms

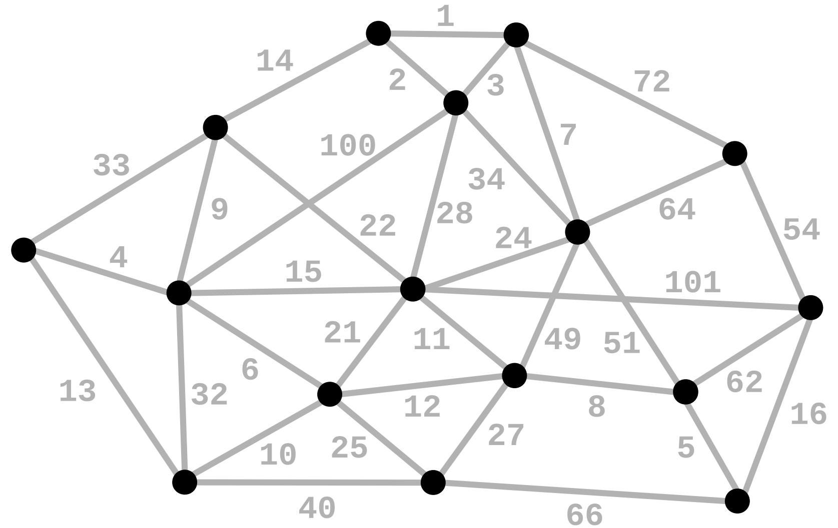
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

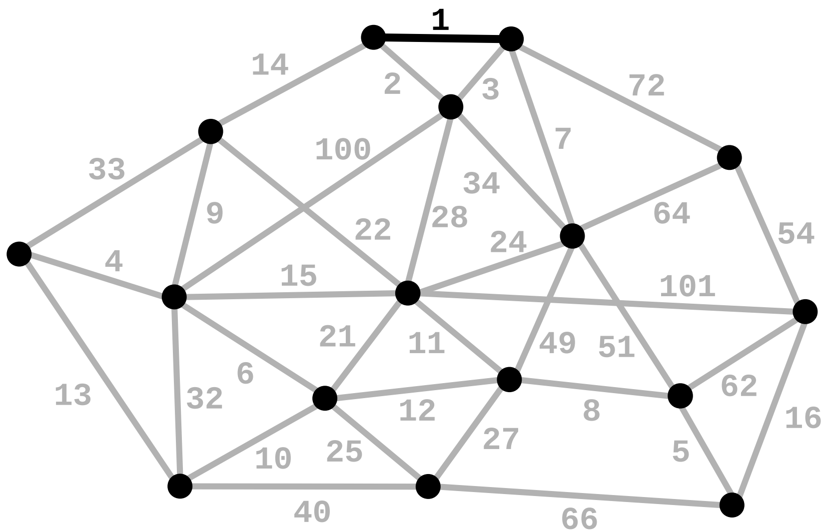
Kruskal's algorithm



3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

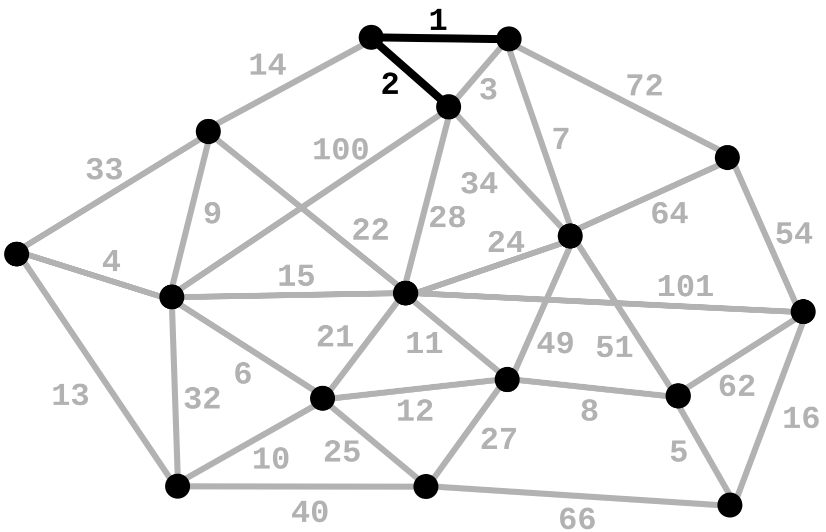
Kruskal's algorithm



3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

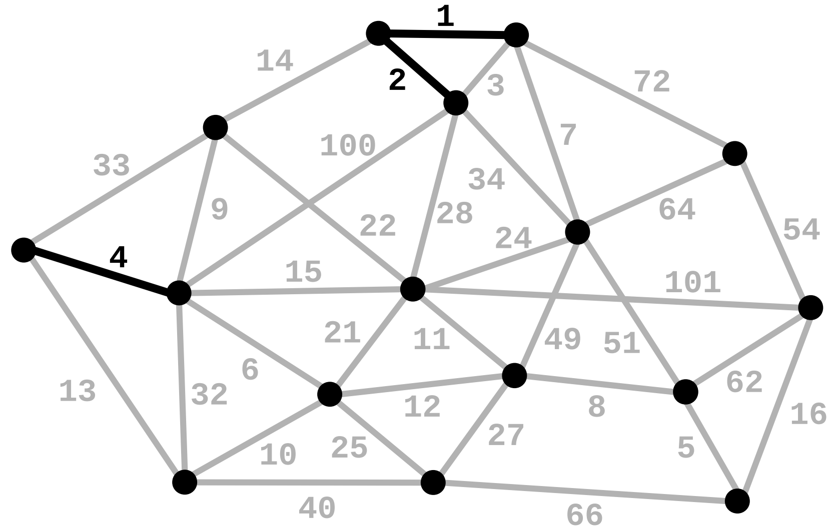
Kruskal's algorithm



3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

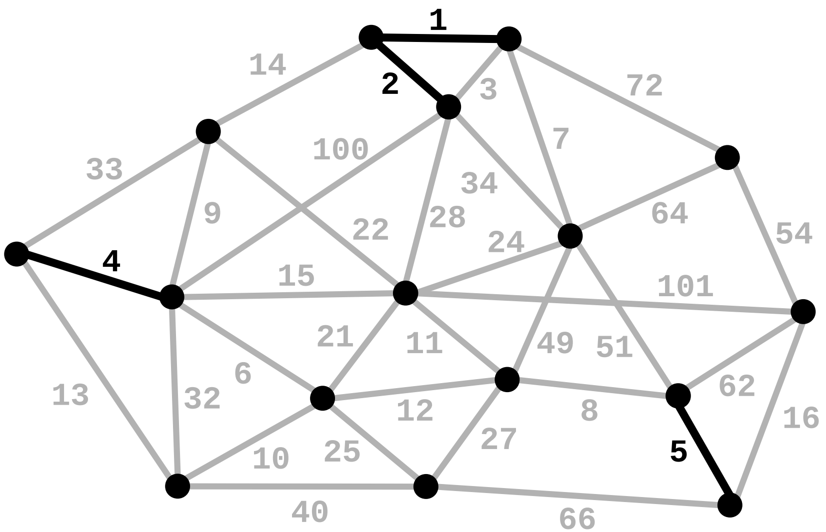
Kruskal's algorithm



3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

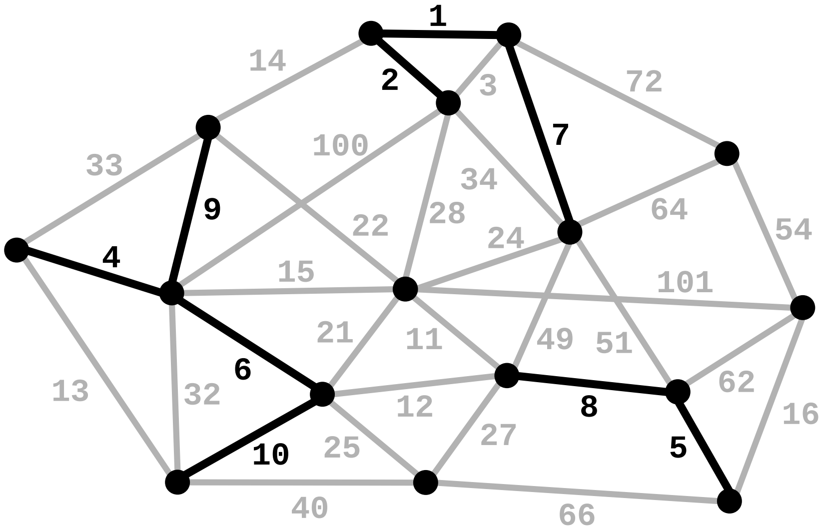
Kruskal's algorithm



3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

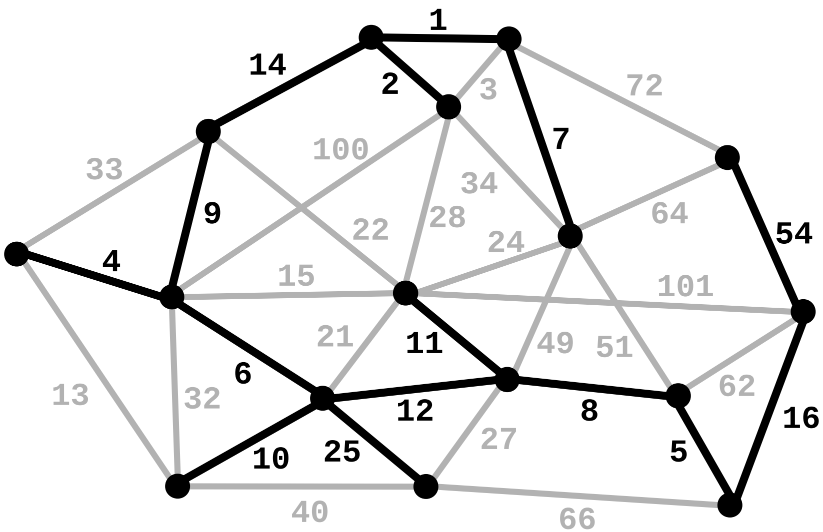
Kruskal's algorithm



3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm

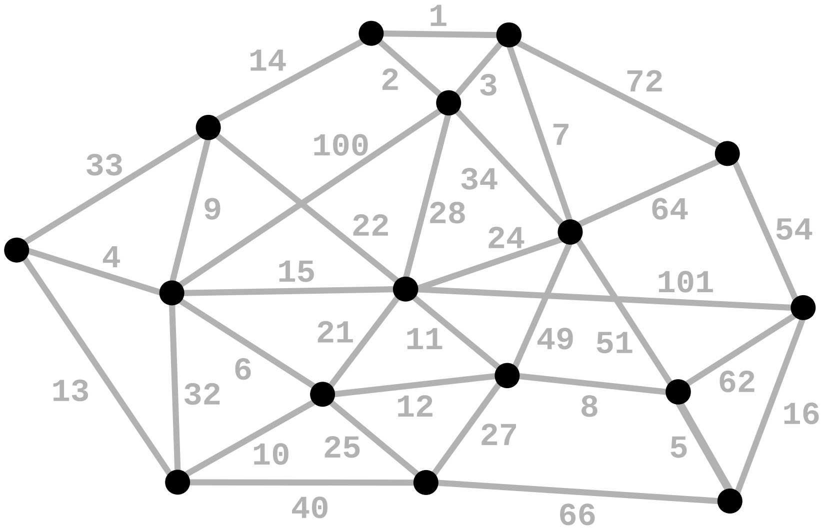


3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm

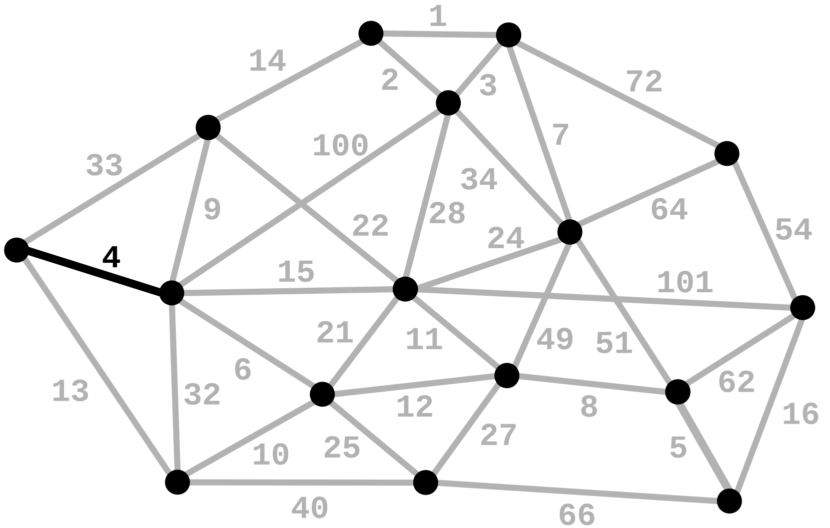
(reminiscent of Dijkstra's algorithm)



3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
(reminiscent of Dijkstra's algorithm)

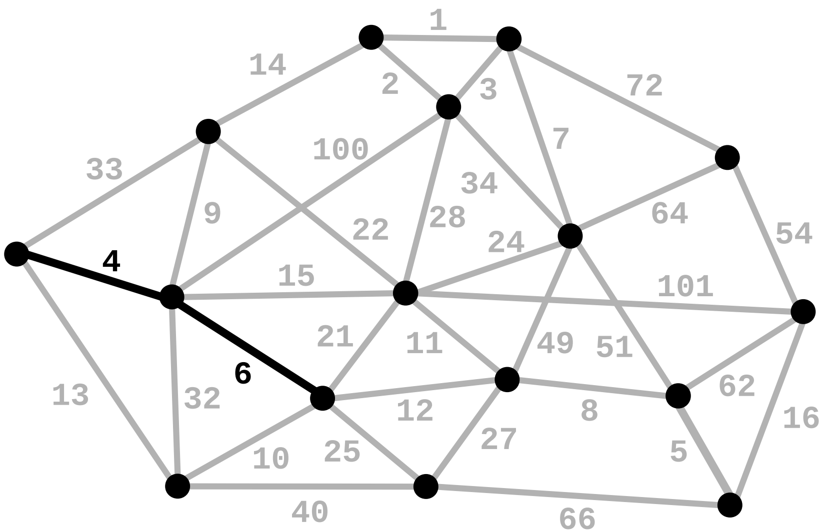


3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm

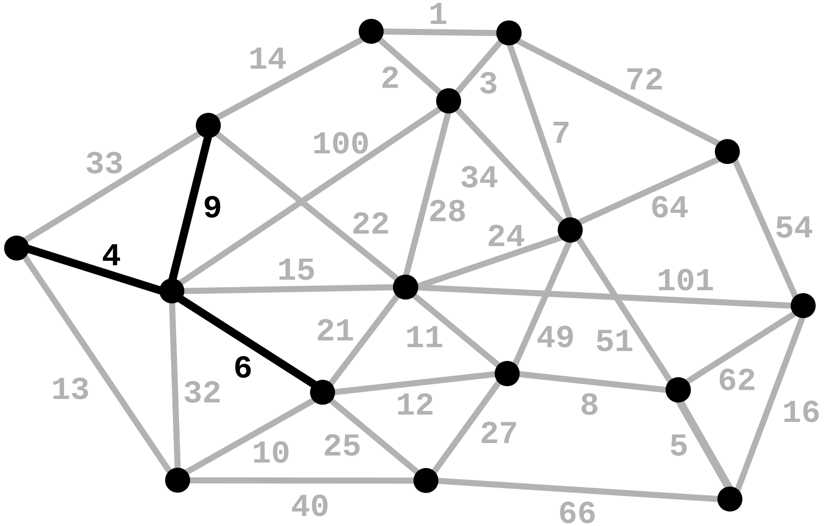
(reminiscent of Dijkstra's algorithm)



3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

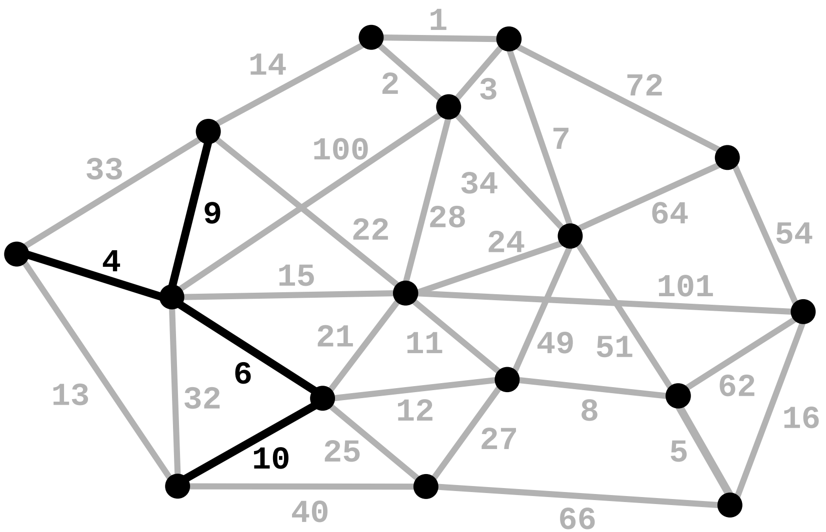
Prim's algorithm
(reminiscent of Dijkstra's algorithm)



3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

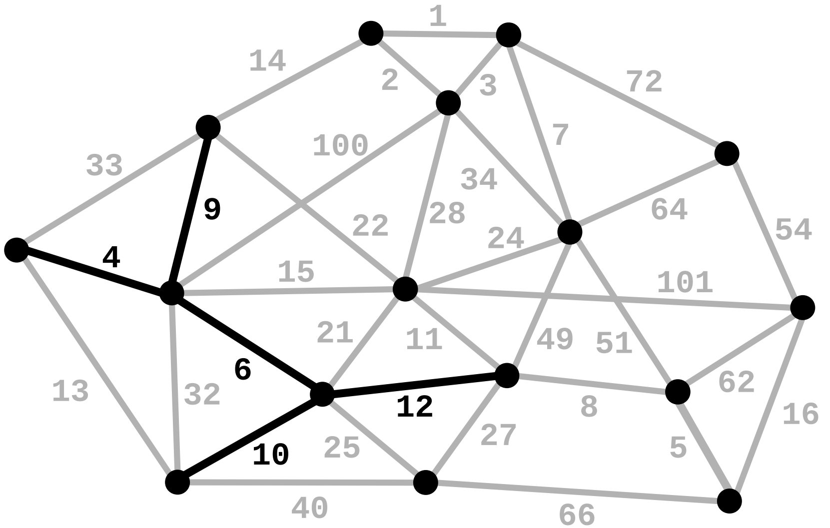
Prim's algorithm
(reminiscent of Dijkstra's algorithm)



3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
(reminiscent of Dijkstra's algorithm)

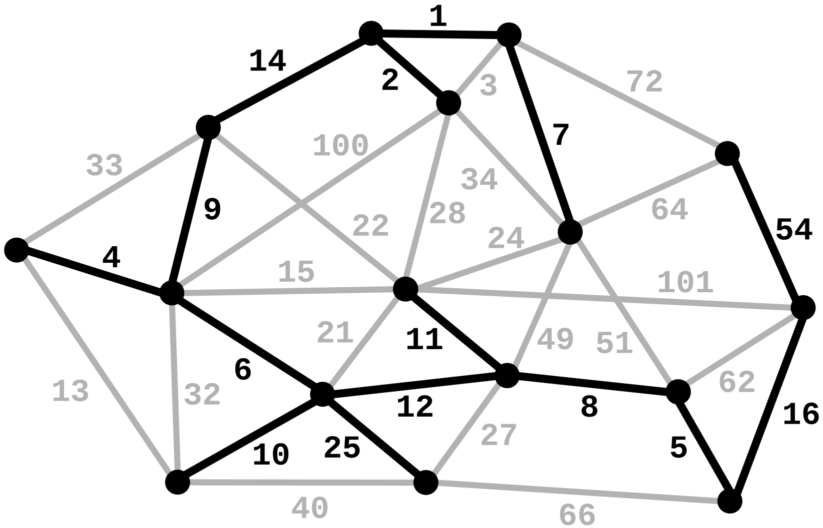


3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

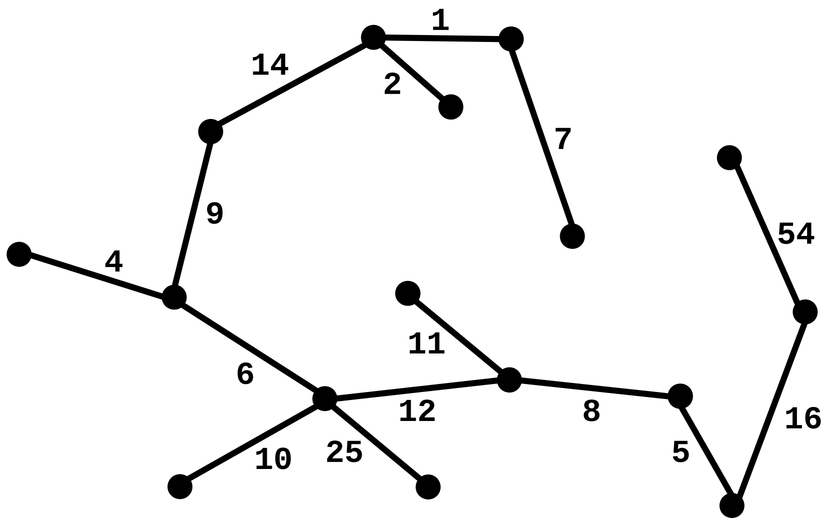
Prim's algorithm

(reminiscent of Dijkstra's algorithm)



3 Greedy Algorithms

All 3 greedy algorithms give the same minimum spanning tree (assuming distinct edge weights)



Analysis?

- How fast do these algorithms run?
- Why are they correct?
- Which is the easiest to implement?

- We'll examine these questions more closely next week.

Exceptions

Runtime Exceptions

Exceptions are usually thrown to indicate that something bad happened

- **`IOException`** on failure to open or read a file
- **`ClassCastException`** if attempted to cast an object to a type that is not a supertype of the dynamic type of the object
- **`NullPointerException`** if tried to dereference **`null`**
- **`ArrayIndexOutOfBoundsException`** if tried to access an array element at index $i < 0$ or $\geq$ the length of the array

Runtime Exceptions

- Exceptions can be caught by the program using a **`try/catch`** block
- **`catch`** clauses are called *exception handlers*

```
Integer x = null;
try {
   x = (Integer)y;
   System.out.println(x.intValue());
} catch (ClassCastException e) {
   System.out.println("y was not an Integer");
} catch (NullPointerException e) {
   System.out.println("y was null");
}
```

Runtime Exceptions

You can define your own exceptions and throw them

```
class MyOwnException extends Exception {}

...

if (input == null) {
   throw new MyOwnException();
}
```

Runtime Exceptions

Any exception you throw must either be caught or declared in the method header

```
void foo(int input) throws MyOwnException {
  if (input == null) {
    throw new MyOwnException();
  }
  ...
}
```

- Note: **`throws`** means "can throw", not "does throw"
- some common exceptions do not have to be declared (e.g., **`NullPointerException`**, **`ClassCastException`**)

How Exceptions are Handled

- If the exception is thrown from inside a try/catch block with a handler for that exception (or a superclass of the exception), then that handler is executed
- Otherwise, the method terminates abruptly and control is passed back to the calling method
- If the calling method can handle the exception (i.e., if the call occurred within a try/catch block with a handler for that exception), then that handler is executed
- Otherwise, the calling method terminates abruptly, etc.
- If none of the calling methods handle the exception, the entire program terminates with an error message

Checking Class Casts

Two ways to check if a class cast will succeed:

- use **`instanceof`**
- just do it, and catch the exception if it fails

```
Integer x = null;
if (y instanceof Integer) {
   x = (Integer)y;
} else {
   System.out.println("y was not an Integer");
}
```

```
Integer x = null;
try {
   Integer x = (Integer)y;
} catch (ClassCastException e) {
   System.out.println("y was not an Integer");
}
```