Wrap Concurrency
Minimum Spanning Trees

CSE 332 Sp25
Lecture 24

Announcements

	Monday	Tuesday	Wed	Thursday	Friday
This Week	Veteran's Day	Ex 11 (prefix prog)due	TODAY Ex 13 (MST,prog) out		Ex 12 (concurrency, GS) due final conflict form due
Next Week	Ex 14 (P/NP, GS) out		Ex 13 due		Ex 14 due

Final Exam information page is up.

If you are requesting a conflict exam, please do so by Friday.

Warm-Up

We said last time bank account should have 1 lock per object, not 1 lock per method that updates the balance. Show a bad interleaving when we have a lock for each method.

```
class BankAccount{
 private int balance = 0;
 private Lock lk = new Lock();
 private Lock lk2 = new Lock();

void withdraw(int amount){
   lk.acquire(); //might block
   int b = getBalance();
   if(amount > b) {
     lk.release();
     throw new
  WithdrawTooLargeException();
   setBalance(b – amount);
   lk.release();
 }
```

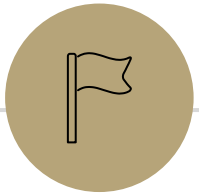
```
void deposit(int amount){
   lk2.acquire();
   int b = getBalance();
   setBalance(b + amount);
   lk2.release();
 }
```

Warm-up

```
void withdraw(int amount){
1   lk.acquire(); //might block
2   int b = getBalance();
3   if(amount > b) {
      lk.release();
      throw new …
8   setBalance(b – amount);
9   lk.release();
  }
```

```
void deposit(int amount){
      lk2.acquire();
4     int b = getBalance();
5     setBalance(b + amount);
6     lk2.release();
7   }
```

This interleaving erases the deposit.
If there were only one lock (just `lk` not `lk2`), this interleaving isn't possible!



Some Java Notes

Real Java locks

A re-entrant lock is available in:

`java.util.concurrent.locks.ReentrantLock`

Methods are `lock()` and `unlock()`

`synchronized`

Java has built-in support for reentrant locks with the keyword `synchronized`

```
synchronized (expression) {
 //Critical section here
 }
```

- Expression must evaluate to an `Object`.
  - Every object “is a lock” in java
  - Lock is acquired at the opening brace and released at the matching closing brace. (Java handles instantiating the lock, remembering which one is which, etc.)
  - Released even if control leaves due to throw/return/etc.

`synchronized`

If your whole method is a critical section

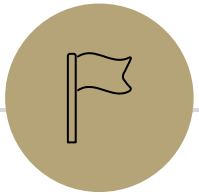
And the object you want for your lock is `this`

You can change the method header to include `synchronized`.

E.g. `private synchronized void getBalance()`

Equivalent of having

`synchronized(this){ }` around entire method body.



Deadlock

Multiple Locks

What happens when you need to acquire more than one lock? What new thing could go wrong with this code?

```
void transferTo(int amt, BankAccount a){
     this.lk.acquire();
     a.lk.acquire();
     this.withdraw(amt);
     a.deposit(amt);
     a.lk.release();
     this.lk.release();
 }
```

Multiple Locks

THREAD 1, FROM ACCT1 TO ACCT2

```
void transferTo(…){
    this.lk.acquire();
    a.lk.acquire();
    this.withdraw(amt);
    a.deposit(amt);
    a.lk.release();
    this.lk.release();
}
```

1

blocks

THREAD 2, FROM ACCT2 TO ACCT1

```
void transferTo(…){
    this.lk.acquire();
    a.lk.acquire();
    this.withdraw(amt);
    a.deposit(amt);
    a.lk.release();
    this.lk.release();
}
```

2

blocks

UH-OH!
Thread 1 needs Thread 2 to let go, but
Thread 2 needs Thread 1 to let go

Deadlock

Deadlock occurs when we have a cycle of dependencies

i.e. we have threads $T_1, \ldots, T_n$ such that

thread T_i is waiting for a resource held by T_{i+1} and

T_n is waiting for a resource held by T_1.

How can we set up our program so this doesn't happen?

Deadlock Solutions

Option 1: Smaller critical section:

- Acquire bank account 1's lock, withdraw, release that lock
- Acquire bank account 2's lock, deposit, release that lock

Maybe ok here, but exposes wrong total amount in bank while blocking.

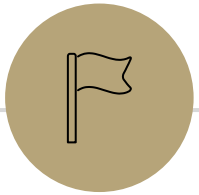
Option 2: Coarsen the lock granularity

- One lock for all accounts.
- Probably too coarse for good behavior

Option 3: All methods acquiring multiple locks acquire them in the same order.

- E.g. in order of account number.

More options – take Operating Systems!



Conventional Wisdom

Conventional Wisdom

There are three types of memory

Thread local (each thread has its own copy)

Immutable (no thread overwrites that memory location)

Shared and mutable
- Synchronization needed to control access.

Whenever possible make memory of type 1 or 2.

If you can minimize/eliminate side-effects in your code, you can make more memory type 2.

Conventional Wisdom

Consistent locking:

Every location that reads or writes a shared resource has a lock.

Even if you can't think of a bad interleaving, better safe than sorry.

When deciding how big to make a critical section:

Start coarse grained (i.e., start with a very large critical section), and move finer if you really need to improve performance.

Conventional Wisdom

Avoid expensive computations or I/O in critical sections.

If possible release the lock, do the long computation, and reacquire the lock.

Just make sure you haven't introduced a race condition.

Think in terms of what operations need to be atomic.

i.e. consider atomicity first, then think about where the locks go.

Conventional Wisdom

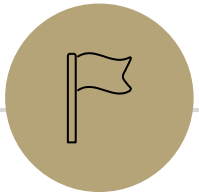
Don’t write your own.

There’s probably a library that does what you need.

Use it.

There are “thread-safe” libraries like `ConcurrentHashMap`.

No need to do it yourself when experts already did it
- and probably did it better.



Data Race

A distinction

A **Race Condition** is an error in parallel code –the output depends on the order of execution of the threads (who wins the race to be executed).

We'll divide into two types

A **data race** is an error where **at (potentially) the same time:**

1. Two threads are writing the same variable.

2. One thread writes to a variable while another is reading it.

A **Bad interleaving**

Is when incorrect behavior (as defined by the user) could result from a particular sequential execution order.
(We **don't** consider deadlock a race condition, but a separate kind of concurrency bug)

Huh?

Consider this code…

```
class Stack<E>{
  private E[] array = (E[])new Object[SIZE];
  private int index = -1;
  synchronized public Boolean isEmpty() { return index==-1;}
  synchronized public void push(E val) {array[++index]=val;}
  synchronized public E pop() {
    if(isEmpty()) { throw new StackEmptyException(); }
    return array[index--];
  }
  public E peek() { E ans = pop(); push(ans); return ans; }
}
```

What's the problem?

That `peek` is, uh, interesting..

Certainly would work as sequential code (albeit probably bad style).

But with multiple threads…?

Well, there aren't any **data races.** The calls to `push` and `pop` are synchronized. We only ever have one thread touching any data (the underlying array or index variable) at a time.

But it certainly isn't correct! Peek has an intermediate state that (if exposed during a **bad interleaving**) leads to incorrect behavior.

Bad Interleaving 1

THREAD A (PEEK)

```
2  E ans = pop();
4  push(ans);
5  return ans;
```

THREAD B (PUSH + ISEMPTY)

```
1  push(x);
3  boolean b = isEmpty();
```

Logical expectation: If we push (and haven't popped) then the stack is not empty.

This is a bad interleaving, without a data race.

Bad Interleaving 2

THREAD A (PEEK)

```
E ans = pop();
push(ans);
return ans;
```

THREAD B (TWO PUSHES)

```
push(x);
push(y);
```

Logical expectation: Pushed values go in LIFO order

Bad Interleaving 2

THREAD A (PEEK)

```
2  E ans = pop();
4  push(ans);
5  return ans;
```

THREAD B (TWO PUSHES)

```
1  push(x);
3  push(y);
```

Logical expectation: Pushed values go in LIFO order

This is a bad interleaving, without a data race.
Notice, this interleaving would be fine if we just had a generic list!

Bad Interleaving 3

THREAD A (PEEK)

```
E ans = pop();
push(ans);
return ans;
```

THREAD B (TWO PUSHES, POP)

```
push(x);
push(y);
E e = pop();
```

Logical expectation: Popped values come in LIFO order

Bad Interleaving 3

THREAD A (PEEK)

```
3  E ans = pop();
5  push(ans);
6  return ans;
```

THREAD B (TWO PUSHES, POP)

```
1  push(x);
2  push(y);
4  E e = pop();
```

Logical expectation: Popped values come in LIFO order

Bad Interleaving 4

THREAD A (PEEK)

```
E ans = pop();
push(ans);
return ans;
```

THREAD B (PEEK)

```
E ans = pop();
push(ans);
return ans;
```

Logical expectation: Peek on a non-empty heap does not throw an exception

Bad Interleaving 4

THREAD A (PEEK)

```
2 E ans = pop();
3 push(ans);
4 return ans;
```

THREAD B (PEEK)

```
1 E ans = pop();
5 push(ans);
6 return ans;
```

Logical expectation: Peek on a non-empty queue does not throw an exception

The Fix: Don’t allow interleaving!

`Peek` needs synchronization

Enlarge the critical section to be the whole method.

Ensures no one is even looking at the stack until we push back the element we popped.

The Fix

Problem so far: `peek` does writes, creating an intermediate state.
The right fix: Make `isEmpty` and `peek` synchronized.

It's tempting to try to fix the code like this (This is the wrong fix!):

`peek`, if "normally" implemented (or `isEmpty`) doesn't actually write anything. Maybe we can skip the synchronization on those?

Do NOT remove the synchronization from peek/isEmpty. You will create a data race!

`isEmpty` reads the same data `push` writes (e.g., the `index` variable). That is **definitionally** a data race

We Can’t Keep Getting Away With It

It might look like `isEmpty` or `peek` not being synchronized won’t lead to errors.

After all, that `push` is just one line! I can’t figure out how to make bad interleavings.

Don’t think just because you can’t figure out a bad interleaving that a data race won’t be a problem.

1. “single steps” aren’t always single steps.

2. A data race is an error **by definition**. The compiler does a lot of optimizations (including reordering sequential code) **assuming** you don’t have data races. If you have them, your code may execute wrong. And good luck figuring that bug out..(see Grossman 7.2)

Summary

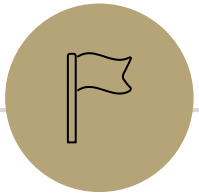
Two kinds of race conditions:

Data race (a thread potentially reads while another writes, or two potentially write simultaneously)

Bad Interleaving: exposes intermediate state to other threads, leads to behavior **we** find incorrect.

Data races are never acceptable, even if you can't find a bad interleaving.

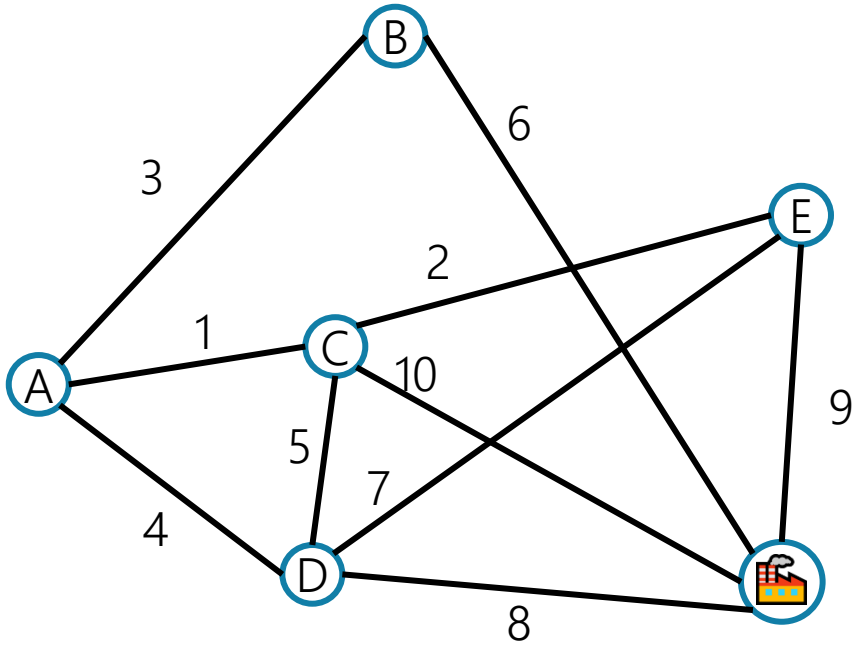
Checking correctness is usually: 1. finding there are no data races, then 2. looking for bad interleavings.



Minimum Spanning Trees

Minimum Spanning Trees

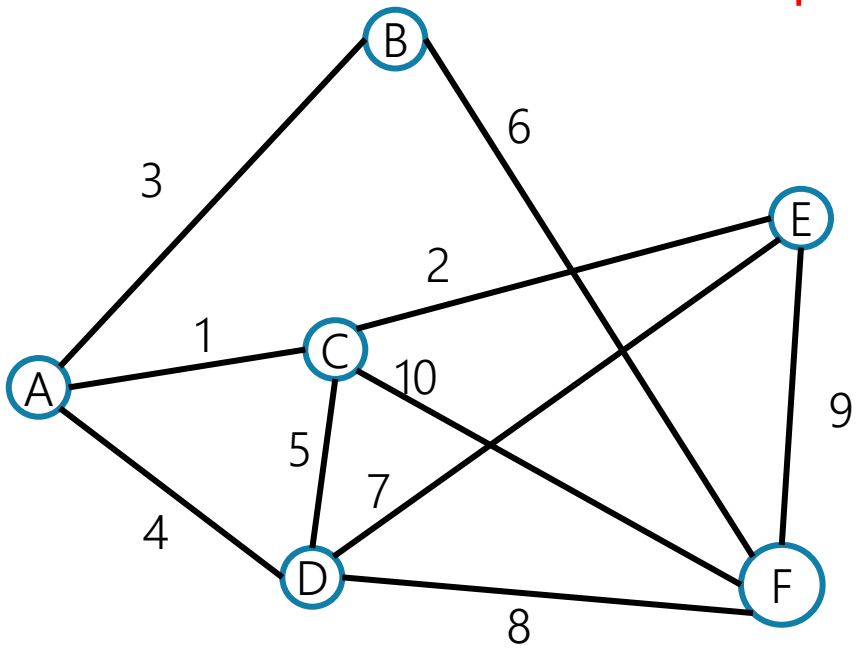
It's the 1920's. Your friend at the electric company needs to choose where to build wires to connect all these cities to the plant.



She knows how much it would cost to lay electric wires between any pair of locations, and wants the cheapest way to make sure electricity from the plant to every city.

Minimum Spanning Trees

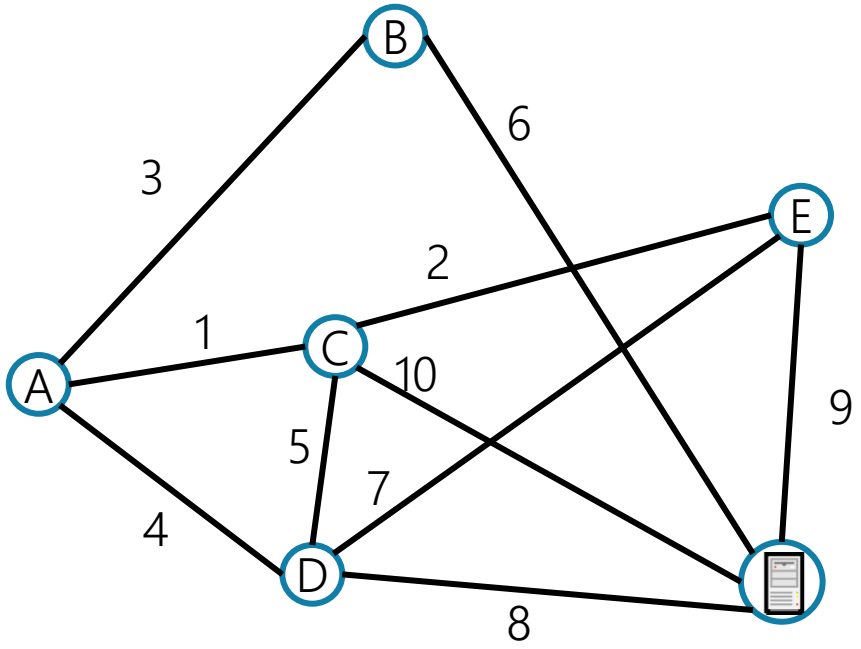
It's the 1950's Your boss at the phone company needs to choose where to build wires to connect all these phones to each other.



She knows how much it would cost to lay phone wires between any pair of locations, and wants the cheapest way to make sure Everyone can call everyone else.

Minimum Spanning Trees

It's today. Your friend at the ISP needs to choose where to build wires to connect all these cities to the Internet.



She knows how much it would cost to lay cable between any pair of locations, and wants the cheapest way to make sure Everyone can reach the server

Minimum Spanning Trees

What do we need? A set of edges such that:

- Every vertex touches at least one of the edges. (the edges **span** the graph)
- The graph on just those edges is **connected**.
  - i.e. the edges are all in the same **connected component.**
  - A connected component is a vertex and everything you can reach from it.
- The minimum weight set of edges that meet those conditions

Claim: The set of edges we pick never has a cycle. Why?

Aside: Trees

On graphs our tees:

- Don't need a root (the vertices aren't ordered, and we can start BFS from anywhere)
- Varying numbers of ~~children~~ neighbors
- Connected and no cycles

Tree (when talking about undirected graphs)

An undirected, connected acyclic graph.

MST Problem

What do we need? A set of edges such that:

- Every vertex touches at least one of the edges. (the edges **span** the graph)
- The graph on just those edges is **connected**.
- The minimum weight set of edges that meet those conditions.

Our goal is a tree!

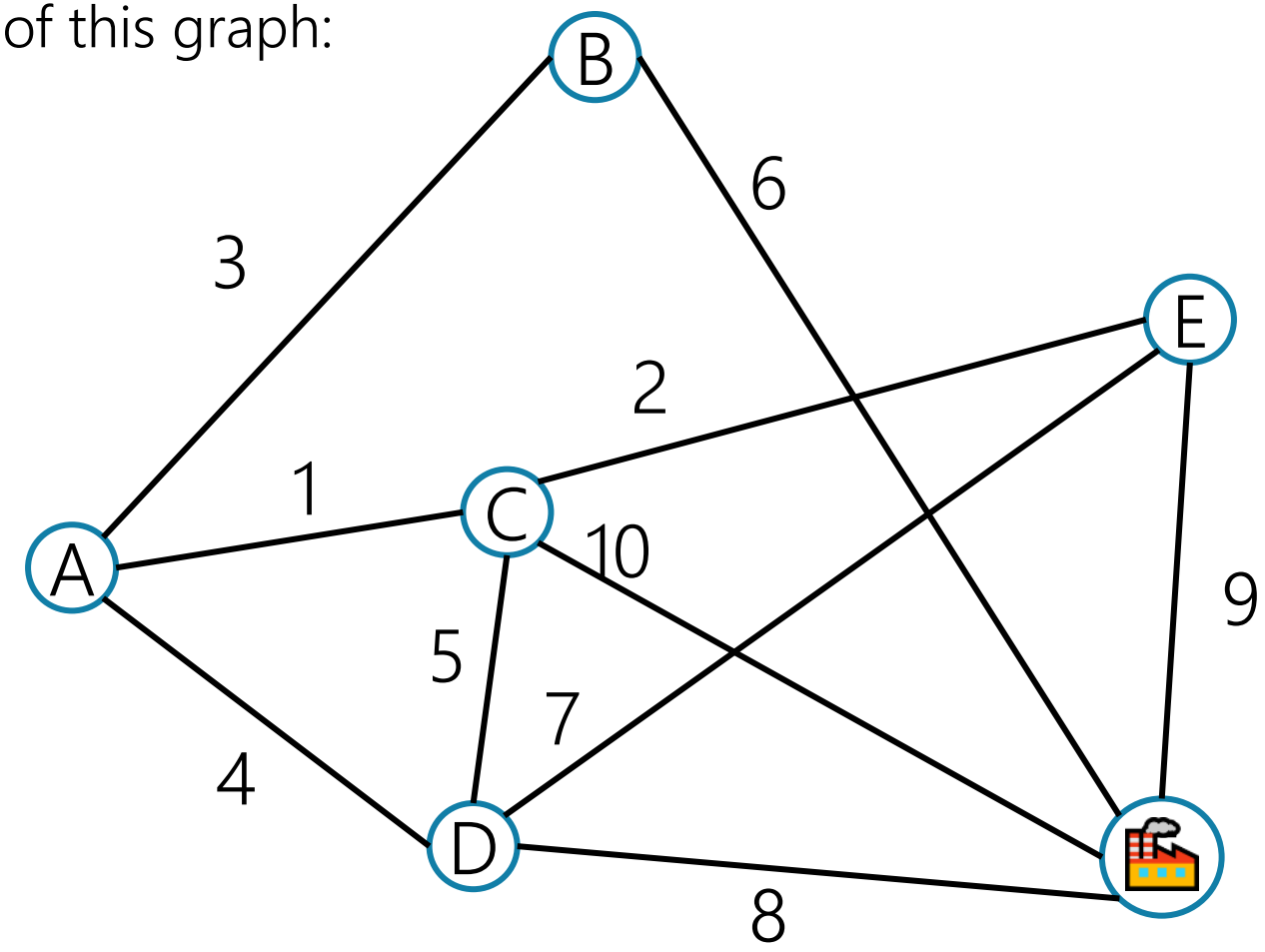
Minimum Spanning Tree Problem

Given: an undirected, (connected, simple) weighted graph G
Find: A minimum-weight set of edges such that you can get from any vertex of G to any other on only those edges.

We'll go through two different algorithms for this problem.

Example

Try to find an MST of this graph:



Prim's Algorithm

Algorithm idea: choose an arbitrary starting point. Add a new edge that:

- Will let you reach more vertices.
- Is as light as possible

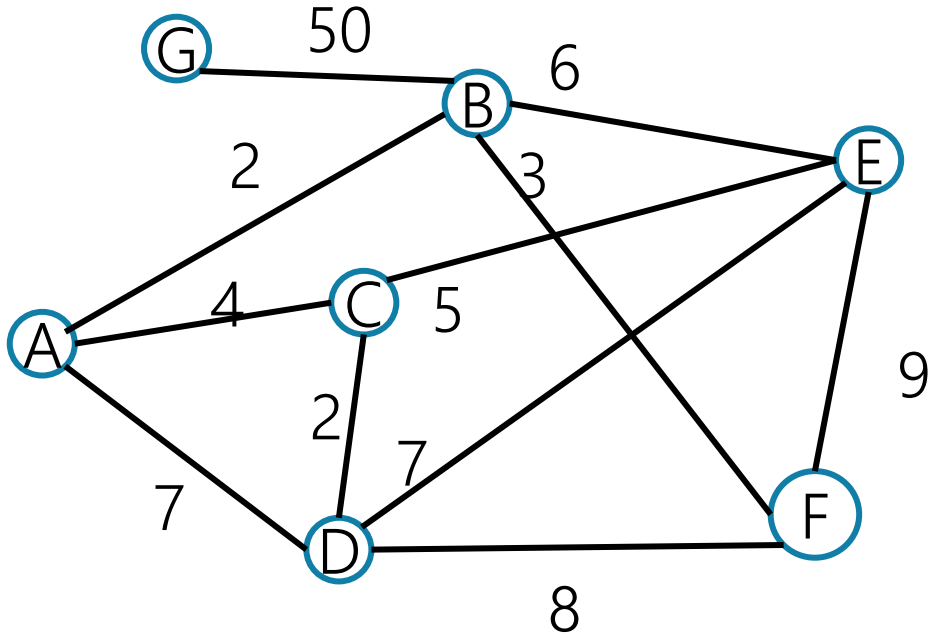
We'd like each not-yet-connected vertex to be able to tell us the lightest edge we could add to connect it.

Code

```
PrimMST(Graph G)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   foreach(edge (source, v) ) {
      v.dist = weight(source,v)
   while(there are unprocessed vertices){
       let u be the closest unprocessed vertex
        add u.bestEdge to spanning tree
        foreach(edge (u,v) leaving u){
           if(weight(u,v) < v.cost){
              v.cost = weight(u,v)
              v.bestEdge = (u,v)
           }
        }
          mark u as processed
     }
   }
```

Try it Out

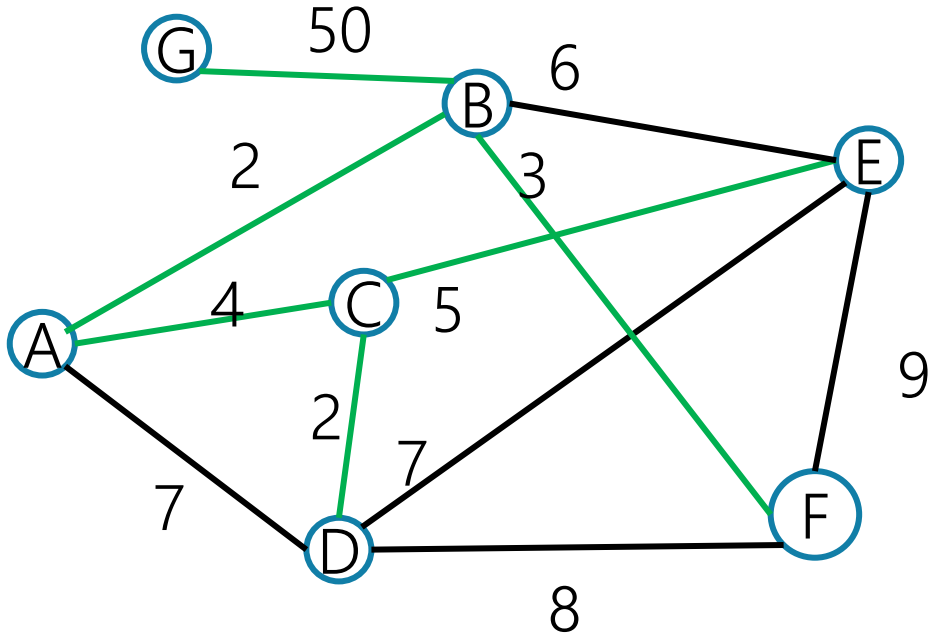
```
PrimMST(Graph G)
   initialize distances to ∞
   mark source as distance 0 //pick arbitrarily
   mark all vertices unprocessed
   foreach(edge (source, v) )
      v.dist = w(source,v)
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      add u.bestEdge to spanning tree
      foreach(edge (u,v) leaving u){
           if(w(u,v) < v.cost){
              v.cost = w(u,v)
              v.bestEdge = (u,v)
           }
       }
       mark u as processed
    }
```



Vertex	Cost	Best Edge	Processed
A			
B			
C			
D			
E			
F			
G			

Try it Out

```
PrimMST(Graph G)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   foreach(edge (source, v) )
      v.dist = w(source,v)
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      add u.bestEdge to spanning tree
      foreach(edge (u,v) leaving u){
           if(w(u,v) < v.cost){
              v.cost = w(u,v)
              v.bestEdge = (u,v)
           }
       }
       mark u as processed
    }
```



Vertex	Cost	Best Edge	Processed
A	--	--	Yes
B	2	(A,B)	Yes
C	4	(A,C)	Yes
D	~~7~~ 2	~~(A,D)~~(C,D)	Yes
E	~~6~~ 5	~~(B,E)~~(C,E)	Yes
F	3	(B,F)	Yes
G	50	(B,G)	Yes

```
PrimMST(Graph G)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed //and add to priority queue
   foreach(edge (source, v) ) {
      v.dist = weight(source,v)
   while(there are unprocessed vertices){
       let u be the closest unprocessed vertex //removeMin!
        add u.bestEdge to spanning tree
        foreach(edge (u,v) leaving u){
           if(weight(u,v) < v.cost){
              v.cost = weight(u,v) //updatePriority!!
              v.bestEdge = (u,v)
            }
        }
          mark u as processed
     }
   }
```

Running time: $\Theta(E \log V)$
Analysis same as Dijkstra, but can assume $E \geq V - 1$

Some Exercise Notes

We'll ask you to implement Prim's in Exercise 13.

You have a few options for the priority queue:

1. Use a Java library priority queue---but it doesn't have updatePriority() so you'll need a workaround:
A. Add edges instead of vertices to the priority queue OR
B. Allow multiple copies of each vertex into the queue (instead of decreasing priority, put in a second copy at the new priority OR

2. Use your (Exercise 2) priority queue instead---call updatePriority!

Will these change the running time? No! $\log(E) = \Theta(\log(V))$ for simple graphs.

Read the paragraph in the spec about this before you get too far. Also see alternate version of pseudocode in section slides tomorrow.

Does This Algorithm Always Work?

Prim's Algorithm is a **greedy** algorithm. Once it decides to include an edge in the MST it never reconsiders its decision.

Greedy algorithms rarely work.

There are special properties of MSTs that allow greedy algorithms to find them.

In fact MSTs are so *magical* that there's more than one greedy algorithm that works.

Why do all of these MST Algorithms Work?

MSTs satisfy two very useful properties:

Cycle Property: The heaviest edge along a cycle is NEVER part of an MST.

Cut Property: Split the vertices of the graph any way you want into two sets A and B. The lightest edge with one endpoint in A and the other in B is ALWAYS part of an MST.

Whenever you add an edge to a tree you create exactly one cycle, you can then remove any edge from that cycle and get another tree out.

This observation, combined with the cycle and cut properties form the basis of all of the greedy algorithms for MSTs.