Spanning Trees

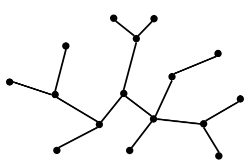
Generics in Java 1.5

Exceptions

Spanning Trees

Undirected Trees

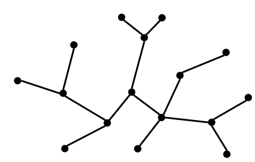
- An undirected graph is a *tree* if there is exactly one simple path between any pair of vertices



Facts About Trees

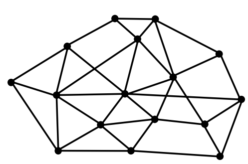
- |E| = |V| – 1
- connected
- no cycles

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



Spanning Trees

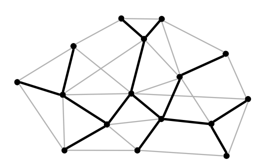
A *spanning tree* of a connected undirected graph (V,E) is a subgraph (V,E') that is a tree



Spanning Trees

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

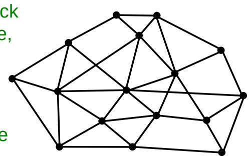
- Same set of vertices V
- $E' \subseteq E$
- (V,E') is a tree



Finding a Spanning Tree

A subtractive method

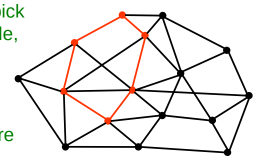
- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected (why?)
- Repeat until no more cycles



Finding a Spanning Tree

A subtractive method

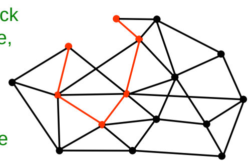
- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected (why?)
- Repeat until no more cycles



Finding a Spanning Tree

A subtractive method

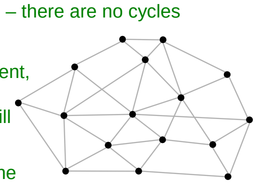
- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected (why?)
- Repeat until no more cycles



Finding a Spanning Tree

An additive method

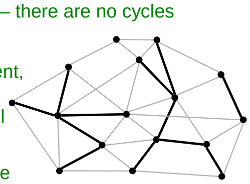
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component



Finding a Spanning Tree

An additive method

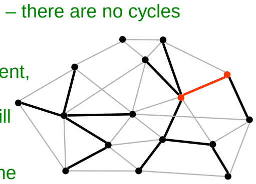
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component



Finding a Spanning Tree

An additive method

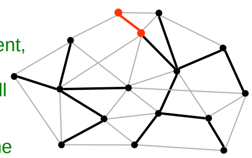
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component



Finding a Spanning Tree

An additive method

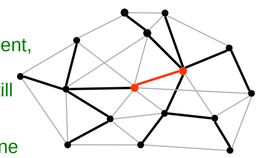
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component



Finding a Spanning Tree

An additive method

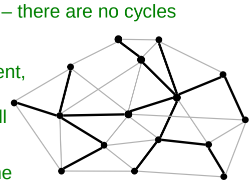
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component



Finding a Spanning Tree

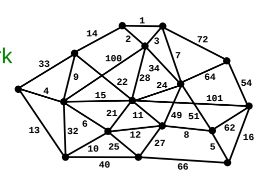
An additive method

- 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



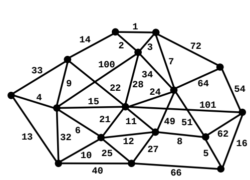
Minimum Spanning Trees

- Suppose edges are weighted, and we want a spanning tree of *minimum cost* (sum of edge weights)
- Useful in network routing & other applications



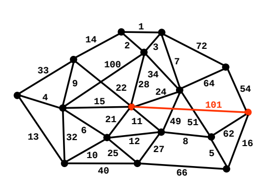
3 Greedy Algorithms

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



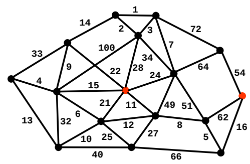
3 Greedy Algorithms

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



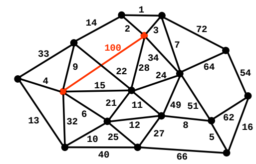
3 Greedy Algorithms

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



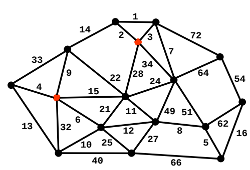
3 Greedy Algorithms

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



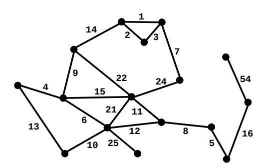
3 Greedy Algorithms

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



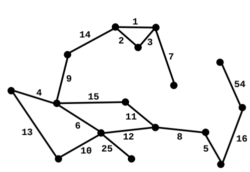
3 Greedy Algorithms

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



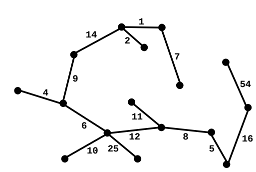
3 Greedy Algorithms

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



3 Greedy Algorithms

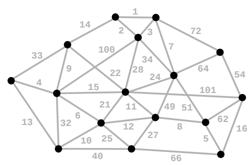
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



3 Greedy Algorithms

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

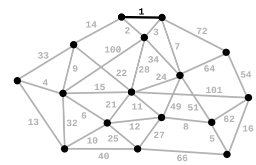
Kruskal's algorithm



3 Greedy Algorithms

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

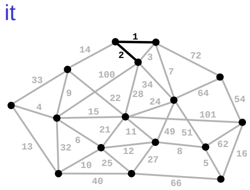
Kruskal's algorithm



3 Greedy Algorithms

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

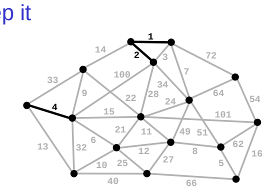
Kruskal's algorithm



3 Greedy Algorithms

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

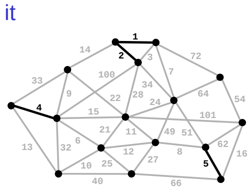
Kruskal's algorithm



3 Greedy Algorithms

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

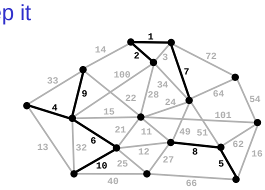
Kruskal's algorithm



3 Greedy Algorithms

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

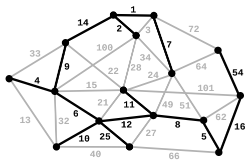
Kruskal's algorithm



3 Greedy Algorithms

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

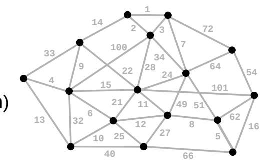
Kruskal's algorithm



3 Greedy Algorithms

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

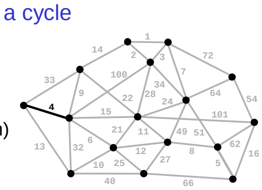
Prim's algorithm
(reminiscent of Dijkstra's algorithm)



3 Greedy Algorithms

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

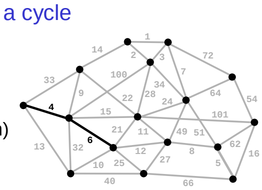
Prim's algorithm
(reminiscent of Dijkstra's algorithm)



3 Greedy Algorithms

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

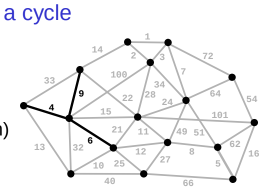
Prim's algorithm
(reminiscent of Dijkstra's algorithm)



3 Greedy Algorithms

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

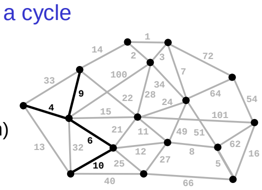
Prim's algorithm
(reminiscent of Dijkstra's algorithm)



3 Greedy Algorithms

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

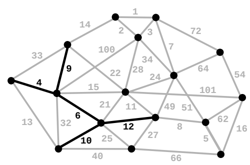
Prim's algorithm
(reminiscent of Dijkstra's algorithm)



3 Greedy Algorithms

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

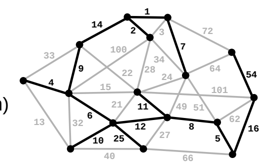
Prim's algorithm
(reminiscent of Dijkstra's algorithm)



3 Greedy Algorithms

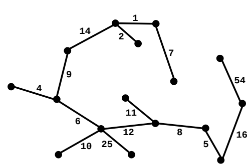
C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

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)



Generics in Java 1.5

Generics in Java 1.5

- When using a collection (e.g. `LinkedList`, `HashSet`, `Hashtable`), we generally have a single type `T` of elements that we store in it (e.g. `Integer`, `String`)
- Before 1.5, when extracting an element, had to cast it to `T` before we could invoke `T`'s methods
- Compiler could not check that the cast was correct, since it didn't know what `T` was (and there was no way to tell it)
- Inconvenient and unsafe, could fail at run time

Generics in Java 1.5

- Generics in Java 1.5 provide a way to communicate `T`, the type of elements of a collection, to the compiler
- Compiler can check that you have used the collection consistently and inserts the correct cast implicitly when extracting values

Example

old

```
//removes 4-letter words from c
//elements must be Strings
static void purge(Collection c) {
  Iterator i = c.iterator();
  while (i.hasNext()) {
    if (((String)i.next()).length() == 4)
      i.remove();
}}
```

new

```
//removes 4-letter words from c
static void purge(Collection<String> c) {
  Iterator<String> i = c.iterator();
  while (i.hasNext()) {
    if (i.next().length() == 4)
      i.remove();
}}
```

Another Example

old

```
Hashtable grades = new Hashtable();
grades.put("John",new Integer(67));
grades.put("Jane",new Integer(88));
grades.put("Fred",new Integer(72));
Integer x = (Integer)grades.get("John");
System.out.println(x.intValue());
```

new

```
Hashtable<String,Integer> grades =
  new Hashtable<String,Integer>();
grades.put("John",new Integer(67));
grades.put("Jane",new Integer(88));
grades.put("Fred",new Integer(72));
Integer x = grades.get("John");
System.out.println(x.intValue());
```

Generics in Java 1.5

- Java inserts the correct cast automatically, based on the declared type
- In this example, `grades.get("John")` is automatically cast to `Integer`

```
Hashtable<String,Integer> grades =
  new Hashtable<String,Integer>();
grades.put("John",new Integer(67));
grades.put("Jane",new Integer(88));
grades.put("Fred",new Integer(72));
Integer x = grades.get("John");
System.out.println(x.intValue());
```

Generics in Java 1.5

- `<T>` is read, "of `T`", e.g. `Stack<Integer>` is read, "`Stack` of `Integer`"
- The type annotation `<T>` informs the compiler that all extractions from this collection should be automatically cast to `T`
- Specify type in declaration, can be checked at compile time – can eliminate all explicit casts

Advantage of Generics

- Declaring `Collection<String> c` tells us something about the variable `c` that holds wherever it is used, and the compiler guarantees it
- On the other hand, a cast tells us something the programmer *thinks* is true at a single point in the code, and the Java virtual machine checks whether the programmer is right only at run time

Subtypes

`Stack<Integer>` is not a subtype of `Stack<Object>`

```
Stack<Integer> s = new Stack<Integer>();
s.push(new Integer(7));
Stack<Object> t = s; //gives compiler error
t.push("bad idea");
System.out.println(s.pop().intValue());
```

However, `Stack<Integer>` *is* a subtype of `Stack` (for backward compatibility with 1.4.2)

```
Stack<Integer> s = new Stack<Integer>();
s.push(new Integer(7));
Stack t = s; //compiler allows this
t.push("bad idea");
System.out.println(s.pop().intValue());
```

Checked Insertion

```
Set<Integer> s = new HashSet<Integer>();
s.add(new Integer(7));
Set<Object> t = s; //gives compiler error
t.add("bad idea");
```

```
Set<Integer> s = new HashSet<Integer>();
s.add(new Integer(7));
Set t = s; //compiler allows this
t.add("bad idea");
```

```
Set<Integer> s =
  Collections.checkedSet(new HashSet<Integer>(),
                         Integer.class);
s.add(new Integer(7));
Set t = s;
t.add("bad idea"); //gives compiler error
```

Creating Your Own Generic Types

```
public interface List<E> { //E is a type variable
   void add(E x);
   Iterator<E> iterator();
}

public interface Iterator<E> {
   E next();
   boolean hasNext();
}
```

- To use the generic type declaration **`List<E>`**, supply an actual type argument, e.g. **`List<Integer>`**
- All occurrences of the formal type parameter (**`E`** in this case) are replaced by the actual type argument (**`Integer`** in this case)

Wildcards

old

```
void printCollection(Collection c) {
  Iterator i = c.iterator();
  while (i.hasNext()) {
    System.out.println(i.next());
}}
```

bad

```
void printCollection(Collection<Object> c) {
  for (Object e : c) {
    System.out.println(e);
}}
```

good

```
void printCollection(Collection<?> c) {
  for (Object e : c) {
    System.out.println(e);
}}
```

Bounded Wildcards

```
static void sort(List<? extends Comparable> c) {
  ...
}
```

Generic Methods

Adding all elements of an array to a **`Collection`**

bad

```
static void a2c(Object[] a, Collection<?> c) {
  for (Object o : a) {
    c.add(o); //compile time error
}}
```

good

```
static <T> void a2c(T[] a, Collection<T> c) {
  for (T o : a) {
    c.add(o); //ok
}}
```

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 ≥ 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");
}
```