

# MA/CSSE 473

## Day 31

(35 in 201720)

### Student questions

Data Compression

Minimal Spanning  
Tree Intro



Choose the locally best next thing ...

## GREEDY ALGORITHMS



More important than ever ...

## DATA COMPRESSION



### Data (Text) Compression

YOU SAY GOODBYE. I SAY HELLO. HELLO, HELLO. I DON'T KNOW WHY YOU SAY GOODBYE, I SAY HELLO.

#### Letter frequencies

SPACE	17	A	4	U	2
O	12	S	4	W	2
Y	9	I	3	N	2
L	8	D	3	K	1
E	6	COMMA	2	T	1
H	5	B	2	APOSTROPHE	1
PERIOD	4	G	2		



- There are 90 characters altogether.
- How many total bits in the ASCII representation of this string?
- We can get by with fewer bits per character (custom code)
  - How many bits per character? How many for entire message?
  - Do we need to include anything else in the message?
  - How to represent the table?
    1. count
    2. ASCII code for each character **How to do better?**



## Compression algorithm: Huffman encoding

- **Named for David Huffman**
  - [http://en.wikipedia.org/wiki/David\\_A.\\_Huffman](http://en.wikipedia.org/wiki/David_A._Huffman)
  - Invented while he was a graduate student at MIT.
  - Huffman never tried to patent an invention from his work. Instead, he concentrated his efforts on education.
  - In Huffman's own words, "My products are my students."
- **Principles of variable-length character codes:**
  - Less-frequent characters have longer codes
  - No code can be a prefix of another code
- We build a tree (based on character frequencies) that can be used to encode and decode messages



## Variable-length Codes for Characters

- Assume that we have some routines for packing sequences of bits into bytes and writing them to a file, and for unpacking bytes into bits when reading the file
  - Weiss has a very clever approach:
    - **BitOutputStream** and **BitInputStream**
    - methods `writeBit` and `readBit` allow us to logically read or write a bit at a time



## A Huffman code: HelloGoodbye message

```
C:\Personal\Courses\CS-230\java-source>type HelloGoodbyeOneLine
YOU SAY GOODBYE. I SAY HELLO. HELLO, HELLO. I DON'T KNOW WHY YOU SAY GOODBYE, I SAY HELLO.
```

```
C:\Personal\Courses\CS-230\java-source>java HuffmanDS <HelloGoodbyeOneLine
Encoding of  is 00 (frequency was 17, length of code is 2)
Encoding of . is 0100 (frequency was 4, length of code is 4)
Encoding of H is 0101 (frequency was 5, length of code is 4)
Encoding of Y is 011 (frequency was 9, length of code is 3)
Encoding of K is 100000 (frequency was 1, length of code is 6)
Encoding of T is 1000010 (frequency was 1, length of code is 7)
Encoding of ' is 1000011 (frequency was 1, length of code is 7)
Encoding of D is 10001 (frequency was 3, length of code is 5)
Encoding of E is 1001 (frequency was 6, length of code is 4)
Encoding of O is 101 (frequency was 12, length of code is 3)
Encoding of I is 11000 (frequency was 3, length of code is 5)
Encoding of B is 110010 (frequency was 2, length of code is 6)
Encoding of , is 110011 (frequency was 2, length of code is 6)
Encoding of S is 11010 (frequency was 4, length of code is 5)
Encoding of A is 11011 (frequency was 4, length of code is 5)
Encoding of U is 111000 (frequency was 2, length of code is 6)
Encoding of G is 111001 (frequency was 2, length of code is 6)
Encoding of N is 111010 (frequency was 2, length of code is 6)
Encoding of W is 111011 (frequency was 2, length of code is 6)
Encoding of L is 1111 (frequency was 8, length of code is 4)
Total bits required for message: 351
```

Decode a  
"message"

Draw part  
of the Tree



## Build the tree for a smaller message

I 1  
R 1  
N 2  
O 3  
A 3  
T 5  
E 8

- Start with a separate tree for each character (in a priority queue)
- Repeatedly merge the two lowest (total) frequency trees and insert new tree back into priority queue
- Use the Huffman tree to encode NATION.

Huffman codes are provably optimal  
among all single-character codes



## What About the Code Table?

- When we send a message, the code table can basically be just the list of characters and frequencies
  - Why?
- Three or four bytes per character
  - The character itself.
  - The frequency count.
- End of table signaled by 0 for char and count.
- Tree can be reconstructed from this table.
- The rest of the file is the compressed message.



## Huffman Java Code Overview

- This code provides human-readable output to help us understand the Huffman algorithm.
- We will deal with Huffman at the abstract level; "real" code to do actual file compression is found in Weiss chapter 12.
- I am confident that you can figure out the other details if you need them.
- Based on code written by Duane Bailey, in his book *JavaStructures*.
- A great thing about this example is the use of various data structures (Binary Tree, Hash Table, Priority Queue).

I do not want to get caught up in lots of code details in class, so I will give a quick overview; you should read details of the code on your own.

## Some Classes used by Huffman

- **Leaf:** Represents a leaf node in a Huffman tree.
  - Contains the character and a count of how many times it occurs in the text.
- **HuffmanTree:** Each node contains the total weight of all characters in the tree, and either a leaf node or a binary node with two subtrees that are Huffman trees.
  - The contents field of a non-leaf node is never used; we only need the total weight.
  - `compareTo` returns its result based on comparing the total weights of the trees.



## Classes used by Huffman, part 2

- **Huffman:** Contains `main`    **The algorithm:**
  - Count character frequencies and build a list of Leaf nodes containing the characters and their frequencies
  - Use these nodes to build a sorted list (treated like a priority queue) of single-character Huffman trees
  - **do**
    - Take two smallest (in terms of total weight) trees from the sorted list
    - Combine these nodes into a new tree whose total weight is the sum of the weights of the new tree
    - Put this new tree into the sorted list
  - **while there is more than one tree left**

The one remaining tree will be an optimal tree for the entire message



## Leaf node class for Huffman Tree

```
class Leaf { // Leaf node of a Huffman tree.

    char ch; // the character represented by this node
    int frequency; // frequency of char in the message.

    public Leaf(char c, int freq) {
        ch = c;
        frequency = freq;
    }
}
```

The code on this slide (and the next four slides) produces the output shown on the *A Huffman code: HelloGoodbye* message slide.



## Highlights of the HuffmanTree class

```
class HuffmanTree implements Comparable<HuffmanTree> {
    BinaryNode root; // root of tree
    int totalWeight; // weight of tree
    static int totalBitsNeeded;
    // bits needed to represent entire message
    // (not including code table).

    public HuffmanTree(Leaf e) {
        root = new BinaryNode(e, null, null);
        totalWeight = e.frequency;
    }

    public HuffmanTree(HuffmanTree left, HuffmanTree right) {
        // pre: left and right non-null
        // post: merge two trees together and add their weights
        this.totalWeight = left.totalWeight + right.totalWeight;
        root = new BinaryNode(null, left.root, right.root);
    }

    public int compareTo(HuffmanTree other) {
        return (this.totalWeight - other.totalWeight);
    }
}
```



## Printing a HuffmanTree

```
public void print() {
    // post: print out strings associated with characters in tree
    totalBitsNeeded = 0;
    print(this.root, "");
    System.out.println("Total bits required for message: "
        + totalBitsNeeded);
}

protected static void print(BinaryNode r, String representation) {
    // post: print out strings associated with chars in tree r,
    //         prefixed by representation
    if (r.getLeft() != null) {
        // interior node
        print(r.getLeft(), representation + "0"); // append a 0
        print(r.getRight(), representation + "1"); // append a 1
    } else { // leaf; print encoding
        Leaf e = (Leaf) r.getElement();
        System.out.println("Encoding of " + e.ch + " is " + representation
            + " (frequency was " + e.frequency + ", length of code is "
            + representation.length() + ")");
        totalBitsNeeded += (e.frequency * representation.length());
    }
}
```

## Highlights of Huffman class part 1

```
import java.util.HashMap;
import java.util.Scanner;
import java.util.PriorityQueue;

public class Huffman {

    public static void main(String args[]) throws Exception {
        Scanner sc = new Scanner(System.in);
        HashMap<Character, Integer> freq = new HashMap<Character, Integer>();
        String oneLine; // current input line.
        // First read the data and count characters
        // Go through the input line, one character at a time.
        System.out.println("Message to be encoded (CTRL-Z to end):");
        while (sc.hasNext()) {
            oneLine = sc.next();
            for (int i = 0; i < oneLine.length(); i++) {
                char c = oneLine.charAt(i);
                if (freq.containsKey(c))
                    freq.put(c, freq.get(c) + 1);
                else
                    // first time we've seen c
                    freq.put(c, 1);
            }
        }
    }
}
```



## Remainder of the main() method

```
// Now the table of frequencies of each character is complete.
// insert each character into a single-node Huffman tree
PriorityQueue<HuffmanTree> treeQueue = new PriorityQueue<HuffmanTree>();
for (char c : freq.keySet())
    treeQueue.add(new HuffmanTree(new Leaf(c, freq.get(c))));

HuffmanTree smallest, secondSmallest;

// merge trees in pairs until only one tree remains
while (true) {
    smallest = treeQueue.poll();
    secondSmallest = treeQueue.poll();
    if (secondSmallest == null)
        break;
    // add bigger tree containing both to the sorted list.
    treeQueue.add(new HuffmanTree(smallest, secondSmallest));
}

// print the only tree that is left.
smallest.print();
}
```

## Summary

- The Huffman code is provably optimal among all single-character codes for a given message.
- Going farther:
  - Look for frequently occurring sequences of characters and make codes for them as well.
- Compression for specialized data (such as sound, pictures, video).
  - Okay to be "lossy" as long as a person seeing/hearing the decoded version can barely see/hear the difference.

**Minimal Spanning Tree (MST) for a connected network G:**  
A tree that contains every node in G

Kruskal and Prim algorithms (both are greedy)

## **ALGORITHMS FOR FINDING A MINIMAL SPANNING TREE**



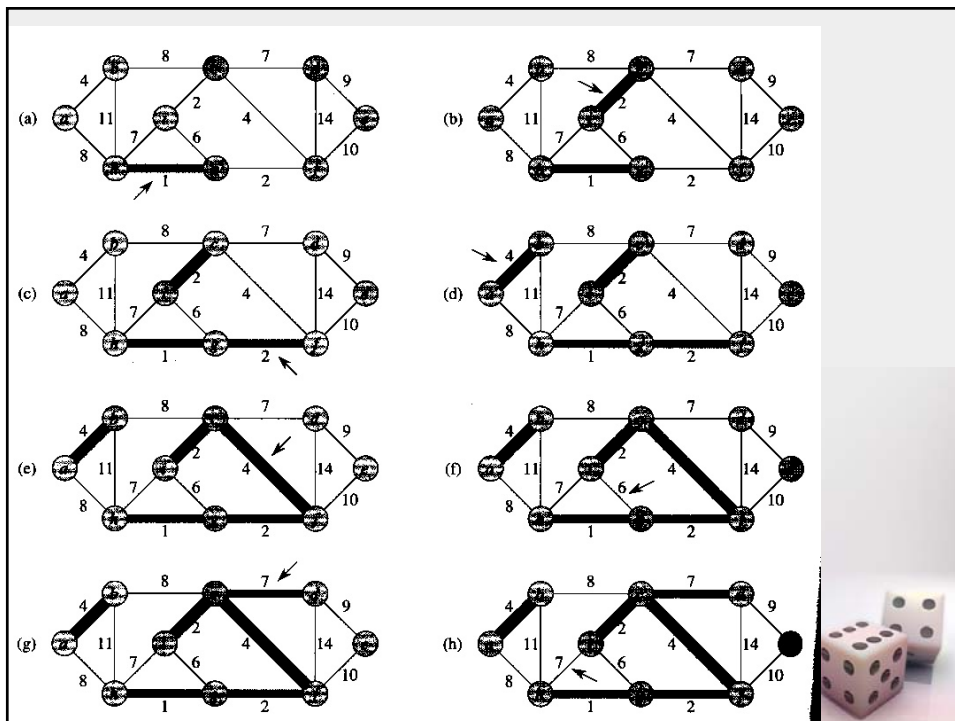
## **Minimal Spanning Tree Definition**

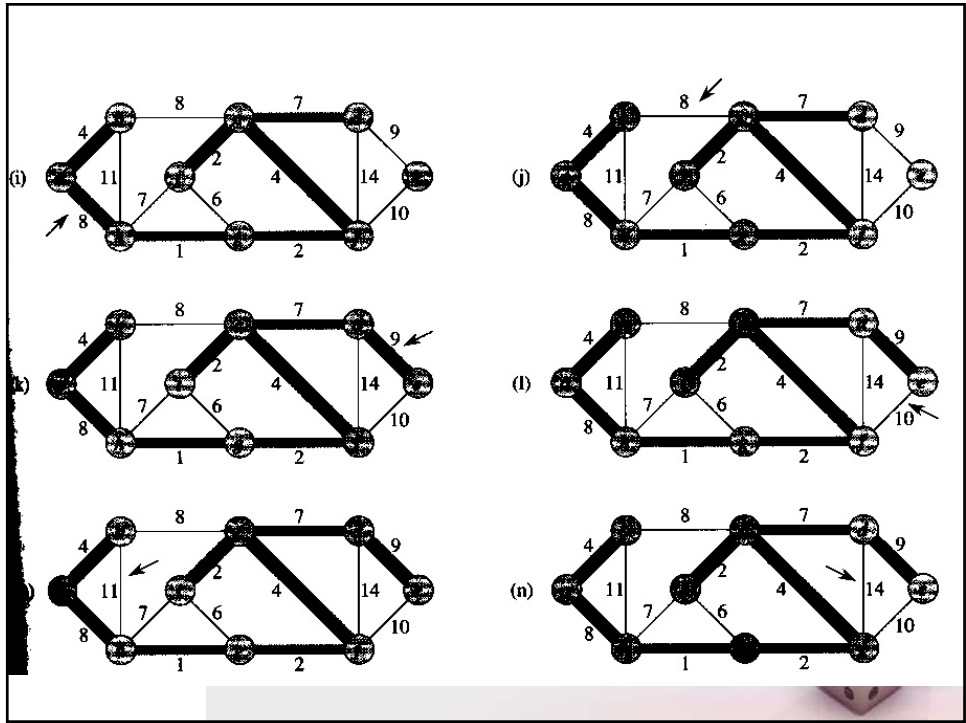
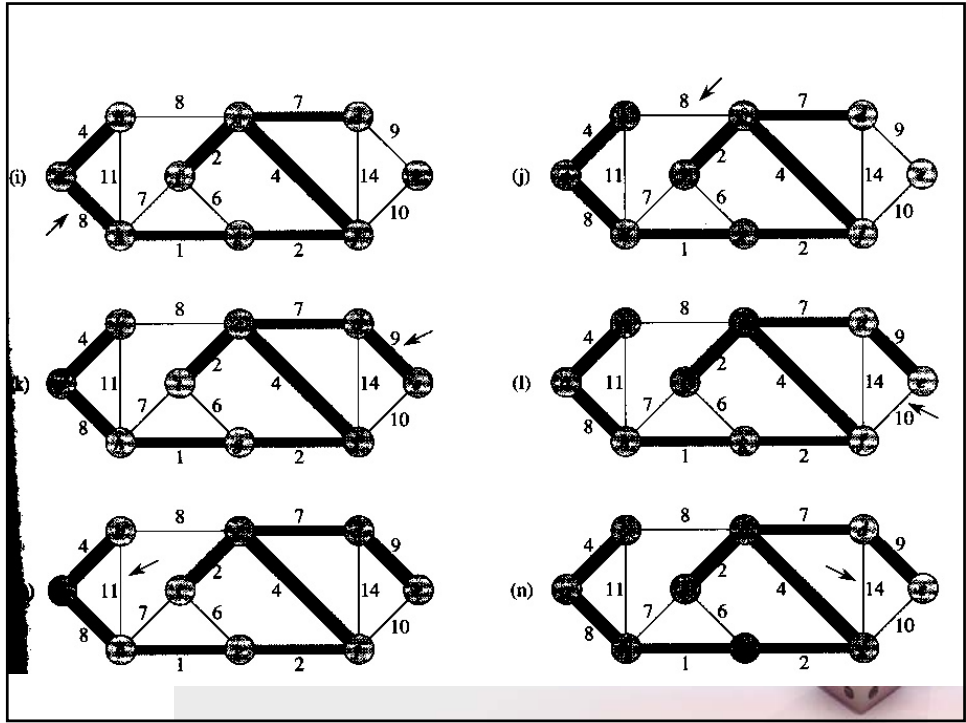
- Let  $G$  be a **network**: a connected graph which has a number (weight) associated with each edge
- A **spanning tree** is a connected subgraph of  $G$  that contains all vertices of  $G$  and is a tree
- Among all spanning trees of  $G$ , a **minimal spanning tree** is one whose total weight is minimal.



## Kruskal's algorithm

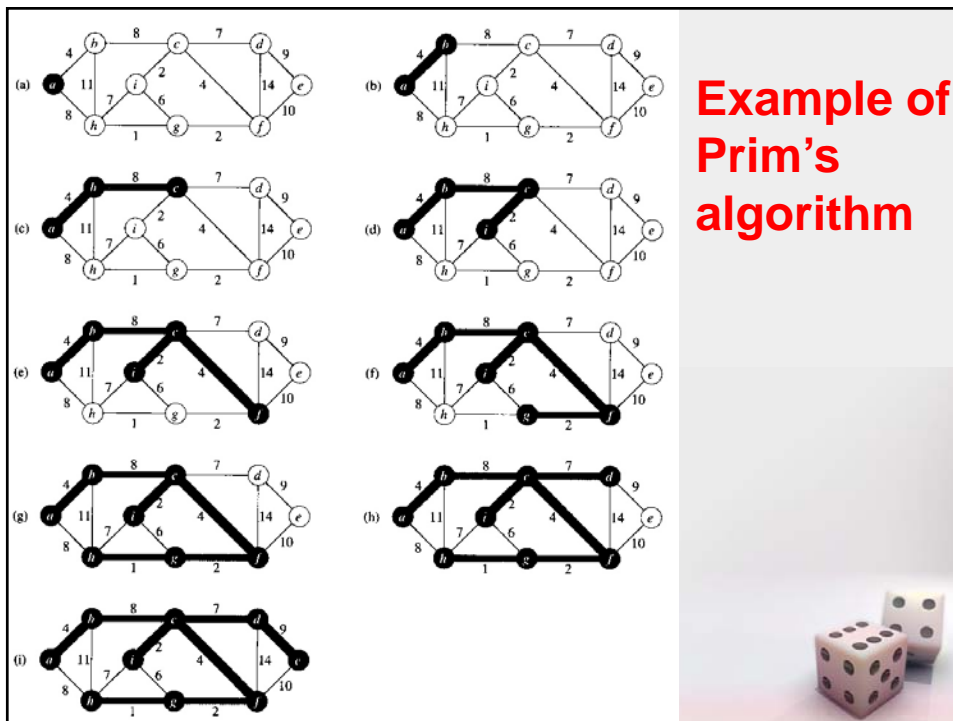
- To find a MST (minimal Spanning Tree):
- Start with a graph  $T$  containing all  $n$  of  $G$ 's vertices and none of its edges.
- for  $i = 1$  to  $n - 1$ :
  - Among all of  $G$ 's edges that can be added without creating a cycle, add to  $T$  an edge that has minimal weight.
  - Details of Data Structures later





## Prim's algorithm

- Start with T as a single vertex of G (which is a MST for a single-node graph).
- for  $i = 1$  to  $n - 1$ :
  - Among all edges of G that connect a vertex in T to a vertex that is not yet in T, add a minimum-weight edge (and the vertex at the other end of that edge).
  - Details of Data Structures later



## Correct?

- These algorithms seem simple enough, but do they really produce a MST?
- We examine a lemma that is the crux of both proofs.
- It is subtle, but once we have it, the proofs are fairly simple.



## MST lemma

- Let  $G$  be a weighted connected graph,
- let  $T$  be any MST of  $G$ ,
- let  $G'$  be any nonempty subgraph of  $T$ , and
- let  $C$  be any connected component of  $G'$ .
- Then:
  - If we add to  $C$  an edge  $e=(v,w)$  that has minimum-weight among all edges that have one vertex in  $C$  and the other vertex not in  $C$ ,
  - $G$  has an MST that contains the union of  $G'$  and  $e$ .

[WLOG,  $v$  is the vertex of  $e$  that is in  $C$ , and  $w$  is not in  $C$ ]

**Summary:** If  $G'$  is a subgraph of an MST, so is  $G' \cup \{e\}$

