CSSE 230 Day 16

Data Compression

Exhaustive search, backtracking, object-oriented Queens

Check out from SVN:
- Queens
- Huffman-Bailey-JFC

Questions

Agenda

- Teams for EditorTrees project
- Greedy Algorithms
- Data Compression
- Huffman's algorithm
- Exhaustive search, backtracking, and object-oriented queens

Exam 2
Tuesday, May 8: 7:00-9:00 PM

EditorTrees project

- Brief description
  Meet your team

EditorTrees project

- In general, *implementation* of a Data Structure is separate from *application*.
- Most CSSE 230 projects have used existing data structures to create an application
- In this project you will create an efficient data structure that could be used for in a text editor.
- But you will not create the GUI application that uses it.
- **EditTree**: A height-balanced (but not AVL) binary tree with rank. Insertion and deletion are by position, not by natural ordering of the inserted elements.
- Log N Operations include
  - Insert, delete, find, concatenate, split, height, size
- Node fields include balance code and rank.

EditorTrees Teams 201230

- csse230-201230-ET-11,amesen,elswicwj,piliseal
- csse230-201230-ET-12,eubankct,harbisjs,murphysw
- csse230-201230-ET-13,goldthea,postcn,rujirasl
- csse230-201230-ET-14,paulbi,woolleld,newmansr
- csse230-201230-ET-15,huangz,namdw,koestedj
- csse230-201230-ET-16,maglioms,mehrinla,rudichza
- csse230-201230-ET-17,mcdonabj,morrista,millerns
- csse230-201230-ET-18,nuanests,shahdk,timaeudg
- csse230-201230-ET-19,sanderej,semmeln,weirjm
- csse230-201230-ET-20,mccullwc,yuhasmj
- csse230-201230-ET-21,bollivbd,davelldf,memeriaj
- csse230-201230-ET-22,faulknks,scroggd,spryct
- csse230-201230-ET-23,fendrirj,hopwoocp,pohltm
- csse230-201230-ET-24,haydr,lawrener,tilleraj
- csse230-201230-ET-25,roetkefj,stewarzt,uphusar
- csse230-201230-ET-26,gartzkds,minardar,ewertbe
- csse230-201230-ET-27,iwemamj,modivr,qinz
- csse230-201230-ET-28,zhangz,taylorem,watterlm
- csse230-201230-ET-29,lius,weil,yuhasem
- csse230-201230-ET-30,meyermc

Meet your partners to plan when you will meet to begin work.

Suggestion: Meet before tomorrow to discuss the project requirements. Formulate a list questions to ask during Tuesday's class.

Whether or not you meet before Tuesday, read the EditorTrees requirements and come with questions.

Greedy algorithms

Q1

- Whenever a choice is to be made, pick the one that seems optimal for the moment, without taking future choices into consideration
  - Once each choice is made, it is irrevocable
- For example, a greedy Scrabble player will simply maximize her score for each turn, never saving any "good" letters for possible better plays later
  - **Doesn't necessarily optimize score for entire game**

Greedy Chess Strategy

- Take a piece or pawn whenever you will not lose a piece or pawn (or will lose one of lesser value) on the next turn
- Not a good strategy for this game either.
- But there are some problems for which greedy algorithms produce optimal solutions.

Data (Text) Compression

Q2-3

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 (20 different).
- 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

Q4-5

- **Named for David 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

- **RECAP: Principles for determining a scheme for creating character codes:**
  1. Less-frequent characters have longer codes so that more-frequent characters can have shorter codes
  2. No code can be a prefix of another code
     - Why is this restriction necessary?
- 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

Q6-9

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?

Q10

- When we send a message, the code table can basically be just the list of characters and frequencies
  - Why?

Huffman Java Code Overview

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

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 implements Comparable**:
  Each node contains the total weight of all characters in its subtree, 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 PriorityQueue of single-character Huffman trees
  - **do**
    - Take two smallest (in terms of total weight) trees from the PQ
    - 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 PQ

  while there is more than one tree left

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

Code Details – several slides

- These are mainly here so that
  - You can see an overview of the most important parts of the code before looking at the code on-line.

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 this
                   // character in message.

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

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 are 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);
  }
```

On this slide:
- fields
- constructors
- compareTo

Printing a HuffmanTree

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

protected static void print(BinaryNode r,
                            String representation) {
  //  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 its code
    Leaf e = (Leaf) r.getElement();
    System.out.println("Encoding of " + e.ch + " is " +
       representation + " (frequency was " +  e.frequency +
       ", length of code is " +  representation.length() + ")");
    totalBits += (e.frequency * representation.length());
  }
}
```

Highlights of Huffman class, part 1

```
public static void main(String args[]) throws Exception {

   Scanner sc = new Scanner(System.in);
   HashMap<Character, Integer> freq =
                         new HashMap<Character,Integer>();
      // List of characters and their frequencies in the //
      // message that we are encoding.
   String oneLine;  // current input line.

   // First read the data and count frequencies
   // Go through each 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);
      }
   }
```

Highlights of Huffman class, part 2

```
// Now the table of frequencies is complete.
// put each character into its own Huffman tree (leaf node)

PriorityQueue<HuffmanTree> treeQueue =
      new  PriorityQueue<HuffmanTree>();
for (char c : freq.keySet())
   treeQueue.add(new HuffmanTree(new Leaf(c, freq.get(c))));

// build the tree bottom up
HuffmanTree smallest, secondSmallest;
// merge trees in pairs until only one tree remains
while (true) {
  smallest = treeQueue.poll();
  secondSmallest = treeQueue.poll();
  if (secondSmallest == null) break; // tree is complete
    // add bigger tree containing both to the sorted list.
  treeQueue.add(new HuffmanTree(smallest, secondSmallest));
}
 // print the only tree left in the PQ of Huffman trees.
 smallest.print();
}
```

Representing the Code Table

- 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.

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.

Exhaustive Search and Backtracking

» A taste of artificial intelligence

Check out Queens from SVN

Exhaustive search

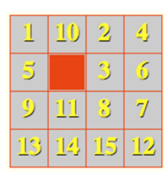
- Given: a (large) set of possible solutions to a problem

The “search space”

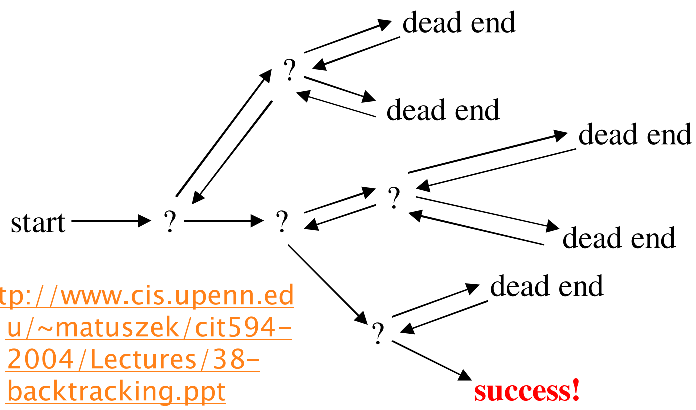
- Goal: Find all solutions (or an optimal solution) from that set
- Questions we ask:
  - How do we represent the possible solutions?
  - How do we organize the search?
  - Can we avoid checking some obvious non-solutions?

In backtracking, we always try to extend a partial solution

- Examples: solving a maze, the “15” puzzle.
- Taken from:
  - http://www.cis.upenn.edu/~matuszek/cit594-2004/Lectures/38-backtracking.ppt



In backtracking, we picture the search as a tree and explore it using a pre-order traversal

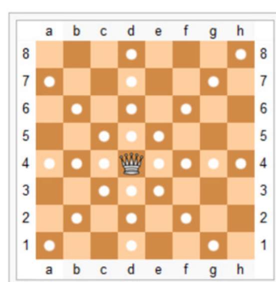


http://www.cis.upenn.edu/~matuszek/cit594-2004/Lectures/38-backtracking.ppt

The non-attacking chess queens problem is a famous example

- In how many ways can N chess queens be placed on an NxN grid, so that none of the queens can attack any other queen?
- I.e. there are not two queens on the same row, same column, or same diagonal.

- There is no "formula" for generating a solution.



Possible moves of the queen are shown

http://en.wikipedia.org/wiki/Queen_(chess)

With a partner, discuss "possible solution" search strategies

- In how many ways can N chess queens be placed on an NxN grid, so that none of the queens can attack any other queen?
  - I.e. no two queens on the same row, same column, or same diagonal.

Two minutes
No Peeking!

Search Space Possibilities 1/5

Q11

- Very naive approach. Perhaps stupid is a better word!
  There are N queens, N^2 squares.
- For each queen, try every possible square, allowing the possibility of multiple queens in the same square.
  - Represent each potential solution as an N-item array of pairs of integers (a row and a column for each queen).
  - Generate all such arrays (you should be able to write code that would do this) and check to see which ones are solutions.
  - Number of possibilities to try in the NxN case:
  - Specific number for N=8:

281,474,976,710,656

Search Space Possibilities 2/5

Slight improvement. There are N queens, N^2 squares. For each queen, try every possible square, notice that we can't have multiple queens on the same square.

- Represent each potential solution as an N-item array of pairs of integers (a row and a column for each queen).
- Generate all such arrays and check to see which ones are solutions.
- Number of possibilities to try in NxN case:
- Specific number for N=8:

178,462,987,637,760

(vs. 281,474,976,710,656)

Search Space Possibilities 3/5

- **Slightly better approach.** There are N queens, N columns. If two queens are in the same column, they will attack each other. Thus there must be exactly one queen per column.
- Represent a potential solution as an N-item array of integers.
  - Each array position represents the queen in one column.
  - The number stored in an array position represents the row of that column's queen.
  - Show array for 4x4 solution.
    - Generate all such arrays and check to see which ones are solutions.
    - Number of possibilities to try in NxN case:
    - Specific number for N=8: 16,777,216

Search Space Possibilities 4/5

- **Still better approach** There must also be exactly one queen per row.

- Represent the data just as before, but notice that the data in the array is a set!
  - Generate each of these and check to see which ones are solutions.
  - **How to generate?** A good thing to think about.
  - Number of possibilities to try in NxN case:
  - Specific number for N=8:

40,320

Search Space Possibilities 5/5

- **Backtracking solution**
- Instead of generating all permutations of N queens and checking to see if each is a solution, we generate "partial placements" by placing one queen at a time on the board
- Once we have successfully placed k<N queens, we try to *extend* the partial solution by placing a queen in the next column.
- When we extend to N queens, we have a solution.

Experimenting with 8 x 8 Case

- Play the game:
  - http://homepage.tinet.ie/~pdpals/8queens.htm
- See the solutions:
  - http://www.dcs.ed.ac.uk/home/mlj/demos/queens

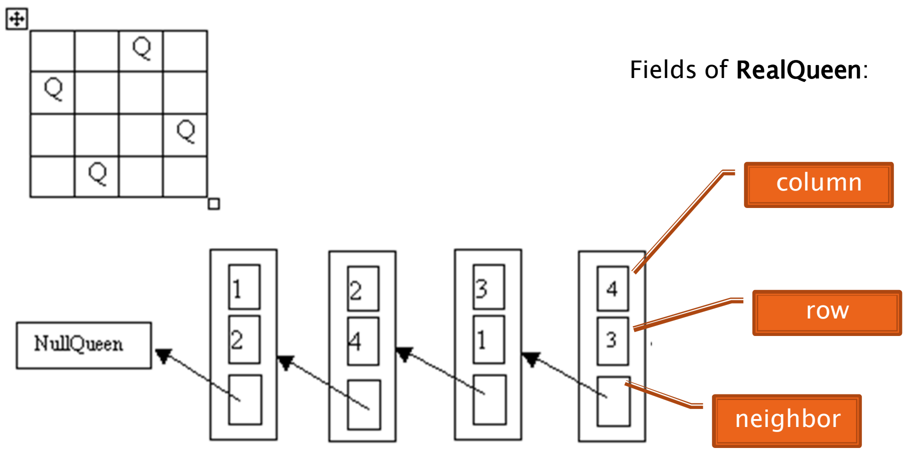
Program output:

```
>java RealQueen 5
SOLUTION:  1 3 5 2 4
SOLUTION:  1 4 2 5 3
SOLUTION:  2 4 1 3 5
SOLUTION:  2 5 3 1 4
SOLUTION:  3 1 4 2 5
SOLUTION:  3 5 2 4 1
SOLUTION:  4 1 3 5 2
SOLUTION:  4 2 5 3 1
SOLUTION:  5 2 4 1 3
SOLUTION:  5 3 1 4 2
```

The Linked List of Queen Objects

Q12–15

- Board configuration represented by a linked list of **Queen** objects



Designed by Timothy Budd
http://web.engr.oregonstate.edu/~budd/Books/oopintro3e/info/slides/chap06/java.htm

Outline of the algorithm

- Each queen sends messages directly to its immediate neighbor to the left (and recursively to all of its left neighbors)
- Return value provides information concerning all of the left neighbors:
- Example: **neighbor.canAttack(currentRow, col)**
  - Message goes to the immediate neighbor, but the real question to be answered by this call is
  - "Hey, neighbors, can any of you attack me if I place myself on this square of the board?"

Queen Methods

16-20

- `findFirst()`
- `findNext()`
- `canAttack(int row, int col)`

Your job (part of WA6):

Understand the job of each of these methods

Javadoc from the Queen interface can help

Fill in the (recursive) details in the RealQueen class

Debug

More details on next slide

More algorithm outline

1. Queen asks its neighbors to find the first position in which none of them attack each other
   - Found? Then queen tries to position itself so that it cannot be attacked.
2. If the rightmost queen is successful, then a solution has been found! The queens cooperate in recording it.
3. Otherwise, the queen asks its neighbors to find the next position in which they do not attack each other
4. When the queens get to the point where there is no next non-attacking position, all solutions have been found and the algorithm terminates