CSC 427: Data Structures and Algorithm Analysis

Fall 2011

Greedy algorithms

- greedy algorithms
  examples: optimal change, job scheduling
- Dijkstra's algorithm (shortest path)
- Huffman codes (data compression)
- greedy vs. backtracking (DFS)
  examples: N-queens, 2-D gels, Boggle

Greedy algorithms

the greedy approach to problem solving involves making a sequence of choices/actions, each of which simply looks best at the moment

local view: choose the locally optimal option
hopefully, a sequence of locally optimal solutions leads to a globally optimal solution

example: optimal change

- given a monetary amount, make change using the fewest coins possible

amount = 16¢ coins?

amount = 96¢ coins?

Example: greedy change

while the amount remaining is not 0:

- select the largest coin that is ≤ the amount remaining
- add a coin of that type to the change
- subtract the value of that coin from the amount remaining

e.g., 96¢ = 50¢ + 25¢ + 10¢ + 10¢ + 1¢

will this greedy algorithm always yield the optimal solution?

for U.S. currency, the answer is YES

for arbitrary coin sets, the answer is NO

- suppose the U.S. Treasury added a 12¢ coin

GREEDY: 16¢ = 12¢ + 1¢ + 1¢ + 1¢ + 1¢ (5 coins)

OPTIMAL: 16¢ = 10¢ + 5¢ + 1¢ (3 coins)

3

Example: job scheduling

suppose you have a collection of jobs to execute and know their lengths

- want to schedule the jobs so as to *minimize* waiting time

Job 1:	5 minutes	Schedule 1-2-3: 0 + 5 + 15 = 20 minutes waiting
Job 2:	10 minutes	Schedule 3-2-1: 0 + 4 + 14 = 18 minutes waiting
Job 3:	4 minutes	Schedule 3-1-2: 0 + 4 + 9 = 13 minutes waiting

GREEDY ALGORITHM: do the shortest job first

i.e., while there are still jobs to execute, schedule the shortest remaining job

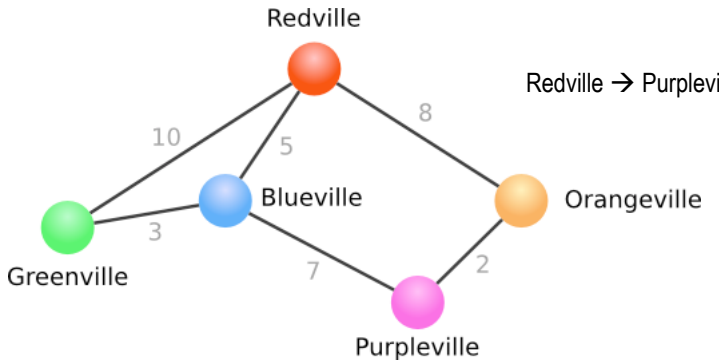
does the greedy approach guarantee the optimal schedule? efficiency?

4

Application: shortest path problem

consider the general problem of finding the shortest path between two nodes in a graph

- flight planning and word ladder are examples of this problem
  - in these cases, edges have uniform cost (shortest path = fewest edges)
- if we allow non-uniform edges, want to find lowest cost/shortest distance path



example from http://www.algolist.com/Dijkstra's_algorithm

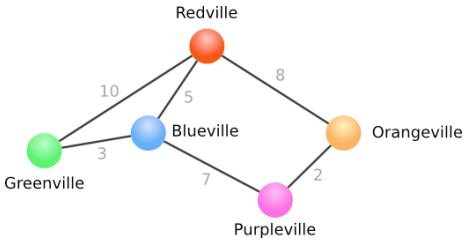
5

Modified BFS solution

we could modify the BFS approach to take cost into account

- instead of adding each newly expanded path to the end (i.e., queue), add in order of path cost (i.e., priority queue)

```
[ [Redville]:0 ]

[ [Redville, Blueville]:5,
  [Redville, Orangeville]:8,
  [Redville, Greenville]:10 ]

[ [Redville, Orangeville]:8,
  [Redville, Blueville, Greenville]:8,
  [Redville, Greenville]:10,
  [Redville, Blueville, Purpleville]:12 ]

[ [Redville, Blueville, Greenville]:8,
  [Redville, Greenville]:10,
  [Redville, Orangeville, Purpleville]:10,
  [Redville, Blueville, Purpleville]:12 ]

[ [Redville, Greenville]:10,
  [Redville, Orangeville, Purpleville]:10,
  [Redville, Blueville, Purpleville]:12 ]

[ [Redville, Orangeville, Purpleville]:10,
  [Redville, Blueville, Purpleville]:12,
  [Redville, Greenville, Blueville]:13 ]
```



note: as before, requires lots of memory to store all the paths

HOW MANY?

6

Dijkstra's algorithm

alternatively, there is a straightforward greedy algorithm for shortest path

Dijkstra's algorithm

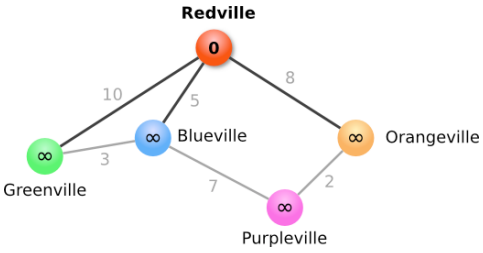
1. Begin with the start node. Set its value to 0 and the value of all other nodes to infinity. Mark all nodes as unvisited.
2. For each unvisited node that is adjacent to the current node:
   a) If (value of current node + value of edge) < (value of adjacent node), change the value of the adjacent node to this value.
   b) Otherwise leave the value as is.
3. Set the current node to visited.
4. If unvisited nodes remain, select the one with smallest value and go to step 2.
5. If there are no unvisited nodes, then DONE.

this algorithm is $O(N^2)$, requires only $O(N)$ additional storage

7

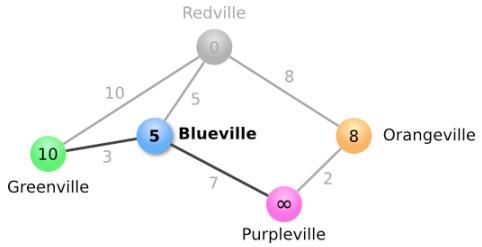
Dijkstra's algorithm: example

suppose want to find shortest path from Redville to Purpleville



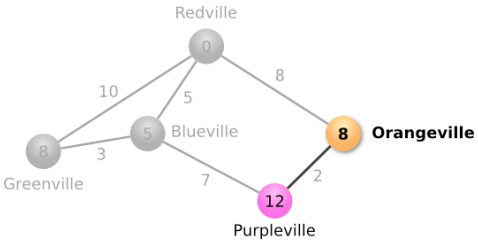
1. Begin with the start node. Set its value to 0 and the value of all other nodes to infinity. Mark all nodes as unvisited

2. For each unvisited node that is adjacent to the current node:
   a) If (value of current node + value of edge) < (value of adjacent node), change the value of the adjacent node to this value.
   b) Otherwise leave the value as is.
3. Set the current node to visited.



8

Dijkstra's algorithm: example cont.



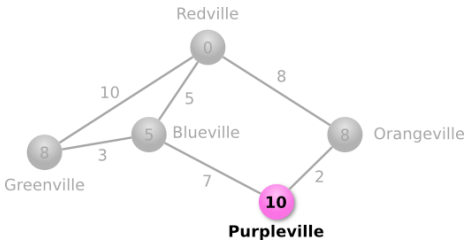
4. If unvisited nodes remain, select the one with smallest value and go to step 2.

Blueville: set Greenville to 8 and Purpleville to 12; mark as visited.
Greenville: no unvisited neighbors; mark as visited.

4. If unvisited nodes remain, select the one with smallest value and go to step 2.

Orangeville: set Purpleville to 10; mark as visited.

5. If there are no unvisited nodes, then DONE.



With all nodes labeled, can easily construct the shortest path – HOW?

9

Another application: data compression

in a multimedia world, document sizes continue to increase

- a 6 megapixel digital picture is 2-4 MB
- an MP3 song is ~3-6 MB
- a full-length MPEG movie is ~800 MB

storing multimedia files can take up a lot of disk space

- perhaps more importantly, downloading multimedia requires significant bandwidth

it could be a lot worse!

- image/sound/video formats rely heavily on data compression to limit file size
  e.g., if no compression, 6 megapixels * 3 bytes/pixel = ~18 MB
- the JPEG format provides 10:1 to 20:1 compression without visible loss

10

Audio, video, & text compression

audio & video compression algorithms rely on domain-specific tricks

- lossless image formats (GIF, PNG) recognize repeating patterns (e.g. a sequence of white pixels) and store as a group
- lossy image formats (JPG, XPM) round pixel values and combine close values
- video formats (MPEG, AVI) take advantage of the fact that little changes from one frame to next, so store initial frame and changes in subsequent frames
- audio formats (MP3, WAV) remove sound out of hearing range, overlapping noises

what about text files?

- in the absence of domain-specific knowledge, can't do better than a fixed-width code
  e.g., ASCII code uses 8-bits for each character

```
'0': 00110000    'A': 01000001    'a': 01100001
'1': 00110001    'B': 01000010    'b': 01100010
'2': 00110010    'C': 01000011    'c': 01100011
 .                .                .
 .                .                .
 .                .                .
```

11

Fixed- vs. variable-width codes

suppose we had a document that contained only the letters a-f

- with a fixed-width code, would need 3 bits for each character

```
a   000        d   011
b   001        e   100
c   010        f   101
```

- if the document contained 100 characters, 100 * 3 = 300 bits required

however, suppose we knew the distribution of letters in the document

```
a:45,  b:13,  c:12,  d:16,  e:9,  f:5
```

- can customize a variable-width code, optimized for that specific file

```
a   0          d   111
b   101        e   1101
c   100        f   1100
```

- requires only 45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4 = 224 bits

12

Huffman codes

Huffman compression is a technique for constructing an optimal* variable-length code for text

*optimal in that it represents a specific file using the fewest bits (among all symbol-for-symbol codes)

Huffman codes are also known as *prefix codes*

- no individual code is a prefix of any other code

```
a   0          d   111
b   101        e   1101
c   100        f   1100
```

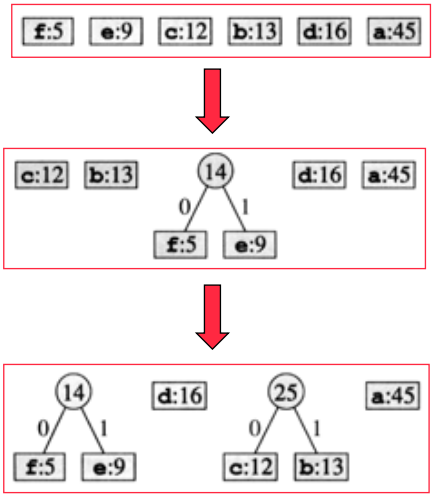
- this makes decompression unambiguous: `101011111000100110 1`
- *note:* since the code is specific to a particular file, it must be stored along with the compressed file in order to allow for eventual decompression

13

Huffman trees

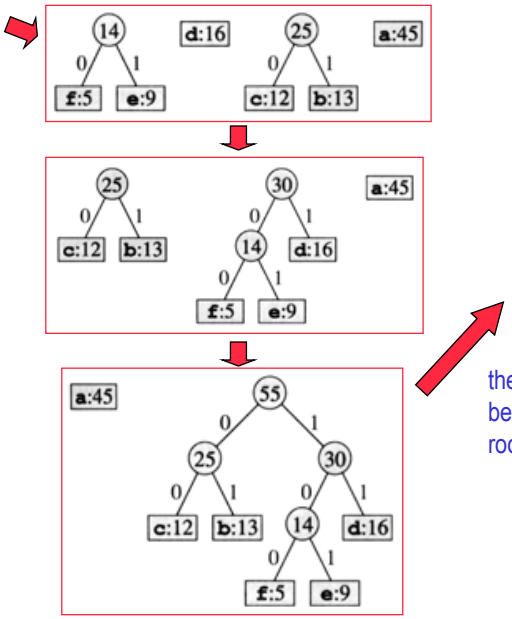
to construct a Huffman code for a specific file, utilize a greedy algorithm to construct a *Huffman tree*:

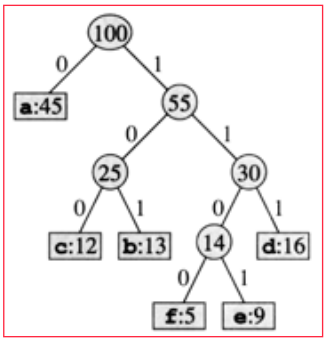
1. process the file and count the frequency for each letter in the file
2. create a single-node tree for each letter, labeled with its frequency
3. repeatedly,
   a. pick the two trees with smallest root values
   b. combine these two trees into a single tree whose root is labeled with the sum of the two subtree frequencies
4. when only one tree remains, can extract the codes from the Huffman tree by following edges from root to each leaf (left edge = 0, right edge = 1)



14

Huffman tree construction (cont.)





the code corresponding to each letter can be read by following the edges from the root: left edge = 0, right edge = 1

```
a: 0        d: 111
b: 101      e: 1101
c: 100      f: 1100
```

15

Huffman code compression

note that at each step, need to pick the two trees with smallest root values

- perfect application for a priority queue (i.e., a min-heap)

- store each single-node tree in a priority queue
- repeatedly,
  remove the two min-value trees from the priority queue
  combine into a new tree with sum at root and insert back into priority queue

while designed for compressing text, it is interesting to note that Huffman codes are used in a variety of applications

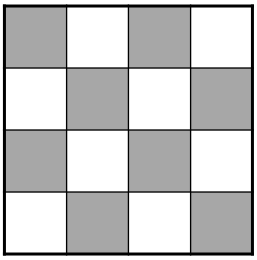
- the last step in the JPEG algorithm, after image-specific techniques are applied, is to compress the resulting file using a Huffman code
- similarly, Huffman codes are used to compress frames in MPEG (MP4)

16

Greed is good?

IMPORTANT: the greedy approach is not applicable to all problems

- but when applicable, it is very effective (no planning or coordination necessary)



GREEDY approach for N-Queens: start with first row, find a valid position in current row, place a queen in that position then move on to the next row

since queen placements are not independent, local choices do not necessarily lead to a global solution

GREEDY does not work – need a more holistic approach

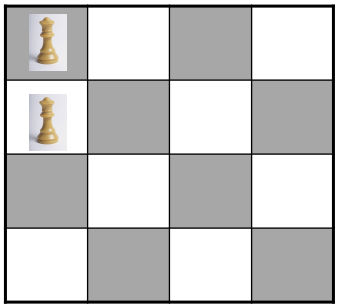
17

Generate & test?

recall the generate & test solution to N-queens

- systematically generate every possible arrangement
- test each one to see if it is a valid solution

$\binom{16}{4}$ = 1,820 arrangements



fortunately, we can do better if we recognize that choices can constrain future choices

- e.g., any board arrangement with a queen at (1,1) and (2,1) is invalid
- no point in looking at the other queens, so can eliminate 16 boards from consideration
- similarly, queen at (1,1) and (2,2) is invalid, so eliminate another 16 boards

18

Backtracking

backtracking is a smart way of doing generate & test

- view a solution as a sequence of choices/actions (*similar to* GREEDY)
- when presented with a choice, pick one (*similar to* GREEDY)
- however, reserve the right to change your mind and backtrack to a previous choice (*unlike* GREEDY)

- you must remember alternatives:
  if a choice does not lead to a solution, back up and try an alternative

- eventually, backtracking will find a solution or exhaust all alternatives

backtracking is essentially depth first search

- add ability to prune a path as soon as we know it can't succeed
- when that happens, back up and try another path

19

N-Queens psuedocode

```
/**
 * Fills the board with queens starting at specified row
 * (Queens have already been placed in rows 0 to row-1)
 */
private boolean placeQueens(int row) {
    if (ROW EXTENDS BEYOND BOARD) {
        return true;
    }
    else {
        for (EACH COL IN ROW) {
            if ([ROW][COL] IS NOT IN JEOPARDY FROM EXISTING QUEENS) {
                ADD QUEEN AT [ROW][COL]

                if (this.placeQueens(row+1)) {
                    return true;
                }
                else {
                    REMOVE QUEEN FROM [ROW][COL]
                }
            }
        }
        return false;
    }
}
```

if row > board size, then all queens have been placed already – return true

place a queen in available column

if can recursively place the remaining queens, then done

if not, remove the queen just placed and continue looping to try other columns

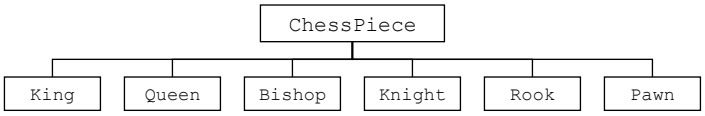
return false if cannot place remaining queens

20

Chessboard class

we could define a class hierarchy for chess pieces

- `ChessPiece` is an abstract class that specifies the common behaviors of pieces
- `Queen`, `Knight`, `Pawn`, … are derived from `ChessPiece` and implement specific behaviors




```
public class ChessBoard {
  private ChessPiece[][] board;                          // 2-D array of chess pieces
  private int pieceCount;                                // number of pieces on the board

  public ChessBoard(int size) {…}                        // constructs size-by-size board
  public ChessPiece get(int row, int col) {…}            // returns piece at (row,col)
  public void remove(int row, int col) {…}               // removes piece at (row,col)
  public void add(int row, int col, ChessPiece p) {…}    // places a piece, e.g., a queen,
                                                         //   at (row,col)
  public boolean inJeopardy(int row, int col) {..}       // returns true if (row,col) is
                                                         //   under attack by any piece
  public int numPieces() {…}                             // returns number of pieces on board
  public int size() {…}                                  // returns the board size
  public String toString() {…}                           // converts board to String
}
```

21

Backtracking N-queens

```
public class NQueens {
  private ChessBoard board;

  . . .

  /**
   * Fills the board with queens.
   */
  public boolean placeQueens() {
    return this.placeQueens(0);
  }


  /**
   * Fills the board with queens starting at specified row
   * (Queens have already been placed in rows 0 to row-1)
   */
  private boolean placeQueens(int row) {
    if (row >= this.board.size()) {
        return true;
    }
    else {
        for (int col = 0; col < this.board.size(); col++) {
            if (!this.board.inJeopardy(row, col)) {
                this.board.add(row, col, new Queen());

                if (this.placeQueens(row+1)) {
                    return true;
                }
                else {
                    this.board.remove(row, col);
                }
            }
        }
        return false;
    }
  }
}
```

in an NQueens class, will have a ChessBoard field and a method for placing the queens

- placeQueens calls a helper method with a row # parameter

BASE CASE: if all queens have been placed, then done.

OTHERWISE: try placing queen in the row and recurse to place the rest

note: if recursion fails, must remove the queen in order to backtrack

note similarity to earlier DFS implementation

22

Why does backtracking work?

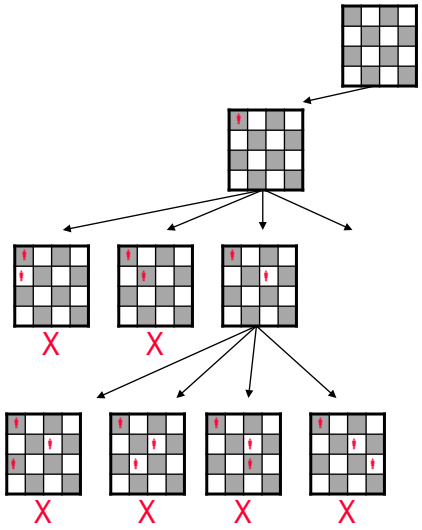
backtracking burns no bridges – all choices are reversible

backtracking provides a systematic way of trying all paths (sequences of choices) until a solution is found

- assuming the search tree is finite, will eventually find a solution or exhaust the entire search space

backtracking is different from generate & test in that choices are made sequentially

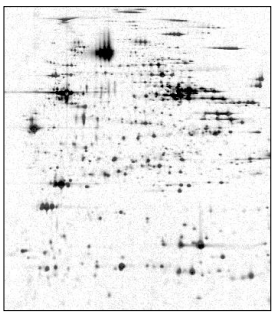
- earlier choices constrain later ones
- can avoid searching entire branches



Another example: blob count

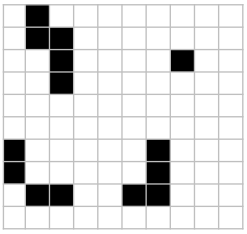
application: 2-D gel electrophoresis

- biologists use electrophoresis to produce a gel image of cellular material
- each "blob" (contiguous collection of dark pixels) represents a protein
- identify proteins by matching the blobs up with another known gel image



we would like to identify each blob, its location and size

- location is highest & leftmost pixel in the blob
- size is the number of contiguous pixels in the blob
- in this small image: Blob at [0][1]: size 5
  Blob at [2][7]: size 1
  Blob at [6][0]: size 4
  Blob at [6][6]: size 4



- can use backtracking to locate & measure blobs

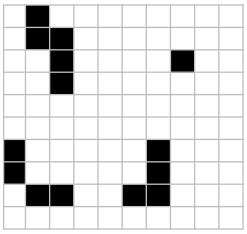
Blob count (cont.)

can use recursive backtracking to get a blob's size

when find a spot:

1 (for the spot) +

size of all connected subblobs (adjacent to spot)



note: we must not double count any spots

- when a spot has been counted, must "erase" it
- keep it erased until all blobs have been counted

pseudocode:

```
private int blobSize(int row, int col) {
    if (OFF THE GRID || NOT A SPOT) {
        return 0;
    }
    else {
        ERASE SPOT;
        return 1 + this.blobSize(row-1, col-1)
                 + this.blobSize(row-1,   col)
                 + this.blobSize(row-1, col+1)
                 + this.blobSize(  row, col-1)
                 + this.blobSize(  row, col+1)
                 + this.blobSize(row+1, col-1)
                 + this.blobSize(row+1,   col)
                 + this.blobSize(row+1, col+1);
    }
}
```

25

Blob count (cont.)

`findBlobs` traverses the image, checks each grid pixel for a blob

`blobSize` uses backtracking to expand in all directions once a blob is found

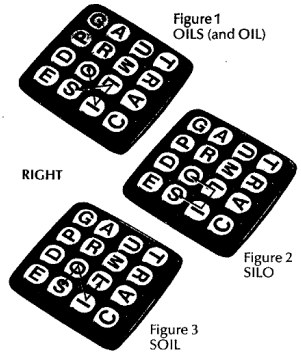
note: each pixel is "erased" after it is processed to avoid double-counting (& infinite recursion)

the image is restored at the end of `findBlobs`

```
public class BlobCounter {
  private char[][] grid;

  . . .

  public void findBlobs() {
    for (int row = 0; row < this.grid.length; row++) {
        for (int col = 0; col < this.grid.length; col++) {
            if (this.grid[row][col] == '*') {
                System.out.println("Blob at [" + row + "][" +
                                   col + "] : size " +
                                   this.blobSize(row, col));
            }
        }
    }

    for (int row = 0; row < this.grid.length; row++) {
        for (int col = 0; col < this.grid.length; col++) {
            if (this.grid[row][col] == 'O') {
                this.grid[row][col] = '*';
            }
        }
    }
  }

  private int blobSize(int row, int col) {
    if (row < 0 || row >= this.grid.length ||
            col < 0 || col >= this.grid.length ||
            this.grid[row][col] != '*') {
        return 0;
    }
    else {
        this.grid[row][col] = 'O';
        return 1 + this.blobSize(row-1, col-1)
                 + this.blobSize(row-1,   col)
                 + this.blobSize(row-1, col+1)
                 + this.blobSize(  row, col-1)
                 + this.blobSize(  row, col+1)
                 + this.blobSize(row+1, col-1)
                 + this.blobSize(row+1,   col)
                 + this.blobSize(row+1, col+1);
    }
  }
}
```

26

Another example: Boggle



recall the game

- random letters are placed in a 4x4 grid
- want to find words by connecting adjacent letters (cannot reuse the same letter)
- for each word found, the player earns points = length of the word
- the player who earns the most points after 3 minutes wins

how do we automate the search for words?

27

Boggle (cont.)

can use recursive backtracking to search for a word

when the first letter is found:

remove first letter & recursively search for remaining letters

again, we must not double count any letters

- must "erase" a used letter, but then restore for later searches

G	A	U	T
P	R	M	R
D	O	L	A
E	S	I	C

pseducode:

```
private boolean findWord(String word, int row, int col) {
    if (WORD IS EMPTY) {
        return true;
    }
    else if (OFF_THE_GRID || GRID LETTER != FIRST LETTER OF WORD) {
        return false;
    }
    else {
        ERASE LETTER;
        String rest = word.substring(1, word.length());
        boolean result = this.findWord(rest, row-1, col-1) ||
                         this.findWord(rest, row-1,   col) ||
                         this.findWord(rest, row-1, col+1) ||
                         this.findWord(rest,   row, col-1) ||
                         this.findWord(rest,   row, col+1) ||
                         this.findWord(rest, row+1, col-1) ||
                         this.findWord(rest, row+1,   col) ||
                         this.findWord(rest, row+1, col+1);
        RESTORE LETTER;
        return result;
    }
}
```

28

BoggleBoard class

can define a `BoggleBoard` class that represents a board

- has public method for finding a word
- it calls the private method that implements recursive backtracking
- also needs a constructor for initializing the board with random letters
- also needs a `toString` method for easily displaying the board

```
public class BoggleBoard {
  private char[][] board;

  . . .

  public boolean findWord(String word) {
    for (int row = 0; row < this.board.length; row++) {
        for (int col = 0; col < this.board.length; col++) {
            if (this.findWord(word, row, col)) {
                return true;
            }
        }
    }
    return false;
  }

  private boolean findWord(String word, int row, int col) {
    if (word.equals("")) {
        return true;
    }
    else if (row < 0 || row >= this.board.length ||
             col < 0 || col >= this.board.length ||
             this.board[row][col] != word.charAt(0)) {
        return false;
    }
    else {
        char safe = this.board[row][col];
        this.board[row][col] = '*';
        String rest = word.substring(1, word.length());
        boolean result = this.findWord(rest, row-1, col-1) ||
                         this.findWord(rest, row-1,   col) ||
                         this.findWord(rest, row-1, col+1) ||
                         this.findWord(rest,   row, col-1) ||
                         this.findWord(rest,   row, col+1) ||
                         this.findWord(rest, row+1, col-1) ||
                         this.findWord(rest, row+1,   col) ||
                         this.findWord(rest, row+1, col+1);
        this.board[row][col] = safe;
        return result;
    }
  }

  . . .
}
```

29

BoggleGame class

a separate class can implement the game functionality

- constructor creates the board and fills `unguessedWords` with all found words
- `makeGuess` checks to see if the word is valid and has not been guessed, updates the sets accordingly
- also need methods for accessing the `guessedWords`, `unguessedWords`, and the board (for display)

see `BoggleGUI`

```
public class BoggleGame {
  private final static String DICT_FILE = "dictionary.txt";
  private BoggleBoard board;
  private Set<String> guessedWords;
  private Set<String> unguessedWords;

  public BoggleGame() {
      this.board = new BoggleBoard();
      this.guessedWords = new TreeSet<String>();
      this.unguessedWords = new TreeSet<String>();

      try {
          Scanner dictFile = new Scanner(new File(DICT_FILE));
          while (dictFile.hasNext()) {
              String nextWord = dictFile.next();
              if (this.board.findWord(nextWord)) {
                  this.unguessedWords.add(nextWord);
              }
          }
      }
      catch (java.io.FileNotFoundException e) {
          System.out.println("DICTIONARY FILE NOT FOUND");
      }
  }

  public boolean makeGuess(String word) {
      if (this.unguessedWords.contains(word)) {
          this.unguessedWords.remove(word);
          this.guessedWords.add(word);
          return true;
      }
      return false;
  }

  . . .
}
```

30