

# CSC 427: Data Structures and Algorithm Analysis

Fall 2008

## Problem-solving approaches

- divide & conquer
  - greedy
  - backtracking
- examples: N-queens, 2-D gels, Boggle

1

## Divide & Conquer

RECALL: the divide & conquer approach tackles a complex problem by breaking it into smaller pieces, solving each piece, and combining them into an overall solution

- e.g., merge sort divided the list into halves, conquered (sorted) each half, then merged the results
- e.g., to count number of nodes in a binary tree, break into counting the nodes in each subtree (which are smaller), then adding the results + 1

divide & conquer is applicable when a problem can naturally be divided into independent pieces

sometimes, the pieces to be conquered can be handled in sequence

- i.e., arrive at a solution by making a sequence of choices/actions
- in these situations, we can consider specialized classes of algorithms
  - greedy algorithms
  - backtracking algorithms

2

## 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?

3

## Example: greedy change

while the amount remaining is not 0:

- select the largest coin that is  $\leq$  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)

4

## 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)

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

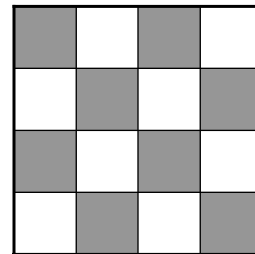
does the greedy algorithm guarantee the optimal schedule? efficiency?

5

## Example: N-queens problem

given an  $N \times N$  chess board, place a queen on each row so that no queen is in jeopardy

GREEDY algorithm: 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

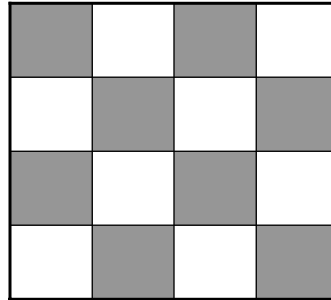
GREEDY does not work – need a more holistic approach

6

## Generate & test

we could take an extreme approach to the N-queens problem

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



this will work (in theory), but the size of the search space may be prohibitive

$$4 \times 4 \text{ board} \rightarrow \binom{16}{4} = 1,820 \text{ arrangements}$$

$$8 \times 8 \text{ board} \rightarrow \binom{64}{8} = 131,198,072 \text{ arrangements}$$

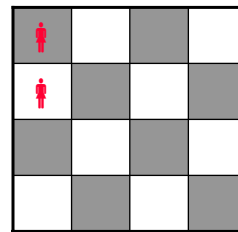
- granted, we could be a little smarter at ruling out possibilities **HOW?**

7

## Backtracking

if we were smart, we could greatly reduce the search space

- 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



backtracking is a smart way of doing generate & test

- view a solution as a sequence of choices/actions
- 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

8

## N-Queens pseudocode

```
/**
 * 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

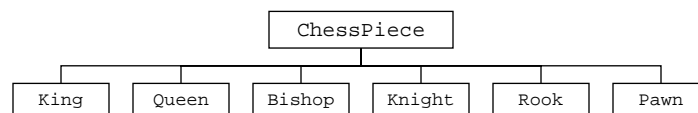
return false if cannot place remaining queens

9

## 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
}
```

10

## 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

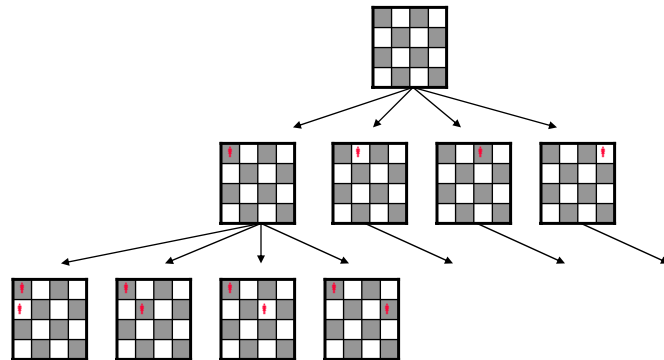
11

## Why does backtracking work?

backtracking burns no bridges – all choices are reversible

think of the search space as a tree

- root is the initial state of the problem (e.g., empty board)
- at each step, multiple choices lead to a branching of the tree
- solution is a sequence of choices (path) that leads from start state to a goal state



12

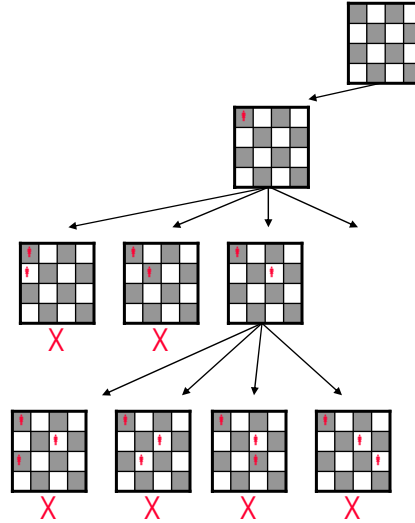
## backtracking vs. generate & test

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

- worst case: exhaustively tries all paths, traversing 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

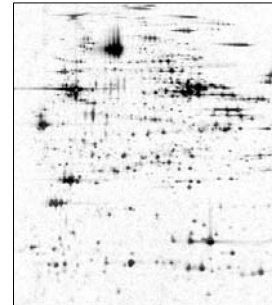


13

## Another example: blob count

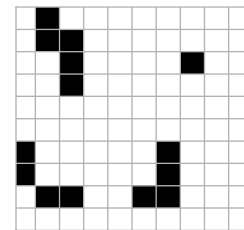
application: 2-D gel electrophoresis

- 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



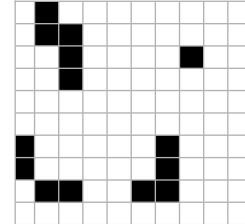
14

## 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);
    }
}
```

15

## 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

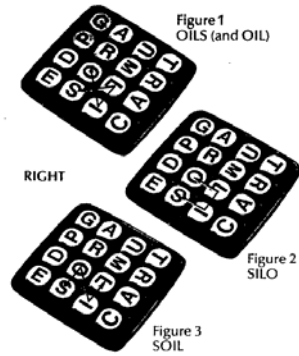
note: each pixel is "erased" after it is processed to avoid double-counting (& infinite recursion)

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);
        }
    }
}
```

16

## 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?

17

## 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

pseudocode:

```
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;
    }
}
```

18

## 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;
        }
    }
    . . .
}
```

19

## 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() {
        board = new BoggleBoard();
        guessedWords = new TreeSet<String>();
        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;
    }
    . . .
}
```

20