

# CSC 427: Data Structures and Algorithm Analysis

Fall 2006

## Problem-solving approaches

- divide & conquer
- greedy
- backtracking
  - examples: N-queens, Boggle, 2-D gels
- hw6: Sudoku solver

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

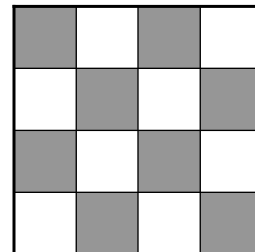
does the greedy algorithm guarantee the optimal schedule? efficiency?

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

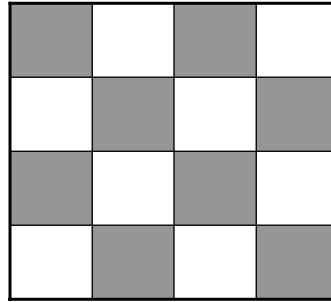
GREEDY does not work – need a more wholistic approach

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

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

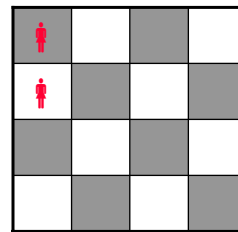
$$8 \times 8 \text{ board} \rightarrow \binom{64}{8} = 131,198,072 \text{ arrangements}$$

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 12 boards from consideration



backtracking is a smart way of doing generate & test

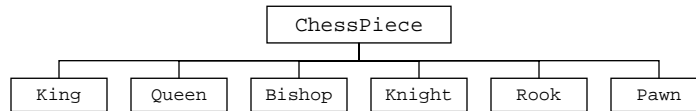
- view a solution as a sequence of choices/actions
- when presented with a choice, pick one (similar to GREEDY)
- however, reserve the right to change your mind and backtrack to a previous choice (unlike GREEDY)
- you must remember alternatives:  
*if a choice does not lead to a solution, back up and try an alternative*
- eventually, backtracking will find a solution or exhaust all alternatives

8

## 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
}
```

9

## Backtracking N-queens

```
/**
 * Fills the board with queens where none are jeopardy
 * @return true if able to successfully fill all rows
 */
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.size()) {
        return true;
    }
    else {
        for (int col = 0; col < this.size(); col++) {
            if (!this.inJeopardy(row, col)) {
                this.add(row, col, new Queen());

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

could add a method for placing queens, which calls a private (recursive backtracking) helper method

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

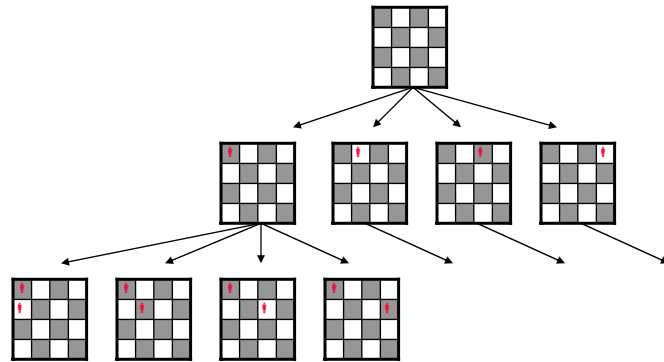
10

## 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



11

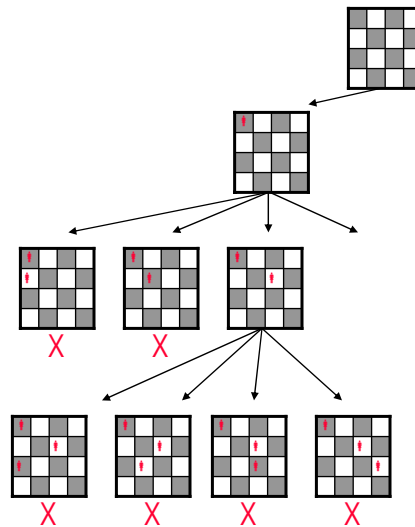
## backtracking vs. generate & test

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



12

## Boggle backtracking

in the boggle code, backtracking was used to search the entire board for all words

findAllWords() :

- starts a fresh search for each position on the board

findAllWords(sofar, row, col) :

- checks to see if (row,col) is valid, then adds the char at (row,col) to the string sofar
- checks to see if that is a word & adds to set if it is
- checks to see if that is a prefix of a word & continues the recursive search if it is (note: replaces char with '\*' to avoid reuse)

```
private void findAllWords() {
    for (int row = 0; row < this.board.length; row++) {
        for (int col = 0; col < this.board.length; col++) {
            this.findAllWords("", row, col);
        }
    }
}

private void findAllWords(String sofar, int row, int col) {
    if (row >= 0 && row < this.board.length &&
        col >= 0 && col < this.board.length &&
        this.board[row][col] != '*') {
        sofar += this.board[row][col];
        if (this.dictionary.contains(sofar)) {
            this.words.add(sofar);
        }

        if (this.dictionary.containsPrefix(sofar)) {
            char saved = this.board[row][col];
            this.board[row][col] = '*';
            this.findAllWords(sofar, row-1, col-1);
            this.findAllWords(sofar, row-1, col);
            this.findAllWords(sofar, row-1, col+1);
            this.findAllWords(sofar, row, col-1);
            this.findAllWords(sofar, row, col+1);
            this.findAllWords(sofar, row+1, col-1);
            this.findAllWords(sofar, row+1, col);
            this.findAllWords(sofar, row+1, col+1);
            this.board[row][col] = saved;
        }
    }
}
```

13

## Boggle (v. 2)

note: the recursive search could have been structured differently

- instead of searching the board for sequences & checking to see if in dictionary, could have traversed the dictionary & checked to see if each word was in board

tradeoffs?

```
private void findAllWords() {
    for (String word : this.dictionary) {
        if (this.findWord(word)) {
            this.words.add(word);
        }
    }
}

private 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;
}

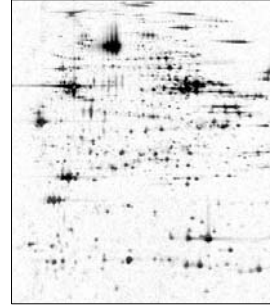
private boolean findWord(String word, int row, int col) {
    if (word.equals("")) {
        return true;
    }
    else if (row < 0 || row >= this.board.length ||
             col < 0 || col >= this.board.length) {
        this.board[row][col] != word.charAt(0) {
            return false;
        }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

14

## Another example: blob count

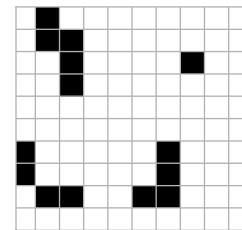
### application: 2-D gel electrophoresis

- 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



15

## Finding blobs

`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 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] == '0') {
                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] = '0';
        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

## HW6: Sudoku solver

for HW6, you are to design and implement a program for solving Sudoku puzzles

- you will need to create a class for representing the board, including a method for solving the board via recursive backtracking

*while there are any empty spots*

- *pick an empty spot & place a number in it*
- *if a conflict occurs, backtrack*

- you will also need to create a GUI for entering the initial board & displaying the solution

		1		2				
	3	7	8					2
2	4						7	3
4						7	1	
			6	8				
	8	2						9
9	5					3	7	
6				5	4	8		
			4		5			

surprisingly: simple, blind backtracking is sufficient for solving even the hardest Sudoku puzzle almost instantaneously