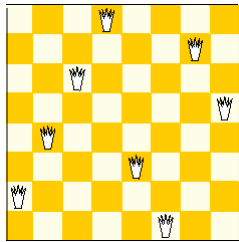


## Eight Queens Problem

- Position 8 queens on a chessboard (8\*8) so that they do not threaten each other
- Suboptimal approaches:
  - try all combinations
  - try all combinations with none in the same columns
  - try all combinations with none in the same column and row



© 2005-06. 324-1-109

## Eight Queens with Backtracking

- Represent chess boards by 8-vectors
  - each entry in vector: position of queen in a row
- k-Vector is promising if:
  - none of the queens threaten each other
- To solve the problem:
  - we need to find a vector that is 8-promising
- Create a tree of vectors
  - edge from u to v if:
    - u is promising k-vector
    - v is promising (k+1)-vector
    - first k elements are the same in both u and v
  - root: empty vector
  - leaves: solutions or dead ends
- Do backtracking on tree to find solutions
  - tree is implicit

© 2005-06. 324-1-110

## Eight Queens with Backtracking

- Works best with depth-first search
- Performance:
  - using permutations of all positions:
    - $\binom{64}{8}$  = about 4 billion possibilities
  - using permutations of 8-vectors
    - $8^8$  = about 16 million possibilities
  - using permutations of 8-vectors with unique numbers
    - (not putting two queens in the same row)
    - $8!$  = 40,320
  - using backtracking
    - number of nodes = 2057

© 2005-06. 324-1-111

## Eight Queens with Backtracking

```
public boolean solve( int y ) {
    for( int i=0; i<8; i++ ) {
        if ( grid[i][y] == 0 ) {           //for each row, try placing a queen
            addQueen( i, y, true );
            if( y == 7 ) {
                return true;              //we have placed all queens successful
            } else {
                if( solve( y+1 ) ) {
                    return true;          //we solved it, return
                } else {
                    addQueen( i, y, false ); // remove the queen
                }
            }
        }
    }
    return false;                          // unable to solve down this branch -- retreat!
} // By Aaron Davidson <http://spaz.ca/>
```

## String Matching

- Problem: finding a string inside another
  - Find “Ball State” in a letter using a word processor
- More advanced: with wildcards
- More advanced: regular expressions
  - example:
    - (Ball State (University)?|BSU)
    - Joh?nson

© 2005-06. 324-1-113

## Brute Force String Matching

```
//return position of pattern in text, -1 if not found
int search( String pattern, String text ) {
    int textIndex = 0;
    while( textIndex + pattern. length() <= text. length()
    {
        int patternIndex = 0;
        while( text[ textIndex + patternIndex ]
            == pattern[ patternIndex ] )
        {
            patternIndex++;
            if( patternIndex >= pattern. length() ) {
                return textIndex;
            }
        }
        textIndex++;
    }
    return -1;
}
```

© 2005-06. 324-1-114

## Brute Force String Matching

- Example:
  - find "001" in "010001"
- Runtime:  $O(\text{text.length() * pattern.length()})$ 
  - average on random text:  $O(\text{text.length() - pattern.length()})$

© 2005-06. 324-1-115

## String Matching: KMP

- Knuth-Morris-Pratt Algorithm
- Example:
  - Find "Tweedledum" in "Tweedledee and Tweedledum"
- Idea:
  - don't process the same part of the text twice
  - create a shift table for a pattern
    - if we hit a mismatch, how much can we shift the pattern?
    - using those comparisons we've already made

© 2005-06. 324-1-116

## Knuth-Morris-Pratt

- Table for Tweedledum:

		T	w	e	e	d	l	e	d	u	m
k	-1	0	1	2	3	4	5	6	7	8	9
shift	1	1	2	3	4	5	6	7	8	9	10

- Table for "pappar"

		p	a	p	p	a	r
k	-1	0	1	2	3	4	5
shift	1	1	2	2	3	3	6

- example: find "pappar" in "panther" and "papaya tree"

© 2005-06. 324-1-117

## Knuth-Morris-Pratt Algorithm

```
int kmpSearch( String text, String pattern ) {
    int[] shift = getKMPShiftTable( pattern );
    int textIndex = 0, int patternIndex = 0;
    while( textIndex + pattern.length() <= text.length() ) {
        //scan the text
        while( text[ textIndex + patternIndex ] == pattern[patternIndex] )
        {
            //scan the pattern
            patternIndex++;
            if( patternIndex >= pattern.length() ) {
                return textIndex; //found it
            }
        }
        textIndex += shift[ patternIndex - 1 ]; //shift
        patternIndex = max( patternIndex - shift[ patternIndex - 1 ], 0 );
    }
    return -1; //not found
}
```

© 2005-06. 324-1-118

## Knuth-Morris-Pratt

- Runtime?
- Based on the loops:
  - $O(\text{text.length() * pattern.length()})$
  - same as brute-force approach!
- However...
  - at start of inner loop
  - trace value of  $2*\text{textIndex} + \text{patternIndex}$ 
    - if inner loop is entered:
      - $\text{patternIndex} += 1$
    - else:
      - $\text{textIndex} += \text{shift}$
      - $\text{patternIndex} -= \text{shift}$
  - $2*\text{textIndex} + \text{patternIndex} \leq 2*\text{text.length() + pattern.length()}$
  - Hence:  $O(\text{text.length() + pattern.length()})$

© 2005-06. 324-1-119