CSE 143
Lecture 18

More Recursive Backtracking

slides created by Marty Stepp
http://www.cs.washington.edu/143/
Exercise: Dominoes

• The game of dominoes is played with small black tiles, each having 2 numbers of dots from 0-6. Players line up tiles to match dots.

• Given a class `Domino` with the following methods:

  ```java
  public int first() // first dots value
  public int second() // second dots value
  public String toString() // e.g. "(3|5)"
  ```

• Write a method `hasChain` that takes a `List` of dominoes and a starting/ending dot value, and returns whether the dominoes can be made into a chain that starts/ends with those values.
  – If the chain's start/end are the same, the answer is always `true`. 
• Suppose we have the following dominoes:

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
```

• We can link them into a chain from 1 to 3 as follows:
  – Notice that the 3|5 domino had to be flipped.

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
```

• We can "link" one domino into a "chain" from 6 to 2 as follows:

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
</table>
```

import java.util.*;   // for ArrayList

public class SolveDominoes {
    public static void main(String[] args) {
        // [(1|4), (2|6), (4|5), (1|5), (3|5)]
        List<Domino> dominoes = new ArrayList<Domino>();
        dominoes.add(new Domino(1, 4));
        dominoes.add(new Domino(2, 6));
        dominoes.add(new Domino(4, 5));
        dominoes.add(new Domino(1, 5));
        dominoes.add(new Domino(3, 5));
        System.out.println(hasChain(dominoes, 5, 5));  // true
        System.out.println(hasChain(dominoes, 1, 5));  // true
        System.out.println(hasChain(dominoes, 1, 3));  // true
        System.out.println(hasChain(dominoes, 1, 6));  // false
        System.out.println(hasChain(dominoes, 1, 2));  // false
    }

    public static boolean hasChain(List<Domino> dominoes, int start, int end) {
        ...
    }
}
public static boolean hasChain(List<Domino> dominoes, int start, int end) {
    if (start == end) {
        return true;  // base case
    } else {
        for (int i = 0; i < dominoes.size(); i++) {
            Domino d = dominoes.remove(i);  // choose
            if (d.first() == start) {  // explore
                if (hasChain(dominoes, d.second(), end)) {
                    return true;
                }
            } else if (d.second() == start) {
                if (hasChain(dominoes, d.first(), end)) {
                    return true;
                }
            }
            dominoes.add(i, d);  // un-choose
        }
    }
    return false;
}
Exercise: Print chain

- Write a variation of your `hasChain` method that also prints the chain of dominoes that it finds, if any.

```python
hasChain(dominoes, 1, 3);

[(1|4), (4|5), (5|3)]
```
public static boolean hasChain(List<Domino> dominoes, int start, int end) {
    Stack<Domino> chosen = new Stack<Domino>();
    return hasChain(dominoes, chosen, start, end);
}

private static boolean hasChain(List<Domino> dominoes, Stack<Domino> chosen, int start, int end) {
    if (start == end) {
        System.out.println(chosen);
        return true;  // base case
    } else {
        for (int i = 0; i < dominoes.size(); i++) {
            Domino d = dominoes.remove(i);  // choose
            if (d.first() == start) {  // explore
                chosen.push(d);
                if (hasChain(dominoes, chosen, d.second(), end)) {
                    return true;
                }
                chosen.pop();
            } else if (d.second() == start) {
                d.flip();
                chosen.push(d);
                if (hasChain(dominoes, chosen, d.second(), end)) {
                    return true;
                }
                chosen.pop();
            }
            dominoes.add(i, d);  // un-choose
        }
        return false;
    }
}
The "8 Queens" problem

- Consider the problem of trying to place 8 queens on a chess board such that no queen can attack another queen.

  - What are the "choices"?
  
  - How do we "make" or "un-make" a choice?
  
  - How do we know when to stop?
• for (each square on board):
  – Place a queen there.
  – Try to place the rest of the queens.
  – Un-place the queen.

  – How large is the solution space for this algorithm?
    • 64 * 63 * 62 * ...
Better algorithm idea

• Observation: In a working solution, exactly 1 queen must appear in each row and in each column.
  
  – Redefine a "choice" to be valid placement of a queen in a particular column.
  
  – How large is the solution space now?
    • $8 \times 8 \times 8 \times \ldots$
Suppose we have a `Board` class with the following methods:

<table>
<thead>
<tr>
<th>Method/Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public Board(int size)</code></td>
<td>construct empty board</td>
</tr>
<tr>
<td><code>public boolean isSafe(int row, int column)</code></td>
<td>true if queen can be safely placed here</td>
</tr>
<tr>
<td><code>public void place(int row, int column)</code></td>
<td>place queen here</td>
</tr>
<tr>
<td><code>public void remove(int row, int column)</code></td>
<td>remove queen from here</td>
</tr>
<tr>
<td><code>public String toString()</code></td>
<td>text display of board</td>
</tr>
</tbody>
</table>

Write a method `solveQueens` that accepts a `Board` as a parameter and tries to place 8 queens on it safely.

- Your method should stop exploring if it finds a solution.
// Searches for a solution to the 8 queens problem
// with this board, reporting the first result found.
public static void solveQueens(Board board) {
    if (!explore(board, 1)) {
        System.out.println("No solution found.");
    } else {
        System.out.println("One solution is as follows:");
        System.out.println(board);
    }
}
// Recursively searches for a solution to 8 queens on this board, starting with the given column, returning true if a solution is found and storing that solution in the board.
// PRE: queens have been safely placed in columns 1 to (col-1)
public static boolean explore(Board board, int col) {
    if (col > board.size()) {
        return true;  // base case: all columns are placed
    } else {
        // recursive case: place a queen in this column
        for (int row = 1; row <= board.size(); row++) {
            if (board.isSafe(row, col)) {
                board.place(row, col);  // choose
                if (explore(board, col + 1)) {  // explore
                    return true;  // solution found
                }
                b.remove(row, col);  // un-choose
            }
        }
        return false;  // no solution found
    }
}