CSE 143
Lecture 18

More Recursive Backtracking

reading: "Appendix R" on course web site

slides adapted from 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 public methods:
  ```java
  int first()  // first dots value
  int second() // second dots value
  void flip()  // inverts 1st/2nd
  boolean contains(int n) // true if 1st/2nd == n
  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.
Domino chains

• Suppose we have the following dominoes:

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

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

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

[(1|4), (4|5), (5|3)]
```
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?
Naive algorithm

- 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?
  * $64 \times 63 \times 62 \times \ldots$
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?
    - \(8 \times 8 \times 8 \times \ldots\)
Exercise

• 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>public <code>Board</code> (int size)</td>
<td>construct empty board</td>
</tr>
<tr>
<td>public boolean <code>isSafe</code> (int row, int column)</td>
<td>true if queen can be safely placed here</td>
</tr>
<tr>
<td>public void <code>place</code> (int row, int column)</td>
<td>place queen here</td>
</tr>
<tr>
<td>public void <code>remove</code> (int row, int column)</td>
<td>remove queen from here</td>
</tr>
<tr>
<td>public String <code>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 (solveQueens(board, 1)) {
        System.out.println("One solution is as follows:");
        System.out.println(board);
    } else {
        System.out.println("No solution found.");
    }
}

...
Exercise solution, cont'd.

// 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 solveQueens(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
    }
}
Suppose we have a Maze class with these methods:

<table>
<thead>
<tr>
<th>Method/Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public Maze(String text)</td>
<td>construct a given maze</td>
</tr>
<tr>
<td>public int getHeight(), getWidth()</td>
<td>get maze dimensions</td>
</tr>
<tr>
<td>public boolean isExplored(int r, int c)</td>
<td>get/set whether you have visited a location</td>
</tr>
<tr>
<td>public void setExploded(int r, int c)</td>
<td>get/set whether you have visited a location</td>
</tr>
<tr>
<td>public void isWall(int r, int c)</td>
<td>whether given location is blocked by a wall</td>
</tr>
<tr>
<td>public void mark(int r, int c)</td>
<td>whether given location is marked in a path</td>
</tr>
<tr>
<td>public void isMarked(int r, int c)</td>
<td>whether given location is marked in a path</td>
</tr>
<tr>
<td>public String toString()</td>
<td>text display of maze</td>
</tr>
</tbody>
</table>
Exercise: solve maze

• Write a method `solveMaze` that accepts a `Maze` and a starting row/column as parameters and tries to find a path out of the maze starting from that position.

  – If you find a solution:
    • Your code should **stop** exploring.
    • You should **mark** the path out of the maze on your way back out of the recursion, using backtracking.

  – (As you explore the maze, squares you set as 'explored' will be printed with a dot, and squares you 'mark' will display an X.)
Recall: Backtracking

A general pseudo-code algorithm for backtracking problems:

Explore(\textit{choices}):  
– if there are no more \textit{choices} to make: stop. 
– else, for each available choice \textit{C}:  
  • Choose \textit{C}. 
  • Explore the remaining \textit{choices}. 
  • Un-choose \textit{C}, if necessary. (backtrack!) 

What are the choices in this problem?
Decision tree

<table>
<thead>
<tr>
<th>position</th>
<th>(row 1, col 7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>choices</td>
<td>←↑↓→ (these never change)</td>
</tr>
</tbody>
</table>

The diagram illustrates a decision-making process in a grid environment, where the agent navigates through walls and visited cells. The choices available for movement are determined by the current position and the walls surrounding it. Each node in the tree represents a possible position, with the arrow directions indicating the available choices: left, up, right, and down. The visited positions are marked with a darker shade, indicating that these cells have already been explored.