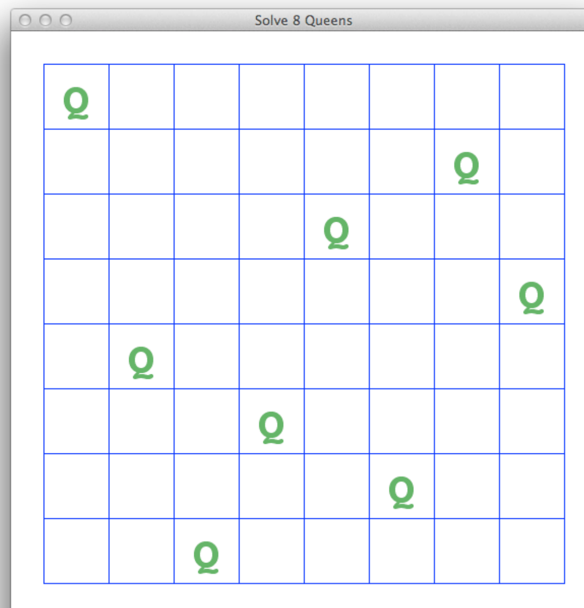
CS106X
Autumn 2012

Handout 19
October 15th, 2012

Recursive Backtracking II

Solving the Eight Queens Problem

The Eight Queens Problem is a classic programming puzzle that asks whether it's possible to place eight queens on an 8 x 8 chessboard in such a way that they can all coexist without attacking each other. Placing nine queens on an 8 x 8 is impossible—there's a pigeonhole principle argument against it, for at least two queens would always need to occupy the same column. But it's not immediately obvious whether eight queens can be placed on an 8 x 8 board, nor is it obvious whether N queens can be placed on an N x N board in general.



One approach—by far the most common programmatic one I know of—uses recursive backtracking to discover a solution, and that approach is spelled out on the next page:

```
static bool solve(GWindow& window, Grid<bool>& board, int col) {
   if (col == board.numCols()) return true;

   for (int rowToTry = 0; rowToTry < board.numRows(); rowToTry++) {
      placeQueen(window, rowToTry, col, kPossiblilityColor);
      if (isSafe(board, rowToTry, col)) {
         board[rowToTry][col] = true;
         placeQueen(window, rowToTry, col, kProvisionalColor);
         if (solve(window, board, col + 1)) {
            placeQueen(window, rowToTry, col, kPermanantColor);
            return true;
         }
         board[rowToTry][col] = false;
      }
      placeQueen(window, rowToTry, col, kNoPossibilityColor);
   }

   return false;
}

static void solve(GWindow& window, Grid<bool>& board) {
   solve(window, board, 0);
}
```

The second of the two versions is called on an empty board, and the first one implements the recursive backtracking. Each call to **`solve`** assumes that queens have been placed in columns 0 through **`col – 1`** in a configuration that allows them to all coexist peacefully. The **`solve`** call systemically searches its own column for a row where yet another queen can be placed without introducing a conflict, and then recurs on **`col + 1`**. If the recursive call on **`col + 1`** returns **`true`**, then that **`true`** is immediately propagated up to whoever called us. If it returns **`false`**, we backtrack by lifting the queen we placed and advancing on to higher rows. Only when **`solve`** has tried to extend the partial solution it inherited in every way possible—and failed every time—does it return **`false`**.

Solving SuDoKu Puzzles [idea by Julie Zelenski]

Recursive backtracking can also be used to solve SuDoKu puzzles by systematically considering every single way to legitimately place a number in some open square that, at least for the moment, works, and then recurring on the same board to see if that decision was a good one.

Original

Solve SuDoKu

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

Mid Progress

Solve SuDoKu

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

Solved

Solve SuDoKu

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

```
static bool solve(GWindow& window, Grid<int>& board) {
   int row, col;
   if (!findEmptyLocation(board, row, col)) return true;

   for (int digit = 1; digit <= 9; digit++) {
      if (isFreeOfConflict(board, row, col, digit)) {
         updateBoardLocation(window, board, row, col, digit);
         if (solve(window, board)) return true;
         updateBoardLocation(window, board, row, col, kEmpty);
      }
   }

   return false;
}
```

For those new to SuDoKu, the challenge is to fill in all empty squares with numbers 1 through 9 so that each digit appears exactly once per row, once per column, and once per 3 x 3 block. The above solution relies on three helper functions, and those helper functions insulate us from some algorithmic detail. There is no denying the above is classic recursive backtracking—even if it's very brute force and not very intelligent.

`isFreeOfConflict` decides, given the current state of the board, whether **`digit`** can be placed at the identified position without violating the rules. **`updateBoardLocation`** updates the board to house the supplied number at the specified coordinate (and updates the visuals as well). The only function students find confusing is **`findEmptyLocation`**. From context, it appears to return a **`true`** if and only if there's some unassigned slot, but what isn't clear is that, when **`true`** is returned, **`row`** and **`col`** are updated (by reference) to some empty location's coordinates. It becomes clearer if you see the code for it, so here it is:

```
static bool findEmptyLocation(Grid<int>& board, int& row, int& col) {
   for (row = 0; row < kBoardDimension; row++) {
      for (col = 0; col < kBoardDimension; col++) {
         if (board[row][col] == KEmpty) return true;
      }
   }

   return false;
}
```

This particular implementation just searched top-to-bottom, left-to-right until it finds something that's empty. It's fairly naïve and results in a solution that takes its time for all but the most trivial of boards. However, it's possible to search not just for some empty square, but for the empty square that is more constrained than any other.