6.009: Fundamentals of Programming

Lecture 4: Recursive Search & Backtracking

[Slides, code to be posted on Calendar page]

Suggestion: download code templates from Calendar page, open your laptop, and work on the lecture problems as we do them!
N Queens Problem

Place eight queens on a chessboard so that no attacks are possible, i.e.,

1. No two queens on same column

2. No two queens on same row

3. No two queens on same diagonal

One of 92 solutions, ignoring symmetries!
Since there is at most one queen in any column, we can use a list to represent the board:

\[ i^{th} \text{ entry in list gives row for queen in column } I \]
\[ 0 \leq \text{row} < N \]

Use -1 if no queen present.

\[ [1, 3, 0, -1] \]
noConflicts(board, col)

Does queen in specified column conflict with queens in columns to the left?

1. No two queens on same column ✓ by construction
2. No two queens on same row
3. No two queens on same diagonal

```python
def noConflicts(board, current):
    """return True if queens in earlier columns don't conflict with queen in current column"""
    
    # row of current queen
    q = board[current]

    # check each previous column
    for i in range(current):
        # check for queen on same row
        if board[i] == q:
            return False
        # check for queen on same diagonal
        if current - i == abs(q - board[i]):
            return False

    # all checks complete
    return True
```
Iterative Solution for 4 Queens

```python
def FourQueens():
    board = [-1] * 4  # initially no queens
    for i in range(4):
        board[0] = i     # place queen in column 0
        for j in range(4):
            board[1] = j   # place queen in column 1
            if not noConflicts(board, 1): continue
        for k in range(4):
            board[2] = k    # place queen in column 2
            if not noConflicts(board, 2): continue
        for l in range(4):
            board[3] = l    # place queen in column 3
            if noConflicts(board, 3):
                print(board)
                return
```

What about 10 queens problem?!

Place queen in column, check for conflicts, repeat for next column

Idea: build a subroutine to do this!
Recursive Solution for N Queens

```python
def nqueens(board, current, size):
    """try to place queen in current column, return True if succeeds, False otherwise. Modifies board to show placement."""

    if current == size:
        # we've filled all the columns without conflict, success!!!
        print(board)
        return True
    else:
        # try placing a queen in each row of current column.
        # If no conflicts, recursively see if solution exists
        # for remaining columns.
        for i in range(size):
            board[current] = i
            if noConflicts(board, current):
                done = nqueens(board, current + 1, size)
                if done:
                    return True
        return False
```

Change to find all solutions?
Recursive Search Design Pattern

- To solve search problem:
  - For each possible move at next position
    - If move is legal
      - Update state to reflect move
      - Recursively search for a solution
      - If succeeds: we’re done, return state
      - Otherwise
        » Restore state
        » Try next possible move
    - If no possible move led to a solution, return False so that previous level knows to try another move
solve_magic_square(grid, sum, choices)

- Grid: list-of-lists representation of 2D magic square

\[
\begin{array}{ccc}
7 & & \\
& 1 & \\
3 & &
\end{array}
\]

- Sum: desired sum for rows, cols, and two diagonals
- Choices: list of numbers to use when filling in blanks (a choice can be used more than once).
- Return False if no solution, otherwise return filled-in grid
solve_magic_square(grid, sum, choices)

If sum = 15, and choices = [1,2,3,4,5,6,7,8,9]:


grid: 

<table>
<thead>
<tr>
<th>7</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

result: 

<table>
<thead>
<tr>
<th>2</th>
<th>7</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>
check_square(grid, magic_sum)

# helper function: return True if okay to proceed
# ie, check sum for rows, col, diag if they don't
# contain any blanks (-1).
def check_square(grid, magic_sum):
    N = len(grid)
    
    # return True if list contains blanks or
    # if it sums to magic_sum
def okay(L):
    return L.count(-1) != 0 or sum(L) == magic_sum
    
    # check rows
    for r in grid:
        if not okay(r):
            return False
    
    # check columns:
    for c in range(N):
        col = [grid[r][c] for r in range(N)]
        if not okay(col):
            return False

    # main diag
diag = [grid[d][d] for d in range(N)]
    if not okay(diag):
        return False

    # other diag
diag = [grid[d][len(grid) - 1 - d] for d in range(N)]
    if not okay(diag):
        return False

    # no non-blank rows, cols, diags violate sum
    return True

Don’t repeat yourself!
• Use local vars
• Use helper functions

Python built-ins make code more readable and more efficient!
solve_magic_square(grid, sum, choices)

def solve_magic_square(grid, magic_sum, choices):
    # find next blank square
    for row, r in enumerate(grid):
        for col, gv in enumerate(r):
            if gv == -1:
                # try all the possible values for this blank
                for choice in choices:
                    grid[row][col] = choice
                    # check if guessed value violates constraint
                    if check_square(grid, magic_sum):
                        # it's okay, recursively solve updated grid
                        result = solve_magic_square(grid, magic_sum, choices)
                        # if we found a solution, return grid
                        if result is not False:
                            return result  # found a solution!
                        # restore blank if we failed to find solution
                        grid[row][col] = -1
                    # no choice lead to a solution, let previous level know
    return False

# no blanks! return filled-in grid
return grid
“Truth-y” and “False-y” in Python

Any object can be tested for truth value, for use in an if or while condition or as operand of the Boolean operations.

By default, an object is considered true unless its class defines either a __bool__() method that returns False or a __len__() method that returns zero, when called with the object. Here are most of the built-in objects considered false:

- constants defined to be false: None and False.
- zero of any numeric type: 0, 0.0, 0j, Decimal(0), Fraction(0, 1)
- empty sequences and collections: '', (), [], {}, set(), range(0)

Operations and built-in functions that have a Boolean result always return 0 or False for false and 1 or True for true, unless otherwise stated. (Important exception: the Boolean operations or and and always return one of their operands.)

```python
>>> 0 == False
True
>>> 0 is False
False
```
solve_sudoku(grid)

Goal: fill in blanks with digits 1-9
- No duplicates in same row
- No duplicates in same column
- No duplicates in same sector
possible_values(grid, r, c)

# return set of possible values for this square
# return empty set if square is already filled
def possible_values(grid, r, c):
    if grid[r][c] != 0: return set()

    # set of all possible values
    possible = {1, 2, 3, 4, 5, 6, 7, 8, 9}

    # remove values on same row
    for i in range(9): possible.discard(grid[r][i])

    # remove values on same col
    for i in range(9): possible.discard(grid[i][c])

    # remove values in same sector
    sr, sc = 3*(r//3), 3*(c//3)
    for i in range(sr, sr+3):
        for j in range(sc, sc+3):
            possible.discard(grid[i][j])

    return possible
```
backtracks = 0
def solve_sudoku(grid):
    global backtracks
    # search for next blank cell
    N = len(grid)
    for row in range(N):
        for col in range(N):
            if grid[row][col] == 0:
                # only try possible values
                for trial in possible_values(grid, row, col):
                    # recursively solve updated grid
                    grid[row][col] = trial
                    result = solve_sudoku(grid)
                    # if we found a result, return it
                    if result is not False:
                        return result
                # otherwise try next possibility
                backtracks += 1
                # nothing worked, restore grid
                grid[row][col] = 0
                return False  # indicate failure
    # no blanks => puzzle solved!
    return grid
```
solve_sudoku Results

<table>
<thead>
<tr>
<th>Puzzle</th>
<th># backtracks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>579</td>
</tr>
<tr>
<td>2</td>
<td>6367</td>
</tr>
<tr>
<td>3</td>
<td>7790</td>
</tr>
</tbody>
</table>

- Backtracking → wasted effort?
- What if we explored most promising square and/or trial value first? (Google for A* search!)
- What if we made inferences about other values based on trial value?
solve_sudoku_opt(grid) - pencil marks!

Pencil marks: set of legal digits for each square

Only one choice? Fill it in!
solvable_sudoku_opt(grid)

# similar to original solution, but when filling in a trial value
# see what other values we might also fill in

def solve_sudoku_opt(grid):
    global backtracks
    # search for next blank cell
    N = len(grid)
    for row in range(N):
        for col in range(N):
            if grid[row][col] == 0:
                # only try possible values
                for trial in possible_values(grid, row, col):
                    # fill in trial value (and perhaps others!)
                    # remember what we filled so we can backtrack
                    fills = fill_in_trial(grid, row, col, trial)
                    # can we solve updated grid?
                    result = solve_sudoku_opt(grid)
                    if result is not False:
                        # success! return result
                        return result
                    # oops, didn't work out, undo fill ins
                    backtracks += 1
                    for r, c in fills: grid[r][c] = 0
                return False  # indicate failure
    # no blanks => puzzle solved!
    return grid
def fill_in_trial(grid, row, col, digit):
    # fill in trial value, return list of filled-in squares
    grid[row][col] = trial
    fills = [(row, col)]
    # your strategy here for filling in other blanks!
    # append coords of any square that gets filled to fills
    # look for other blank squares with only 1 possible value
    for r in range(9):
        for c in range(9):
            if grid[r][c] == 0:
                # blank square
                possible = possible_values(grid, r, c)
                if len(possible) == 1:
                    d = possible.pop()
                    # one pass of inference
                    grid[r][c] = d
                    fills.append((r, c))
    return fills
solve_sudoku_opt Results

<table>
<thead>
<tr>
<th>Puzzle</th>
<th># backtracks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>original</td>
</tr>
<tr>
<td>1</td>
<td>579</td>
</tr>
<tr>
<td>2</td>
<td>6363</td>
</tr>
<tr>
<td>3</td>
<td>7790</td>
</tr>
</tbody>
</table>
solve_sudoku_opt(grid) - pencil marks!

Make more inferences each time we fill a square.
fill_in_trial with recursive inferencing

# fill in trial value, return list of filled-in squares
def fill_in_trial(grid, row, col, trial):
    # fill in current blank
    grid[row][col] = trial
    fills = [(row, col)]

    # your strategy here for filling in other blanks!
    # append coords of any square that gets filled to fills

    # look for other blank squares with only 1 possible value
    for r in range(9):
        for c in range(9):
            if grid[r][c] == 0:
                # blank square
                possible = possible_values(grid, r, c)
                if len(possible) == 1:
                    d = possible.pop()
                    fills.extend(fill_in_trial(grid, r, c, d))
    return fills

    # return list of filled-in squares
return fills
# solve_sudoku_opt Results

<table>
<thead>
<tr>
<th>Puzzle</th>
<th># Backtracks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>original</td>
</tr>
<tr>
<td>1</td>
<td>579</td>
</tr>
<tr>
<td>2</td>
<td>6363</td>
</tr>
<tr>
<td>3</td>
<td>7790</td>
</tr>
</tbody>
</table>