Backtracking

July 2, 2018
Backtracking

- Wikipedia: backtracking is a general algorithm for finding all (or some) solutions to some computational problems
- Problems are usually constraint satisfaction problem
- Incrementally builds candidates to the solutions
- Abandons each partial candidate (backtracks) as soon as it determines that it cannot possibly be a valid solution
- Essentially depth first search
- Build up a search tree
Permutation

- Write a program to print all permutations of a given string
- A string of length n has n! permutations
- ABC: ABC ACB BAC BCA CBA CAB
public List<String> permute(String str) {
    List<String> list = new ArrayList<>();
    backtrack(list, "", str);
    return list;
}

public void backtrack(List<String> list, String temp, String str) {
    if (str.length() == temp.length()) {
        list.add(temp); // add temp to the result list
    } else {
        for (int i = 0; i < str.length(); i++) {
            char c = str.charAt(i);
            if (temp.indexOf(c) == -1) {// c already exists, skip
                temp += c;
                backtrack(list, temp, str);
                temp = temp.substring(0, temp.length() - 1);
            }
        }
    }
}
N-Queens

- No two queens share the same row, column, or diagonal
- How many solutions from a n-queens puzzle?
public List<List<String>> solveNQueens(int n) {
    char[][] board = new char[n][n];
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            board[i][j] = '.';
    List<List<String>> res = new ArrayList<List<String>>();
    backtrack(board, 0, res);
    return res;
}
private boolean validate(char[][] board, int x, int y) {
    for(int i = 0; i < x; i++) {
        for(int j = 0; j < board.length; j++) {
            if(board[i][j] == 'Q' && (x + j == y + i || x + y == i + j || y == j))
                return false;
        }
    }
    return true;
}
private void backtrack(char[][] board, int rowIndex, 
List<List<String>> res) {
    if(rowIndex == board.length) {
        res.add(construct(board));
        return;
    }
    for(int j = 0; j < board.length; j++) {
        if(validate(board, rowIndex, j)) {
            board[rowIndex][j] = 'Q'; //Place this queen in board
            backtrack(board, rowIndex + 1, res); //do the rest
            board[rowIndex][j] = '.'; //backtrack
        }
    }
}

A rat is at top-left corner of a maze
- The rat can only move either down or right at any point
- The rat is trying to reach the bottom-right corner
- 0 means the block is dead and 1 means the block can be used.

![Maze Image](image from geeksforgeeks)
boolean validate(int maze[][], int x, int y){
    if(x >= 0 && x < N && y >= 0 &&
       y < N && maze[x][y] == 1)
        return true;
    else
        return false;
}
boolean backtrack(int maze[][], int x, int y, int sol[][]) {
    if (x == N - 1 && y == N - 1) {
        sol[x][y] = 1;
        return true;
    }

    if (validate(maze, x, y)) {
        sol[x][y] = 1; // mark x,y as part of solution path
        // Move forward in x direction
        if (backtrack(maze, x + 1, y, sol))
            return true;
        // If moving in x direction fails, then move in y direction
        if (backtrack(maze, x, y + 1, sol))
            return true;
    } /* If none of the above movements work then */
    BACKTRACK: unmark x,y as part of solution path */
    sol[x][y] = 0;
    return false;
}

return false;
}
Backtracking - Other examples

- Knight’s tour
- Hamiltonian Cycle
- Sudoku
- ...
The knight is placed on the first block of an empty board and, moving according to the rules of chess, must visit each square exactly once.
- Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once.
- Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path (you end up where you started from).
The objective is to fill a 9 by grid with digits so that each column, each row, and each of the nine 3 by 3 subgrids that compose the grid contains all of the digits from 1 to 9.

The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution.