

# Recursive backtracking

Robin Jadoul



OLYMPIADE BELGE D'INFORMATIQUE  
BELGISCHE INFORMATICA OLYMPIADE

April 7, 2016

# Table of Contents

Complete Search

Recursive Complete Search

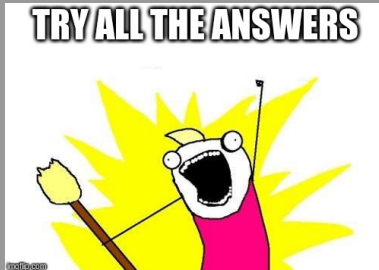
Pruning

Bitmasks

# Complete Search

Try all the answers

- ▶ Try (almost) every possibility
- ▶ Should always be **AC** (given enough time)
- ▶ However, frequently **TLE**
- ▶ Use the right algorithm/structure
- ▶ Sometimes necessary (eg. enumerate all permutations)



# Complete Search

## Approaches

- ▶ Generating
  - ▶ Usually iteratively
  - ▶ (Sometimes heavily nested) loops
  - ▶ Covers *every* possibility
- ▶ Filtering
  - ▶ Usually recursively
  - ▶ Recursive function(s)
  - ▶ Can *prune* along the way (see later)

# Complete Search

Easy iterative example

```
// count the number of pairs (a, b) (natural numbers)  
// such that a + b == c  
unsigned countSumPairs(unsigned c) {  
    unsigned num = 0;  
    for (unsigned a = 0; a <= c; a++) {  
        for (unsigned b = 0; b <= c; b++) {  
            if (a == b)  
                ++num;  
        }  
    }  
    return num;  
}
```

# Table of Contents

Complete Search

Recursive Complete Search

Pruning

Bitmasks

# Recursive Complete Search

(Simplified) sudoku solving

- ▶ Running example: Solving a sudoku puzzle
- ▶ Simplified: without  $3 \times 3$  box constraint
- ▶ Note: there are other ways to solve these (constraint satisfaction)
- ▶ Representation: a  $9 \times 9$  grid of integers (0 is an unfilled cell)

# Checking a filled sudoku

```
typedef int Grid[9][9];
bool correct(Grid& grid) {
    //Simple, but can be faster
    //Does not consider 3x3 boxes
    for (int i = 0; i < 9; i++)
        for (int j = 0; j < 9; j++)
            for (int k = j + 1; k < 9; k++)
                if (grid[i][j] <= 0 || grid[i][j] > 9
                    || grid[i][k] <= 0 || grid[i][k] > 9
                    || grid[j][i] <= 0 || grid[j][i] > 9
                    || grid[k][i] <= 0 || grid[k][i] > 9
                    || grid[i][j] == grid[i][k]
                    || grid[j][i] == grid[k][i])
                    return false;
    return true;
}
```



# Recursive Complete Search

## Filling the sudoku

```
bool solve_slow(Grid& grid, int i=0, int j=0) {
    if (i >= 9)
        return correct(grid);

    int nextj = (j + 1) % 9;
    int nexti = i + ((j + 1) / 9);

    if (grid[i][j] == 0) { //Not yet filled
        for (int val = 1; val <= 9; val++) {
            grid[i][j] = val;
            if (solve_slow(grid, nexti, nextj))
                return true;
        }
        grid[i][j] = 0;
        return false;
    } else {
        return solve_slow(grid, nexti, nextj);
    }
}
```

# Recursive Complete Search

## Filling the sudoku

## Some (really big) problems with this version

- ▶ Really slow (filling an empty grid took my computer x minutes)
- ▶ We can still improve the correct (but this will not be necessary, as we will see)
- ▶ We keep going, even if we can see that the solution already fails!

[illegible]

- Even this grid is generated:

# Table of Contents

Complete Search

Recursive Complete Search

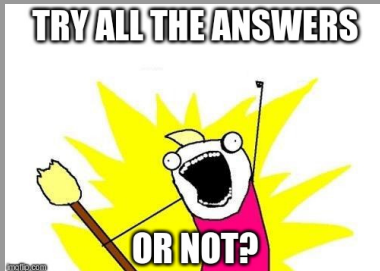
Pruning

Bitmasks

# Pruning

Try all the answers

- ▶ Stop processing when you know it is wrong
- ▶ Alternatively: stop processing when you know you have a better answer already (aka alpha-beta pruning)
- ▶ Applied to our sudoku: don't fill numbers that cannot go there anymore



# Pruning

Sudoku: check a single cell

```
bool good(Grid& grid, int r, int c) {  
    for (int i = 0; i < 9; i++) {  
        if (i != r && grid[i][c] == grid[r][c])  
            return false;  
        if (i != c && grid[r][i] == grid[r][c])  
            return false;  
    }  
    return true;  
}
```

# Pruning

Sudoku: solve it

```
bool solve(Grid& grid , int i=0, int j=0) {  
    if (i >= 9)  
        //Check happens before going to the next cell  
        //So we have found a solution  
        return true;  
  
    int nextj = (j + 1) % 9;  
    int nexti = i + ((j + 1) / 9);  
  
    if (grid[i][j] == 0) { //Not yet filled  
        for (int val = 1; val <= 9; val++) {  
            grid[i][j] = val;  
            //Short circuiting to the rescue  
            if (good(grid , i , j) && solve(grid , nexti , nextj))  
                return true;  
        }  
        grid[i][j] = 0;  
        return false;  
    } else {  
        return solve(grid , nexti , nextj);  
    }  
}
```

# Pruning

Sudoku: wrap up

- ▶ Very similar
- ▶ Just with pruning this time
- ▶ Filling an empty maze: instant (0.002s)

# Table of Contents

Complete Search

Recursive Complete Search

Pruning

Bitmasks



# bitmasks

## Passing booleans

- ▶ Sometimes, you need to pass around booleans to recursive calls
- ▶ Note: not just some boolean flags, but a boolean that indicates for example if an element has been *taken*
- ▶ How can we best do this?
- ▶ Passing around a `vector<bool>`: needs copying every time
- ▶ Better: `bitset<N>`, a statically sized boolean collection
- ▶ If the size is small enough ( $size \leq 32$  or  $size \leq 64$ ), store it in an integer

# bitmasks

Using ints

- ▶ Needed operations:  $\ll$ ,  $\&$ ,  $|$
- ▶ set at index  $i$ : `bitmask |= 1 << i`
- ▶ test at index  $i$ : `bitmask & (1 << i)`