Backtracking explained

Backtracking is the programmer's swiss army knife: Most problems where you can't find another solution for are solved by backtracking.

Therefore I want to describe how to do backtracking using simple examples.

A (only little) more advanced example is solving Sudokus using backtracking as implemented in YasSS, a Sudoku Solver.

The Problem of Placing Eight Queens

A very old (and often solved) Problem in computer science is how to place eight queens on a checkers board in a way that no queen can hit another one.

A checkers board consists of 8x8 cells. Queens may move an arbitrary number of cells vertically, horizontally or diagonally.

To reduce the amount of output that our program will produce, we don't want to know the exact position of the queens, only the number of possible solutions.

More General: Placing N Queens

It's not hard to generalize the problem above to n Queens on a NxN checkers board.

We will write a program that answers the following question:

How many possiblities are there to places N Queens on a NxN checkers board so that no two queens can hit each other.

Now what's Backtracking?

To put it short: backtracking means trying all possiblities, e.g. Trial and Error.

Usually it works this way: When you search for a solution, you try a possibility, and if you are stuck, you try the next one. You repeat that until you found your solution.

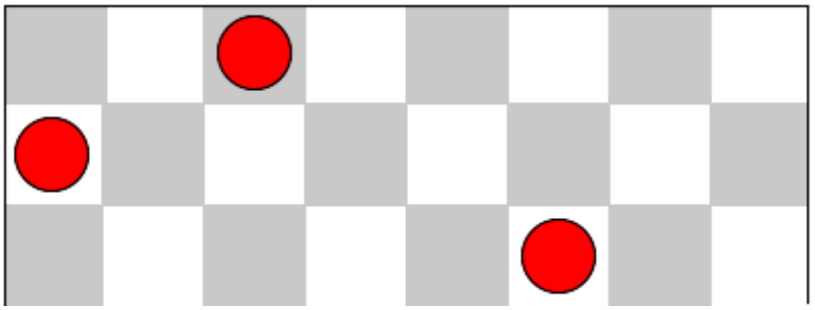
Solution

Let's discuss a solution to the problem above using the programming language C.

First you have to decide how to store the field.

The first idea is often to use a two dimensional array, one byte for each cell.

If you think about it a little bit, you will find that you don't need that much memory. In each row there can only be one queen, so you just need to store where the queen is.



A part of a checkers field, you see the first three rows.

In this example our array describing the field would be [1, 0, 5] (we start to count at zero).

We start by defining a type for our field, and by declaring two global variables. I kno that global variables are evil, but in this small example they are appropriate.

```
typedef signed short int field_type;


int N = 8;
unsigned long int Count = 0;
```

field_type is `signed`, because later we will use differences that can be negative. `N` is the size of the field, and `Count` the number of solutions.

Now it's time to do the work:

```
void search(field_type *field, field_type row){
        field_type i = 0;
        if (row == N){
                /* We have found a solution. We're done, let's return */
                Count++;
                return;
        }
        /* We've got to fill more rows: */
        for (i = 0; i < N; i++){
                field[row] = i;
                if (! collision(field, row)){
                        search(field, row + 1);
                }
        }
}
```

To say it in words: In row `row` place a queen at cell number `0`. If the queen does not collide with another one, start searching in the next row.

Then move on to cell number `1` and so on.

We called the function `int collision(field_type*, field_type)` which we haven't implemted yet. Here it is:

```
int collision(field_type *field, field_type row){
        field_type i = 0;
        field_type tmp = 0;
```

```
        for(i = 0; i < row; i++){
                tmp = field[i] - field[row];
                if (tmp == 0 || tmp == i - row || tmp == row - i){
                        return 1;
                }
        }
        return 0;
}
```

This function checks, if the queen in row `row` collides with one of the other queens.

Now we just have to call the function `search` with `row = 0` and an empty array.

But we can nearly double the speed if consider that there is a symmetrie: If we exchange the first and the last row, the second and the next-to-last etc., we don't generate or destroy collisions.

So in the first row we only need to check the first half of the row, and then take twice the result.

That has the disadvantage that it only works with an even field size. We could modify our algorithm to work with odd `N`, but it's not worth the trouble.

(It would work like this: set the first row the values 0 .. (N-1)/2, double the result, set the first row to (N-1)/2 + 1, search again and add these two values).

```
int main(int argc, char** argv){
        /* Initialisiere Array */
        field_type field[N];
        field_type i = 0;
        for (i = 0; i < N / 2; i++){
                field[0] = i;
                search(field, 1);
        }
        printf("Number of solutions: %ld (with %d queens)\n", 2 * Count, N);
        return 0;
}
```

Now only routine tasks are left, like including the

apropriate header files and parsing the command line argument to set the field size.

Here is the full source code:

```
/* Copyright by Moritz Lenz (C) 2006. */
/* You may use this file under the terms of the MIT license */
#include <stdio.h>
#include <stdlib.h>

typedef signed short int field_type;

int N = 8;
unsigned long int Count = 0;

int collision(field_type *field, field_type row){
        field_type i = 0;
        field_type tmp = 0;
        for(i = 0; i < row; i++){
                tmp = field[i] - field[row];
                if (tmp == 0 || tmp == i - row || tmp == row - i){
                        return 1;
                }
        }
        return 0;
}

void search(field_type *field, field_type row){
        field_type i = 0;
        if (row == N){
                Count++;
                return;
        }

        for (i = 0; i < N; i++){
                field[row] = i;
                if (! collision(field, row)){
                        search(field, row + 1);
                }
        }
}



int main(int argc, char** argv){
        if (argc == 2) {
                /* Die Groesse des Feldes wurde auf der Kommandozeile */
```

```
                /* angegeben */
                N = atoi(argv[1]);
                if (N < 4 || N % 2 == 1){
                        printf("%s: Warning: Ignoring count '%d' since it "
                                        "must be an event integer > 3",
                                        argv[0], N);
                        N = 8;
                }
        }

        field_type field[N];
        field_type i = 0;
        for (i = 0; i < N / 2; i++){
                field[0] = i;
                search(field, 1);
        }
        printf("Number of solutions: %ld (with %d queens)\n", 2 * Count, N);
        return 0;
}
```

You can download the source code here.

Results

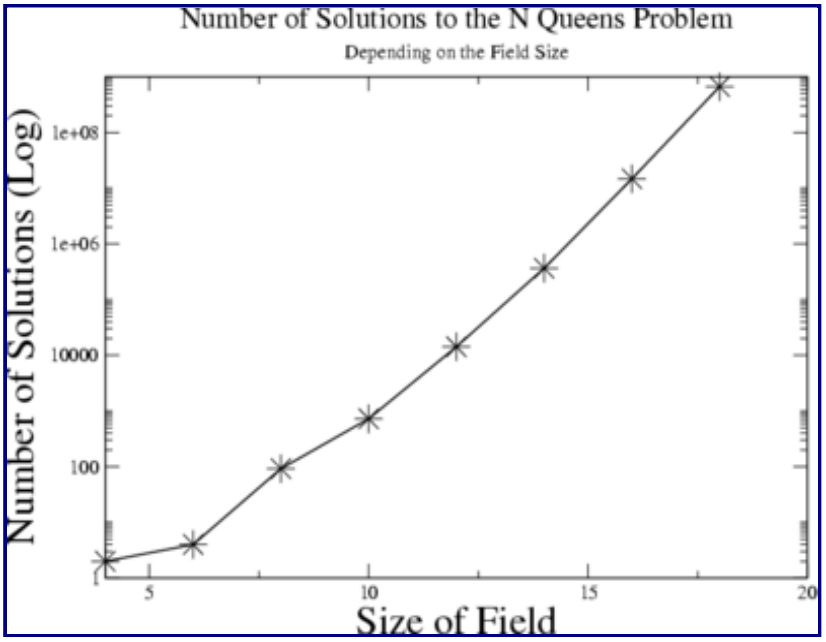
I hope that you know understand a little bit how backtracking works. Apart from that, the results are as follows:

Number of Soltuions

Size	Number of Solutions
4	2
6	8
8	**92**
10	724
12	14200
14	365596
16	14772512
18	666090624

As a chart (logarithmic scaling:)

Click on the image to get a larger version.



(Available as PostScript-file as well)

You can see that the number of solutions increases approximately linear in logarithmic scale, so the overall dependency is about `exp(N)`.

The problem is that with large `N` the program needs much time, the consumed time grows about exponentially as well.
