Backtracking

Dov Kruger

Department of Electrical and Computer Engineering
Stevens Institute of Technology

November 6, 2022

Backtracking is a recursive search that implements the equivalent of n nested loops

In order to understand backtracking

- Get a feeling for exponential functions and their growth
- Learn a couple of algorithms for permutations used in backtracking
- Examine some classic backtracking problems

Overview

Backtracking is a recursive search that implements the equivalent of n nested loops

- Backtracking can solve huge problems
- There must be a way to quickly reject most cases or it is too slow
- Complexity is $O(n^n)$ or $O(n!)$. Polynomial is nothing by comparison!

Comparing Big Functions

n	2^n	10^n	$n!$	n^n
1	2	10	1	1
10	1024	10^{10}	3628800	10^{10}
20	10^6	10^{20}	2.4×10^{18}	10^{26}
30	10^9	10^{30}	2.6×10^{30}	2.05×10^{44}

Reasons why $2^n < 10^n < n! < n^n$

Why isn't $2^n = 10^n$ since complexity $log_2 n = log_3 n$ (differs by a constant)?

Because as n grows, 2n grows by 2 each time, while 10^n grows by 10

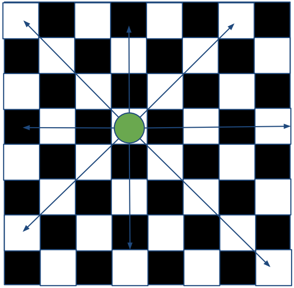
$5^n/2^n = 2.5^n$

10^n	$10 * 10 * 10 * 10 * \ldots$	
$n!$	$n * (n-1) * (n-2) * \ldots$	n $>$ 10
n^n	$n \quad n \quad n \quad n$	

First Problem: n Queens

On an n x n chessboard, place n queens such that none can take each other

Queens in chess can move in any straight line horizontally, vertically, or diagonally



If each row has a queen at one position $n * n * ... * n = n^n$ combinations

However, each column can only be used once:
$n(n-1)(n-2)...1 = n!$ permutations

Even this is an overestimate.

Constraints may be used to limit the number of board positions checked

Minimum Size of N Queens

There are no solutions for n=3

Q shows position of queens, x shows where they cannot be

Q		
x	x	Q
x	x	x

	Q	
x	x	x

		Q
Q	x	x
x	x	x

The first board shows no solution with a queen in the top-left

The second shows one of the solutions for $n = 4$

Q			
x	x	Q	
x	x	x	x

	Q		
x	x	x	Q
Q	x	x	x
x	x	Q	x

n queens requires n nested for loops $O(n^n)$

Problem: changing the size of the problem requires rewriting the code

```
for (int a = 0; a < 4; a++) {
  for (int b = 0; b < 4; b++) {
    if (a == b) // TODO: only checks same column
      continue; // not allowed to use same column
    for (int c = 0; c < 4; c++) {
      if (a == c || b == c) //TODO: only checks column
        continue;
      for (int d = 0; d < 4; d++) {
        if (a == d || b == d || c == d)
          continue;
        //solution here!
```

Backtracking: Recursive Function Equivalent of Nested Loops

There are many backtracking algorithms

We will cover two from Sedgewick

First is direct but requires $2n!$ (still $O(n!)$)

Second is slightly more complicated, but only $n!$

Problems still limited to very small n unless you can truncate search somehow!

Time Estimates for Large Problems

Assuming your CPU can do on the order of 10^9 operations/sec...

n	$n!$	Time
10	3628800	milliseconds
16	2.0×10^{13}	day
17	3.5×10^{14}	days
18	6.4×10^{15}	weeks
19	1.2×10^{17}	months
20	2.4×10^{18}	years

Backtracking, algorithm 1

```
permute(n)
  if n == 0
    // found one, doit!
    return
  end
  for c ← 0 to n
    swap(c, n)
    permute(n−1)
    swap(c, n)
  end
end
```

```
permute(n)
  if n == 0
    // found one, doit!
    return
  end
  for c $\leftarrow$ 0 to n
    permute(n-1)
    if n mod 2 != 0
      swap(0, n)
    else
      swap(i, n)
    end
  end
end
```

Magic Squares

A Magic Square is a number puzzle

- Square of size n x n
- Numbers from 1 to n^2
- Each row, sum and diagonal sums to the same number

1	15	14	4	34
10	11	8	5	34
7	6	9	2	34
16	2	3	13	34

34 34 34 34 34 34

Sudoku Board

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

Complexity Analysis of Sudoku

Brute force: place 9^2 numbers from 1 to 9
on the board: $9^{81} = 1.9 \times 10^{77}$

But they must be unique in each row: $(9!)9 == 1.09 \times 10^{50}$

These two analysis overestimate the complexity because

- There are huge constraints
- Most solutions are not legal

Still, the problem is huge

Building up a Sudoku

First Row

- Any order is equally good
- No constraints
- Straight permutation problem
- $O(9!)$

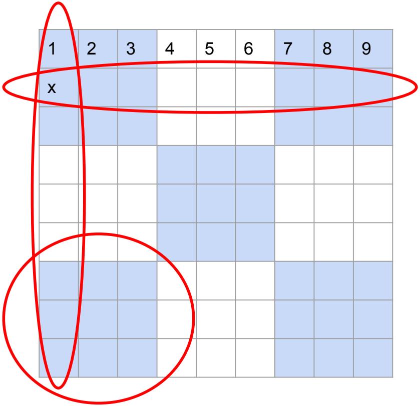
1	2	3	4	5	6	7	8	9

Building up a Sudoku, part 2

Second Row

- Can only select numbers not used already (row, col, box)
- Need a high-speed method to tell
- Bit vectors can help

```
bool usedInRow(int row, int num);
bool usedInCol(int col, int num);
bool usedInBox(int col, int num);
```



Testing Already Used

Bit vector can be used to speed the test for finding the next available candidate in a row

For a 64-bit machine, limits $n \leq 64$ but that is huge

Brute force approach O(9)	bool usedInRow(int row, int num) { for (int col = 0; col < 9; col++) if (grid[row][col] == num) return true; return false; }
Stored as bits O(1)	bool usedInRow(int row, int num) { return bits[row] & (1 « num); }