Gmail **Kyle Lang <kyle.everett.lang@gmail.com>**

An Introduction To Backtracking

1 message

Daily Coding Problem <founders@dailycodingproblem.com> Sun, Sep 1, 2019 at 5:27 PM
To: kyle.everett.lang@gmail.com

Daily Coding Problem

Hey there,

Today I'd like to give you some tips on how to solve backtracking questions. Backtracking is an effective technique for solving algorithmic problems. In backtracking, we search depth-first for solutions, backtracking to the last valid path as soon as we hit a dead end.

Backtracking reduces the search space since we no longer have to follow down any paths we know are invalid. This is called **pruning**. We must be able to test partial solutions: for example, we can't find a global optimum using backtracking, since we have no idea if the solution we're currently on can lead to it or not. But we can, for example, solve Sudoku using backtracking. We can know immediately if our solution so far is invalid by testing if two of the same number appear in the same row, column, or square.

Let's go through several examples of problems that can be nicely solved with backtracking to drill this concept down.

1. The N queens puzzle

The N queens puzzle is the classic backtracking problem. The question is this:

You have an N by N board. Write a function that returns the number of

possible arrangements of the board where N queens can be placed on the board without threatening each other, i.e. no two queens share the same row, column, or diagonal.

Before continuing, you should take some time to try to solve it on your own!

First off, let's describe the brute force solution to this problem, which means trying every single combination of N queens in each of N * N spots. That's n^2 choose n, which is painfully slow. We can immediately improve the runtime of this algorithm by noticing that there's no point in ever placing two queens on the same row (or column), so we really only need to have one queen per row. Now, using brute force, we need to iterate over each row and over each spot on each row. Since we have N rows and N columns, our runtime will be $O(N^N)$. That's still hella slow, though.

It's helpful to ask ourselves three questions to determine whether we can apply backtracking to a problem.

Can we construct a partial solution?

Yes, we can tentatively place queens on the board.

Can we verify if the partial solution is invalid?

Yes, we can check a solution is invalid if two queens threaten each other. To speed this up, we can assume that all queens already placed so far do not threaten each other, so we only need to check if the last queen we added attacks any other queen.

Can we verify if the solution is complete?

Yes, we know a solution is complete if all N queens have been placed.

Now that we are confident that we can use backtracking, let's apply it to this problem. We'll loop through the first row and try placing a queen in column 0..N, and then the second, and so on up until N. What

differs here from brute force is that we'll be adding the queens incrementally instead of all at once.

We'll create an `is_valid` function that will check the board on each incremental addition. `is_valid` will look at the last queen placed and see if any other queen can threaten it. If so, then we prune the branch since there's no point pursuing it. Otherwise, we'll recursively call ourselves with the new incremental solution. We only stop once we hit the base case: we've placed all queens on the board already.

We can represent our board as just a 1D array of integers from 1..N, where the value at the index i that represents the column the queen on row i is on. Since we're working incrementally, we don't even need to have the whole board initialized. We can just append and pop as we go down the stack.

Here's the actual code in Python:

```
def n_queens(n, board=[]):
    if n == len(board):
        return 1

    count = 0
    for col in range(n):
        board.append(col)
        if is_valid(board):
            count += n_queens(n, board)
        board.pop()
    return count

def is_valid(board):
    current_queen_row, current_queen_col = len(board) - 1,
board[-1]
    # Check if any queens can attack the last queen.
    for row, col in enumerate(board[:-1]):
        diff = abs(current_queen_col - col)
        if diff == 0 or diff == current_queen_row - row:
            return False
    return True
```

Let's try it out.

```
for i in range(10):
    print(n_queens(i))
1
1
0
0
2
10
4
40
92
352
```

Looks correct, according to OEIS!

2. Flight itinerary problem

The flight itinerary problem is as follows:

Given an unordered list of flights taken by someone, each represented as (origin, destination) pairs, and a starting airport, compute the person's itinerary. If no such itinerary exists, return null. All flights must be used in the itinerary.

For example, given the following list of flights:

- **HNL → AKL**
- **YUL → ORD**
- **ORD → SFO**
- **SFO → HNL**

and starting airport **YUL**, you should return **YUL → ORD → SFO → HNL → AKL**. (This also happens to be the actual itinerary for the trip I'm about to take.)

You should take some time to try to solve it on your own! Notice that a greedy solution won't work, since it's possible to have a cycle in the graph.

Let's again describe the brute force solution to this problem, which is to try every permutation of flights and verify that it's a valid itinerary. That would be O(n!). Now let's ask ourselves if we can improve this with backtracking.

Can we construct a partial solution?

Yes, we can build an (incomplete) itinerary and extend it by adding more flights to the end.

Can we verify if the partial solution is invalid?

Yes, we can check a solution is invalid if 1) there are no flights leaving from our last destination and 2) there are still flights remaining that can be taken. Since we must use all flights, this means we're at a dead end.

Can we verify if the solution is complete?

Yes, we can check if a solution is complete if our itinerary uses all the flights.

Let's use this to construct our solution:

```
def get_itinerary(flights, current_itinerary):
    # If we've used up all the flights, we're done
    if not flights:
        return current_itinerary
    last_stop = current_itinerary[-1]
    for i, (origin, destination) in enumerate(flights):
        # Make a copy of flights without the current one to
mark it as used
        flights_minus_current = flights[:i] + flights[i + 1:]
        current_itinerary.append(destination)
        if origin == last_stop:
            return get_itinerary(flights_minus_current,
current_itinerary)
        current_itinerary.pop()
    return None
```

3. Sudoku

Here's the problem: solve a well-posed sudoku puzzle.

Let's try using backtracking:

Can we construct a partial solution?

Yes, we can fill in some portions of the board.

Can we verify if the partial solution is invalid?

Yes, we can check that the board is valid so far if there are no rows, columns, or squares that contain the same digit.

Can we verify if the solution is complete?

Yes, the solution is complete when the board has been filled up.

Let's try to solve it using our template. We'll try filling each empty cell one by one, and backtrack once we hit an invalid state. That'll look something like this:

```
EMPTY = 0

def sudoku(board):
    if is_complete(board):
        return board

    r, c = find_first_empty(board)
    # set r, c to a val from 1 to 9
    for i in range(1, 10):
        board[r][c] = i
        if valid_so_far(board):
            result = sudoku(board)
            if is_complete(result):
                return result
        board[r][c] = EMPTY
    return board

def is_complete(board):
    return all(all(val is not EMPTY for val in row) for row
in board)
```

```
def find_first_empty(board):
    for i, row in enumerate(board):
        for j, val in enumerate(row):
            if val == EMPTY:
                return i, j
    return False

def valid_so_far(board):
    if not rows_valid(board):
        return False
    if not cols_valid(board):
        return False
    if not blocks_valid(board):
        return False
    return True

def rows_valid(board):
    for row in board:
        if duplicates(row):
            return False
    return True

def cols_valid(board):
    for j in range(len(board[0])):
        if duplicates([board[i][j] for i in
range(len(board))]):
            return False
    return True

def blocks_valid(board):
    for i in range(0, 9, 3):
        for j in range(0, 9, 3):
            block = []
            for k in range(3):
                for l in range(3):
                    block.append(board[i + k][j + l])
            if duplicates(block):
                return False
    return True

def duplicates(arr):
    c = {}
    for val in arr:
```

```
        if val in c and val is not EMPTY:
            return True
        c[val] = True
    return False
```

Best of luck!

Marc

If you liked this guide, feel free to forward it along! As always, shoot us an email if there's anything we can help with!
