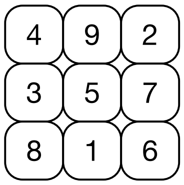
Backtracking I

Backtracking is a very simple method, and sometimes effective method to solve combinatorial problems that avoids full enumeration and relies on recursion.

Abstractly, we want to solve a problem that can be thought of as finding a path subject to various constraints. Whenever the solver encounters a dead-end, the solver backtracks.

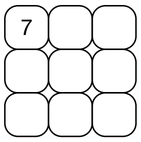
It is probably easiest to understand backtracking using examples. A first example is finding a magic square. A magic square is a 3-by-3 square filled with the numbers between 1 and 9 (ends included) such that the sum the numbers of each row and each column as well as the sum of the numbers in the two diagonals are equal. (Since the sum of the row sums is equal to the sum of the numbers from 1 to 9, i.e. 45, the sums have to be equal to 15). Figure 1 gives one example of a magic square.



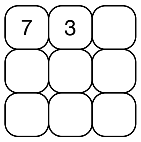
(1) The numbers 1-9 are all used, exactly once
(2) The sum of the numbers in each row is 15
(3) The sum of the numbers in each column is 15
(4) The sum of the numbers in the two long diagonals is 15

Fig 1: A magic square and the conditions on it.

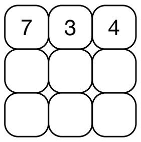
Backtracking starts in one random, empty cell by guessing a number among the available numbers.



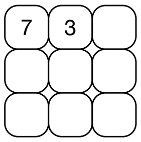
Then the solver calls itself again finding another cell to fill in. So far so good, this is a good *partial* solution.



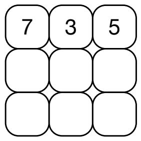
After filling in this random number, we have a problem. The square is no longer a valid *partial* solution, it is *infeasible.*



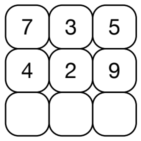
So, we undo what we did and backtrack and get to the previous state.



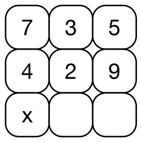
After possibly some repetition, we have guessed a number that makes the solution partially valid.



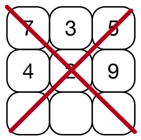
We then continue assigning. Let's say we reached this state:



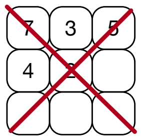
Whatever we assign to the lower left corner, will violate a constraint. The only x that does not lead to a violation that the sum of the first column is 15 is 4, but that violates the condition that all numbers are different. This means that there is no solution from the previous state and we roll back.



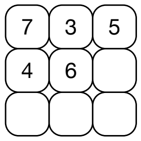
So, in this case, the solver having tried all possibilities, decides that the position on the right is infeasible.



This implies that the previous state is also infeasible.



Therefore, the solver backtracks one more step and assigns a new value to the center cell, trying out whether this one will work.



To be able to backtrack, we need to remember our past path, and for this we can use a stack. However, computing already uses stacks, namely the call stack. A programmer can implement backtracking by just using recursion judiciously.

There are three components of a backtracking program, or more precisely, three aspects of a problem that can be solved with backtracking:

1. Can we solve the problem by iteratively developing partial solutions?
2. Can we verify if the partial solution is invalid?
3. Can we verify if the problem is complete.

In the case of the magic square, we have partial solutions that fill in some but not all cells. A partial solution is invalid if it uses numbers outside of the range and re-uses at least one of the numbers. A solution is complete, if all the cells are filled in.

In this case, we can build a backtracking program using the following scheme:

```
def solver(board):
     If complete return board
     Find next step
     For all possible ways to take the next step:
          take step this way
          if we are in a valid partial state:
               result = solver(board)
               if result is complete:
                    return result
               undo the step
     Return board
```

When we are in a certain state, we try out all possible positions. If we find a feasible position, we recursively continue from this position in line (7). Otherwise, we undo.

Implementation:

Our underlying data structure is a two-dimension core-Python matrix. (Aficionados of NumPy will role their eyes.) We model a non-filled in cell by a 0 value.

```
matrix = [ [0 for i in range(3)] for j in range(3) ]
```

We check whether we are done by testing if there are still zeroes in the matrix.

```
def is_complete(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                return False
    return True
```

Finding the next possible step to take involves finding a free cell.

```
def find_first_free_cell(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                return i, j
```

To check for feasibility, we check first whether the numbers assigned are all different. Then we check whether the sums of completely filled in rows, columns, and diagonals match. In this code fragment, I left in the commented print statements that I used for debugging as a reminder that you need to check each and every method.

```
def is_valid(board):
    numbers = []
    for i in range(3):
        for j in range(3):
            if board[i][j] != 0:
                if board[i][j] in numbers:
                    #print('numbers not different')
                    return False
                else:
                    numbers.append(board[i][j])
    #all numbers are different
    for i in range(3):
        if board[i][0] and board[i][1] and board[i][2]:
            if board[i][0]+board[i][1]+board[i][2]!=15:
                #print('rows')
                return False
    for j in range(3):
        if board[0][j] and board[1][j] and board[2][j]:
            if board[0][j]+board[1][j]+board[2][j] !=15:
                #print('cols')
                return False
    if board[0][0] and board[1][1] and board[2][2]:
        if board[0][0]+board[1][1]+board[2][2] != 15:
            #print('main diagonal')
            return False
    if board[0][2] and board[1][1] and board[2][0]:
        if board[0][2]+board[1][1]+board[2][0] != 15:
            #print('opp diagonal')
            return False
    return True
```

Finally, we implement the blue-print:

```
def back_track(matrix):
    if is_complete(matrix):
        return matrix
    i, j = find_first_free_cell(matrix)
    for num in range(1,9+1):
        matrix[i][j] = num
        if is_valid(matrix):
            result = back_track(matrix)
            if is_complete(result):
                return result
        matrix[i][j] = 0
    return matrix
```

If you look at this code, it is very in-efficient. Certain conditions are going to be checked over and over again when we can infer heir veracity. Thus, this code is not the fastest. But because we only look at 9^9 possible positions and exclude many of them, it still runs almost instantaneously. In real life, you would spend some time improving the performance of the algorithm. This is not the case with your task, which is big enough that you will notice the execution time.

Your task

You are to use the pattern for backtracking in order to find simple Balanced Incomplete Block Design (BIBD). BIBD were originally invented in the 1930s in order to determine the effect of several causes (such as fertilizing, crop rotation, farming method, irrigation etc.) on farm yield. Ideally, we would want to evaluate the influence of all possible combinations but this quickly becomes impossible. Since then, BIBD have started a Mathematical Community - Design Theory -, but there results have also been used in Computer Science, for example in the layout of disk arrays. Luckily for you, you do not need to know any of this.

A BIBD consists of a set of *blocks* that contain symbols. The blocks have all the same length and no symbol is repeated in a block. Each pair of symbols appears together in the same number of blocks. A block is characterized by:

1. v — The number of different symbols. We will write the symbols as numbers $1, 2, \ldots, v$.
2. b — The number of blocks.
3. k — The number of points in a block.
4. λ — The number of blocks that contain a given pair of blocks.
5. r — The number of blocks that contain a given point.

Here is an example BIBD:

1234, 1235, 1267, 1368, 1456, 1478, 1578, 2378, 2457, 2468, 2568, 3458, 3467, 3567

This BIBD has $v = 8, b = 14, k = 4, \lambda = 3$, and $r = 7$. BIBD are characterized in fact by the parameters (v, k, λ), but that involves a bit of Mathematics.

You are to use backtracking in order to find a BIBD with ten blocks of length three on six elements. Each element has to be five times in a block. Any pair of elements will appear in two blocks together.

You have to use the schema laid out for magic squares. However, this is going to take a long time to execute. So, we use one heuristic, namely that we can add elements to blocks in increasing order.

In backtrack, you have to find the first free spot. This is the first block with a zero and an index into the block. Then you only try numbers that are larger than the maximum number of elements already in the block.

```
nr_blocks = 10
pts_per_block = 3
nr_elements = 6
distinct = 2
blks_with_point = 5


blocks = [ pts_per_block*[0] for _ in range(nr_blocks) ]


def bibd(blocks):
    if is_complete(blocks):
        return blocks
    blk, i = find_first_empty(blocks)
                  #bl is a block, i an index, this is the first block with a 0
    for num in range(max(blk)+1,nr_elements+1):
        %%%%%%%%%%%%
        %%%%%%%%%%%%%%%%%%%%:
            %%%%%%%%%%%%%%%%%%%%
            %%%%%%%%%%%%%%%%%%%:
                %%%%%%%%%%%%%%%%%
        %%%%%%%%%%%
    %%%%%%%%%%%%
```

Extra credit (but I doubt you can do it) equal to 10% on the midterm / final:

Find the example BIBD with $v = 8$, $b = 14$, $k = 4$, $\lambda = 3$, and $r = 7$ via back-tracking. You will need to implement further heuristics into the basic backtracking routine and make the test really efficient in order to use backtracking. If you think you can do it, make sure you are not using your computer for the foresee-able future and that you are not running on battery.

Deliverable:

Your complete code.

The output of your code.

The execution time and the type and rating of your processor (e.g. I5 at 2.2 GHz).