# Algorithms for Data Science
# Lab 10
# Backtracking

Given the code below for the nQueens backtracking problem, you are to implement a backtracking solution for the graphColoring problem (not the weighted version).

A few notes on the nQueens code below. It is implemented with an explicit stack data structure to perform a depth-first search through the state space tree. Included is our stack ADT from Lab 5. Each state is a Python list that represents the current state of the chessboard. For example [1,3] means the queen in Column 0 has been placed in Row 1, and the queen in Column 1 has been placed in Row 3. This list of two items also implies that we have yet to place the queen in Columns 2 and 3 (for four queens). After we have placed all the queens, we have found a solution. This code simply prints out that solution and continues to find more solutions. If we are not at a solution, then the code pushes the feasible children onto the stack for future processing.

It is your task in this lab to implement a similar backtracking solution for graphColoring. Your testing code should look like the following:

```
# Adjacency matrix representation of a graph
# This particular graph is the one from the videos
graph = [[False, True,  False, False, False, True ],
         [True,  False, True,  False, False, True ],
         [False, True,  False, True,  True,  False],
         [False, False, True,  False, True,  False],
         [False, False, True,  True,  False, True ],
         [True, True,  False, False, True,  False]]

colors = ['r', 'g', 'b']

graphColoring(graph, colors)
```

Your backtracking function should take in an adjacency matrix and a list of colors. Note that this is the six-node (A–F) example from the videos/PowerPoint slides. Recall from the lecture that both nQueens and graphColoring can be thought of as a depth-first search through a state space tree. Thus, the code should be very similar. In nQueens, we created a list of columns that had queens placed in them. In graphColoring, you should create a list of nodes that have been colored so far. Thus, that list should start empty just like in nQueens, and when it reaches the number of nodes in the graph you have found a solution. Note that to get the number of nodes from the adjacency matrix representation of the graph you can do the following:

```
n = len(graph)
```

There are a couple of differences in the code. One is the feasibility check. In nQueens we checked to see if the queen we wanted to place conflicted with the row or diagonal of any previously placed queen. In graphColoring we also want to check the previously colored nodes, but here we check to make sure they aren't the same color as the one we are trying to color, if they are adjacent to each other. Note that the adjacencyMatrix form of the graph is highly efficient for answering that particular question.

The other main difference is that for nQueens, after we printed any solution we found, we kept going to find all the other solutions. For graphColoring, I want you to simply stop after finding the first solution. For example, your solution might be

```
['r', 'g', 'r', 'b', 'g', 'b']
```

Meaning node A/0 is colored 'r', node B/1 is colored 'g', and so on.

---

```
# Finds ALL ways to place n nonattacking queens on a n x n board
# NOTE: State[i] is the row for the queen on Column i
# NOTE: There are solutions for n>3

# Stack ADT with list implementation from Lab 5
class MyStack(object):
    def init  (self, type): # Creates an empty list
        self.elemType = type
        self.state = []  # Empty list
    def str  (self): # for print
        return str(self.state)
    def empty(self):
        return len(self.state) == 0
    def push(self, elem): # Adds an element to the top of a stack
        assert type(elem) == self.elemType
        self.state.append(elem)
    def pop(self): # Removes an element from the top of the stack
        if self.empty():
            raise ValueError("Requested top of an empty stack")
        else:
            return self.state.pop()
    def top(self): # Returns the top of a nonempty stack
        if self.empty():
            raise ValueError("Requested top of an empty stack")
        else:
            return self.state[-1]

def nQueens(n):
    # Each state will include only the queens that have been placed so far
    initialState = [] # Initial empty state

    s = MyStack(list) # For a depth first search
    s.push(initialState) # Push the initial state onto the Stack

    # While we still have states to explore
    while not s.empty():
```

```
        currentState = s.pop() # Grab the next state
        currentCol = len(currentState)

        # See if we found a solved state at a leaf node
        # That is, we have filled in every column with a queen
        if currentCol == n:
            print(currentState)  # Display the solution
        else:
            # Produce the state's children (if they are feasible)
            # Note children are produced backward so they come off the
            #  stack later left to right
            for currentRow in range(n,0,-1):
                # Check horizontal and both diagonals of previous queens
                feasible = True
                for previousCol in range(currentCol):
                    if (currentState[previousCol] == currentRow) or \
                       abs(currentState[previousCol]-currentRow) ==
(currentCol - previousCol):
                        feasible = False
                        break
                if feasible:
                    # Create child by making a copy and appending new col
                    childState = currentState.copy()
                    childState.append(currentRow)
                    s.push(childState) # Push child onto data structure

# Testing code (check 4,5,6,7)
for n in range(4,8):
    nQueens(n)
```