**Backtracking and Forward-Checking | N-Queens.**

Thomas Mulvey

3/11/19

Dr. Doboli





**CODE:**

import time

import random

class n\_queens:

def \_\_init\_\_(self, N, queen\_pos):

self.N = N # SIZE OF BOARD

self.queen\_pos = queen\_pos #ARR[i] RETURNS ROW POS OF COL I QUEEN

self.nodes\_visited = 0

# fill queen pos array

for i in range(0,N):

queen\_pos.append(-2\*N) #fill array with junk for now

def checkPositions(self, row, col):

# given a queen in column 'col' and in row 'row', is it valid vs the rest of the board

row\_to\_check = row

diag1\_check = row - col

diag2\_check = row + col

# loop thru queen pos array to validate positions

for c in range(0, len(self.queen\_pos) ):

if c == col:

continue #dont want to validate against self

cur\_row = self.queen\_pos[c]

cur\_diag1 = self.queen\_pos[c] - c

cur\_diag2 = self.queen\_pos[c] + c

if row\_to\_check == cur\_row or diag1\_check == cur\_diag1 or diag2\_check==cur\_diag2:

return False

return True

def print\_sol(self):

res = [ [0] \* self.N for \_ in range(self.N) ]

for i in range(0, self.N):

res[i][self.queen\_pos[i]] = self.queen\_pos[i]+1

for j in range(0, self.N):

s = ""

for k in range(0, self.N):

if res[j][k] == 0:

s += "- " #print("- ")

else:

s+= str(res[j][k]) + " " #print(str(res[j][k]) + " ")

print(s)

class backtracking(n\_queens):

def \_\_init\_\_(self, N, queen\_pos):

n\_queens.\_\_init\_\_(self, N, queen\_pos)

def backtrack(self, col): # col is current col being checked

self.nodes\_visited += 1

if self.N==2 or self.N==3:

print("NO SOLUTIONS EXIST FOR THE GIVEN N")

return False #no sols. exist for n=2 or 3

if self.N==col:

self.print\_sol()

return True #done

for r in range(0, self.N,): #loop over every row trying to get a valid solution

self.queen\_pos[col] = r

if self.checkPositions(r, col) == True:

if self.backtrack(col+1) == True:

return True

else:

self.queen\_pos[col] = -2\*self.N; #failed, go back to def value

class forwardtracking(n\_queens):

def \_\_init\_\_(self, N, queen\_pos):

n\_queens.\_\_init\_\_(self, N, queen\_pos)

self.domains = dict()

temp = [x for x in range(N)]

for i in range(N):

random.shuffle(temp)

self.domains[i]=set(temp[:]) # domains is a dict int->set,

# set is int of rows left in domain

def domain\_wipeout(self, old\_domains, row, col):

#1. eliminate rows

#2. eliminate r-c, diag1

#3. eliminate r+c, diag2

# old\_domains is a copy of current stacks current domains

#1. rows

for i in range(self.N):

if i==col:

continue

old\_domains[i] = (old\_domains[i]).difference({row})

#2. diag1, row-col

dif = row - col

for i in range(self.N):

if i==col:

continue

old\_domains[i] = (old\_domains[i]).difference({i+dif})

#3. diag2, row+col

add = row + col

for i in range(self.N):

if i==col:

continue

old\_domains[i] = (old\_domains[i]).difference({add-i})

return old\_domains

def forwardcheck(self,col, domains):

self.nodes\_visited += 1

# invalid game size 2 or 3.

if self.N == 2 or self.N == 3:

return False

# if N == col, N queens finished

if self.N == col:

self.print\_sol()

return True

# if curent columns domain size is zero, backtrack

if len(domains[col])==0:

return False

# while there is still a row to try in current domain

while (domains[col]).\_\_len\_\_() >0 :

r = (domains[col]).pop()

# assume current row is good

self.queen\_pos[col] = r

# if valid

if self.checkPositions(r, col) == True:

# domain wipeout!

temp = domains.copy()

new\_domains = self.domain\_wipeout(temp, r, col )

# if any of the domains are 0, kill stack

valid = True

for i in range(col+1,self.N):

if new\_domains[i].\_\_len\_\_() == 0 :

valid = False

if not valid:

break

# print("CUR COL : " + str(col))

if self.forwardcheck(col+1, new\_domains) == True:

return True

else:

self.queen\_pos[col] = -2\*self.N

def main():

print("n\_queens mulvey\n")

n = int(input("ENTER size of board ? : "))

print("\n====!!BACKTRACKING!!====")

start\_time = time.time()

a = backtracking(n, [])

a.backtrack(0)

end\_time = time.time() - start\_time

print("BACKTRACKING took " + str(end\_time) + "secs to run (and print) with " + str(a.nodes\_visited) + " nodes visited")

print("\n\n====!!FORWARD-CHECING!!====")

start\_time = time.time()

b = forwardtracking(n, [])

b.forwardcheck(0, b.domains)

end\_time = time.time() - start\_time

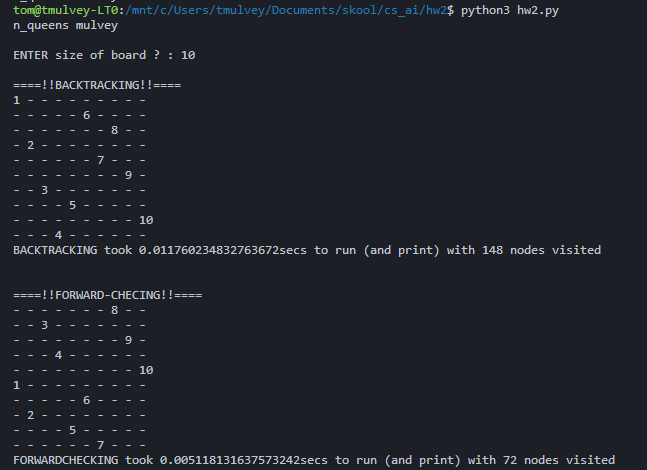
print("FORWARDCHECKING took " + str(end\_time) + "secs to run (and print) with " + str(b.nodes\_visited) + " nodes visited")

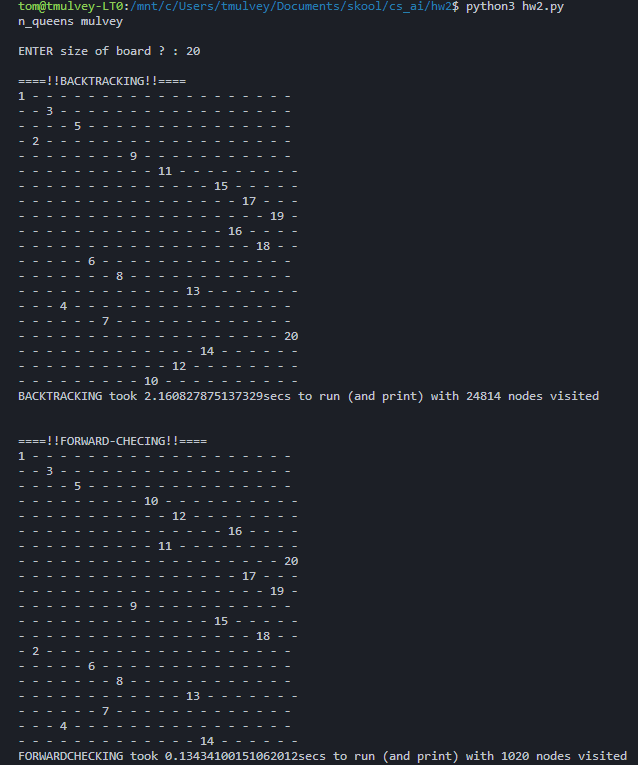
return

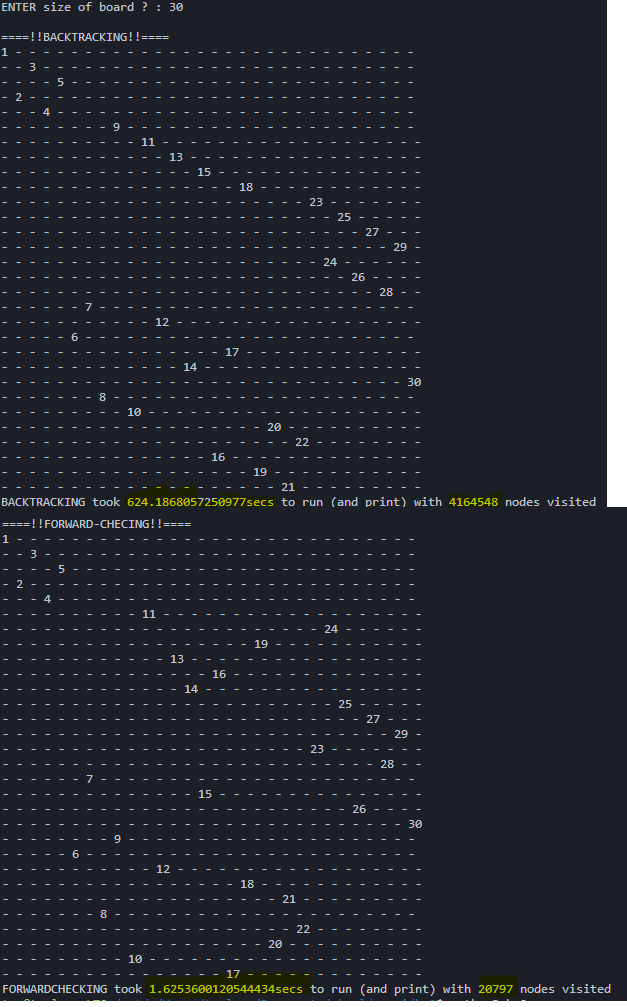
#-----#

main()

**SAMPLE OUTPUTS**





****



**COMPARISONS:**

For Backtracking vs Forward-Checking, I only ran tests for sizes 10, 20, and 30. Since backtracking took over 10 minutes for N=30, I decided that was enough of that. I ran My Forward-Checking for N=40 and N=50. Here is a table of N, Time Elapsed, and Nodes Visited (with a 1 CPU core and 0.5GB RAM VM).



For smaller Ns, N<10, Backtracking seems to visit about 2x as many nodes as the forward-checking will. When N got a little larger, 20, backtracking visited 24x more nodes than forward checking. At the largest tested N, 30, backtracking visited over **200x** more nodes than its counterpart! With my forward-checking algorithm, 40x40 board seemed to be the limit because N=50 took 48 minutes to run and visited 15,426,815 nodes. The total # of possible nodes for a NxN board is . Considering for N=50, there are 9.0630451\*10^84 nodes, and only 1.54x10^7 were visited, that is pretty good, but can be improved upon.

**CONCLUSION:**

Forward checking was always faster than backtracking due to the fewer nodes it had to visit. Much more calculations had to be done for forward-checking, but they obviously paid off as you can see in the chart above. Backtracking basically brute forced its way through the NxN chessboard, while forward-checking did the same but was able to tell when a future Queen would have nowhere to go, and it would go back before going through many other possibilities where that same queen would also fail. The forward checking can still be improved upon by implementing a Priority Queue of the domain lengths of each queen. The priority queue would try the smallest domain (most constrained variable). Within that domain, it would choose the value/row that would eliminate the least amount of values from other domains (Least constraining value heuristic). The most constrained variable and least constraining value heuristic would make the forward-checking even faster.

Both backtracking and forward-checking are exponential in terms of time, and the forward-checking with the two heuristics is also exponential time complexity. Space complexity wise, backtracking will be better off than its forward-checking counterparts because they will require extra space for domains and advanced data structures. The extra space used by forward-checking does result in many fewer nodes visited than backtracking though, and in CSP like N-queens, every solution is a valid one, so why not go get the fastest one? When implementing N-Queens recursively, it does not explicitly show the backtracking. Following the stack traces though, does show giving up on a certain queen spot and trying a different one. Like mentioned earlier, the Forward-checking can be improved upon with a PriorityQueue().

There is also another approach called “Hill Climbing” which is a very rudimentary heuristic. It goes as follows:

* Select an initial (random or pre-determined) assignment of Queens
* WHILE there are threatened queens
  + Select Random Threatened Queen
  + Move that queen to another row that minimizes conflicts

The heuristic is # of queens threatening that piece, and obviously the lower the better. This is much faster than the forward checking or backtracking but is a heuristic-based approach. Russel and Norvig claim in their book “Artificial Intelligence: A Modern Approach” that they had an average of 50 moves for 1 million queens with this method. I would assume they used more efficient methods (for example, break a recursive stack after x iterations of trying to move the queen), but this method would be the fastest to solve many queens.

My forward-checking algorithm was not too great, and it would probably benefit greatly with a priority queue implemented with the two heuristics. It would still be slower than hill climbing. Hill climbing will “pick best random” and forward-checking will brute-force, but take some of the branching depth and width down. A simple Hill-Climbing algorithm for me was able to perform N=50 in 27 seconds and N=100 in 521 seconds, so not as good as Russel and Norvig’s implementation.

