CP 468 Term Project: N-Queens with Min-Conflicts Algorithm

Jacob Cabral, Mathumithan Manimaran, Adrian Vuong, Muhammad Ali, Gavan Singh
CP486 - Artificial Intelligence, section A
Dr. Ilias S. Kotsireas
December 4, 2022

Table of Contents

Discussion of Design Choices

The program implements a CSP data structure that holds the domains, variables and constraints. This is done in order to make the access of the main pieces of the CSP easier during program execution and keep things neat. The Board itself is also a data structure, containing the value of n, the queens positions on the rows and diagonals. This was implemented to keep track of the queens and update their constraints quickly. The board class also contains helper methods to place, remove and get the number of conflicting queens. These methods are implemented to remove boilerplate code when working on this project. Of note, setting and removing a queen automatically updates the constraints.

With regards to execution, the program has a max_steps, this is simply to ensure that the program does not run for an extended period of time with no result, after the maximum steps had been reached the program will halt execution and recommend increasing maximum steps, which can be done via altering the variable in the definition for the min_conflict. If the value of N is less than or equal to 100, a simulated chessboard will be drawn with “-” representing an empty space and “Q” representing a queen placed on the board. However if the board becomes too large, the board is printed to a text file named “output.txt” which is inside the program’s directory.

How to compile and execute the N-Queens solver

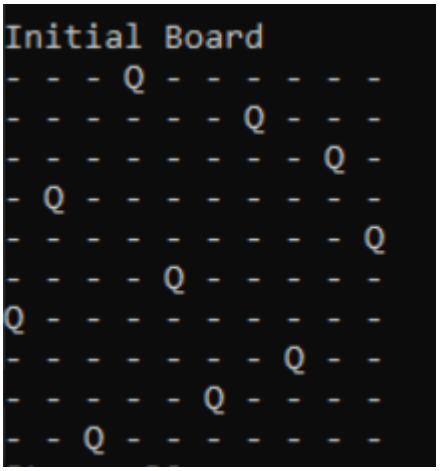
The Solver is composed of two files, main.py and n-queens.py. Ensure both files are in the same directory. You can run the file with your command prompt window by using the ‘cd’ command to navigate to the directory that the N queens Solver files are in, alternatively, if you have installed Python from the Windows Store, you can follow the command to open a PowerShell window at the current location you are in. Regardless, once you have navigated to the directory of both files execute the command; “python3 main.py” to run the program. If all is successful, you will be prompted for a value of N, enter it and press enter. The program will then return your desired results.

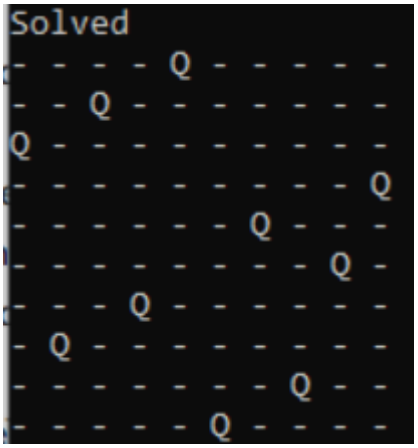
Results for certain Specified Values of N

We are required to submit results of values for n = 10, 100, 1000, 10000, 100000

N = 10

Steps:10
Solve Time: 0.00100 seconds





N = 100

Steps:31
Solve Time: 0.00100 seconds

Note: post n = 100, the size of the board would become too large, thus boards above 100 x 100 will not be shown here.

N = 1000

Steps:31
Solve Time: 0.018 seconds

N = 10000

Steps:55
Solve Time: 0.27400 seconds

N = 100000

Steps:
Solve Time: seconds

N = 1000000

Steps:
Solve Time: seconds

Poster with graphical representation of solution

N-Queens using CSP and Min-Conflicts

Gavan Singh, Adrian Vuong, Jacob Cabral, Muhammad Ali, Mathumithan Manimaran

Faculty of Science - Wilfrid Laurier - CP468 - Fall 2022

Objectives

Our objectives for this project are as stated given:

- Find any n-queen solution within the shortest possible time.
- Find any n-queen solution with the largest possible n.
- Optimize any possible solution with the knowledge used within the textbook.

Introduction

The algorithmic problem noted **N-Queens**, is as follows; given that an n by n board exists, what is any possible orientation of queens that can exist on a board that can produce no possible winners in one move, in other words, which possible orientation of queens could make it so that no queens interfere with each other or can possibly kill one another. Given the knowledge gained throughout the course, a good solution for this problem is the most beneficial outcome.

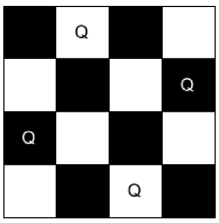


Figure 1:An example solution of 4-Queens

Solution: Min Conflict Algorithm

A search algorithm to solve constraint satisfaction problems.
ADVANTAGES:

- It is very fast: can solve N-Queens in almost constant time

DISADVANTAGES:

- May not search entire state space: only begins from initial state
- Doesn't allow worse moves: Can be caught on local maxima
- Might not find a solution even if one exists

Min Conflict Algorithm Process

This algorithm has a simple process:

- Begin with a random board state
- 1: count the conflicts, if != 0, continue. If 0 conflicts, problem solved
- 2: Randomly select an element (queen) that is attacking or being attacked
- 3: Assign the queen a position on the board that minimizes the number of conflicts
- Our algorithm places queens in separate rows, shifts by column

Figure 2:Algorithm Process

Solution for n = 100

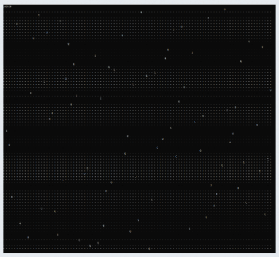


Figure 3:Result for n=100, any value of n past this becomes too large to display properly.

Design Choices

The program implements a CSP data structure that holds the domains, variables and constraints. This is done in order to make the access of the main pieces of the CSP easier during program execution and keep things neat. The Board itself is also a data structure, containing the value of n, the queens positions on the rows and diagonals. The board class also contains helper methods to place, remove and get the number of conflicting queens. These methods are implemented to remove boilerplate code when working on this project.

Compiling and Executing

The Solver is composed of two files, main.py and n-queens.py.

- Ensure both files are in the same directory.
- Using the 'cd' command to navigate to the directory that the N queens Solver files are in.
- IF PYTHON IS INSTALLED FROM WINDOWS STORE **THEN**:
- Follow the command to open a PowerShell window at the current location you are in.
- Once both files are navigated and in the same directory.
- Execute command 'python3 main.py' to run the program
- GREAT SUCCESS!
- you will be prompted for a value of N, enter it and press enter. The program will then return your desired results.

Code Implementation

main.py

```
import time
from nqueens import *
from random import choice

#option = input("Choose value of
n:\n1.10\n2.100\n3.1000\n4.10000\n5.100000\n6.10000000\n")
n = int(input("Choose value of n.\n"))
max_steps = int(input("Choose value of max_steps\n"))
# match option:
#     case "1":
#         n = 10
#     case "2":
#         n = n*n
#     case "3":
#         n = n*n*n
#     case "4":
#         n = n*n*n*n
#     case "5":
#         n = n*n*n*n*n
#     case "6":
#         n = n*n*n*n*n*n
#     case _:
#         print("Invalid value")

# Initialize the board as well as the other variables like the constraints, domains and
variables
start_time = time.time()
board = Board(n)
all_vars = [i for i in range(1, n+1)]
constraints = {i: 0 for i in all_vars}
y = choice(all_vars)
doms = {1: y}
all_vars.remove(y)
board.place_queen(1, y, constraints)

# Iterates on the top row of the board to find the most optimal placements of the first queen
for i in range(2, n+1):
    y = least_conf(i, all_vars, board)
    doms[i] = y
    board.place_queen(i, y, constraints)
    all_vars.remove(y)

# Creates a CSP so that all the constraints, domains and variables are all on
cspqueens = CSP(all_vars, doms, constraints)
```

```
# Prints the Initial Board, given it is under 100 x 100
if(n <= 100):
    print('\nInitial Board')
    init_board = [['-']*n for i in range(n)]
    for key, val in cspqueens.doms.items():
        if constraints[key] > 0:
            init_board[val-1][key -1] = 'Q'
        else:
            init_board[val - 1][key - 1] = 'Q'

    for i in init_board:
        for j in i:
            print(j, end=' ')
        print()

solve_time = time.time()

# Calculates the optimal board using the min conflicts algorithm
assign = min_conflicts(cspqueens, n, board, max_steps)

# Prints the Final Board, given it is under 100 x 100
if assign:
    end_time = time.time()
    print('Solve Time: {:0.5f} seconds'.format(end_time - solve_time))
    print('Total Time: {:0.5f} seconds'.format(end_time - start_time))

    if(n <= 100):
        print('\nSolved')
        solved = [['-']*n for i in range(n)]
        for key, val in assign.doms.items():
            solved[val - 1][key - 1] ='Q'

        for i in solved:
            for j in i:
                print(j, end=' ')
            print()

        print("printing to file")
        # Also prints final board to file using coordinates, rather than a graphic
    with open("output.txt", "w") as f:
        for i in cspqueens.doms:
            print(i,cspqueens.doms[i], file = f, end = "\n")

else:
    print('This cannot be solved, please increase max steps or do a different value of n!')
```

nqueens.py

```
from random import choice

class Board:
    # Initializes the board with diagonal values
    def __init__(self, n):
        self._n = n
        self.rows = {i: set() for i in range(1, n+1)}
        # Defines the forward diagonal
        self.fordiag = {i: set() for i in range(1, 2 * n)}
        # Defines the opposite diagonal
        self.oppdiag = {i: set() for i in range(1, 2 * n)}

    # Removes the constrains of a queen as an auxiliary function
    def remove_aux(self, x, y):
        self.rows[y].remove(x)
        self.fordiag[y+(x-1)].remove(x)
        self.oppdiag[y+(self._n - x)].remove(x)

    # Adds the constrains of a queen as an auxiliary function
    def add_aux(self, x, y):
        self.rows[y].add(x)
        self.fordiag[y+(x-1)].add(x)
        self.oppdiag[y+(self._n - x)].add(x)

    # Combined the constraints of the row, forward diagonal and the opposite diagonal using
bitwise OR
    def combined_constraints(self, x, y):
        return self.rows[y] | self.fordiag[y+(x-1)] | self.oppdiag[y + (self._n - x)]

    # Places a queen on the board and updates the constraints
    def place_queen(self, x, y, constraints):
        comb_constraints = self.combined_constraints(x, y)
        for i in comb_constraints:
            constraints[i] += 1
        self.add_aux(x, y)
        constraints[x] = len(comb_constraints)

    # Removes a queen on the board and updates the constraints
    def remove_queen(self, x, y, constraints):
        comb_constraints = self.combined_constraints(x, y)
        for i in comb_constraints:
            constraints[i] -= 1
        self.remove_aux(x, y)
        constraints[x] = 0

    # Determines the number of conflicts for the
```

```
    def conflict_count(self, x, y):
        comb_constraints = self.combined_constraints(x, y)
        return len(comb_constraints)

class CSP:
    # Initializes for the CSP
    def __init__(self, var , doms, constraints):
        # Initializes the Variables, Domains and Constraints for use with CSP class
        self.var = var
        self.doms = doms
        self.constraints = constraints

# The min conflicts algorithm
def min_conflicts(cspqueens, n, board, max_steps):
    # Using Tabu list, this makes local search signiicantly faster and avoid repeats
    var_list = {}
    removed_queen = None
    x = 0

    if(n>=100):
        x = 50
    else:
        x = n//2

    for i in range(1, max_steps+1):
        # Creates a list of conflicts within the CSP
        conflicted_list = [i for i, j in cspqueens.constraints.items() if j != 0]
        if conflicted_list == []:
            print('Current Steps: {}'.format(i))
            return cspqueens
        # Removes queen if it exists within the conflict list
        if removed_queen is not None and removed_queen in conflicted_list:
            conflicted_list.remove(removed_queen)
        # Takes a random queen from the conflicted list and removes it from the board, and
then updates the constraints
        random_queen = choice(conflicted_list)
        removed_queen = random_queen
        board.remove_queen(random_queen, cspqueens.doms[random_queen],
cspqueens.constraints)
        if random_queen in var_list:
            if cspqueens.doms[random_queen] not in var_list[random_queen]:
                var_list[random_queen].append(cspqueens.doms[random_queen])
        else:
            var_list[random_queen] = [cspqueens.doms[random_queen]]
        # Checks if there is conflicting variables within the board and calculates the
        conflict_val = conflicts(random_queen, n, var_list[random_queen], board)
        if len(var_list[random_queen]) >= x: var_list[random_queen].pop(0)
        cspqueens.doms[random_queen] = conflict_val
```

```
            board.place_queen(random_queen, conflict_val, cspqueens.constraints)
        return False

# Calculating a queen with the least conflicts heuristic
def least_conf(x, is_possible, board):
    list_of_conflicts = [is_possible[0]]
    mincounts = board.conflict_count(x, is_possible[0])
    # Checks the conflict count of x given that it is possible
    for i in is_possible[1:]:
        count = board.conflict_count(x, i)
        if mincounts > count:
            mincounts = count
            list_of_conflicts = [i]
        elif mincounts == count:
            list_of_conflicts.append(i)
    # Returns with a random queen within the list of conflicts if none are found
    return choice(list_of_conflicts)

# Calculates the conflicts for the min conflicts algorithm
def conflicts(x, n, impossible, board):
    list_of_conflicts = []
    mincounts = None
    # Checks if certain moves are impossible, and if there are places that currently have no
conflicts
    for i in range(1, n+1):
        count = board.conflict_count(x, i)
        if mincounts is not None and mincounts > count:
            mincounts = count
            list_of_conflicts = [i]
        elif mincounts is not None and mincounts == count:
            list_of_conflicts.append(i)
        elif mincounts is None:
            mincounts = count
            list_of_conflicts = [i]
    clist = list(set(list_of_conflicts) - set(impossible))
    # Returns with a random queen from a substraction of the list of conflicts and list of
impossible moves
    if clist != []:
        return choice(clist)
    # Returns with a random queen within the conflict list if none are found
    return choice(list_of_conflicts)
```