Solving the n-Queens Problem using Local Search

Instructions

Total Points: undergrad 10, graduate students 11

Complete this notebook and submit it. The notebook needs to be a complete project report with

- your implementation (you can use libraries like math, numpy, scipy, but not libraries that implement inteligent agents or search algorithms),
- documentation including a short discussion of how your implementation works and your design choices, and
- experimental results (e.g., tables and charts with simulation results) with a short discussion of what they mean.

Use the provided notebook cells and insert additional code and markdown cells as needed.

The n-Queens Problem

- **Goal:** Find an arrangement of n queens on a $n \times n$ chess board so that no queen is on the same row, column or diagonal as any other queen.
- **State space:** An arrangement of the queens on the board. We restrict the state space to arrangements where there is only a single queen per column. We represent a state as an integer vector $\mathbf{q} = \{q_1, q_2, \dots, q_n\}$, each number representing the row positions of the queens from left to right. We will call a state a "board."
- **Objective function:** The number of pairwise conflicts (i.e., two queens in the same row/column/diagonal). The optimization problem is to find the optimal arrangement $\mathbf{q}^*$ of n queens on the board can be written as:

  > minimize: $\mathrm{conflicts}(\mathbf{q})$
  >
  > subject to: $\mathbf{q} \ \text{contains only one queen per column}$

  Note: the constraint (subject to) is enforced by the definition of the state space.
- **Local improvement move:** Move one queen to a different row in its column.
- **Termination:** For this problem there is always an arrangement $\mathbf{q}^*$ with $\mathrm{conflicts}(\mathbf{q}^*) = 0$, however, the local improvement moves might end up in a local minimum.

Helper functions

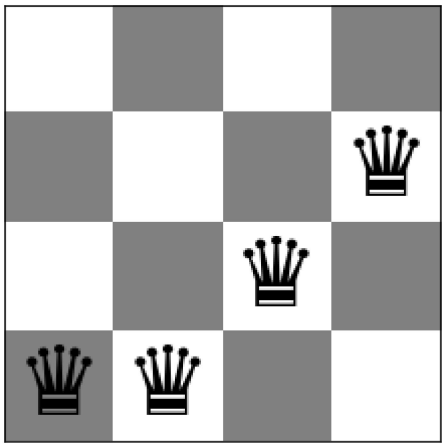
In [18]:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors

np.random.seed(1234)


def random_board(n):
    """Creates a random board of size n x n. Note that only a single queen is placed in

    return(np.random.randint(0,n, size = n))

def comb2(n): return n*(n-1)//2 # this is n choose 2 equivalent to math.comb(n, 2); //

def conflicts(board):
    """Caclulate the number of conflicts, i.e., the objective function."""

    n = len(board)

    horizontal_cnt = [0] * n
    diagonal1_cnt = [0] * 2 * n
    diagonal2_cnt = [0] * 2 * n

    for i in range(n):
        horizontal_cnt[board[i]] += 1
        diagonal1_cnt[i + board[i]] += 1
        diagonal2_cnt[i - board[i] + n] += 1

    return sum(map(comb2, horizontal_cnt + diagonal1_cnt + diagonal2_cnt))


def show_board(board, cols = ['white', 'gray'], fontsize = 48):
    """display the board"""

    n = len(board)

    # create chess board display
    display = np.zeros([n,n])
    for i in range(n):
        for j in range(n):
            if (((i+j) % 2) != 0):
                display[i,j] = 1

    cmap = colors.ListedColormap(cols)
    fig, ax = plt.subplots()
    ax.imshow(display, cmap = cmap,
              norm = colors.BoundaryNorm(range(len(cols)+1), cmap.N))
    ax.set_xticks([])
    ax.set_yticks([])

    # place queens. Note: Unicode u265B is a black queen
    for j in range(n):
        plt.text(j, board[j], u"\u265B", fontsize = fontsize,
                 horizontalalignment = 'center',
                 verticalalignment = 'center')

    print(f"Board with {conflicts(board)} conflicts.")
    plt.show()
```

Create a board

In [19]:
```
board = random_board(4)

show_board(board)
print(f"Queens (left to right) are at rows: {board}")
print(f"Number of conflicts: {conflicts(board)}")
```

```
Board with 4 conflicts.
```

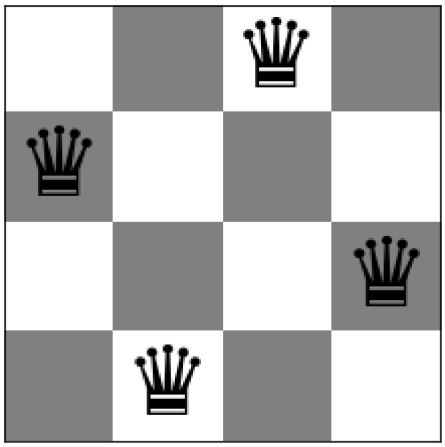


```
Queens (left to right) are at rows: [3 3 2 1]
Number of conflicts: 4
```

A board 4×4 with no conflicts:

In [20]:
```
board = [1,3,0,2]
show_board(board)
```

```
Board with 0 conflicts.
```



Steepest-ascend Hill Climbing Search [3 Points]

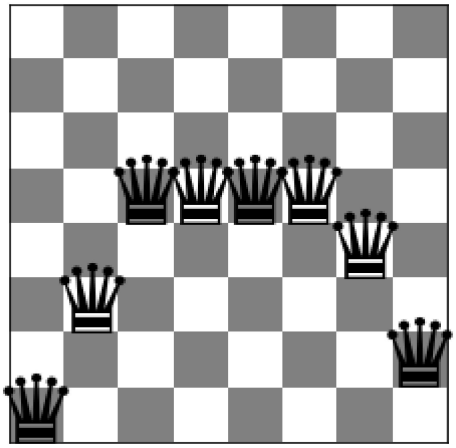
Calculate the objective function for all local moves (move each queen within its column) and always choose the best among all local moves. If there are no local moves that improve the objective, then you have reached a local optimum.

In [46]:
```
def steepest(board):
```

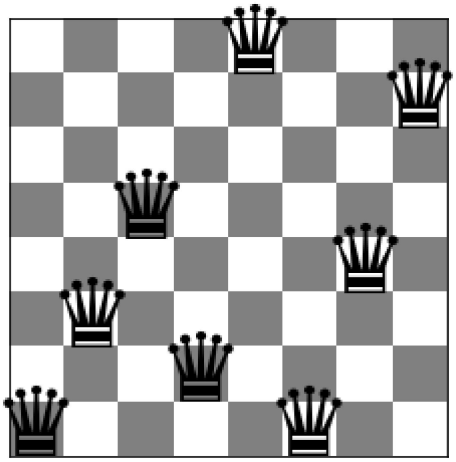
```
    current = board
    if (conflicts(board) == 0):
        return board
    while True:
        temp = list(current)

        successor = calc_successor_steepest(temp)

        if (successor[2] == "Initial State"): #found the optimal solution or a local mi

            return current
        else:

            num = successor[1] - current[successor[0]]

            newNum = current[successor[0]] + num

            current[successor[0]] = newNum




def calc_successor_steepest(boardState):
    n = len(boardState)

    successors = np.zeros([n,n])
    initial = conflicts(boardState)

    minimum = initial
    message = "Initial State"
    x_idx = 0
    y_idx = 0
    for x in range(n):
        successors[boardState[x]][x] = initial  #plug in inital points


    #calculate all potential moves and return the lowest one
    for x in range(n):
        tempState = list(boardState)
        i = boardState[x]
        check = True
        count = 1
        for y in range(n - 1):
            if (i + count < n and check == True):
                tempState[x] = i + count
                count += 1
                successors[tempState[x]][x] = conflicts(tempState)
                num = conflicts(tempState)
                if (num < minimum):
                    y_idx = tempState[x]
                    x_idx = x
                    minimum = num
                    message = "New State"
            else:
                if (check == True):
                    check = False
                    count = 1
                tempState[x] = i - count
                count += 1
                successors[tempState[x]][x] = conflicts(tempState)
                num = conflicts(tempState)
```

```
                if (num < minimum):
                    y_idx = tempState[x]
                    x_idx = x
                    minimum = num
                    message = "New State"

    return (x_idx,y_idx,message)



board = random_board(8)
show_board(board)
optimal = steepest(board)
show_board(optimal)
optimal
```

```
Board with 10 conflicts.
```



```
Board with 1 conflicts.
```



Out[46]:
```
array([7, 5, 3, 6, 0, 7, 4, 1])
```

Stochastic Hill Climbing 1 [2 Point]

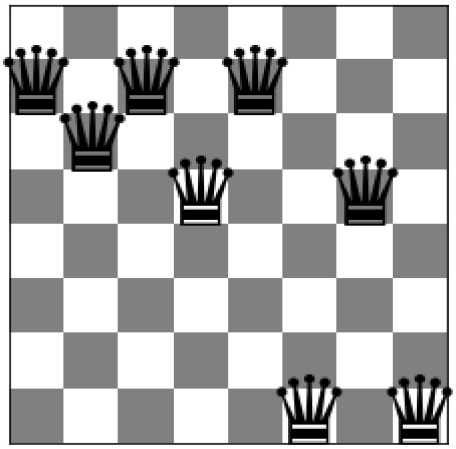
Chooses randomly from among all uphill moves. Make the probability of the choice proportional to the steepness of the uphill move (i.e., with the improvement in conflicts).

In [48]:
```
import numpy as np
```

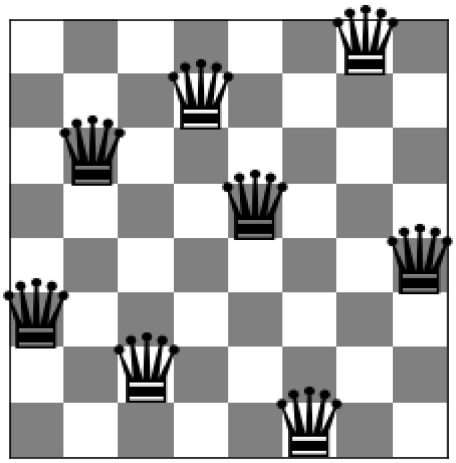
```
def stochastic1(board):
    current = board
    if (conflicts(board) == 0):
        return board
    while True:
        temp = list(current)

        successor = calc_successor_stochastic1(temp)

        if (successor[1] == "Initial State"): #same logic here as steepest ascent

            return current
        else:

            num = successor[0][1] - current[successor[0][0]]

            newNum = current[successor[0][0]] + num

            current[successor[0][0]] = newNum



def calc_successor_stochastic1(boardState):
    n = len(boardState)

    successors = np.zeros([n,n])
    initial = conflicts(boardState)

    message = "Initial State"
    conflictsList = []
    coordinates = []
    conflictsList.clear
    coordinates.clear
    index =0
    for x in range(n):
        successors[boardState[x]][x] = initial

    #calculate all possible moves and add them to
    #a list if they are lower than the initial state
    for x in range(n):
        tempState = list(boardState)
        i = boardState[x]
        check = True
        count = 1
        for y in range(n - 1):
            if (i + count < n and check == True):
                tempState[x] = i + count
                count += 1
                successors[tempState[x]][x] = conflicts(tempState)
                num = conflicts(tempState)
                if (num < initial):
                    conflictsList.append(num)
                    coordinates.append((x,tempState[x]))

                    message = "New State"
            else:
                if (check == True):
                    check = False
                    count = 1
                tempState[x] = i - count
```

```
                count += 1
                successors[tempState[x]][x] = conflicts(tempState)
                num = conflicts(tempState)
                if (num < initial):
                    conflictsList.append(num)
                    coordinates.append((x,tempState[x]))
                    message = "New State"

    totalConflicts = sum(conflictsList)
    probabilities = []

    #pick a random move from the list
    if (totalConflicts != 0):
        for x in range(len(conflictsList)):
            probabilities.append(conflictsList[x]/totalConflicts)
        index = np.random.choice(len(coordinates),p=probabilities)

    if not coordinates:
        return (0,message)
    else:
        return (coordinates[index], message)



board = random_board(8)
show_board(board)
optimal = stochastic1(board)
show_board(optimal)
optimal
```

Board with 9 conflicts.



Board with 0 conflicts.



Out[48]: array([5, 2, 6, 1, 3, 7, 0, 4])

Stochastic Hill Climbing 2 [2 Point]

A popular version of stochastic hill climbing generates only a single random local neighbor at a time and accept it if it has a better objective function value than the current state. This is very efficient if each state has many possible successor states. This method is called "First-choice hill climbing" in the textbook.

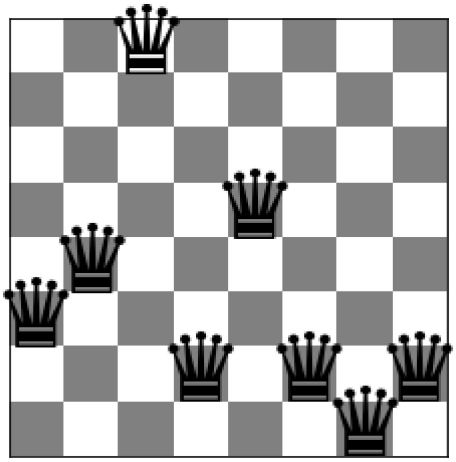
Notes:

- Detecting local optima is tricky! You can, for example, stop if you were not able to improve the objective function during the last x tries.

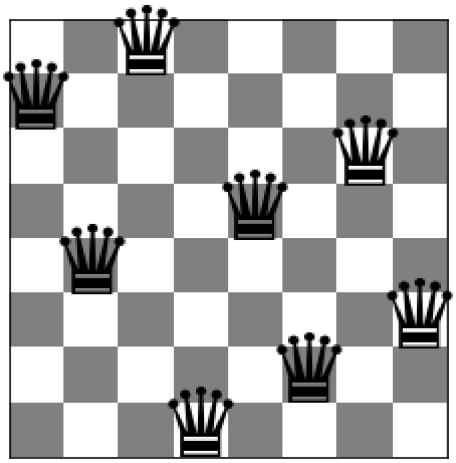
In [49]:
```
import numpy as np
def stochastic2(board,cap):
    current = board
    attempts = 0
    if (conflicts(board) == 0):
        return board
    while True:
        #show_board(current)
        temp = list(current)

        successor = calc_successor_stochastic2(temp)

        temp2 = list(current)

        num = successor[1] - temp2[successor[0]]

        newNum = temp2[successor[0]] + num

        temp2[successor[0]] = newNum

        #if the returned state has the same number of conflicts
        #as the current state, run it again until it either finds
        #a lower state, or it reaches the set cap of iterations
        if (conflicts(temp2) < conflicts(current)):
            attempts = 0
            current[successor[0]] = newNum
```

```
        elif (conflicts(temp2) == conflicts(current)):
            attempts += 1
        if (attempts >= cap):

            return current



def calc_successor_stochastic2(boardState):
    n = len(boardState)



    arr=[]
    for i in range(n):
        arr.append(i)

    #randomly pick a set of coordinates to move to
    x = np.random.choice(arr)
    y = np.random.choice(arr)
    index = (x,y)
    return index




board = random_board(8)
show_board(board)
optimal = stochastic2(board,100)
show_board(optimal)
optimal
```

```
Board with 8 conflicts.
```



```
Board with 2 conflicts.
```



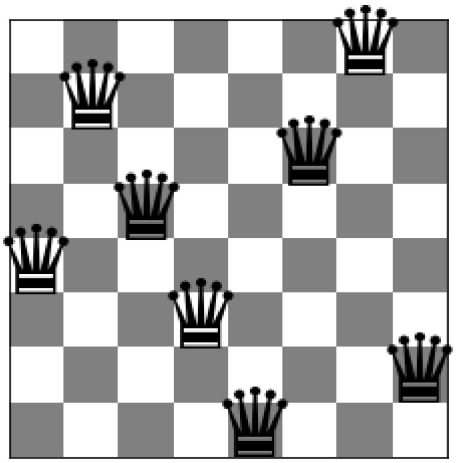
Out[49]: array([1, 4, 0, 7, 3, 6, 2, 5])

Hill Climbing Search with Random Restarts [1 Point]

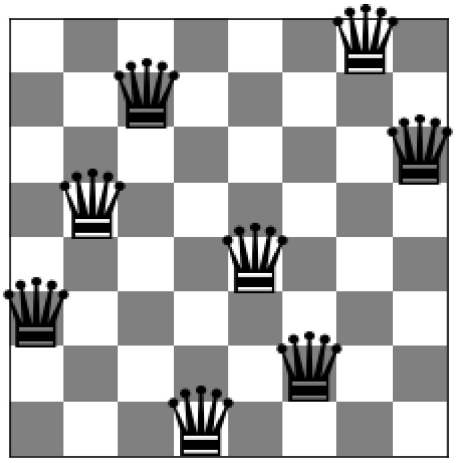
Hill climbing will often end up in local optima. Restart the each of the three hill climbing algorithm up to 100 times with a random board to find a better (hopefully optimal) solution. Note that restart just means to run the algoithm several times with a new random initialization.

In [24]:
```
print("Steepest")
for x in range(100):
    board = random_board(8)
    optimal = steepest(board)
    if (conflicts(optimal) == 0):
        break

show_board(optimal)

print("Stochastic1")
for x in range(100):
    board = random_board(8)
    optimal = stochastic1(board)
    if (conflicts(optimal) == 0):
        break

show_board(optimal)

print("Stochastic2")
for x in range(100):
    board = random_board(8)
    optimal = stochastic2(board,100)
    if (conflicts(optimal) == 0):
        break

show_board(optimal)
```

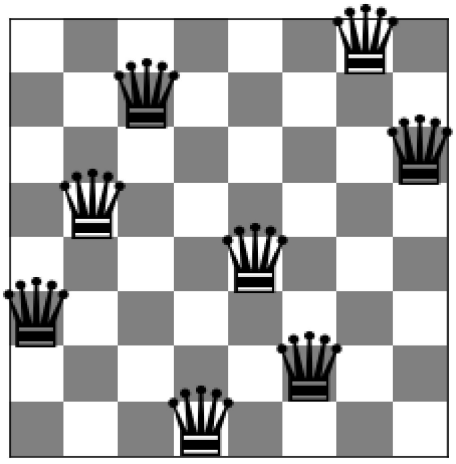
```
Steepest
Board with 0 conflicts.
```



```
Stochastic1
Board with 0 conflicts.
```



```
Stochastic2
Board with 0 conflicts.
```



Running all three algorithms with random restarts over the course of 100 iterations seems to guarantee that they will reach an optimal solution. It would be very unlikely for one of them not to reach 0 conflicts.

Compare Performance [2 Points]

Use runtime and objective function value to compare the algorithms.

- Use boards of different sizes to explore how the different algorithms perform. Make sure that you run the algorithms for each board size several times (at least 10 times) with different starting boards and report averages.
- How do the algorithms scale with problem size?
- What is the largest board each algorithm can solve in a reasonable amount time?

The example below times creating 100 random boards and calculating the conflicts. Reported is the average run time over `N = 100` runs.

For timing you can use the `time` package.

In [34]:

```
import time

N = 100
total = 0

for i in range(N):
    board = random_board(8)
    t0 = time.time()



    steepest(board)
    t1 = time.time()
    total += t1 - t0

tm = total/N
print("Time for 100 iterations of steepest: ")
print(f"This took: {tm * 1e3} milliseconds")

total = 0

for i in range(N):
    board = random_board(8)
    t0 = time.time()



    stochastic1(board)
    t1 = time.time()
    total += t1 - t0

tm = total/N
print("Time for 100 iterations of stochastic1: ")
print(f"This took: {tm * 1e3} milliseconds")

total = 0

for i in range(N):
    board = random_board(8)
    t0 = time.time()
```

```
    stochastic2(board,50)
    t1 = time.time()
    total += t1 - t0

tm = total/N
print("Time for 100 iterations of stochastic2 with a cap of 50: ")
print(f"This took: {tm * 1e3} milliseconds")

total = 0

for i in range(N):
    board = random_board(8)
    t0 = time.time()



    stochastic2(board,1)
    t1 = time.time()
    total += t1 - t0

tm = total/N
print("Time for 100 iterations of stochastic2 with a cap of 1: ")
print(f"This took: {tm * 1e3} milliseconds")
```

```
Time for 100 iterations of steepest:
This took: 5.760011672973633 milliseconds
Time for 100 iterations of stochastic1:
This took: 8.46034288406372 milliseconds
Time for 100 iterations of stochastic2 with a cap of 50:
This took: 31.98038101196289 milliseconds
Time for 100 iterations of stochastic2 with a cap of 1:
This took: 0.33000946044921875 milliseconds
```

The `timit` package is useful to measure time for code that is called repeatedly.

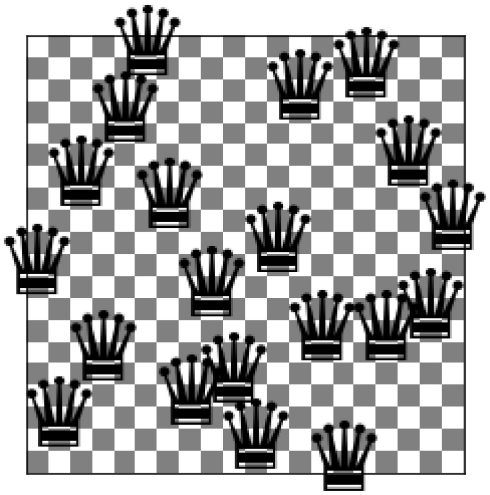
From the time data I just gathered, it is clear to see that stochastic hill climbing that picks one random neighbor and has the lowest cap possible is the quickest. However, running this algorithm with a cap of only 1 runs a very great risk of stopping much too early. Running the same algorithm with a cap of 50 increases the time taken significantly, but will most likely guarantee reaching the optimal solution or a local minimum. This being the case, I believe that steepest ascent is the best of the three in terms of time complexity and how well it performs getting to the solution. I will now try running the algorithms on much larger boards to see how they do.

In [41]:

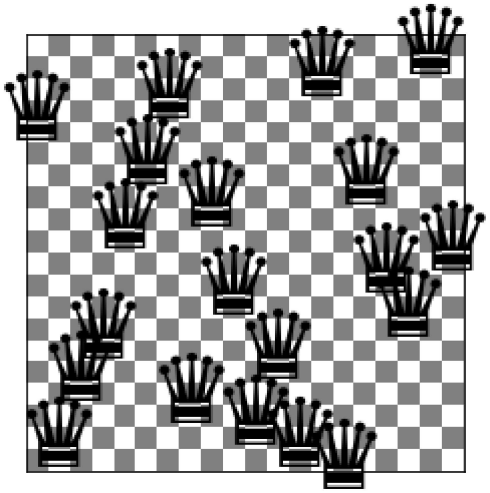
```
total = 0

for i in range(5):
    board = random_board(20)
    t0 = time.time()



    solution = steepest(board)
    t1 = time.time()
    print(conflicts(solution))
    show_board(solution)
    total += t1 - t0
```

```
tm = total/5
print("Time for 5 iterations of steepest on board size 20: ")
print(f"This took: {tm * 1e3} milliseconds")
print("")
print("")
print("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++")
total = 0

for i in range(5):
    board = random_board(20)
    t0 = time.time()



    solution = stochastic1(board)
    t1 = time.time()
    print(conflicts(solution))
    show_board(solution)
    total += t1 - t0

tm = total/5
print("Time for 5 iterations of stochastic1 on board size 20: ")
print(f"This took: {tm * 1e3} milliseconds")
print("")
print("")
print("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++")
total = 0

for i in range(5):
    board = random_board(20)
    t0 = time.time()



    solution = stochastic2(board,100)
    t1 = time.time()
    print(conflicts(solution))
    show_board(solution)
    total += t1 - t0

tm = total/5
print("Time for 5 iterations of stochastic2 with a cap of 100 on board size 20: ")
print(f"This took: {tm * 1e3} milliseconds")
```

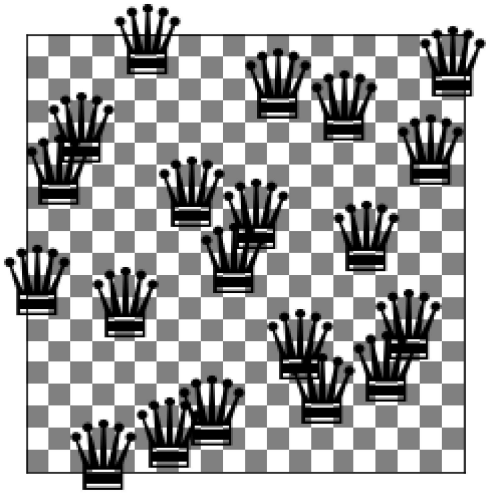
```
1
Board with 1 conflicts.
```



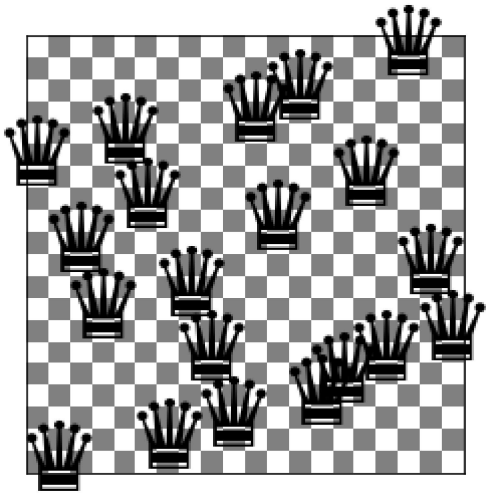
2
Board with 2 conflicts.



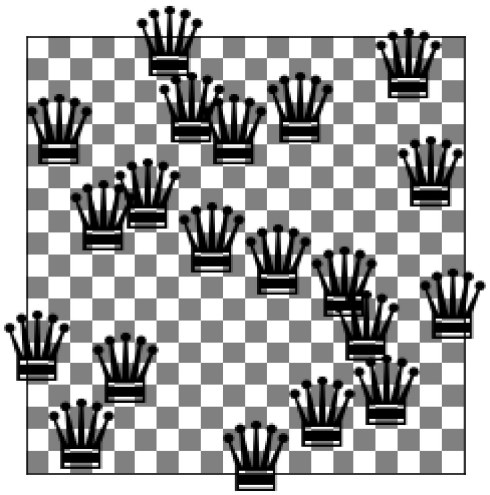
2
Board with 2 conflicts.



2
Board with 2 conflicts.

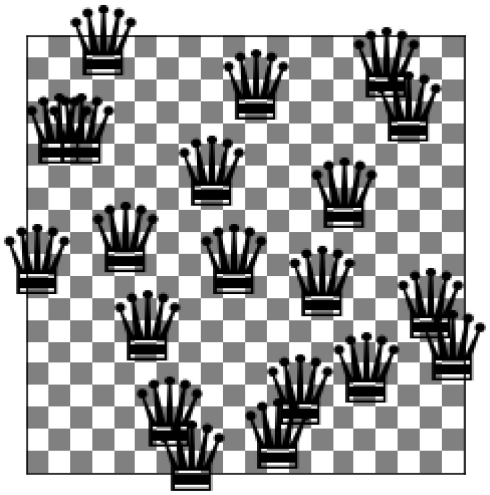


2
Board with 2 conflicts.



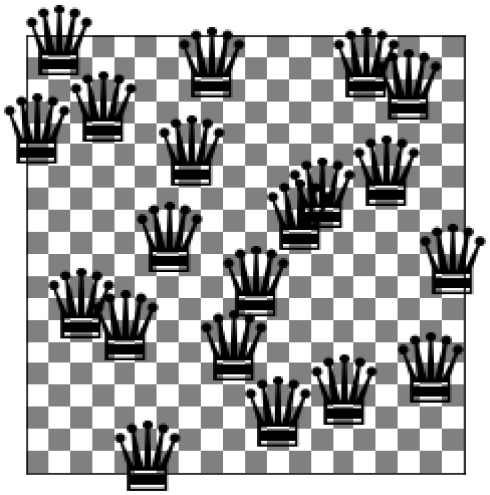
Time for 5 iterations of steepest on board size 20:
This took: 201.42874717712402 milliseconds

++
4
Board with 4 conflicts.

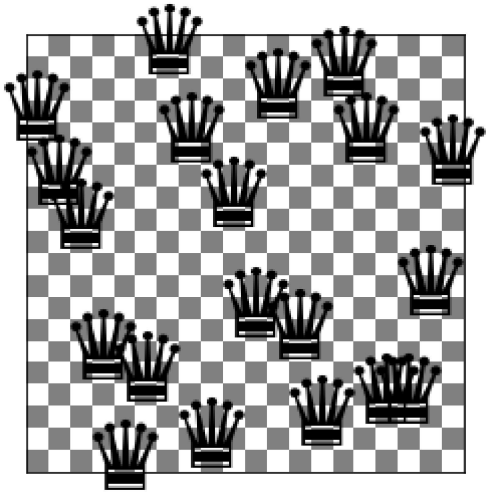


2

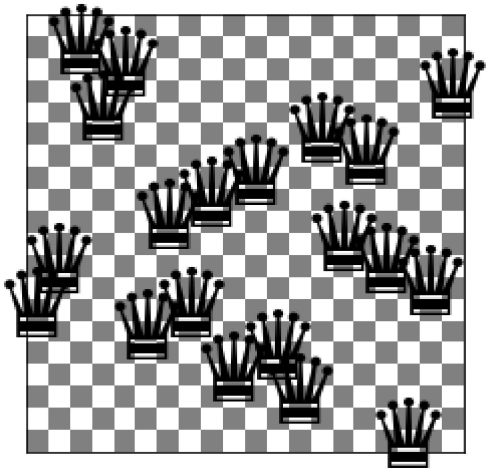
```
Board with 2 conflicts.
```



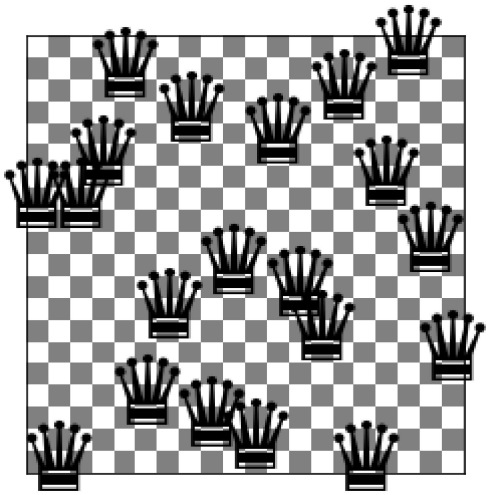
```
3
Board with 3 conflicts.
```



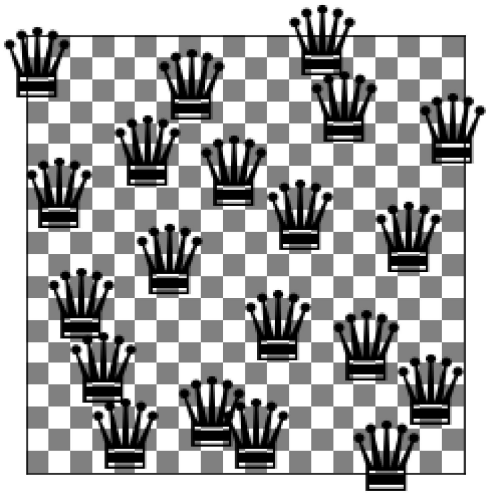
```
2
Board with 2 conflicts.
```



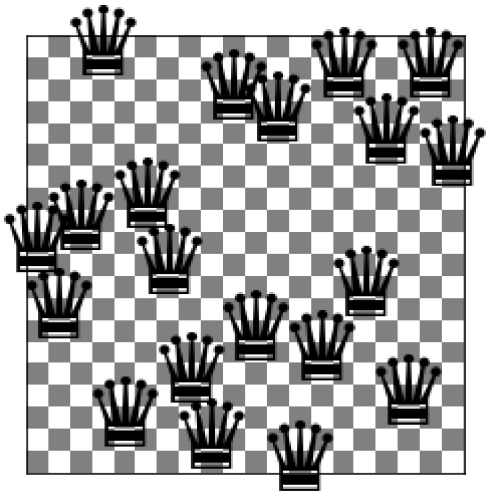
```
2
Board with 2 conflicts.
```



```
Time for 5 iterations of stochastic1 on board size 20:
This took: 342.8374767303467 milliseconds


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
1
Board with 1 conflicts.
```

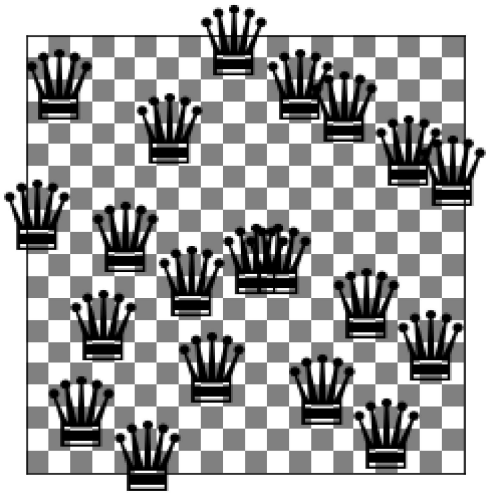


```
2
Board with 2 conflicts.
```

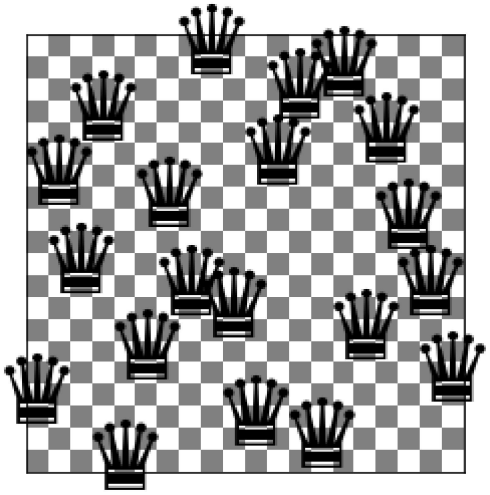


```
2
```

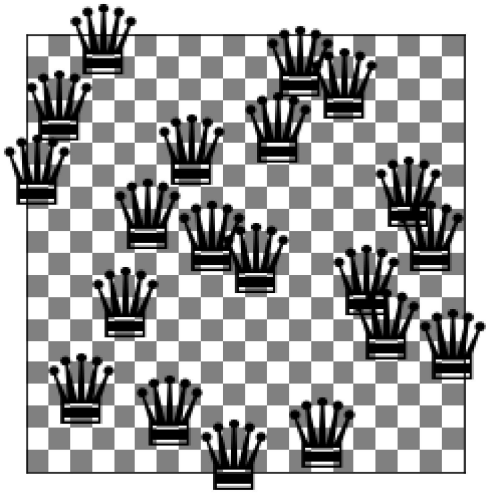
```
Board with 2 conflicts.
```



```
3
Board with 3 conflicts.
```



```
4
Board with 4 conflicts.
```



```
Time for 5 iterations of stochastic2 with a cap of 100 on board size 20:
This took: 290.0290012359619 milliseconds
```

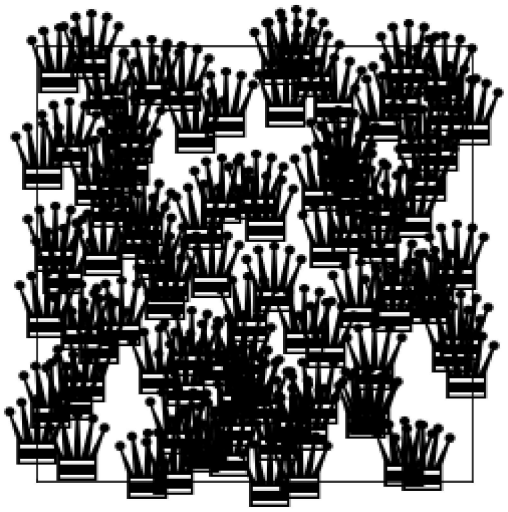
Again, we see similar results, with steepest ascent being the quickest of the three, however this time, stochastic1 was the slowest as opposed to stochastic2. In terms of how good of a solution they got,

all three performed similarly. Now for a 100x100 board.

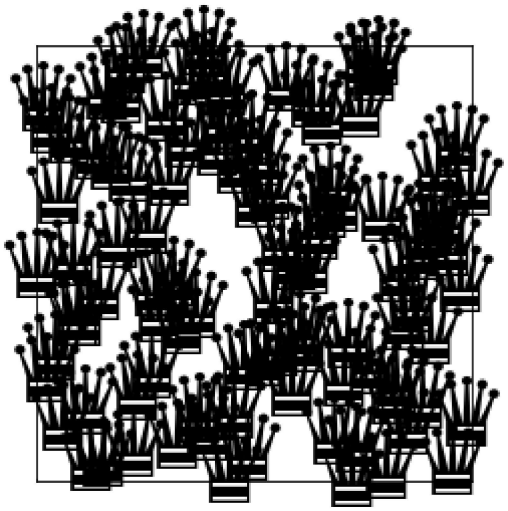
In [43]:

```
total = 0


board = random_board(100)
t0 = time.time()
solution = steepest(board)
t1 = time.time()
print(conflicts(solution))
show_board(solution)
total += t1 - t0

tm = total
print("Time for 1 iteration of steepest on board size 100: ")
print(f"This took: {tm * 1e3} milliseconds")

total = 0


board = random_board(100)
t0 = time.time()
solution = stochastic1(board)
t1 = time.time()
print(conflicts(solution))
show_board(solution)
total += t1 - t0

tm = total
print("Time for 1 iteration of stochastic1 on board size 100: ")
print(f"This took: {tm * 1e3} milliseconds")

total = 0


board = random_board(100)
t0 = time.time()
solution = stochastic2(board,100)
t1 = time.time()
print(conflicts(solution))
show_board(solution)
total += t1 - t0

tm = total
print("Time for 1 iteration of stochastic2 with cap of 100 on board size 100: ")
print(f"This took: {tm * 1e3} milliseconds")
```

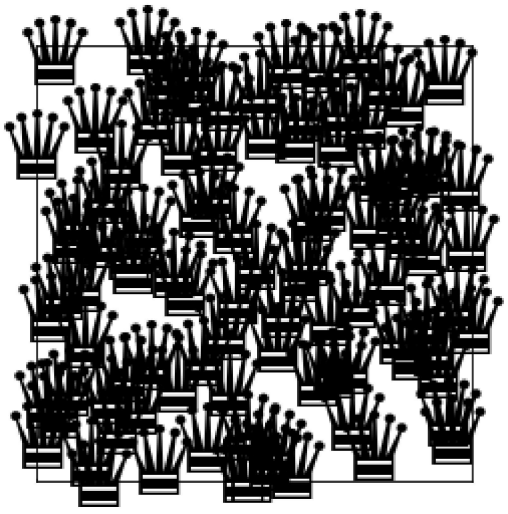
```
5
Board with 5 conflicts.
```



```
Time for 1 iteration of steepest on board size 100:
This took: 118931.4079284668 milliseconds
3
Board with 3 conflicts.
```



```
Time for 1 iteration of stochastic1 on board size 100:
This took: 254209.2685699463 milliseconds
11
Board with 11 conflicts.
```



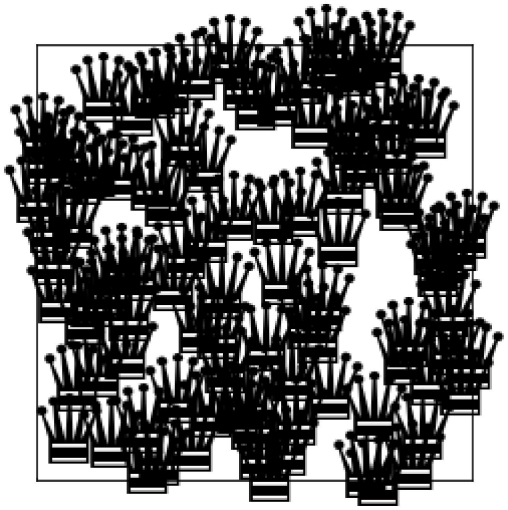
```
Time for 1 iteration of stochastic2 with cap of 100 on board size 100:
This took: 4083.7674140930176 milliseconds
```

Looks like stochastic2 needs a higher cap. Let's try that again.

In [45]:

```
total = 0


board = random_board(100)
t0 = time.time()
solution = stochastic2(board,1000)
t1 = time.time()
print(conflicts(solution))
show_board(solution)
total += t1 - t0

tm = total
print("Time for 1 iteration of stochastic2 with cap of 1000 on board size 100: ")
print(f"This took: {tm * 1e3} milliseconds")
```

```
4
Board with 4 conflicts.
```



```
Time for 1 iteration of stochastic2 with cap of 1000 on board size 100:
This took: 61018.484354019165 milliseconds
```

That is looking much better! From these we can see that as the board gets larger and larger, steepest ascent and stochastic1 take an incredibly long time because they calculate each possible move every time, whereas stochastic2 does not take that long at all. This is because it only has to calculate 1 move each time. Steepest ascent took around twice as long as stochastic2 and stochastic1 took around four times as long. Since they all perfromed pretty well, it is clear to me that stochastic2 is the best algorithm for very large boards.

Graduate student advanced task: Simulated Annealing [1 Point]

Undergraduate students: This is a bonus task you can attempt if you like [+1 Bonus point].

Simulated annealing is a form of stochastic hill climbing that avoid local optima by also allowing downhill moves with a probability proportional to a temperature. The temperature is decreased in

every iteration following an annealing schedule. You have to experiment with the annealing schedule (Google to find guidance on this).

Implement simulated annealing for the n-Queens problem and compare its performance with the previouse algorithms.

In [11]:
```
# Code and description go here
```

More things to do

Implement a Genetic Algorithm for the n-Queens problem.

In [12]:
```
# Code and description go here
```