Solving the n-Queens Problem using Local Search

Instructions

Total Points: Undergrads 100 / Graduate students 110

Complete this notebook. Use the provided notebook cells and insert additional code and markdown cells as needed. Submit the completely rendered notebook as a PDF file.

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 $\mathbf{q} = \{q_1, q_2, \ldots, q_n\}$, 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: $\mathbf{q}$ 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.

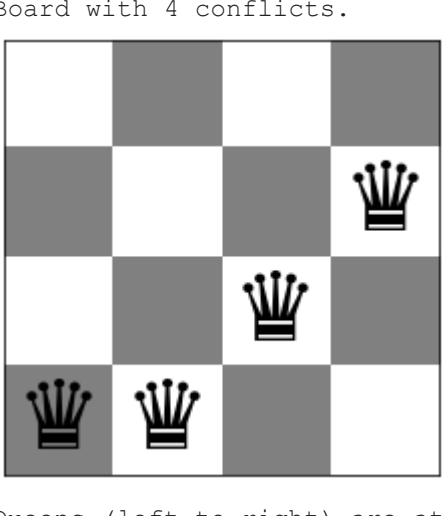
Helper functions

In [127…
```python
import random

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 each column!"""

    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); // is int division

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 [128…
```python
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.
```

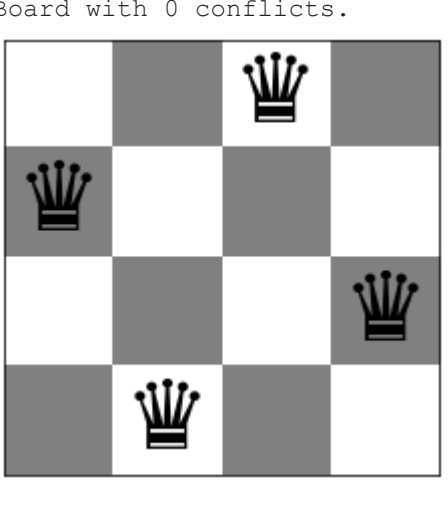


```
Queens (left to right) are at rows: [3 3 2 1]
Number of conflicts: 4
```

A board 4×4 with no conflicts:

In [129…
```python
board = [1,3,0,2]
show_board(board)
```

```
Board with 0 conflicts.
```



Tasks

General [10 Points]

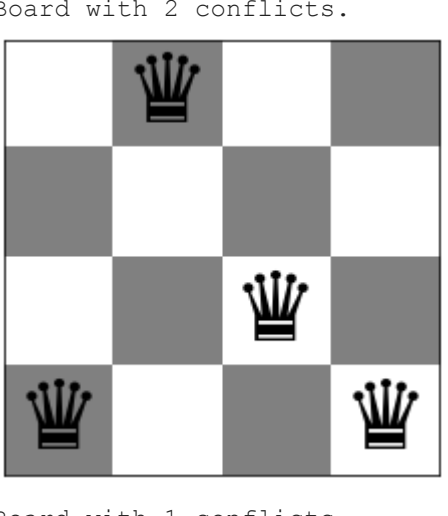
1. Make sure that you use the latest version of this notebook. Sync your forked repository and pull the latest revision.
2. Your implementation can use libraries like math, numpy, scipy, but not libraries that implement inteligent agents or complete search algorithms. Try to keep the code simple! In this course, we want to learn about the algorithms and we often do not need to use object-oriented design.
3. You notebook needs to be formated professionally.
   - Add additional markdown blocks for your description, comments in the code, add tables and use mathplotlib to produce charts where appropriate
   - Do not show debugging output or include an excessive amount of output.
   - Check that your PDF file is readable. For example, long lines are cut off in the PDF file. You don't have control over page breaks, so do not worry about these.
4. Document your code. Add a short discussion of how your implementation works and your design choices.

Task 1: Steepest-ascend Hill Climbing Search [30 Points]

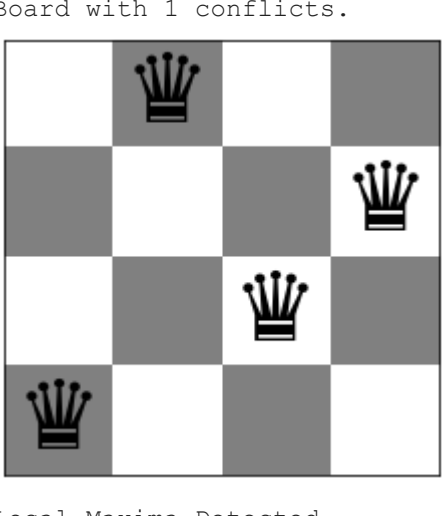
Calculate the objective function for all local moves (see definition of local moves above) and always choose the best among all local moves. If there are no local moves that improve the objective, then you have reached a local optimum.

In [204…
```python
# Code and description go here
def getHighestValuedSuccessor(board):

    highestValue=conflicts(board)
    if highestValue == 0:
        #print("Global Maxima Detected")
        return False
    positionToChange=[0]*2
    ind=0
    betterValue=False
    for i in board:
        hold=i
        for index in range(len(board)):
            if i < len(board)-1:
                i=i+1
            elif i == len(board)-1:
                i=0
            board[ind]=i

            if conflicts(board)<highestValue:
                highestValue=conflicts(board)
                positionToChange[0]=ind
                positionToChange[1]=i
                betterValue=True



        board[ind]=hold
        ind+=1

    if betterValue:
        board[positionToChange[0]]=positionToChange[1]
    if not betterValue and conflicts(board)!=0:
        print("Local Maxima Detected")

    show_board(board)
    return betterValue

def hillClimbing(board):
    cont=True
    while cont is True:
        cont = getHighestValuedSuccessor(board)


board = random_board(4)

hillClimbing(board)
show_board(board)
```

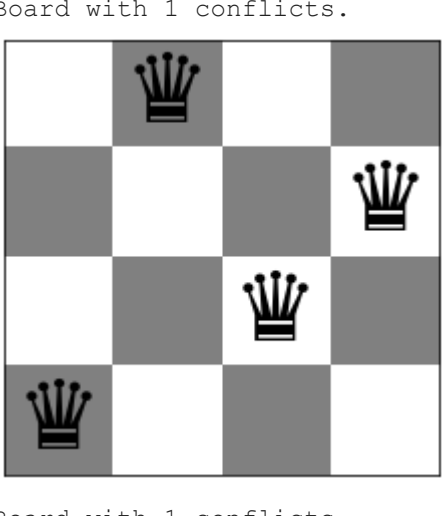
```
Board with 2 conflicts.
```



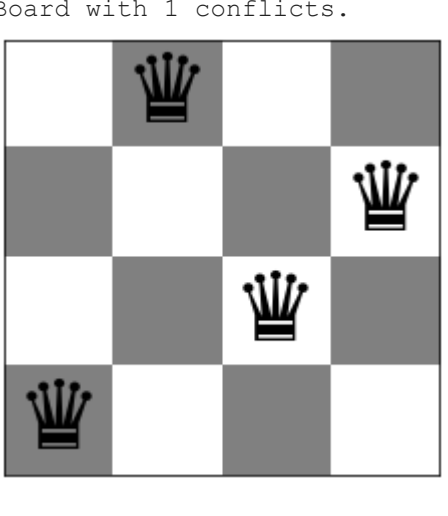
```
Board with 1 conflicts.
```



```
Local Maxima Detected
Board with 1 conflicts.
```



```
Board with 1 conflicts.
```

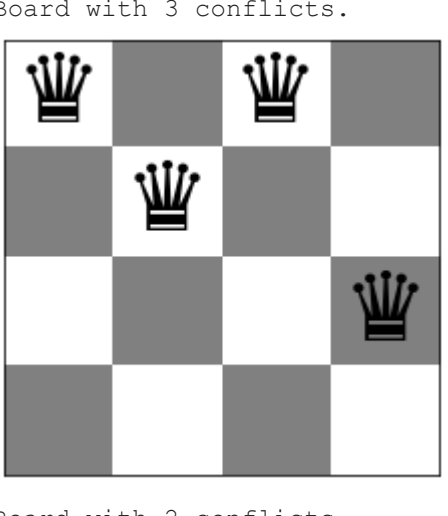


Task 2: Stochastic Hill Climbing 1 [10 Points]

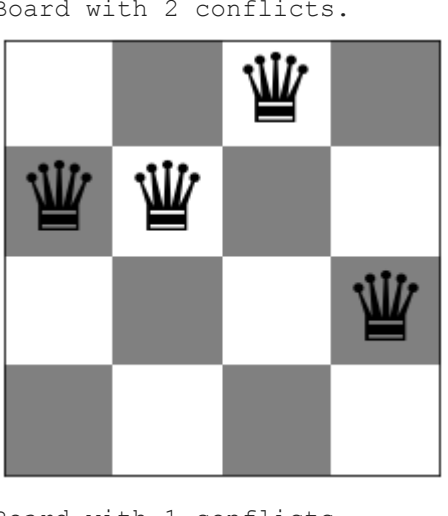
Chooses randomly from among all uphill moves till you have reached a local optimum.

In [206…
```python
# Code and description go here
# Picks the first better neighbor it finds
def getRandomSuccessor(board):
    highestValue=conflicts(board)
    if highestValue == 0:
        #print("Global Maxima Detected")
        return False
    positionToChange=[0]*2
    ind=0
    for i in board:
        hold=i
        for index in range(len(board)):
            if i < len(board)-1:
                i=i+1
            elif i == len(board)-1:
                i=0
            board[ind]=i

            if conflicts(board)<highestValue:
                positionToChange[0]=ind
                positionToChange[1]=i
                board[positionToChange[0]]=positionToChange[1]
                show_board(board)
                return True



        board[ind]=hold
        ind+=1


    if conflicts(board)!=0:
        print("Local Maxima Detected")
        return False
    show_board(board)
    return False


def stochasticHillClimbing1(board):
    cont=True
    while cont is True:
        cont = getRandomSuccessor(board)

board = random_board(4)
show_board(board)
stochasticHillClimbing1(board)
```

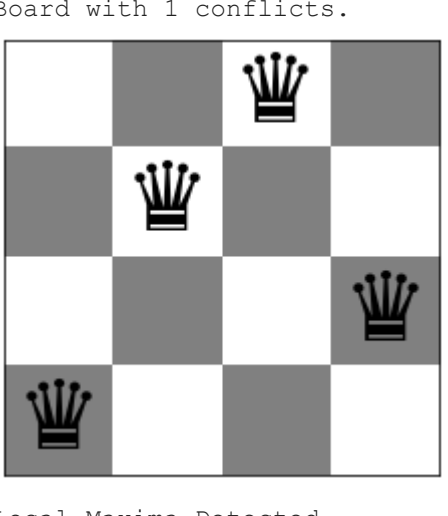
```
Board with 3 conflicts.
```



```
Board with 2 conflicts.
```



```
Board with 1 conflicts.
```



```
Local Maxima Detected
```

Task 3: Stochastic Hill Climbing 2 [20 Points]

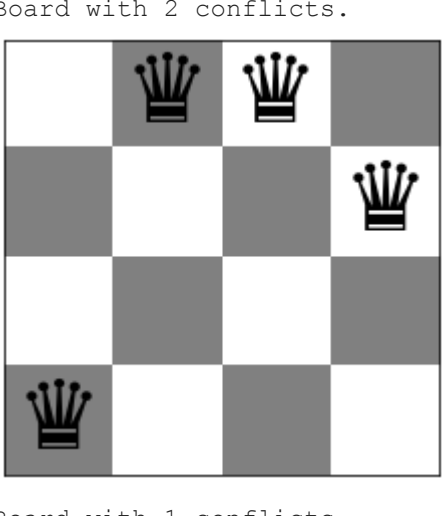
A popular version of stochastic hill climbing generates only a single random local neighbor at a time and accept it if it has a better objective function value than the current state. This is very efficient if each state has many possible successor states. This method is called "First-choice hill climbing"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

Notes:

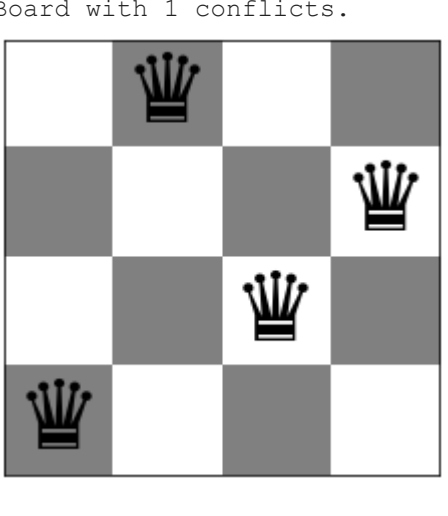
- Detecting local optima is tricky! You can, for example, stop if you were not able to improve the objective function during the last x tries.

In [208…
```python
# Code and description go here
def genRandomNeighbor(board):
    tries=0
    while tries<50:
        currConflicts=conflicts(board)
        if currConflicts==0:
            #print("Global optima found")
            return
        originalIndex=random.randint(0, len(board)-1)
        originalValue=board[originalIndex]
        i=board[originalIndex]
        for index in range(len(board)):
            if i < len(board)-1:
                i=i+1
            elif i == len(board)-1:
                i=0
            board[originalIndex]=i

            if conflicts(board)<currConflicts:
                tries=0
                show_board(board)
                break
            else:
                tries+=1
                board[originalIndex]=originalValue

def stochastic2(board):
    genRandomNeighbor(board)

board = random_board(4)
show_board(board)
stochastic2(board)
```

```
Board with 2 conflicts.
```



```
Board with 1 conflicts.
```



Task 4: Hill Climbing Search with Random Restarts [10 Points]

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 starting with a new random board.

In [133…
```python
# Code and description go here
#steepest
for i in range(10):
    board=random_board(4)
    hillClimbing(board)

#stochastic
for i in range(10):
    board=random_board(4)
    stochasticHillClimbing1(board)

#stochastic 2
for i in range(10):
    board = random_board(4)
    stochastic2(board)
```

Task 5: Compare Performance [20 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? Use tables and charts.
- What is the largest board each algorithm can solve in a reasonable amount time?

See Profiling Python Code for help about how to measure runtime in Python.

In [201…
```python
# Code, results, charts and discussion go here
import time

currConflicts = 0
totalTime=0
loops=1
boardsize=100

# TESTING FOR STEEPEST FIRST
# for i in range(loops):
#     board=random_board(boardsize)
#     start_time = time.time()
#     hillClimbing(board)
#     end_time = 1000*(time.time()-start_time)
#     currConflicts+=conflicts(board)
#     totalTime+=end_time
#
# print(currConflicts/loops)
# print(end_time/loops)


# TESTING FOR STOCHASTIC 1
# currConflicts=0
# for i in range(loops):
#     board=random_board(boardsize)
#     start_time = time.time()
#     stochasticHillClimbing1(board)
#     end_time = 1000*(time.time()-start_time)
#     currConflicts+=conflicts(board)
#     totalTime+=end_time
#
# print(currConflicts/loops)
# print(end_time/loops)


# TESTING FOR STOACHASTIC 2
currConflicts=0
for i in range(loops):
    board=random_board(boardsize)
    start_time = time.time()
    stochastic2(board)
    end_time = 1000*(time.time()-start_time)
    currConflicts+=conflicts(board)
    totalTime+=end_time

print(currConflicts/loops)
print(end_time/loops)
```

```
78.0
23.392915725708008
```

title title title title Once the board size gets to around 30 seconds, the first two algorithms begin to take a relatively long time (>20 seconds). Each of the above data points was the average of 50 simulations.

Graduate student advanced task: Simulated Annealing [10 Points]

Undergraduate students: This is a bonus task you can attempt if you like [+5 Bonus Points].

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).

1. Implement simulated annealing for the n-Queens problem.
2. Compare the performance with the previous algorithms.
3. Discuss your choice of annealing schedule.

In [147…
```python
# Code and description go here
```

More things to do

Implement a Genetic Algorithm for the n-Queens problem.

In [148…
```python
# Code and description go here
```

In [ ]: