Solving the n-Queens Problem using Local Search

Student Name: Vishakha Kishor Satpute

I have used the following AI tools: [list tools]

I understand that my submission needs to be my own work: VS

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: $\text{conflicts}(\mathbf{q})$
  >
  > 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 $\text{conflicts}(\mathbf{q}^*) = 0$, however, the local improvement moves might end up in a local minimum.

Helper functions

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors

np.random.seed(1234)
```

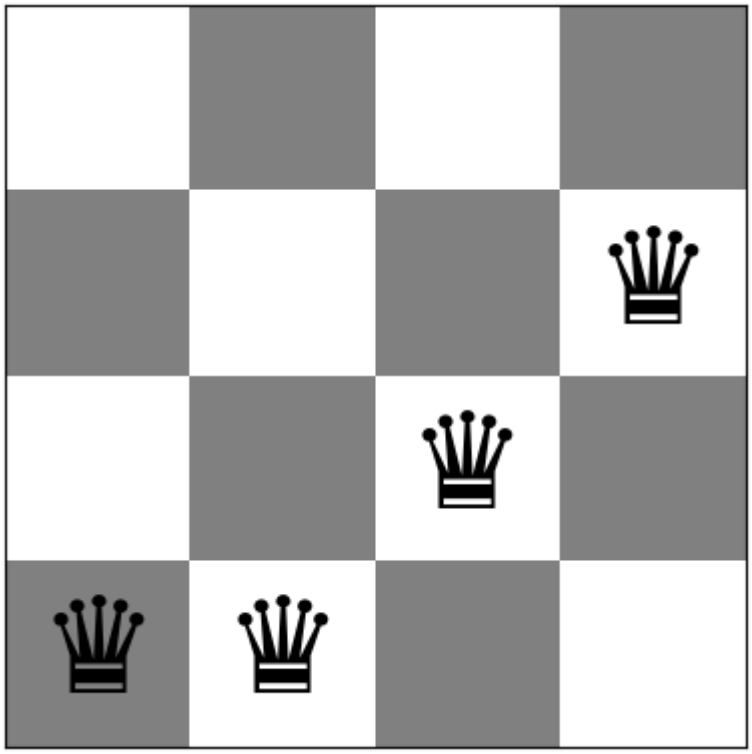
```
def random_board(n):
    """Creates a random board of size n x n. Note that only a single queen is placed

    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):
    """Calculate 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))

# decrease the fontsize to fit larger boards
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 [3]:

```
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.
```



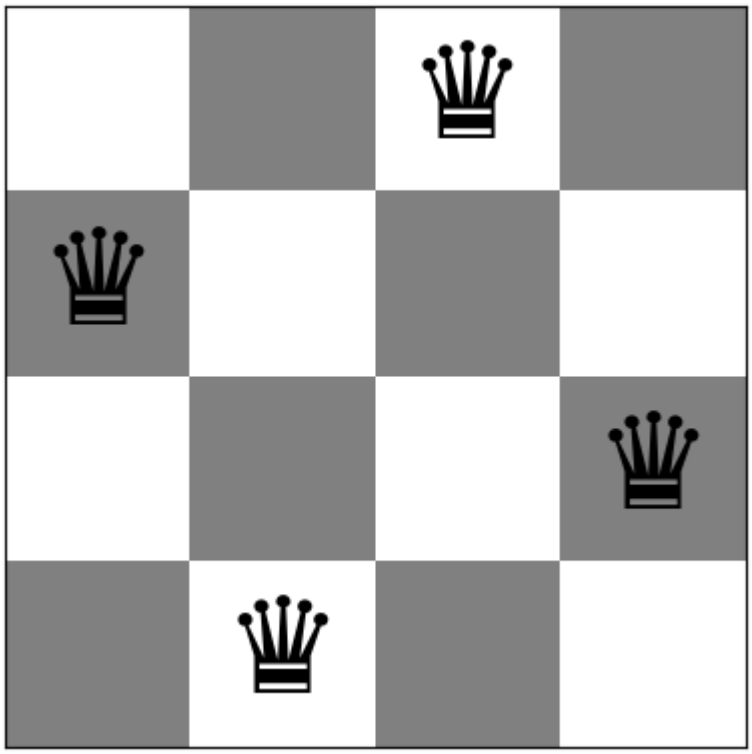
```
Queens (left to right) are at rows: [3 3 2 1]
Number of conflicts: 4
```

A board 4×4 with no conflicts:

In [4]:

```
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 intelligent 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 formatted 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 [117…

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

Below code finds the co-ordinate where the conflicts for the queen is minimum and shift the queen to the co-ordinates where the conflict is minimum.

In [150…

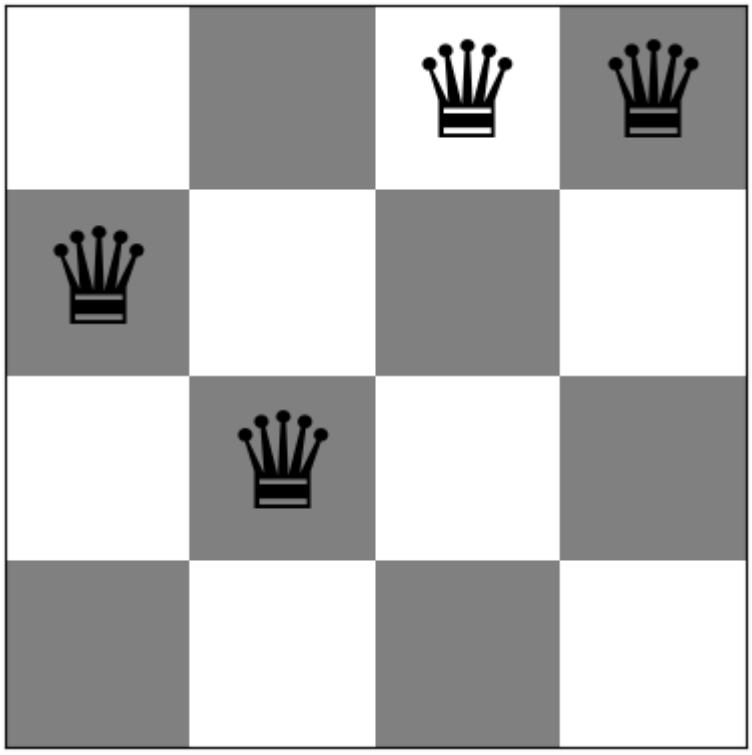
```
def steepest_ascend_hill_climbing(board,verbose = True):
  if verbose: show_board(board)
  n = len(board)
  curr_con = conflicts(board) #finds the current conflict
  bval = [[-1 for _ in range(n)] for _ in range(n)]
  steps = 0
  while True:
    for i in range(n):
      a = board[i]
      for j in range(n):
        board[i] = j
        bval[j][i] = conflicts(board)
      board[i] = a
    con = np.min(bval)

    steps = steps +1

    if(curr_con > con):
      loc = np.where(bval == con)
      new_loc = [a for a in zip(loc[0], loc[1])]
      #print(new_loc)
      new_loc = new_loc[np.random.randint(0, len(new_loc))]
      board[new_loc[1]] = new_loc[0]
```

```
      curr_con = con
      if verbose: show_board(board)
    else: return(board)
  print(steps, "Number_of_steps")
```

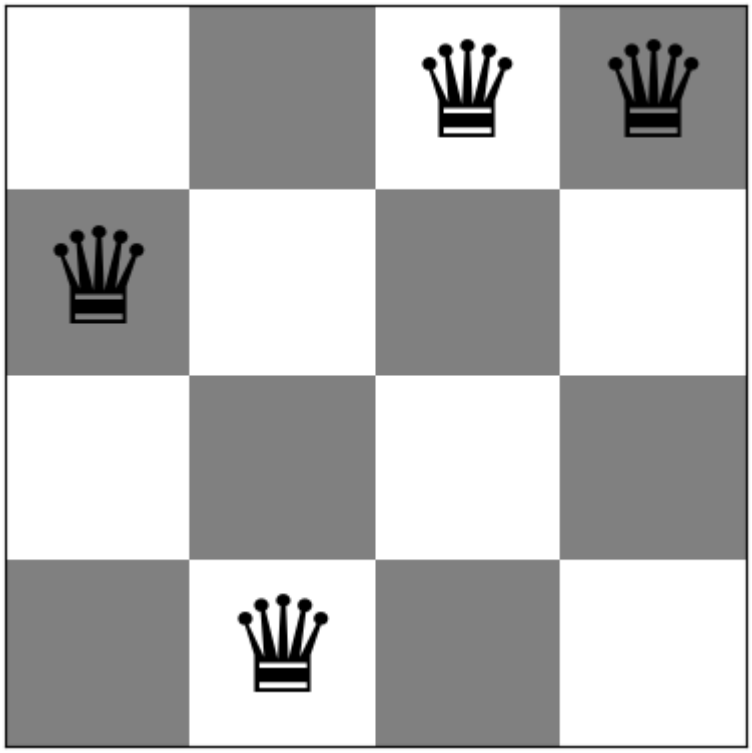
In [154…

```
board = random_board(4)
b = steepest_ascend_hill_climbing(board)
```

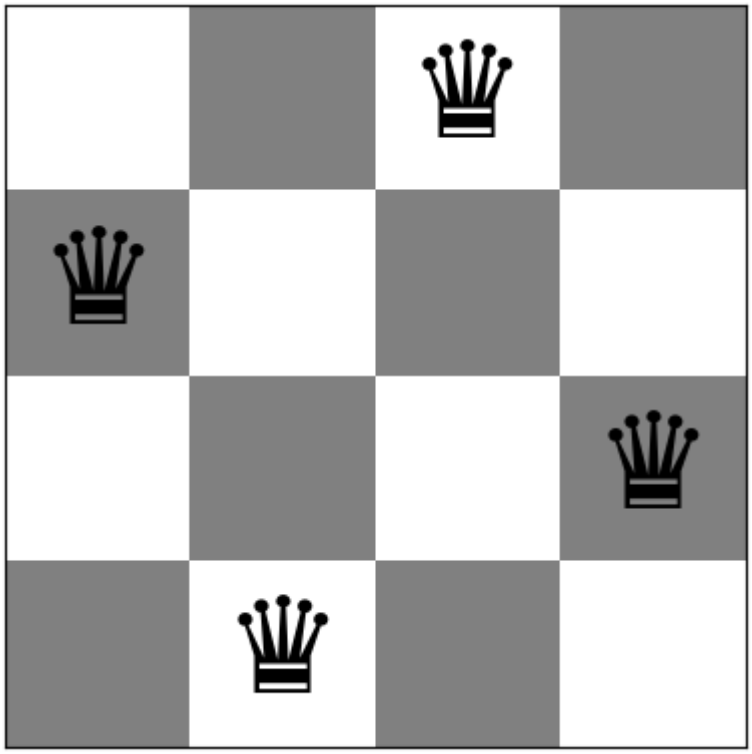
```
Board with 3 conflicts.
```



```
Board with 1 conflicts.
```



```
Board with 0 conflicts.
```



Task 2: Stochastic Hill Climbing 1 [10 Points]

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

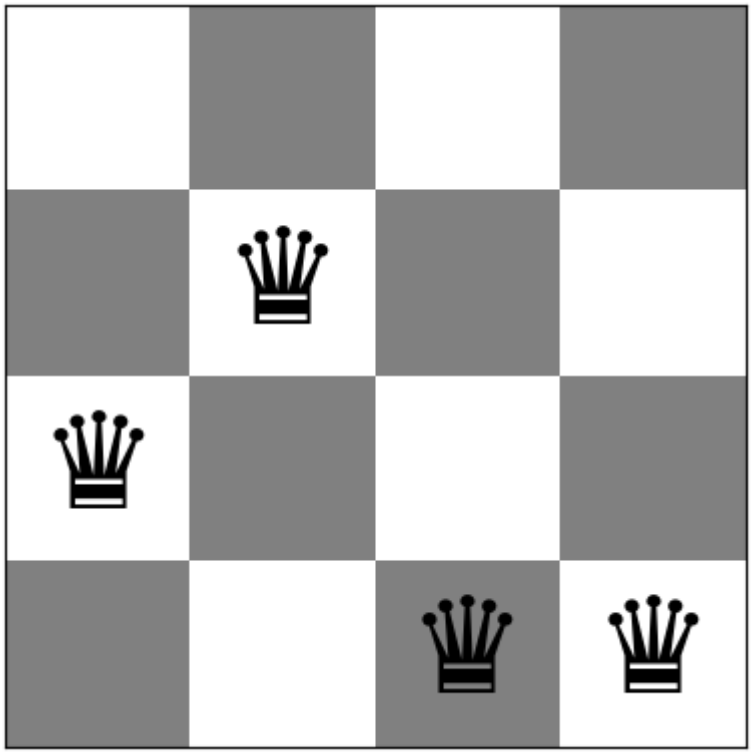
Description: Below code explores and selects uphill moves at random while attempting to minimize the number of conflicts. It continues this process until no more uphill moves are available or the current configuration is already at the global minimum (ideally, zero conflicts), representing a solution to the N-Queens problem.

In [129…

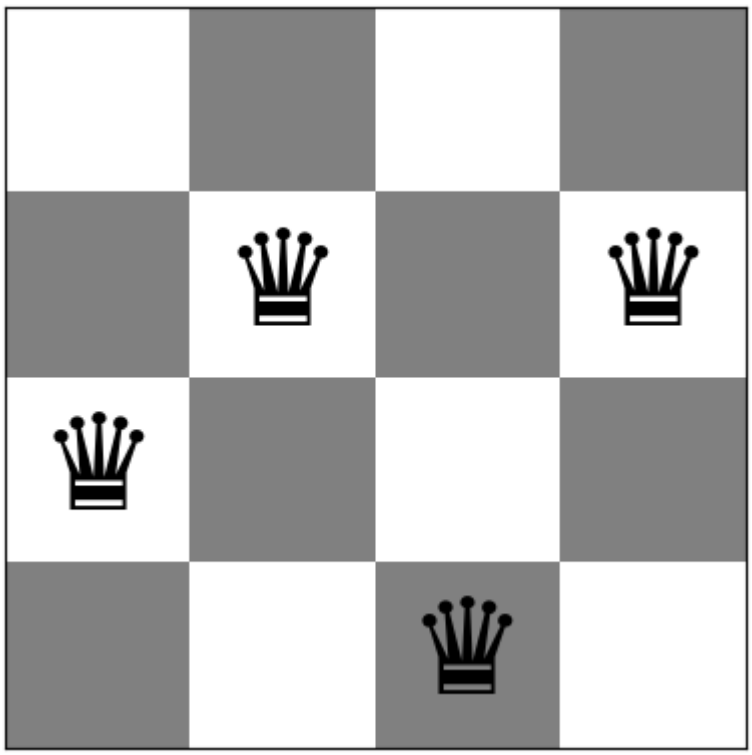
```
board = random_board(4)
def Stochastic_Hill_Climbing(board,verbose = True):
  if verbose: show_board(board)
  n = len(board)
  curr_con = conflicts(board)
  bval = [[-1 for _ in range(n)] for _ in range(n)]
  while True:
    uphill = []
    #Finding Uphill moves and choosing randomly from those moves
    for i in range(n):
      a = board[i]
      for j in range(n):
        board[i] = j
        bval[j][i] = conflicts(board)
      board[i] = a

    for i in range(n):
      for j in range(n):
        if bval[i][j] < curr_con:
          uphill.append(bval[i][j])
    if len(uphill) > 0:
      new_val = np.random.choice(uphill)
      if(curr_con > new_val):
        for i in bval:
          for j in i:
            if j == new_val:
              loc = (i,j)
```

```
        loc = np.where(bval == new_val)
        new_loc = [a for a in zip(loc[0], loc[1])]
        new_loc = new_loc[np.random.randint(0, len(new_loc))]
        board[new_loc[1]] = new_loc[0]
        curr_con = new_val
        if verbose: show_board(board)
      else: return(board)
    elif uphill == []: break
  return(board)
b = Stochastic_Hill_Climbing(board)
```

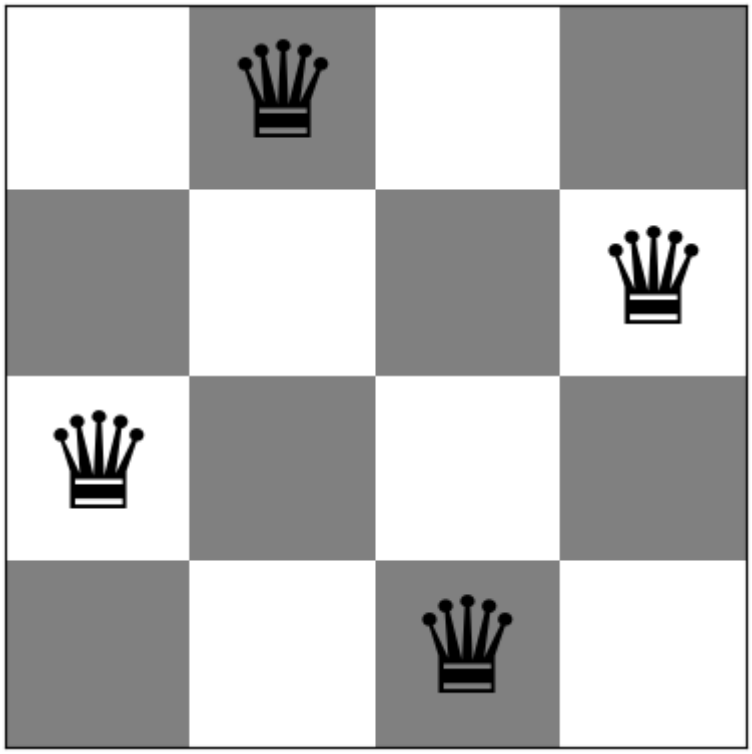
```
Board with 3 conflicts.
```



```
Board with 2 conflicts.
```



```
Board with 0 conflicts.
```



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.

Description: First-Choice Hill Climbing algorithm explores random moves and accepts the first move that leads to fewer conflicts. It continues this process for a specified number of iterations or until no improvements are observed for a consecutive number of steps. This algorithm focuses on finding a local minimum rather than a global one.

In [130…

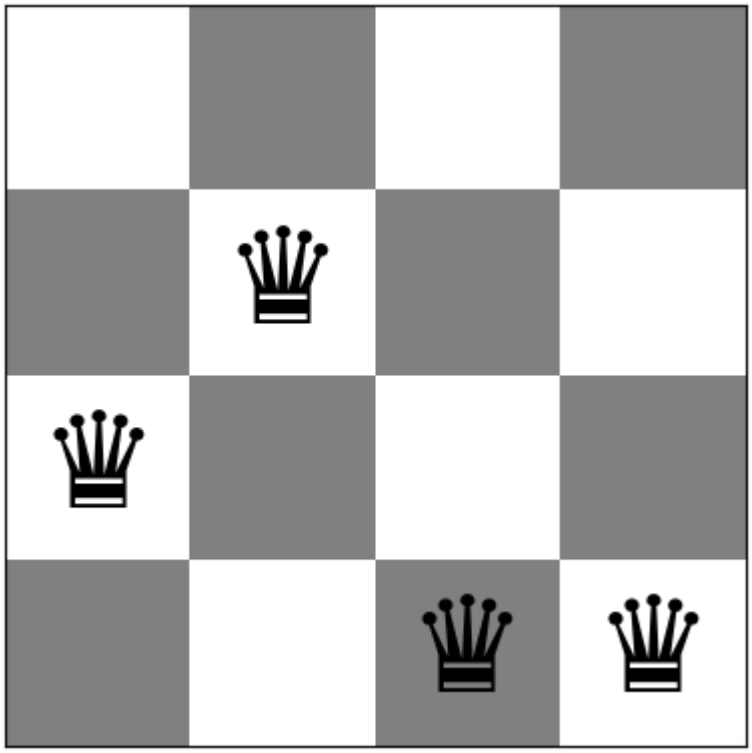
```
def first_choice_hill_climbing(board,verbose=True):

  # Initialize variables
  board_size = len(board)
  best_val = conflicts(board)
  vals = [[-1 for _ in range(board_size)] for _ in range(board_size)]
  num_steps = 0
  num_not_improving_steps = 0
  limit = np.ceil(100 * board_size)

  while num_steps < 20000:
    # Evaluate the conflicts for each possible move
    for col in range(board_size):
      original_row = board[col]
      for row in range(board_size):
        board[col] = row
        vals[row][col] = conflicts(board)
```

```
      board[col] = original_row

    # Make a random move and evaluate the conflicts
    random_col = np.random.randint(0, board_size)
    random_row = np.random.randint(0, board_size)
    new_val = vals[random_row][random_col]

    num_steps += 1

    # Make the move if it improves the conflicts
    if best_val > new_val:
      num_not_improving_steps = 0
      best_positions = [(row, col) for row in range(board_size) for col in range(boa
      best_row, best_col = best_positions[np.random.randint(0, len(best_positions))]
      board[best_col] = best_row
      best_val = new_val
      if verbose: show_board(board)
      if best_val == 0:
        break
    else:
      num_not_improving_steps += 1
      if num_not_improving_steps > limit:
        break

  return board
```

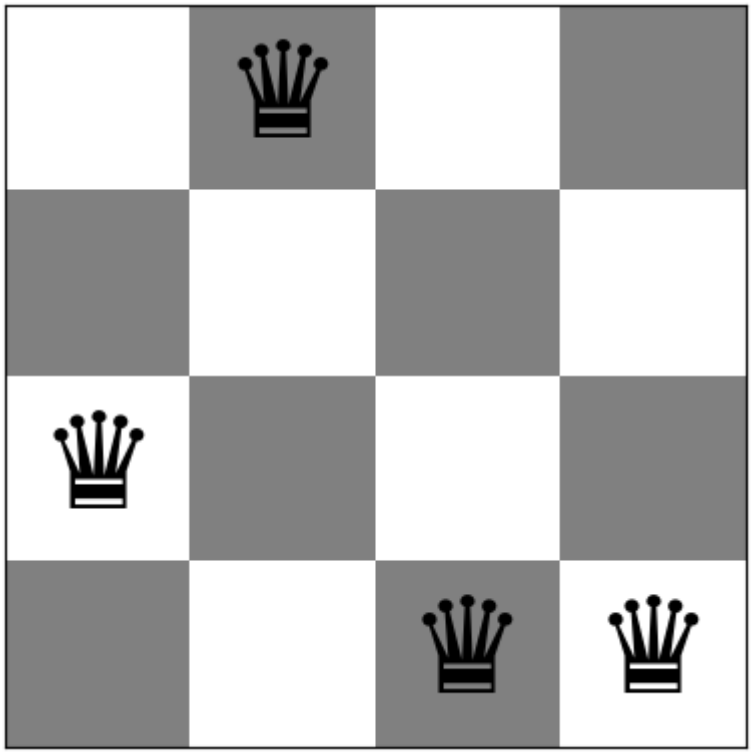
In [131…

```
board = random_board(4)
b = first_choice_hill_climbing(board)
```

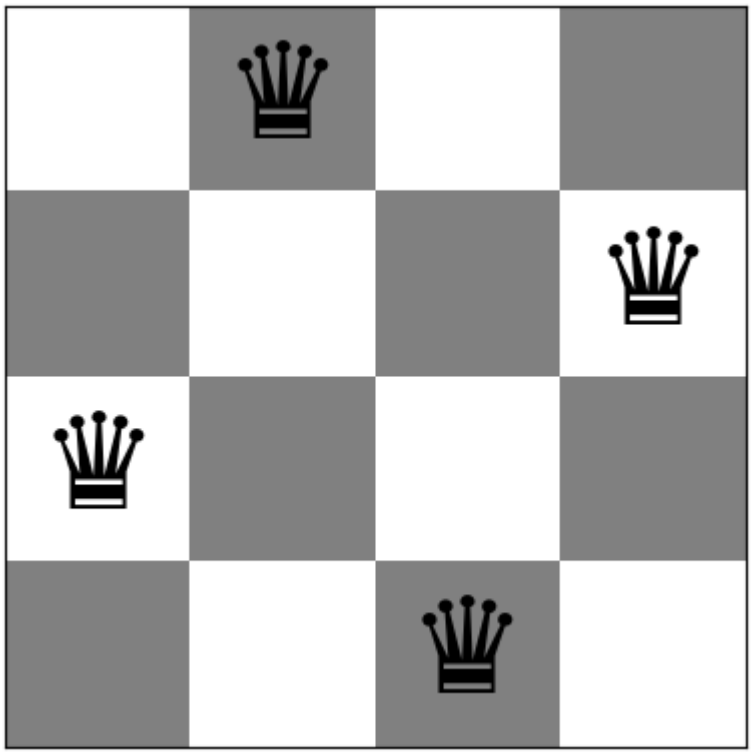
```
Board with 3 conflicts.
```



```
Board with 1 conflicts.
```



```
Board with 0 conflicts.
```



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

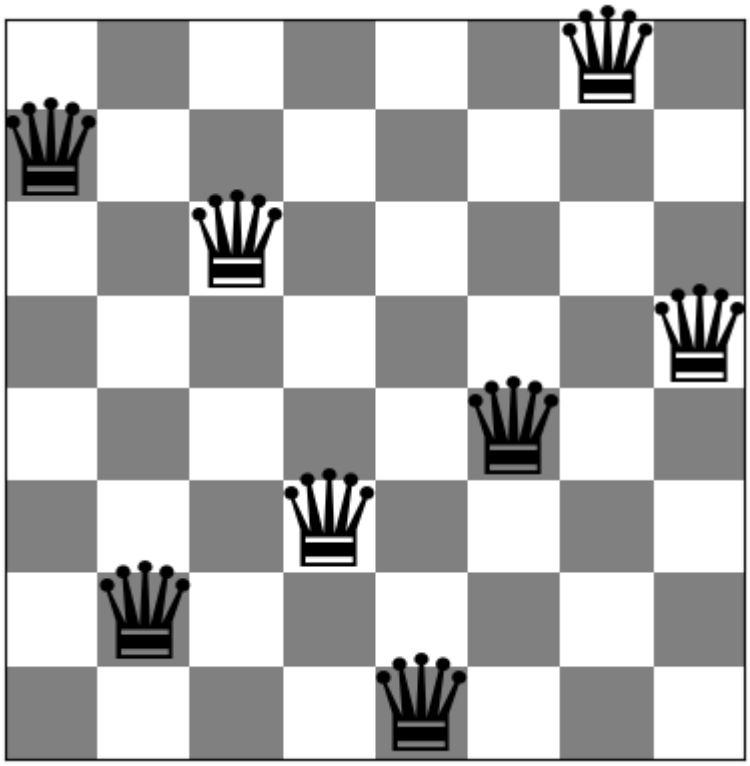
Hill climbing will often end up in local optima. Restart the each of the three hill climbing algorithm up to 100 times with a random board to find a better (hopefully optimal) solution. Note that restart just means to run the algorithm several times starting with a new random board.

In [161…

```
def Steepest_Ascend_Hill_Cimbing_random_restarts(b, func, verbose =True):
  n = len(b)
```

```
  min_val = conflicts(b)
  for i in range(100):
    board = random_board(n)
    board = func(board, verbose = False)
    value = conflicts(board)
    if value < min_val:
      b = board
      min_val = value
    if value == 0:
      if verbose: print (f"Restarts: {i+1}")
      return(board)
  return b
board = random_board(8)
board = Steepest_Ascend_Hill_Cimbing_random_restarts(board, steepest_ascend_hill_cli
show_board(board)
```

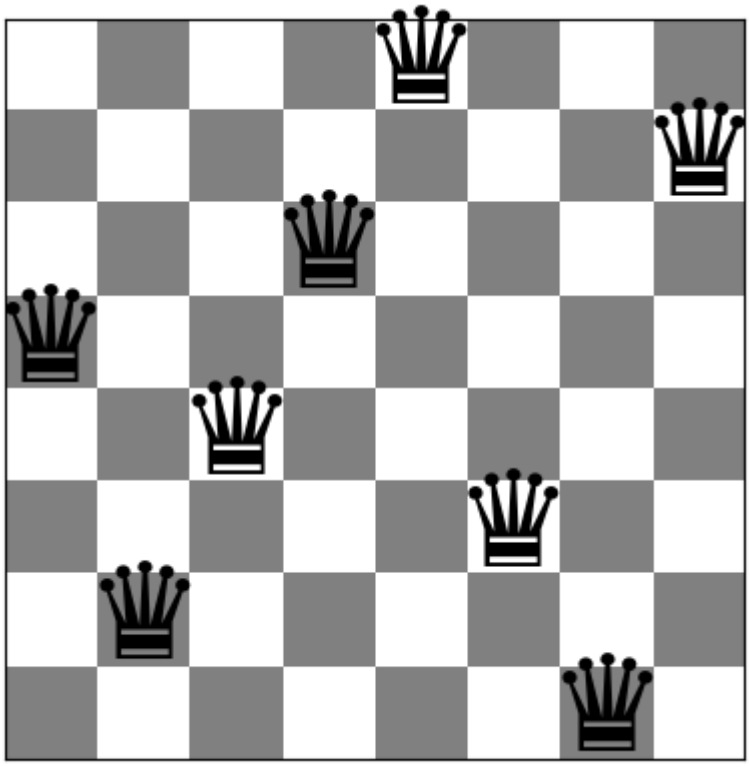
```
Restarts: 6
Board with 0 conflicts.
```



In [163…

```
def Stochastic_Hill_Cimbing_random_restarts(b, func, verbose =True):
  n = len(b)
  min_val = conflicts(b)
  for i in range(100):
    board = random_board(n)
    board = func(board, verbose = False)
    value = conflicts(board)
    if value < min_val:
      b = board
      min_val = value
    if value == 0:
      if verbose: print (f"Restarts: {i+1}")
      return(board)
  return b
board = random_board(8)
board = Stochastic_Hill_Cimbing_random_restarts(board, Stochastic_Hill_Climbing)
show_board(board)
```

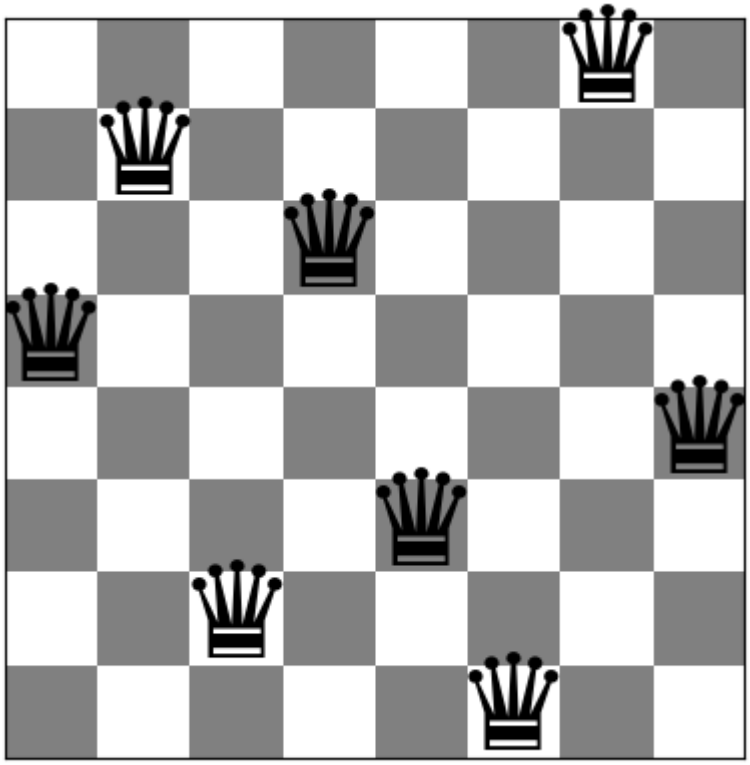
```
Restarts: 7
Board with 0 conflicts.
```



In [164…

```
def First_Choice_Hill_Climbing_random_restarts(b, func, verbose =True):
  n = len(b)
  min_val = conflicts(b)
  for i in range(100):
    board = random_board(n)
    board = func(board, verbose = False)
    value = conflicts(board)
    if value < min_val:
      b = board
      min_val = value
    if value == 0:
      if verbose: print (f"Restarts: {i+1}")
      return(board)
  return b
board = random_board(8)
board = First_Choice_Hill_Climbing_random_restarts(board, first_choice_hill_climbing
show_board(board)
```

```
Restarts: 3
Board with 0 conflicts.
```



Task 5: Simulated Annealing [10 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 [169…

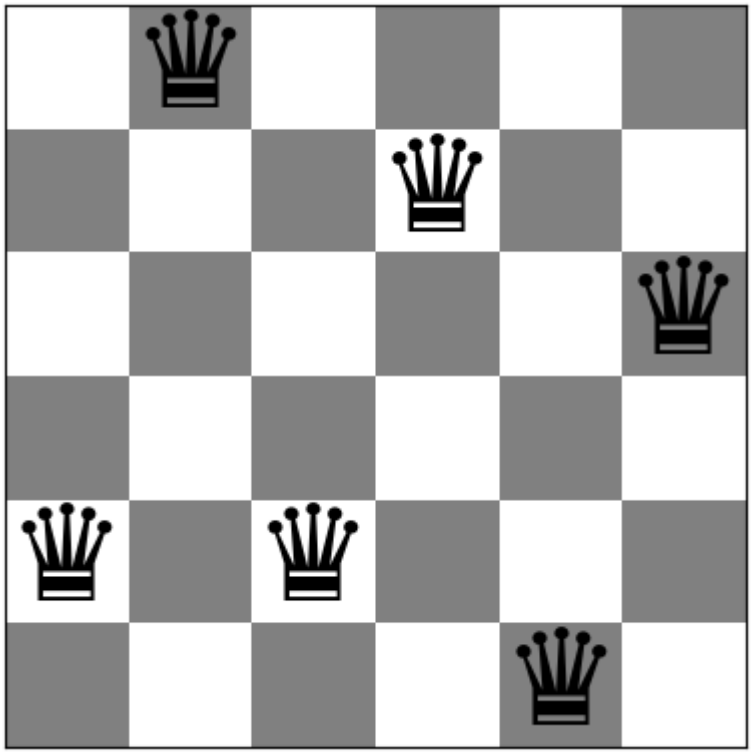
```
import random

def simulated_annealing(board, verbose=False, T0=None, alpha=0.999, epsilon=1e-1):
  board_size = len(board)
  best_val = conflicts(board)
  vals = [[-1 for _ in range(board_size)] for _ in range(board_size)]
  prob = []
  num_steps = 0

  # Calculate T0
  deltaE = (board_size ** 2) / 2
  probability_T0 = 0.9
  T0 = -(board_size ** 2) / (2 * np.log(probability_T0))
  T = T0

  while T > epsilon:
    T = T0 * alpha ** num_steps

    for col in range(board_size):
      original_row = board[col]
      for row in range(board_size):
        board[col] = row
        vals[row][col] = conflicts(board)
```

```
      board[col] = original_row

    random_col = random.randint(0, board_size - 1)
    random_row = random.randint(0, board_size - 1)
    new_val = vals[random_row][random_col]
    deltaE = new_val - best_val

    if deltaE <= 0:
      w = np.where(vals == new_val)
      best = [a for a in zip(w[0], w[1])]
      board[best_col] = best_row
      best_val = new_val
      if best_val == 0:
        break
    else:
      probability = np.exp(-deltaE / T)
      prob.append(probability)
      if random.random() < probability:
        w = np.where(vals == new_val)
        best_row, best_col = w[0][0], w[1][0]
        board[best_col] = best_row

    num_steps += 1

    if verbose:
      show_board(board)

  # Calculate the average probability of accepting bad moves.
  avg_prob = np.mean(prob)

  # Add the avg_prob to the return value.
  return avg_prob
```

In [188…

```
board = random_board(6)
b = simulated_annealing(board, verbose = True) #plt.plot(c)
#plt.xlabel("Iteration")
#plt.ylabel("Probability of making bad decision") #plt.show()
show_board(b)
conflicts(b)
```

```
Board with 2 conflicts.
```



Out[188… 2

Task 6: Compare Performance [10 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?

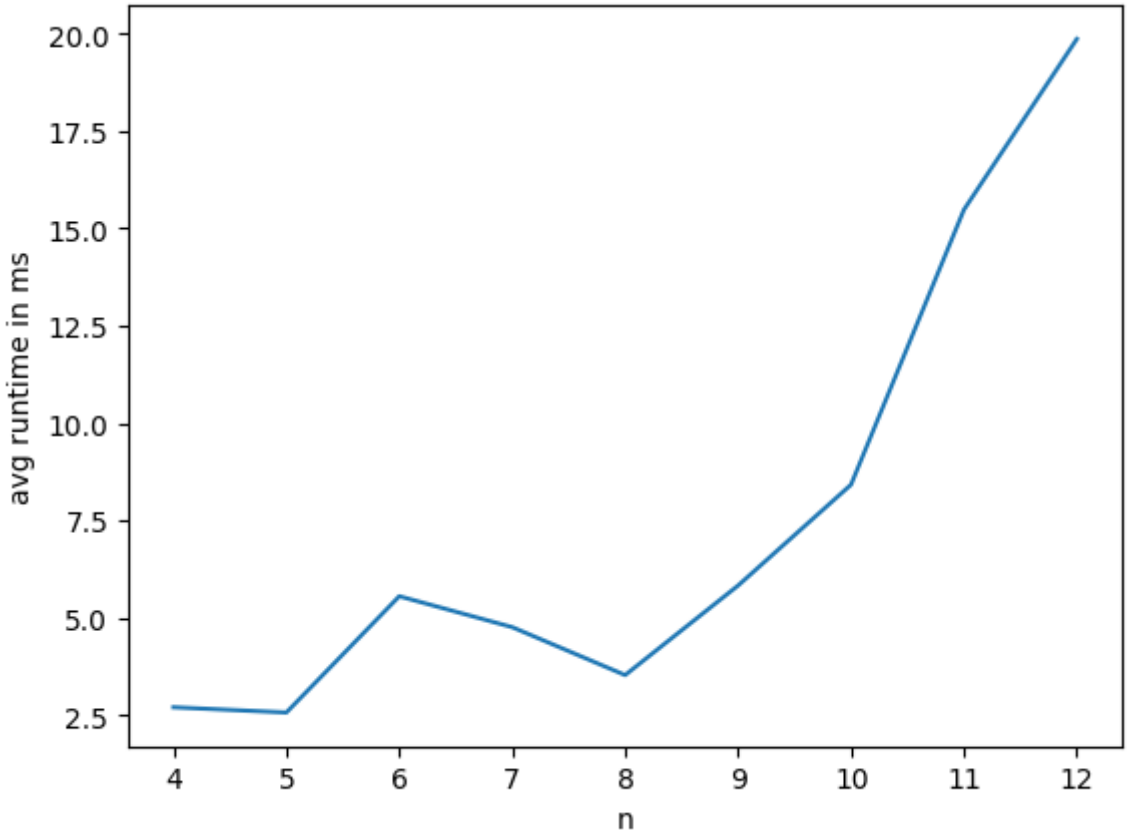
See Profiling Python Code for help about how to measure runtime in Python.

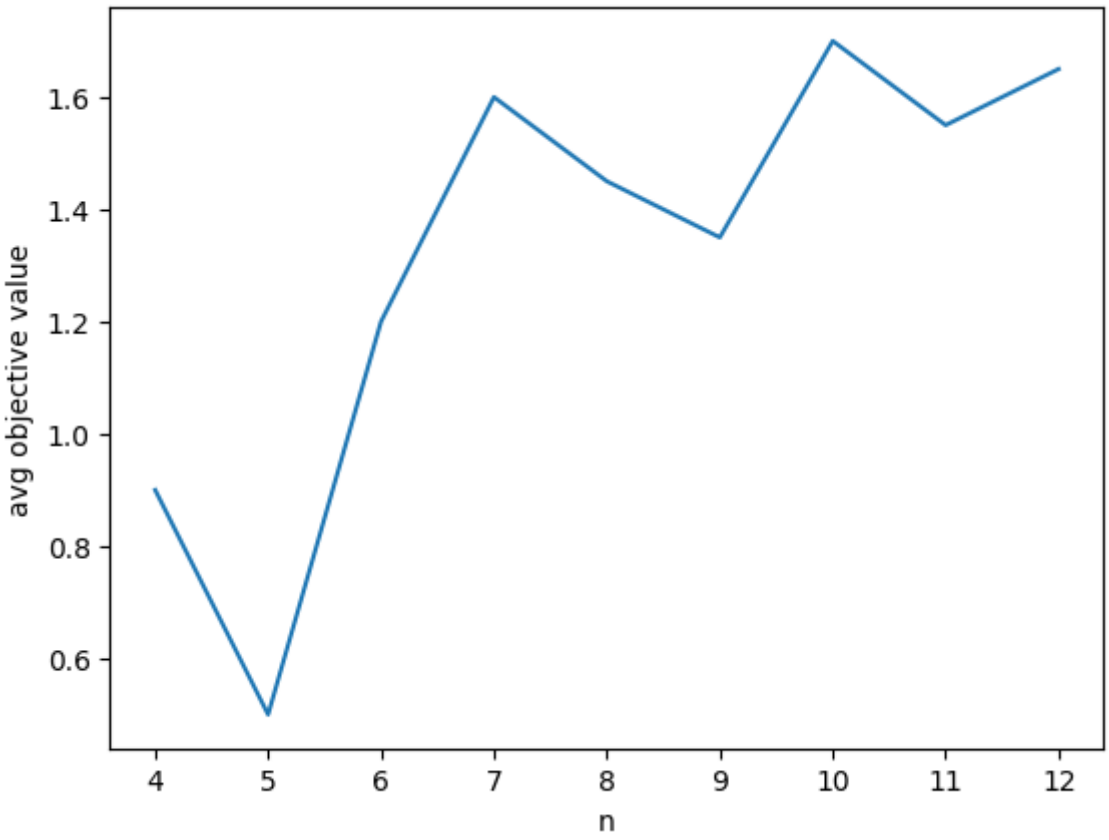
In [59]:

```
import time

board_sizes = [4, 5, 6, 7, 8, 9, 10, 11, 12]
mean_objective_values = []
mean_times = []
for board_size in board_sizes:
    objective_values = []
    times = []
    for i in range(20):
        board = random_board(board_size)
        start_time = time.time()
        solution = steepest_ascend_hill_climbing(board, False)
        end_time = time.time()
        objective_value = conflicts(solution)
        time_taken = end_time - start_time
        objective_values.append(objective_value)
        times.append(time_taken)
    mean_objective_value = np.average(objective_values)
    mean_time = np.average(times)
```

```
    mean_objective_values.append(mean_objective_value)
    mean_times.append(mean_time)
    print(f"n = {board_size}: \tmean(time) = {round(mean_time*1e3, 2)} ms\tmean(obje
plt.plot(board_sizes, [mean_time*1e3 for mean_time in mean_times])
plt.xlabel("n")
plt.ylabel("avg runtime in ms")
plt.show()
plt.plot(board_sizes, mean_objective_values)
plt.xlabel("n")
plt.ylabel("avg objective value")
plt.show()
```

```
n = 4:  mean(time) = 2.7 ms     mean(objective) = 0.9
n = 5:  mean(time) = 2.56 ms    mean(objective) = 0.5
n = 6:  mean(time) = 5.55 ms    mean(objective) = 1.2
n = 7:  mean(time) = 4.76 ms    mean(objective) = 1.6
n = 8:  mean(time) = 3.52 ms    mean(objective) = 1.45
n = 9:  mean(time) = 5.83 ms    mean(objective) = 1.35
n = 10:         mean(time) = 8.42 ms    mean(objective) = 1.7
n = 11:         mean(time) = 15.48 ms   mean(objective) = 1.55
n = 12:         mean(time) = 19.87 ms   mean(objective) = 1.65
```



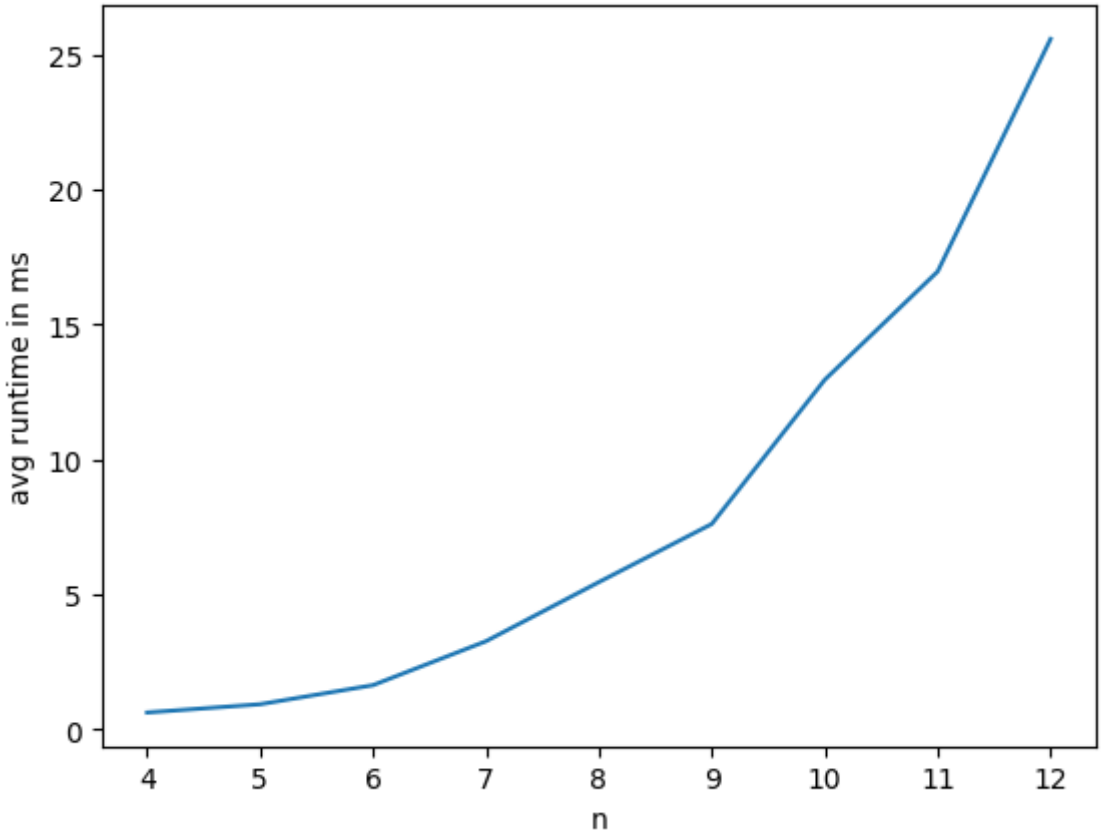


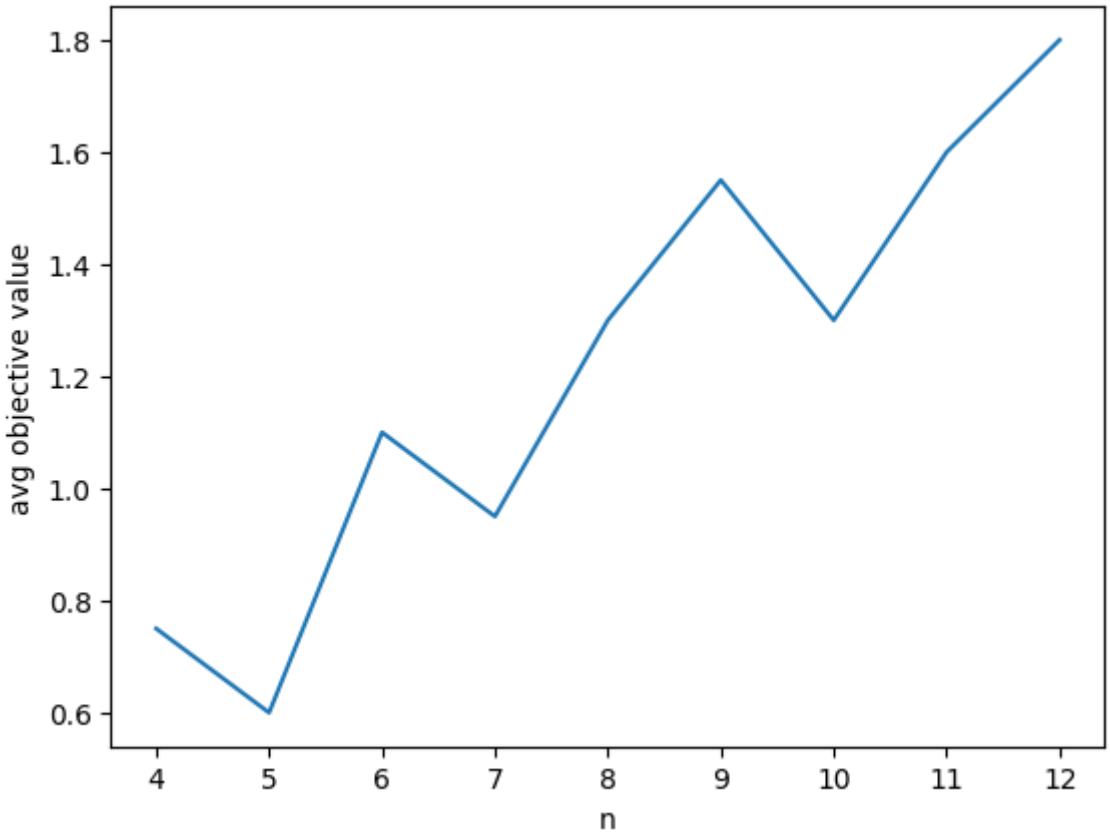
In [65]:

```
import time

board_sizes = [4, 5, 6, 7, 8, 9, 10, 11, 12]
mean_objective_values = []
mean_times = []
for board_size in board_sizes:
    objective_values = []
    times = []
    for i in range(20):
        board = random_board(board_size)
        start_time = time.time()
        solution = Stochastic_Hill_Climbing(board, False)
        end_time = time.time()
        objective_value = conflicts(solution)
        time_taken = end_time - start_time
        objective_values.append(objective_value)
        times.append(time_taken)
    mean_objective_value = np.average(objective_values)
    mean_time = np.average(times)
    mean_objective_values.append(mean_objective_value)
    mean_times.append(mean_time)
    print(f"n = {board_size}: \tmean(time) = {round(mean_time*1e3, 2)} ms\tmean(obje
plt.plot(board_sizes, [mean_time*1e3 for mean_time in mean_times])
plt.xlabel("n")
plt.ylabel("avg runtime in ms")
plt.show()
plt.plot(board_sizes, mean_objective_values)
plt.xlabel("n")
plt.ylabel("avg objective value")
plt.show()
```

```
n = 4:  mean(time) = 0.63 ms    mean(objective) = 0.75
n = 5:  mean(time) = 0.93 ms    mean(objective) = 0.6
n = 6:  mean(time) = 1.64 ms    mean(objective) = 1.1
n = 7:  mean(time) = 3.27 ms    mean(objective) = 0.95
n = 8:  mean(time) = 5.46 ms    mean(objective) = 1.3
```

```
n = 9:  mean(time) = 7.61 ms    mean(objective) = 1.55
n = 10:         mean(time) = 12.95 ms   mean(objective) = 1.3
n = 11:         mean(time) = 16.96 ms   mean(objective) = 1.6
n = 12:         mean(time) = 25.58 ms   mean(objective) = 1.8
```



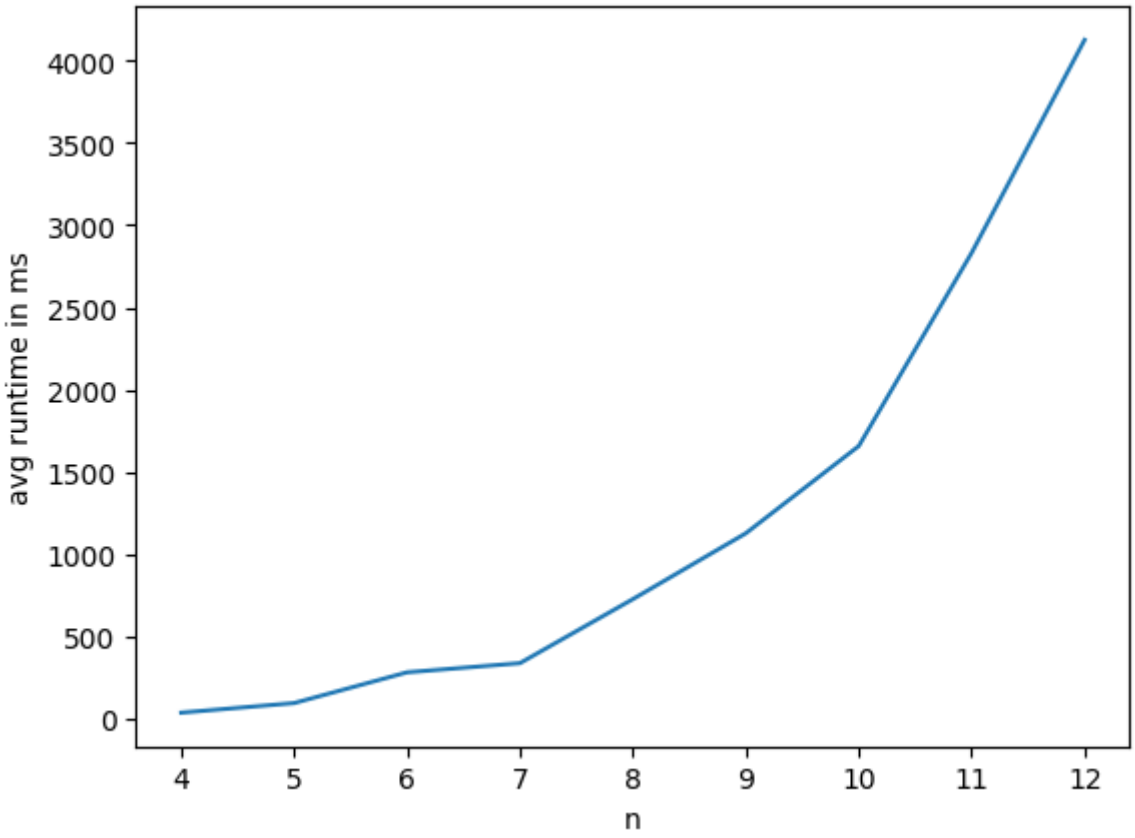


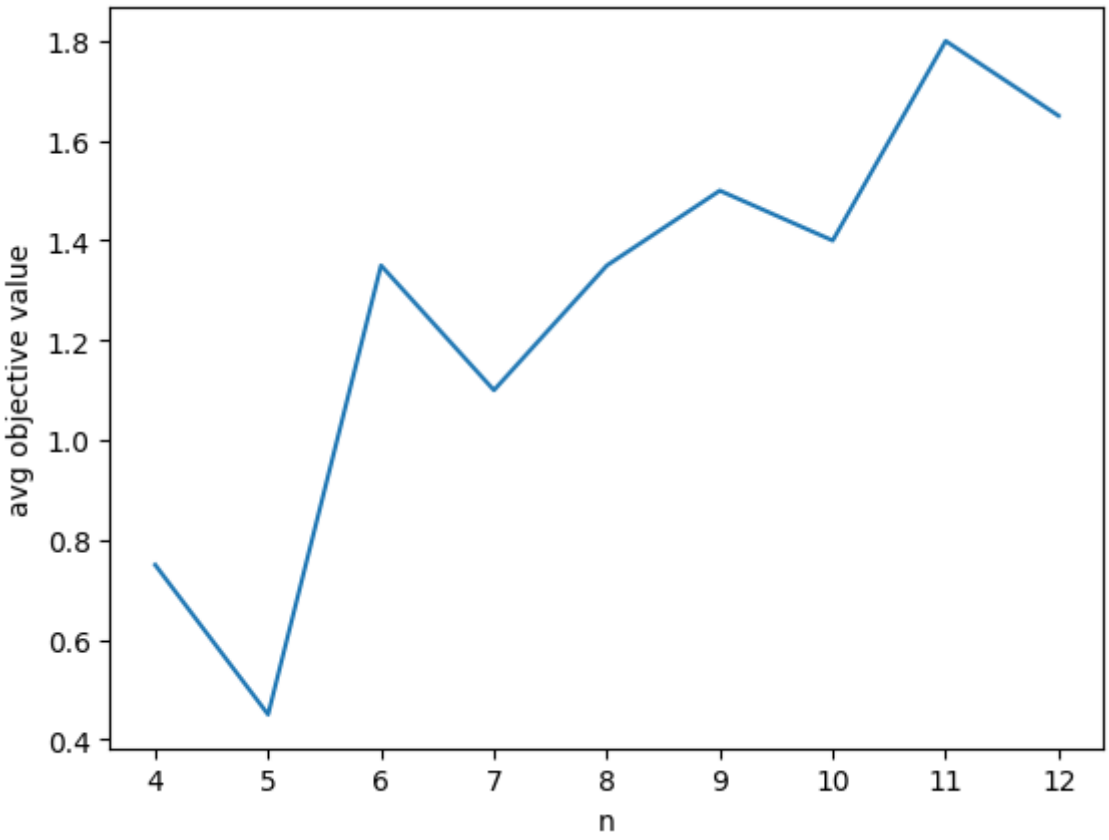
In [73]:

```python
import time

board_sizes = [4, 5, 6, 7, 8, 9, 10, 11, 12]
mean_objective_values = []
mean_times = []
```

```python
for board_size in board_sizes:
    objective_values = []
    times = []
    for i in range(20):
        board = random_board(board_size)
        start_time = time.time()
        solution = first_choice_hill_climbing(board, False)
        end_time = time.time()
        objective_value = conflicts(solution)
        time_taken = end_time - start_time
        objective_values.append(objective_value)
        times.append(time_taken)
    mean_objective_value = np.average(objective_values)
    mean_time = np.average(times)
    mean_objective_values.append(mean_objective_value)
    mean_times.append(mean_time)
    print(f"n = {board_size}: \tmean(time) = {round(mean_time*1e3, 2)} ms\tmean(obje
plt.plot(board_sizes, [mean_time*1e3 for mean_time in mean_times])
plt.xlabel("n")
plt.ylabel("avg runtime in ms")
plt.show()
plt.plot(board_sizes, mean_objective_values)
plt.xlabel("n")
plt.ylabel("avg objective value")
plt.show()
```

```
n = 4:  mean(time) = 36.7 ms    mean(objective) = 0.75
n = 5:  mean(time) = 95.02 ms   mean(objective) = 0.45
n = 6:  mean(time) = 281.91 ms  mean(objective) = 1.35
n = 7:  mean(time) = 337.66 ms  mean(objective) = 1.1
n = 8:  mean(time) = 726.94 ms  mean(objective) = 1.35
n = 9:  mean(time) = 1127.09 ms mean(objective) = 1.5
n = 10:         mean(time) = 1657.35 ms mean(objective) = 1.4
n = 11:         mean(time) = 2832.17 ms mean(objective) = 1.8
n = 12:         mean(time) = 4126.14 ms mean(objective) = 1.65
```



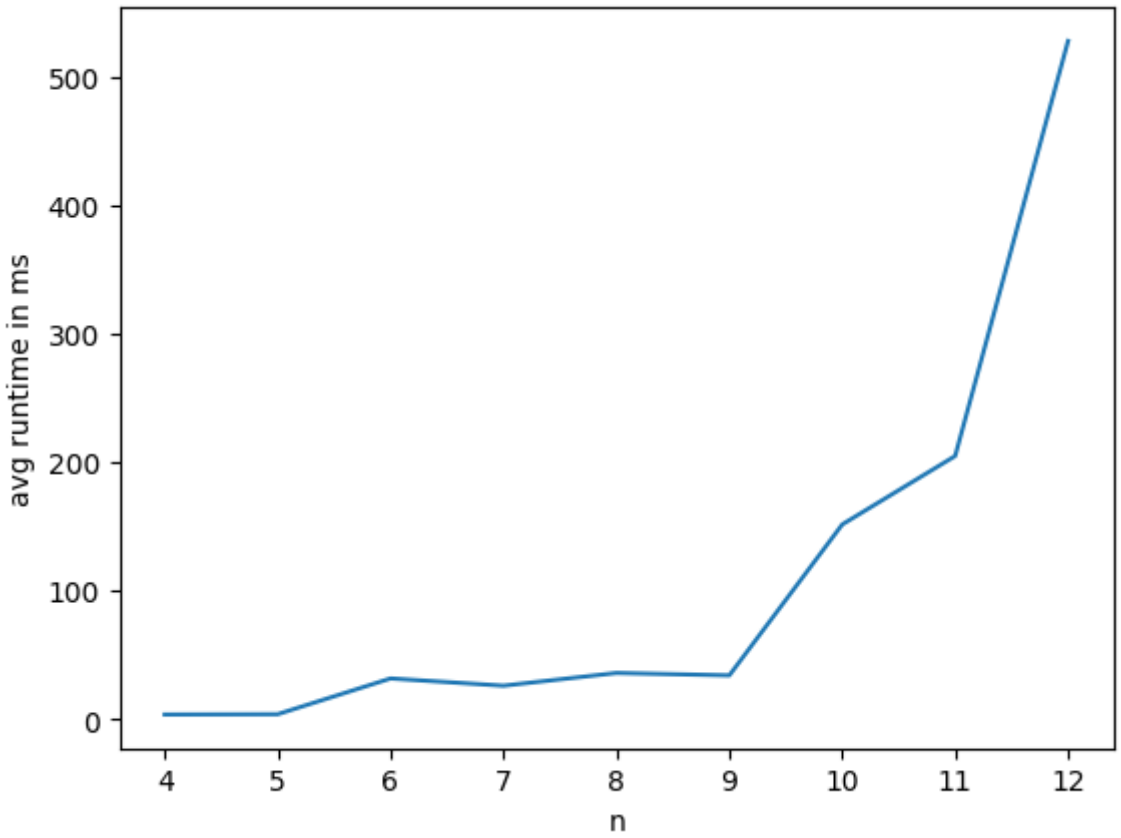


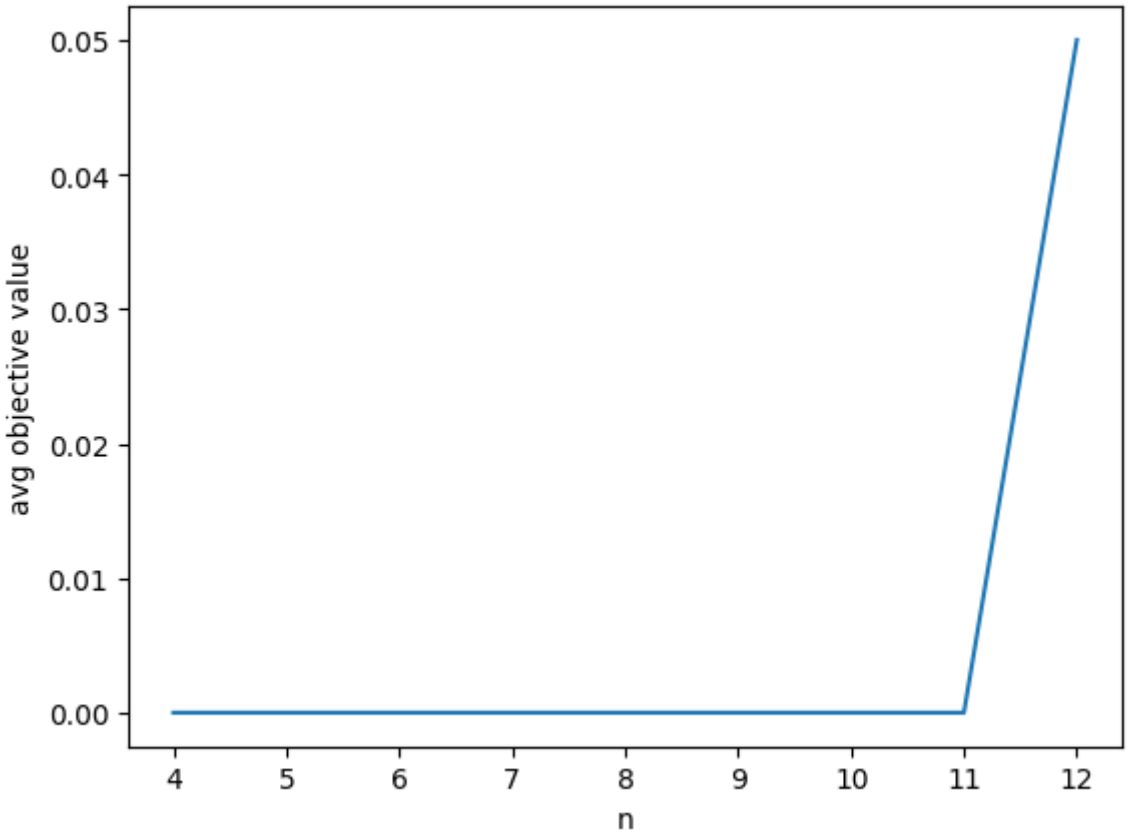
In [166…

```
import time

board_sizes = [4, 5, 6, 7, 8, 9, 10, 11, 12]
mean_objective_values = []
mean_times = []
for board_size in board_sizes:
    objective_values = []
    times = []
    for i in range(20):
        board = random_board(board_size)
        start_time = time.time()
        solution = Steepest_Ascend_Hill_Cimbing_random_restarts(board, steepest_asce
        end_time = time.time()
        objective_value = conflicts(solution)
        time_taken = end_time - start_time
        objective_values.append(objective_value)
        times.append(time_taken)
    mean_objective_value = np.average(objective_values)
    mean_time = np.average(times)
    mean_objective_values.append(mean_objective_value)
    mean_times.append(mean_time)
    print(f"n = {board_size}: \tmean(time) = {round(mean_time*1e3, 2)} ms\tmean(obje
plt.plot(board_sizes, [mean_time*1e3 for mean_time in mean_times])
plt.xlabel("n")
plt.ylabel("avg runtime in ms")
plt.show()
plt.plot(board_sizes, mean_objective_values)
plt.xlabel("n")
plt.ylabel("avg objective value")
plt.show()
```

```
n = 4:  mean(time) = 3.51 ms    mean(objective) = 0.0
n = 5:  mean(time) = 3.72 ms    mean(objective) = 0.0
n = 6:  mean(time) = 31.56 ms   mean(objective) = 0.0
n = 7:  mean(time) = 25.92 ms   mean(objective) = 0.0
n = 8:  mean(time) = 35.83 ms   mean(objective) = 0.0
```

```
n = 9:  mean(time) = 33.96 ms   mean(objective) = 0.0
n = 10:         mean(time) = 151.43 ms  mean(objective) = 0.0
n = 11:         mean(time) = 204.71 ms  mean(objective) = 0.0
n = 12:         mean(time) = 527.71 ms  mean(objective) = 0.05
```



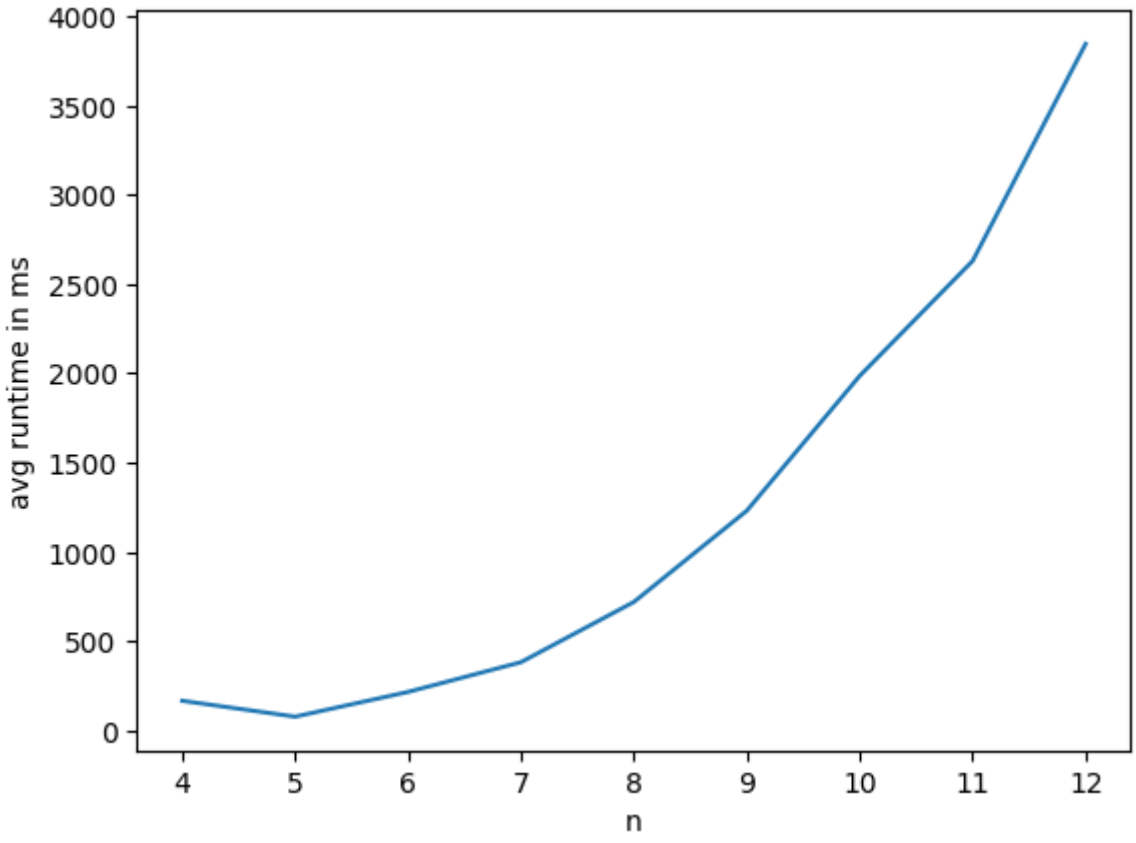


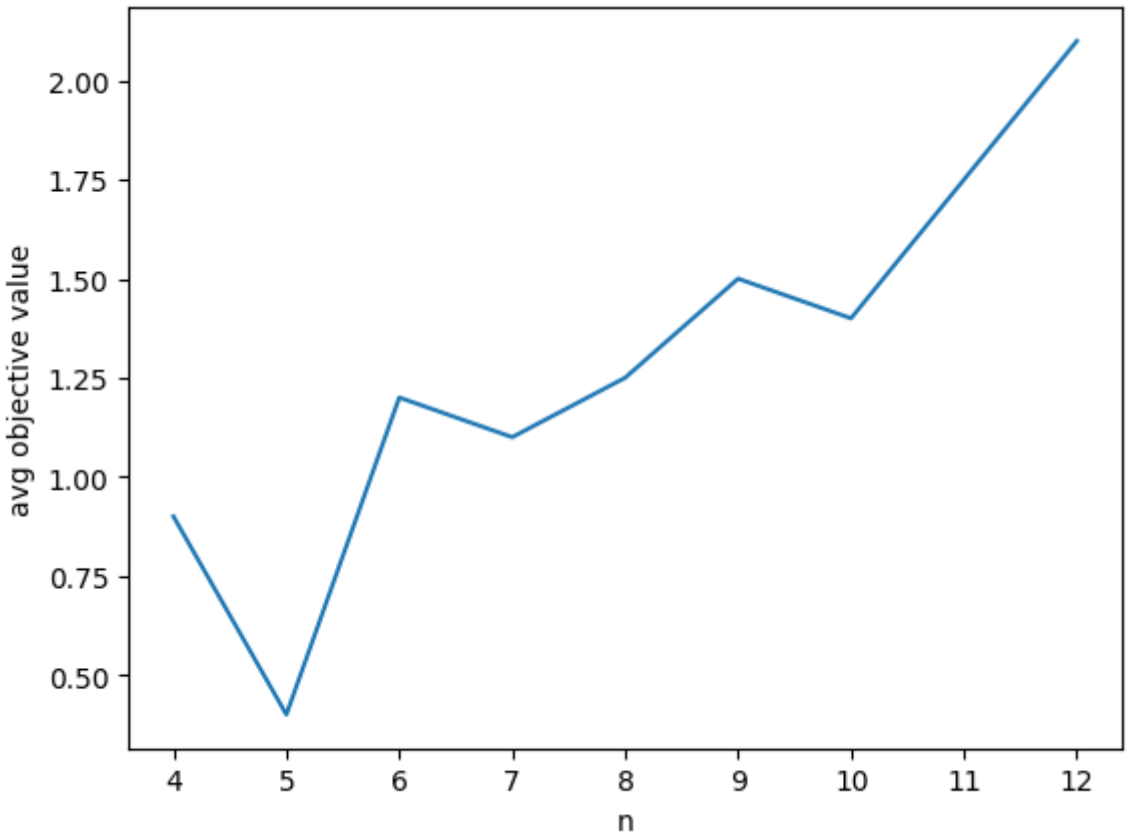
In [190…

```python
import time

board_sizes = [4, 5, 6, 7, 8, 9, 10, 11, 12]
mean_objective_values = []
mean_times = []
```

```python
for board_size in board_sizes:
    objective_values = []
    times = []
    for i in range(20):
        board = random_board(board_size)
        start_time = time.time()
        solution = Stochastic_Hill_Cimbing_random_restarts(board, first_choice_hill_
        end_time = time.time()
        objective_value = conflicts(solution)
        time_taken = end_time - start_time
        objective_values.append(objective_value)
        times.append(time_taken)
    mean_objective_value = np.average(objective_values)
    mean_time = np.average(times)
    mean_objective_values.append(mean_objective_value)
    mean_times.append(mean_time)
    print(f"n = {board_size}: \tmean(time) = {round(mean_time*1e3, 2)} ms\tmean(obje
plt.plot(board_sizes, [mean_time*1e3 for mean_time in mean_times])
plt.xlabel("n")
plt.ylabel("avg runtime in ms")
plt.show()
plt.plot(board_sizes, mean_objective_values)
plt.xlabel("n")
plt.ylabel("avg objective value")
plt.show()
```

```
n = 4:  mean(time) = 166.15 ms  mean(objective) = 0.9
n = 5:  mean(time) = 77.21 ms   mean(objective) = 0.4
n = 6:  mean(time) = 215.69 ms  mean(objective) = 1.2
n = 7:  mean(time) = 382.21 ms  mean(objective) = 1.1
n = 8:  mean(time) = 719.48 ms  mean(objective) = 1.25
n = 9:  mean(time) = 1230.53 ms mean(objective) = 1.5
n = 10:         mean(time) = 1985.8 ms  mean(objective) = 1.4
n = 11:         mean(time) = 2629.31 ms mean(objective) = 1.75
n = 12:         mean(time) = 3843.94 ms mean(objective) = 2.1
```





Graduate student advanced task: Exploring other Local Moves [10 Points]

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

Implement a few different local moves. Implement:

- moving a queen only one square at a time
- switching two columns
- more moves which move more than one queen at a time.

Compare the performance of these moves for the 8-Queens problem using your stochastic hill climbing 2 implementation from above. Also consider mixing the use of several types of local moves (e.g., move one queen and moving two queens).

Describe what you find out about how well these moves and combinations of these moves work.

The psuedo code for the above question is:

function one_square_move(board):

```
best_board = board
for each queen in board:
    original_row,original_column = queen's current row,queen's current
column
    for each possible new_row:
        move queen to new_row
        if new_board has fewer conflicts than best_board:
            best_board = copy of new_board
        move queen back to original_row
```

```
    return best_board
```

function switch_columns(board):

```
    best_board = board
    for each pair of columns:
        create a new board by swapping queens in the two columns
        if new_board has fewer conflicts than best_board:
            best_board = copy of new_board
    return best_board
```

function multi_queen_move(board, num_queens):

```
    best_board = board
    for num_queens times:
        pick num_queens random queens
        for each queen in the selected queens:
            original_row = queen's current row
            for each possible new_row:
                move queen to new_row
                if new_board has fewer conflicts than best_board:
                    best_board = copy of new_board
                move queen back to original_row
    return best_board
```

function first_choice_hill_climbing(board):

```
    while not reached the maximum number of iterations:
        if board has no conflicts:
            return board
        choose a random move (one_square_move, switch_columns, or
    multi_queen_move)
        new_board = apply the chosen move to the board
        if new_board has fewer conflicts than the current board:
            board = new_board
    return the best board found
```

More things to do

Implement a Genetic Algorithm for the n-Queens problem.

In [ ]:

```
# Code and description go here
```