CSE 4705: Assignment 03 - Local Search - 8-queens - Part 1

In this lab you will:

- Implement various local search algorithms that we've discussed in class, including:

1. Steepest ascent hill climbing
2. Stochastic hill climbing
3. First choice hill climbing
4. Random restart hill climbing.

Outline

0.1 Problem Statement

In this assignment you will implement a number of local search algorithms for solving the n-queens problem. As we discussed in class, this problem consists of determining how to arrange queen pieces on an n x n chess board so that no two queens are attacking each other. The diagram below shows one of a number of successful arrangmeents of queens on an 8 x 8 chess board:

No description has been provided for this image

Your algorithms will start with a random arrangement of queens on the board, and then will utilize the algorithms to approach, or hopefully, successfully find, a goal state in which no two queens are attacking each other.

The algorithms you will implement include the following:

1. [25 points] Hill Climbing (Steepest Ascent)
2. [30 points] Stochastic Hill Climbing
3. [30 points] First Choice Hill Climbing
4. [15 points] Random Restart Hill Climbing

0.2 Objective Function - Number of Attacking Pairs

As discussed in class, a commonly used metric for the objective function for the 8-queens problem is the number of pairs of attacking queens for a given 8-queens assignment (i.e., 8-queens arrangement on the board). The goal state is a queens assignment for which attack pairs = 0.

In the slide deck from Lecture 04 - Local Search, there is a calculation for the attack pairs metric for the given queens assignment for each square on the board, as shown below. We'll discuss this example in more detail a bit later.

No description has been provided for this image

You will use the attack pairs metric for guiding the search in your algorithm implementations.

0.3 Queens Assignment Representation

You will use a numpy array to represent the assignment of a set of locations for queens on the chess board. It will take on a form consistent with the following example:

$$\text{queens} = \text{ndarray}([3, 2, 1, 4, 3, 2, 1, 2])$$

This numpy array indicates the row position for each queen located in each of the 8 columns on the board. Note that the row indices are 0-based. So, the bottom row is indicated with a 0, the second row from the bottom by a 1, and so on up through top row indicated by a 7.

The example above gives the representation for the arrangement of queens in the image above. That is, queens[0] = 3 indicates the queen in the first column is in the fourth row. Meanwhile, queens[1] = 2 indicates the queen in the second column is in the third row. This same reasoning follows for the rest of the 8 queens in the array.

In [ ]:
```
import pandas as pd
import numpy as np
```

```python
from random import choices
from queue import PriorityQueue
```

1 - Steepest Ascent Hill Climbing

[25 points]

You will implement the steepest ascent hill climbing algorithm in the cells below.

1.1 Exercise - Function Implementation - Count attacking pairs for a given queens assignment

[10 points]

Below, the attack_paris() function is intended to return the number of pairs of queens attacking each other for a given queens assignment, passed in as an input argument, in the form of an numpy array (as described in section 0.3.

Implement this function according to the specs given for the function.

There are some simple (but not comprehensive) unit tests after the function to indicate whether you are on the right track.

In [ ]:
```python
def attack_pairs(queens):
    """
    computes number of pairs of attacking queens
    Args:
      queens (ndarray (n, ))  : represents the assignment of queens on the board.  n = 8 for the 8-queens problem.
    Returns
      attack_pairs scaler     : number of pairs of attacking queens for the given queens assignment on the board.
    """
    attack_pairs = 0


    ### START CODE HERE
    for i in range(len(queens)):
        for j in range(i+1,len(queens)):
            if queens[i]==queens[j]:
                attack_pairs+=1
            if abs(queens[i]-queens[j])==abs(i-j): #check diagonal attacks
                attack_pairs+=1

    ### END CODE HERE

    return attack_pairs
```

In [ ]:
```python
# UNIT TEST 1 - attack_pairs()

queens = np.array([3, 2, 1, 4, 3, 2, 1, 2])
ap = attack_pairs(queens)

print(f'number of attacking pairs for [3, 2, 1, 4, 3, 2, 1, 2]: {ap}')
```

```
number of attacking pairs for [3, 2, 1, 4, 3, 2, 1, 2]: 17
```

Expected Reult: number of attacking pairs for [3, 2, 1, 4, 3, 2, 1, 2]: 17

In [ ]:
```python
# UNIT TEST 2 - attack_pairs()

queens = np.array([0, 2, 1, 4, 3, 2, 1, 2])
ap = attack_pairs(queens)

print(f'number of attacking pairs for [0, 2, 1, 4, 3, 2, 1, 2]: {ap}')
```

```
number of attacking pairs for [0, 2, 1, 4, 3, 2, 1, 2]: 14
```

Expected Reult: number of attacking pairs for [0, 2, 1, 4, 3, 2, 1, 2]: 14

1.2 Exercise - Function Implementation - Count attacking pairs for successors of queens assignment

[5 points]

The attack_paris_board() function computes the number of pairs of queens attacking each other when you've moved one queen in one column to a different row within the same column. (Moves of queens within their same columns are what constitutes successors for the purposes of our approach.)

Refer to our example:

No description has been provided for this image

This diagram shows, for example, that if we move the queen in the first column from the 3rd row (from the bottom) to the 4th row, the number of attacking pairs will change from its current value of 17 to 15. On the other hand, as an another example, if we move the queen in the third column from its current position in the 2nd row to the top row, the number of pairs of attacking queen changes from 17 to 12.

So, attack_pairs_board() computes all of these numbers and returns them in the form of an n x n Numpy array.

Implement the function in the space provided. **You should make use of the attack_pairs() function you implemented, above, in your code for this function.**

The unit test below will help you determine whether your implementation is on the right track.

In [ ]:
```python
def attack_pairs_board(queens):
    """
    computes the number of pairs of attacking queens for each successor queen assignment to the one passed in as an
    input argument.
    Args:
      queens (ndarray (n, ))   : represents the assignment of queens on the board.  n = 8 for the 8-queens problem.
    Returns
      counts (ndarray (n, n))  : number of pairs of attacking queens for when the queen of each column is moved from
                                 its current row to the row of each respective cell.
    """

    n = len(queens)

    ### START CODE HERE


    counts = np.zeros((n, n))

    for i in range(n):

      # create nxn matrix for if queen in column was in another row
      queens_matrix = np.tile(queens, (n, 1))
```

```
    # put queen in ith row of each coloumn
    queens_matrix[:, i] = np.arange(n)

    #compute attack pairs if queen is moved to each row in column
    for j,q in enumerate(queens_matrix):
      counts[j,i] = attack_pairs(q)



  ### END CODE HERE

  return counts
```

In [ ]:
```
# UNIT TEST 1 - attack_pairs_board()

queens = np.array([3, 2, 1, 4, 3, 2, 1, 2])
ap_board = attack_pairs_board(queens)

print(f'successors attacking pairs for [3, 2, 1, 4, 3, 2, 1, 2]: \n\n{ap_board}')
```

```
successors attacking pairs for [3, 2, 1, 4, 3, 2, 1, 2]:

[[14. 14. 13. 17. 12. 14. 12. 18.]
 [18. 14. 17. 15. 15. 14. 17. 16.]
 [17. 17. 16. 18. 15. 17. 15. 17.]
 [17. 14. 17. 15. 17. 14. 16. 16.]
 [15. 14. 14. 17. 13. 16. 13. 16.]
 [14. 12. 18. 13. 15. 12. 14. 14.]
 [14. 16. 13. 15. 12. 14. 12. 16.]
 [18. 12. 14. 13. 13. 12. 14. 14.]]
```

Expected Reult:

successors attacking pairs for [3, 2, 1, 4, 3, 2, 1, 2]:

[[14. 14. 13. 17. 12. 14. 12. 18.]
[18. 14. 17. 15. 15. 14. 17. 16.]
[17. 17. 16. 18. 15. 17. 15. 17.]
[17. 14. 17. 15. 17. 14. 16. 16.]
[15. 14. 14. 17. 13. 16. 13. 16.]
[14. 12. 18. 13. 15. 12. 14. 14.]
[14. 16. 13. 15. 12. 14. 12. 16.]
[18. 12. 14. 13. 13. 12. 14. 14.]]

In [ ]:
```
# UNIT TEST 2 - attack_pairs_board()

queens = np.array([0, 2, 1, 4, 3, 2, 1, 2])
ap_board = attack_pairs_board(queens)

print(f'successors attacking pairs for [0, 2, 1, 4, 3, 2, 1, 2]: \n\n{ap_board}')
```

```
successors attacking pairs for [0, 2, 1, 4, 3, 2, 1, 2]:

[[14. 13. 12. 14. 11. 12. 10. 16.]
 [18. 13. 14. 12. 13. 11. 14. 13.]
 [17. 14. 15. 15. 13. 14. 12. 14.]
 [17. 11. 14. 12. 14. 10. 12. 12.]
 [15. 11. 12. 14. 12. 13. 10. 13.]
 [14. 10. 15. 10. 13. 10. 11. 11.]
 [14. 14. 11. 11. 10. 11. 10. 13.]
 [18. 10. 12. 10. 10.  9. 11. 12.]]
```

Expected Reult:

successors attacking pairs for [0, 2, 1, 4, 3, 2, 1, 2]:

[[14. 13. 12. 14. 11. 12. 10. 16.]
[18. 13. 14. 12. 13. 11. 14. 13.]
[17. 14. 15. 15. 13. 14. 12. 14.]
[17. 11. 14. 12. 14. 10. 12. 12.]
[15. 11. 12. 14. 12. 13. 10. 13.]
[14. 10. 15. 10. 13. 10. 11. 11.]
[14. 14. 11. 11. 10. 11. 10. 13.]
[18. 10. 12. 10. 10. 9. 11. 12.]]

1.3 Exercise - Function Implementation - Steepest Ascent Hill Climb

[10 points]

The steepest_ascent_hill_climb() function implements the algorithm after which it was named, where at each state it moves to an adjacent state offering a minimum value of attacking pairs among the set of successors.

You should make use of the attack_pairs_board() function above in your logic for choosing a successor state and for determining whether you've reached a local minimum.

In [ ]:
```
def steepest_ascent_hill_climb(n):
    """
    performs a steepest ascent hill climb toward a goal state of a queens assignment (represented in the form of a
    Numpy array of size (n, )) in which there are no pairs of queens attacking each other.  Not every execution
    of this function will result in a success - often a local optimum will be reached (i.e., a local min in which
    the number of attacking pairs is > 0, but no neighbors offer any improvement.

    Args:
      n (scalar))                      : dimension of the board.  For 8-queens, n = 8 (but we could use this to
                                         solve say, 10-queens)
    Returns
      current_attack_pairs (scalar)    : count of attacking pairs of the local optimum it found (0 if goal state found)
      queens (ndarray (n, ))           : locally optimum queens assignment, or, if attack pairs = 0, a globally optimum
                                         assignment
    """

    # start with a random assignment of queens on the board.
    queens = np.random.randint(n, size=n)


    ### START CODE HERE

    cur_attack_pair = attack_pairs(queens)

    while True:

        neighbor_attack_pairs = attack_pairs_board(queens)

        lowest_pair = neighbor_attack_pairs.min()

        if lowest_pair < cur_attack_pair:

            row, col = np.where(neighbor_attack_pairs == lowest_pair)
            #move queen to row with lowest attack pair
            queens[col[0]] = row[0]
            cur_attack_pair = lowest_pair

        else:
            break
    return cur_attack_pair, queens




    ### END CODE HERE
```

In [ ]:
```
# UNIT TEST 1 - steepest_ascent_hill_climb()

# This test runs steepest ascent 100 times, giving us the chance to to observe the frequency with which
# it arrives at a solution for 100 randomly chosen starting queen assignments.  We know from the literature
# that the overall average is about 14%.

np.random.seed(0)  # reset seed to produce the same set of starting queen assignments with every execution

num_successes = 0
for i in range(100):
    attack_pairs_count, queens = steepest_ascent_hill_climb(8)
    if attack_pairs_count == 0:
        print(f'Success: {queens}')
        num_successes += 1

print(f'\nNumber of successes: {num_successes}')
```

```
Success: [6 0 2 7 5 3 1 4]
Success: [2 6 1 7 5 3 0 4]
Success: [4 1 3 6 2 7 5 0]
Success: [4 0 7 5 2 6 1 3]
Success: [3 7 4 2 0 6 1 5]
Success: [3 5 0 4 1 7 2 6]
Success: [1 3 5 7 2 0 6 4]
Success: [2 5 3 1 7 4 6 0]
Success: [6 4 2 0 5 7 1 3]
Success: [6 2 7 1 4 0 5 3]
Success: [3 5 7 1 6 0 2 4]
Success: [3 1 4 7 5 0 2 6]

Number of successes: 12
```

Expected Result:

Success: [6 0 2 7 5 3 1 4]
Success: [2 6 1 7 5 3 0 4]
Success: [4 1 3 6 2 7 5 0]
Success: [4 0 7 5 2 6 1 3]
Success: [3 7 4 2 0 6 1 5]
Success: [3 5 0 4 1 7 2 6]
Success: [1 3 5 7 2 0 6 4]
Success: [2 5 3 1 7 4 6 0]
Success: [6 4 2 0 5 7 1 3]
Success: [6 2 7 1 4 0 5 3]
Success: [3 5 7 1 6 0 2 4]
Success: [3 1 4 7 5 0 2 6]

Number of successes: 12

2 - Stochastic Hill Climbing

[30 points]

You will implement the stochastic hill climbing algorithm in the cells below.

2.1 Exercise - Function Implementation - Probability distribution based on queens assignment

[10 points]

Stochastic hill climbing involves selecting successors from a probability distribution instead of picking the one that has the largest improvement in the objective function. Therefore, in order to implement this technique, we need to build a function that returns a probability distribution upon which our selection of a state's successor will be based.

The probability distribution will be developed using the following approach:

1. Determine the maximum number of attacking pairs possible for a set of n queens on a board. From you days in CSE 2500 you may recall that this is n "choose" 2, that is:

$$\text{worst case attack pairs count} = \binom{n}{2} = \frac{n(n-1)}{2}$$

2. Determine the fitness for each successor cell on the board according to the following formula:

$$\text{successors\_fitness} = \text{worst case attack pairs count} - \text{successors\_counts}$$

You should use the attack_pairs_board() function you developed above to find the array of successors counts values for all the cells on the board, for a given queens assignment.

This formula will be applied to every cell on the board, to each respective successor count. For example, for an 8-queens instance, you should have an 8 x 8 array of successor count values (from calling attack_pairs_board()) to which you should broadcast the fitness calculation above to get an 8 x 8 grid of successor_fitness values.

3. Scale the successor_fitness array with an constant, k, that prescribes the ratio of the max fitness value over the min fitness value, that is:

$$k = \max(\text{successors\_fitness})/\min(\text{successors\_fitness})$$

This value of k will be pre-determined and will serve as an input to this function for scaling the probabilities in your distribution to be developed by this function.

4. Calculate the scaled successor fitness values as follows:

$$\text{scaled\_successors\_fitness} = \frac{\text{successors\_fitness} \cdot (k-1)}{(x_2 - x_1)} + \frac{x_2 - k \cdot x_1}{(x_2 - x_1)}$$

where

$$x_2 = \max(\text{successors\_fitness})$$

and

$$x_1 = \min(\text{successors\_fitness})$$

This step should yield an n x n ndarray where the following principle holds:

$$\max(\text{successors\_fitness}) = k \times \min(\text{successors\_fitness})$$

5. Build the probabilities by dividing these scaled successor fitness values by their sum.

$$\text{probabilities} = \frac{\text{scaled\_successors\_fitness}}{\text{scaled\_successors\_fitness.sum()}}$$

This yields an n x n ndarray of values between 0 and 1 which serves as the distribution returned by the function.

Notice that cells with lower attack pair values will be assigned higher probabilities and vice versa and that the sum of these values is 1 (as required for a probability distribution).

In [ ]:
```python
def successors_probs(queens, k):
    """
    returns a probability distribution whose values correspond to the attack pair counts for a queens arrangement
    that is passed in as an input argument.  That is, cells with lower attack pair counts are assigned higher
    probabilities and those with higher counts are assigned lower probabilities.

    Args:
      queens ((n, ) ndarray)         : queens assignment on a board
      k (scalar)                     : scaling factor for probabilities.  (max_prob = k x min_prob - see above)

    Returns
      probs ((n**2, ) ndarray)         : 1D array of probs whose length is n**2, giving a probability for each cell
                                         in the n x n grid of successors.
    """
    n = len(queens)

    ### BEGIN CODE HERE
    # worst_case_attack_pairs_count} = {n \choose 2} = \frac{n (n-1)}{2}$$
    worst_case= n * (n - 1) / 2

    successors_counts = attack_pairs_board(queens)
    successors_fitness = worst_case - successors_counts

    max_fitness = np.max(successors_fitness)
    min_fitness = np.min(successors_fitness)

    scaled_fitness = (successors_fitness * (k - 1) + max_fitness - k * min_fitness) / (max_fitness - min_fitness)

    probs = scaled_fitness / np.sum(scaled_fitness)

    probs = probs.ravel()

    ### END CODE HERE

    return probs
```

In [ ]:
```python
# UNIT TEST 1 - successors_probs()

queens = np.array([3, 2, 1, 4, 3, 2, 1, 2])
successors_probs(queens, k=20)
```

Out[ ]:
```
array([0.01942207, 0.01942207, 0.02392231, 0.00592136, 0.02842255,
       0.01942207, 0.02842255, 0.00142113, 0.00142113, 0.01942207,
       0.00592136, 0.01492184, 0.01492184, 0.01942207, 0.00592136,
       0.0104216 , 0.00592136, 0.00592136, 0.0104216 , 0.00142113,
       0.01492184, 0.00592136, 0.01492184, 0.00592136, 0.00592136,
       0.01942207, 0.00592136, 0.01492184, 0.00592136, 0.01942207,
       0.0104216 , 0.0104216 , 0.01492184, 0.01942207, 0.01942207,
       0.00592136, 0.02392231, 0.0104216 , 0.02392231, 0.0104216 ,
       0.01942207, 0.02842255, 0.00142113, 0.02392231, 0.01492184,
       0.02842255, 0.01942207, 0.01942207, 0.01942207, 0.0104216 ,
       0.02392231, 0.01492184, 0.02842255, 0.01942207, 0.02842255,
       0.0104216 , 0.00142113, 0.02842255, 0.01942207, 0.02392231,
       0.02392231, 0.02842255, 0.01942207, 0.01942207])
```

Expected Result:

array([0.01942207, 0.01942207, 0.02392231, 0.00592136, 0.02842255,
0.01942207, 0.02842255, 0.00142113, 0.00142113, 0.01942207,
0.00592136, 0.01492184, 0.01492184, 0.01942207, 0.00592136,
0.0104216 , 0.00592136, 0.00592136, 0.0104216 , 0.00142113,
0.01492184, 0.00592136, 0.01492184, 0.00592136, 0.00592136,
0.01942207, 0.00592136, 0.01492184, 0.00592136, 0.01942207,
0.0104216 , 0.0104216 , 0.01492184, 0.01942207, 0.01942207,
0.00592136, 0.02392231, 0.0104216 , 0.02392231, 0.0104216 ,
0.01942207, 0.02842255, 0.00142113, 0.02392231, 0.01492184,
0.02842255, 0.01942207, 0.01942207, 0.01942207, 0.0104216 ,
0.02392231, 0.01492184, 0.02842255, 0.01942207, 0.02842255,
0.0104216 , 0.00142113, 0.02842255, 0.01942207, 0.02392231,
0.02392231, 0.02842255, 0.01942207, 0.01942207])

2.2 Exercise - Function Implementation - Stochastic Hill Climbing

[20 points]

Using the successors_probs() function you created above, you will apply it in the implementation of the stochastic hill climbing algorithm you'll code in the cell below.

In stochastic hill climbing, the algorithm picks a successor state based on a probability distribution, not on a steepest ascent metric. You'll call the successors_probs() function to create a distribution that biases in favor of states that offer larger improvement in the attack_pairs metric, but allows the possibility of a successor with a smaller improvement, or even a negative change.

How to terminate this algorithm? Allow this function to iterate 1000 times. If it finds a goal state before that, it should return the goal state queens assignment. Otherwise, return whatever it has after 1000 iterations.

Notice that we do not stop the algorithm if we hit a local minimum - we simply keep on picking states randomly until we hit a goal or 1000 iterations.

In [ ]:
```python
def stochastic_hill_climb(n, k):
    """
    implements the stochastic hill climbing algorithm, starting with a random queens assignment and repeatedly
    picking successor states randomly (according to a probability distribution proportionate to states' fitness
    levels) until either a goal state is found (no attacking pairs) or until 1000 iterations have been executed.

    Args:
      n  (scalar)                    : size of the board.  In 8-queens, n = 8
      k (scalar)                     : scaling factor for probabilities

    Returns
      attack_pairs (scalar)          : count of attacking pairs of queens when the algorithm is finished (0 if
                                       it finds a goal state)
      queens ((n, ) ndarray)         : queens assignment when the algorithm is finished
    """


    # start with a random assignment of queens on the board.
    queens = np.random.randint(n, size=n)

    ### START CODE HERE

    for _ in range(1000):
        # get probability distribution for successors
        probs = successors_probs(queens, k)

        # n * n possible indices
        choices = np.arange(n**2)

        selected_index = np.random.choice(choices, p=probs)

        row = selected_index // n
        col = selected_index % n
        queens[col] = row

        if attack_pairs(queens) == 0:
            return 0, queens


    ### END CODE HERE

    return attack_pairs(queens), queens
```

In [ ]:
```python
# UNIT TEST 1 - stochastic_hill_climb()

# np.random.seed(0)  # reset seed to produce the same set of starting queen assignments with every execution
num_successes = 0
for i in range(100):
    attack_pairs_count, queens = stochastic_hill_climb(n = 8, k = 5000)
    if attack_pairs_count == 0:
        print(f'Success: {queens}')
        num_successes += 1

print(f'\nNumber of successes: {num_successes}')
```

```
Success: [3 7 4 2 0 6 1 5]
Success: [6 0 2 7 5 3 1 4]
Success: [5 2 4 6 0 3 1 7]
Success: [4 0 7 3 1 6 2 5]
Success: [4 6 3 0 2 7 5 1]
Success: [4 6 1 5 2 0 7 3]
Success: [3 1 6 4 0 7 5 2]
Success: [4 0 7 3 1 6 2 5]
Success: [3 1 6 2 5 7 4 0]
Success: [5 7 1 3 0 6 4 2]
Success: [4 6 1 5 2 0 7 3]
Success: [4 7 3 0 6 1 5 2]
Success: [6 1 5 2 0 3 7 4]

Number of successes: 13
```

Example Result: (your result will likely vary from this):

Success: [5 3 6 0 2 4 1 7]
Success: [5 7 1 3 0 6 4 2]
Success: [3 0 4 7 1 6 2 5]
Success: [7 3 0 2 5 1 6 4]
Success: [2 5 1 6 4 0 7 3]
Success: [0 6 4 7 1 3 5 2]
Success: [2 5 1 6 0 3 7 4]
Success: [4 2 0 6 1 7 5 3]
Success: [2 0 6 4 7 1 3 5]
Success: [2 5 3 1 7 4 6 0]
Success: [3 6 4 2 0 5 7 1]
Success: [5 2 4 7 0 3 1 6]

Number of successes: 12

Problem 3 - First Choice Hill Climbing

[30 points]

You will implement the first choice hill climbing algorithm in the cells below.

3.1 Exercise - Function Implementation - First choice

[10 points]

Implement the first_choice() function which repeatedly picks successor states until one is found that is better than the current state; that is, has a lower attacking pairs count than that of the current state.

This function takes a queens assignment as an input parameter and a scaling factor, k, which gives determines the character of the probability distribution.

Use the attack_pairs() function, the attack_pairs_board() function, and the successors_probs() functions you implemented above to help you code the implementation for this function. The value of k passed in as an input parameter is the parm you'll pass to the successors_probs function.

first_choice() implementation - Q: Do we have an infinite loop concern? - A: No, if instructions are followed...

Note that you should **not** need to be concerned with this function, first_choice(), entering an infinite loop because of a possible edge case of queens (the input array) being a local min whose attack_pairs() count is less than all successors. This is because your first_choice_hill_climb() function (to be implemented next) should only call this function if queens is **not** a local min.

In [ ]:
```
def first_choice(queens, k):
    """
    returns an index value of a successor state picked randomly, but which offers an improvement in the attack pairs
    metric over the current state of the queens assignment passed in as input.  You will use the random.choices()
    function to pick a value based on the probability distribution.
    Args:
      queens ((n, ) ndarray)        : queens assignment on a board
      k  (scalar)                   : scaling factor for the probability distribution.  needed by successors_probs()
    Returns
      select_index (scalar)         : index value of the cell in the attack_pairs_board(queens) array which gives the
                                      successor chosen to the queens array passed in as input.  Note you will need
                                      to use the following to map back to a row and index in the 2D attack_pairs_board()
                                      array:  row = select_index // n,  column = select_index % n
    """


    n = len(queens)

    ### START CODE HERE

    current_attack_pairs = attack_pairs(queens)
    probs = successors_probs(queens, k)

    possible_indices = np.arange(n * n)
    select_index = None
    while True:
        # Randomly select a successor based on the probabilities
        select_index = np.random.choice(possible_indices, p=probs)

        # Check if  successor has fewer attacking pairs than current state
        col = select_index // n
        row = select_index % n

        temp_queens = queens.copy()
        temp_queens[col] = row

        if attack_pairs(temp_queens) < current_attack_pairs:
          break




    ### END CODE HERE
    return select_index
```

In [ ]:
```
# UNIT TEST 1 - first_choice()

# np.random.seed(0)  # reset seed to produce the same set of starting queen assignments with every execution

queens = np.array([3, 2, 1, 4, 3, 2, 1, 2])

n = len(queens)
print(f'queens: {queens}')

ap = attack_pairs(queens)
print(f'attack_pairs(queens): {ap}')

select_index = first_choice(queens, 5)
row_move = select_index // n
col_move = select_index % n

print(f'select_index: {select_index}')
print(f'row_move: {row_move}')
print(f'col_move: {col_move}')

# move to successor state (move queen...)
queens[col_move] = row_move
print(f'queens: {queens}')

ap_new = attack_pairs(queens)
print(f'attack_pairs(queens): {ap_new}')
```

```
queens: [3 2 1 4 3 2 1 2]
attack_pairs(queens): 17
select_index: 45
row_move: 5
col_move: 5
queens: [3 2 1 4 3 5 1 2]
attack_pairs(queens): 12
```

Example Result: (your result will likely vary from this):

queens: [3 2 1 4 3 2 1 2]

attack_pairs(queens): 17

select_index: 5

row_move: 0

col_move: 5

queens: [3 2 1 4 3 0 1 2]

attack_pairs(queens): 14

3.2 Exercise - Function Implementation - First choice hill climbing

[20 points]

Implement the first choice hill climbing algorihm in the cell below, utilizing the first_choice() function you coded above for choosing the successor at each step.

Note that your implementation should *test whether the current state is a local min before calling first_choice()*. This prevents first_choice() from entering an infinite loop, as discussed in the comments to the last exercise.

Execute the algo loop 1000 times in your implementation. If a goal state is found, return the attack pairs count of 0 and the queens assignment. If no goal state is found, return the attack pairs count and queens assignment at the last step of the algorithm.

In [ ]:
```
def first_choice_hill_climb(n, k):
    """
    implements the first choice hill climbing algorithm, starting with a random queens assignment and repeatedly
    picking successor states using the first choice function until either a goal state is found (no attacking pairs)
    or until 1000 iterations have been executed.

    Args:
      n  (scalar)                    : size of the board.  In 8-queens, n = 8
      k  (scalar)                    : scaling factor for the probability distribution.  needed by successors_probs()
    Returns
      attack_pairs (scalar)          : count of attacking pairs of queens when the algorithm is finished (0 if
                                       it finds a goal state)
      queens ((n, ) ndarray)         : queens assignment when the algorithm is finished
    """

    # start with a random assignment of queens on the board.
    queens = np.random.randint(n, size=n)

    ### START CODE HERE
    for _ in range(1000):
        current_attacks = attack_pairs(queens)

        # If current state is a goal state
        if current_attacks == 0:
            return 0, queens

        # Check for local min
        attacks_board = attack_pairs_board(queens)
        if np.min(attacks_board) >= current_attacks:
            break  # Exit if local minimum

        # Else find sucessor
        select_index = first_choice(queens, k)
        col = select_index // n
        row = select_index % n
```

```python
        queens[col] = row

        #check if updated state is a goal state
        if attack_pairs(queens) == 0:
            return 0, queens


    ### END CODE HERE

    return attack_pairs(queens), queens
```

In [ ]:
```python
# UNIT TEST 1 - first_choice_hill_climb()

# np.random.seed(0)  # reset seed to produce the same set of starting queen assignments with every execution

num_successes = 0
for i in range(100):
    attack_pairs_count, queens = first_choice_hill_climb(n = 8, k = 10)
    if attack_pairs_count == 0:
        print(f'Success: {queens}')
        num_successes += 1
print(f'\nNumber of successes: {num_successes}')
```

```
Success: [3 1 6 2 5 7 0 4]
Success: [4 6 1 5 2 0 3 7]
Success: [3 5 7 2 0 6 4 1]
Success: [7 1 3 0 6 4 2 5]
Success: [6 2 0 5 7 4 1 3]
Success: [5 2 0 7 4 1 3 6]
Success: [4 1 3 6 2 7 5 0]
Success: [5 2 0 7 3 1 6 4]
Success: [3 1 7 5 0 2 4 6]
Success: [5 1 6 0 2 4 7 3]
Success: [2 4 1 7 0 6 3 5]

Number of successes: 11
```

Example Result: (your result will likely vary from this):

Success: [1 7 5 0 2 4 6 3]

Success: [4 1 5 0 6 3 7 2]

Success: [1 7 5 0 2 4 6 3]

Success: [4 1 5 0 6 3 7 2]

Success: [4 7 3 0 6 1 5 2]

Success: [4 7 3 0 6 1 5 2]

Success: [4 1 5 0 6 3 7 2]

Success: [4 7 3 0 2 5 1 6]

Success: [4 1 5 0 6 3 7 2]

Success: [4 1 5 0 6 3 7 2]

Success: [4 7 3 0 6 1 5 2]

Success: [4 1 5 0 6 3 7 2]

Success: [4 1 5 0 6 3 7 2]

Success: [4 1 5 0 6 3 7 2]

Success: [4 1 5 0 6 3 7 2]

Number of successes: 15

Problem 4 - Random Restart Hill Climbing

[15 points]

You will implement the random restart hill climbing algorithm in the cells below.

4.1 Exercise - Function Implementation - Random Restart Hill Climbing

[15 points]

Random restart hill climbing is essentially repeated executions of the steepest ascent hill climbing algorithm. Implement this algorithm in the cell below.

In [ ]:
```python
def random_restart_hill_climb(n, attempts):
    """
    implements the random restart hill climbing algorithm, executing the steepest ascent hill climbing algorithm
    until a goal state is found or until an attempt limit has been reached.

    Args:
      n  (scalar)                    : size of the board.  In 8-queens, n = 8
      attempts  (scalar)             : the number of attempts to take at the steepest ascent hill climbing algorithm
    Returns
      attack_pairs (scalar)          : count of attacking pairs of queens when the algorithm is finished (0 if
                                       it finds a goal state)
      queens ((n, ) ndarray)         : queens assignment when the algorithm is finished, goal state if one if found
    """

    best_ap = 9999
    best_queens = np.zeros(8)

    ### START CODE HERE

    best_ap = float('inf')  # Initialize as inf
    best_queens = None  # Initialize as none

    for _ in range(attempts):
        # Run steepest ascent hill climb algo
        ap, queens = steepest_ascent_hill_climb(n)

        # Update sol if better found
        if ap < best_ap:
            best_ap = ap
            best_queens = queens

        if best_ap == 0:
            break

    ### END CODE HERE

    return best_ap, best_queens
```

In [ ]:
```python
# UNIT TEST 1 - random_restart_hill_climb()

np.random.seed(0)  # reset seed to produce the same set of starting queen assignments with every execution

num_successes = 0
for i in range(100):
    attack_pairs_count, queens = random_restart_hill_climb(n = 8, attempts=7)
    if attack_pairs_count == 0:
        print(f'Success: {queens}')
        num_successes += 1
print(f'\nNumber of successes: {num_successes}')
```

```
Success: [6 0 2 7 5 3 1 4]
Success: [2 6 1 7 5 3 0 4]
Success: [4 1 3 6 2 7 5 0]
Success: [4 0 7 5 2 6 1 3]
Success: [3 7 4 2 0 6 1 5]
Success: [3 5 0 4 1 7 2 6]
Success: [1 3 5 7 2 0 6 4]
Success: [2 5 3 1 7 4 6 0]
Success: [6 4 2 0 5 7 1 3]
Success: [6 2 7 1 4 0 5 3]
Success: [3 5 7 1 6 0 2 4]
Success: [3 1 4 7 5 0 2 6]
Success: [3 1 7 4 6 0 2 5]
Success: [1 7 5 0 2 4 6 3]
Success: [0 5 7 2 6 3 1 4]
Success: [4 6 0 2 7 5 3 1]
Success: [4 7 3 0 2 5 1 6]
Success: [0 5 7 2 6 3 1 4]
Success: [3 1 6 2 5 7 4 0]
Success: [1 4 6 0 2 7 5 3]
Success: [1 6 4 7 0 3 5 2]
Success: [5 0 4 1 7 2 6 3]
Success: [2 5 7 1 3 0 6 4]
Success: [4 2 0 6 1 7 5 3]
Success: [1 6 2 5 7 4 0 3]
Success: [1 5 7 2 0 3 6 4]
Success: [5 2 6 1 3 7 0 4]
Success: [4 7 3 0 6 1 5 2]
Success: [2 6 1 7 4 0 3 5]
Success: [5 2 4 7 0 3 1 6]
Success: [3 1 7 4 6 0 2 5]
Success: [4 7 3 0 2 5 1 6]
Success: [3 1 7 5 0 2 4 6]
Success: [4 2 0 5 7 1 3 6]
Success: [3 7 0 4 6 1 5 2]
Success: [6 2 0 5 7 4 1 3]
Success: [3 6 4 2 0 5 7 1]
Success: [1 4 6 0 2 7 5 3]
Success: [4 1 7 0 3 6 2 5]
Success: [2 5 1 6 0 3 7 4]
Success: [5 7 1 3 0 6 4 2]
Success: [4 6 0 3 1 7 5 2]
Success: [0 4 7 5 2 6 1 3]
Success: [4 1 3 6 2 7 5 0]
Success: [0 6 4 7 1 3 5 2]
Success: [2 5 7 0 4 6 1 3]
Success: [3 6 0 7 4 1 5 2]
Success: [5 2 6 1 3 7 0 4]
Success: [6 4 2 0 5 7 1 3]
Success: [1 4 6 0 2 7 5 3]
Success: [5 1 6 0 2 4 7 3]
Success: [5 7 1 3 0 6 4 2]
Success: [4 1 3 5 7 2 0 6]
Success: [5 7 1 3 0 6 4 2]
Success: [5 2 4 7 0 3 1 6]
Success: [5 1 6 0 2 4 7 3]
Success: [5 1 6 0 2 4 7 3]
Success: [1 7 5 0 2 4 6 3]

Number of successes: 58
```

Expected Result:

Success: [6 0 2 7 5 3 1 4]
Success: [2 6 1 7 5 3 0 4]
Success: [4 1 3 6 2 7 5 0]
Success: [4 0 7 5 2 6 1 3]
Success: [3 7 4 2 0 6 1 5]
Success: [3 5 0 4 1 7 2 6]
Success: [1 3 5 7 2 0 6 4]
Success: [2 5 3 1 7 4 6 0]
Success: [6 4 2 0 5 7 1 3]
Success: [6 2 7 1 4 0 5 3]
Success: [3 5 7 1 6 0 2 4]
Success: [3 1 4 7 5 0 2 6]
Success: [3 1 7 4 6 0 2 5]
Success: [1 7 5 0 2 4 6 3]
Success: [0 5 7 2 6 3 1 4]
Success: [4 6 0 2 7 5 3 1]
Success: [4 7 3 0 2 5 1 6]
Success: [0 5 7 2 6 3 1 4]
Success: [3 1 6 2 5 7 4 0]
Success: [1 4 6 0 2 7 5 3]
Success: [1 6 4 7 0 3 5 2]
Success: [5 0 4 1 7 2 6 3]
Success: [2 5 7 1 3 0 6 4]
Success: [4 2 0 6 1 7 5 3]
Success: [1 6 2 5 7 4 0 3]
Success: [1 5 7 2 0 3 6 4]
Success: [5 2 6 1 3 7 0 4]
Success: [4 7 3 0 6 1 5 2]
Success: [2 6 1 7 4 0 3 5]
Success: [5 2 4 7 0 3 1 6]
Success: [3 1 7 4 6 0 2 5]
Success: [4 7 3 0 2 5 1 6]
Success: [3 1 7 5 0 2 4 6]
Success: [4 2 0 5 7 1 3 6]
Success: [3 7 0 4 6 1 5 2]
Success: [6 2 0 5 7 4 1 3]
Success: [3 6 4 2 0 5 7 1]
Success: [1 4 6 0 2 7 5 3]
Success: [4 1 7 0 3 6 2 5]
Success: [2 5 1 6 0 3 7 4]
Success: [5 7 1 3 0 6 4 2]
Success: [4 6 0 3 1 7 5 2]
Success: [0 4 7 5 2 6 1 3]
Success: [4 1 3 6 2 7 5 0]
Success: [0 6 4 7 1 3 5 2]
Success: [2 5 7 0 4 6 1 3]
Success: [3 6 0 7 4 1 5 2]
Success: [5 2 6 1 3 7 0 4]
Success: [6 4 2 0 5 7 1 3]
Success: [1 4 6 0 2 7 5 3]
Success: [5 1 6 0 2 4 7 3]
Success: [5 7 1 3 0 6 4 2]
Success: [4 1 3 5 7 2 0 6]
Success: [5 7 1 3 0 6 4 2]
Success: [5 2 4 7 0 3 1 6]
Success: [5 1 6 0 2 4 7 3]
Success: [5 1 6 0 2 4 7 3]
Success: [1 7 5 0 2 4 6 3]

Number of successes: 58

5 Congratulations!

In this lab you:

- implemented four significant local search, hill climbing algorithms - steepest ascent, stochastic, first choice, and randomized restart.