CAGATAY DUYGU

48369962

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

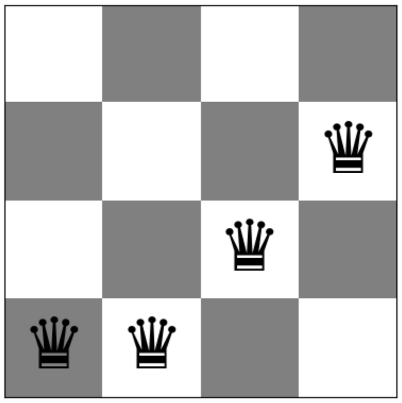
In [215…

```python
import math
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 [216…
```
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.
```

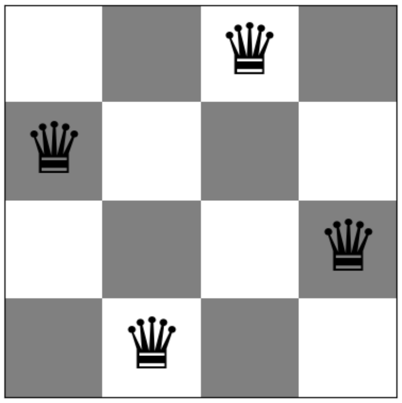


```
Queens (left to right) are at rows: [3 3 2 1]
Number of conflicts: 4
```

A board 4×4 with no conflicts:

In [217…
```
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.

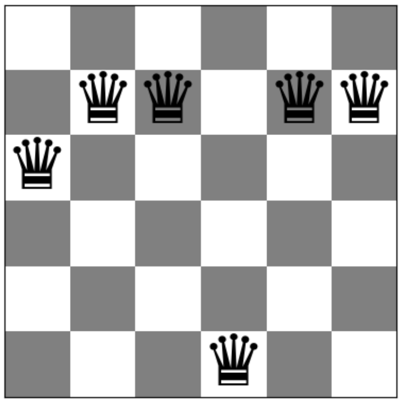
Implementation

In [316…
```
def sahc(n, Verbose):

    first_board = random_board(n)
    a_shape = (n, n)
    fill_value = -1

    #I made a conflict array which
    #includes the objective function
    #of every possible action.
    #This is the initialization (an array full of -1)
    num_of_conflict_array = np.full(a_shape, fill_value)

    if Verbose==True:
        print("First Board",first_board)
        show_board(first_board)
    conflict_first_board=conflicts(first_board)

    if Verbose==True:
        print("First Conflict", conflict_first_board)

    #make a copy of first board
    #so that it does not change
    current_board=np.copy(first_board)
    flag=1

    while True:
        action_list = []
        num_of_conflict_array
        for i in range(0,n):
            t = i
            action_list.append(t)


        for i in range (n):
            potential_board=np.copy(current_board)


            for m in action_list:
                potential_board[i] = m

                num_of_conflict_array[m,i] = conflicts(potential_board)

        #num of conflict aray calculates the conflicts for every possible action.
        #then, it will select the minimum one.

        current_board_flag=np.copy(current_board)
        c_flag=conflicts(current_board_flag)
        idx_min_conflict=np.unravel_index(num_of_conflict_array.argmin(), num_of_conflict_array.shape)

        #make the action that we will satisfy minimum
        #number of conflicts.
        current_board[idx_min_conflict[1]]=idx_min_conflict[0]


        if Verbose==True:
            show_board(current_board)

        flag=flag+1

        if Verbose==True:
            print("flag:", flag)
            print("New selected board:")

        #if zero conflict, break.
        if conflicts(current_board)==0:
            break
        #if the conflict is same with the previous one
        #local optima is reached.
        elif c_flag==conflicts(current_board):
            current_board=np.copy(current_board_flag)
            if Verbose==True:
                print("Encountered Local Minium")
            break

    if Verbose==True:
        print("Last Board:", current_board,"\n")
    #returns last number of conflicts
    return(conflicts(current_board))
```

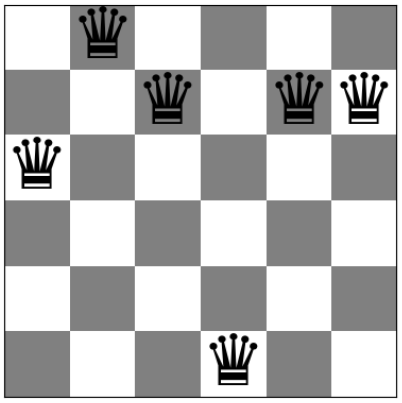
Example Run

In [306…
```
n=6
sahc(n,True)
```

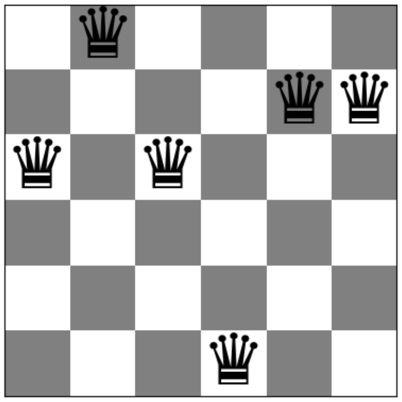
```
First Board [2 1 1 5 1 1]
Board with 8 conflicts.
```



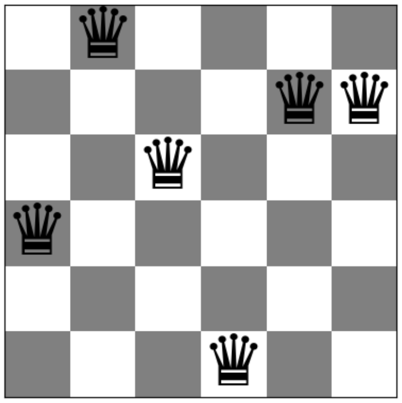
```
First Conflict 8
Board with 5 conflicts.
```



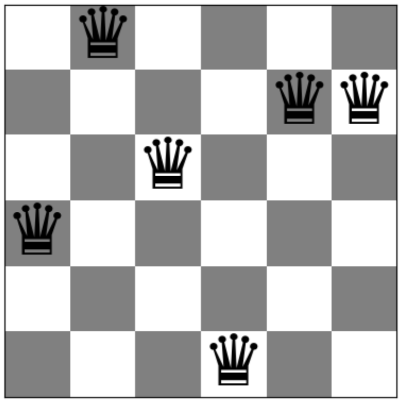
```
flag: 2
New selected board:
Board with 3 conflicts.
```



```
flag: 3
New selected board:
Board with 1 conflicts.
```



```
flag: 4
New selected board:
Board with 1 conflicts.
```



```
flag: 5
New selected board:
Encountered Local Minium
Last Board: [3 0 2 5 1 1]
```

Out[306]: 1

Steepest Ascent Hill Climbing without Printing, returns last number of conflicts

In [314…
```
sahc(4,False)
```

Out[314]: 0

Task 2: Stochastic Hill Climbing 1 [10 Points]

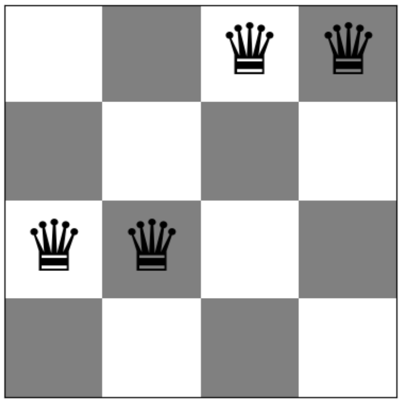
Chooses randomly from among all uphill and equal moves.

In [221…
```
import random
```

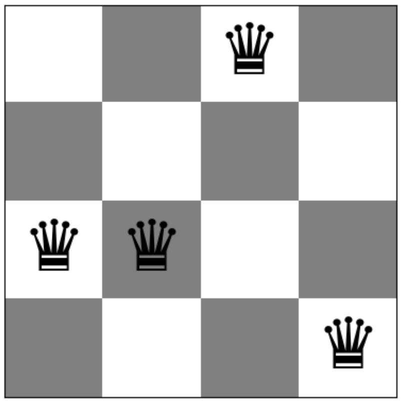
In [222…
```
def stoch1(n, Verbose):
    #create random board
    first_board = random_board(n)
    #initialize the array that will
    # represent the number of conflicts
    #for every possible neighbor in that
    #state
    a_shape = (n, n)
    fill_value = -1
    num_of_conflict_array = np.full(a_shape, fill_value)

    if Verbose==True:
        print("First Board",first_board)
        show_board(first_board)
    #save the # of conflicts of first board
    conflict_first_board=conflicts(first_board)

    #c_flag_list of is the list that
    #has
    c_flag_list=[]
    current_board=np.copy(first_board)
    flag=1
    while True:
 #make an action list
        action_list = []
        num_of_conflict_array
        for i in range(0,n):
            t = i
            action_list.append(t)


        for i in range (n):
            potential_board=np.copy(current_board)

            for m in action_list:
                potential_board[i] = m
                num_of_conflict_array[m,i] = conflicts(potential_board)


        current_board_flag=np.copy(current_board)
        #c_flag is the number of conflict in current board.
        # i will use it in an
        #if statement (i should break the algorithm)

        c_flag=conflicts(current_board_flag)
        if Verbose==True:
            print("Number of Conflict Array", num_of_conflict_array)
        result = np.where(num_of_conflict_array == np.amin(num_of_conflict_array))

        listOfCordinates = list(zip(result[0], result[1]))
        if Verbose==True:
            print("Minimum coordinates list", listOfCordinates)
        #randomly select an action from the list
        #that includes the potential actions
        #with smallest number of conflicts
        a=random.choice(listOfCordinates)
        if Verbose==True:
            print("Queen in column", a[1], "goes to row", a[0] )

        #Board after the random action
        current_board[a[1]]=a[0]
        if Verbose==True:
            print("Conflicts:", conflicts(current_board))
            show_board(current_board)

        c_flag_list.append(conflicts(current_board))
        flag=flag+1

        if Verbose==True:
            print("flag:", flag)
        #if no conflict, break
        if conflicts(current_board)==0:
            break
        #if same conflict with the previous one, break. (local optima)
        elif c_flag==conflicts(current_board):
            current_board=np.copy(current_board_flag)
            if Verbose==True:
                print("Encountered Local Optima")
                print("Number of Conflict in the Last Board is:",conflicts(current_board))
            break



    if Verbose==True:
        print("Last Board:", current_board,"\n")
    return conflicts(current_board)
```

In [223…
```
n=4
stoch1(n, True)
```

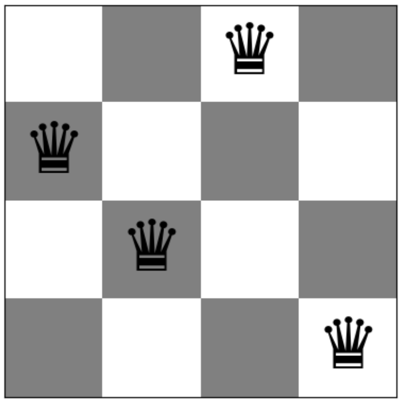
```
First Board [2 2 0 0]
Board with 4 conflicts.
```



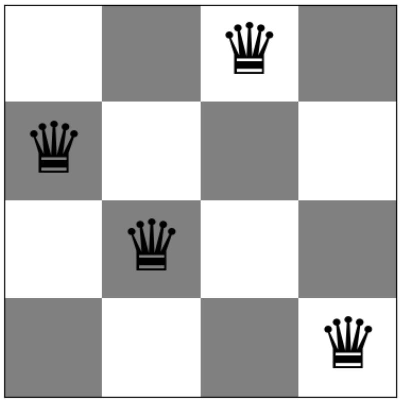
```
Number of Conflict Array [[4 4 4 4]
 [3 4 4 3]
 [4 4 4 4]
 [4 3 3 2]]
Minimum coordinates list [(3, 3)]
Queen in column 3 goes to row 3
Conflicts: 2
Board with 2 conflicts.
```



```
flag: 2
Number of Conflict Array [[2 2 2 4]
 [1 4 2 3]
 [2 2 4 4]
 [2 3 3 2]]
Minimum coordinates list [(1, 0)]
Queen in column 0 goes to row 1
Conflicts: 1
Board with 1 conflicts.
```



```
flag: 3
Number of Conflict Array [[2 2 1 3]
 [1 3 3 3]
 [2 1 3 2]
 [2 1 4 1]]
Minimum coordinates list [(0, 2), (1, 0), (2, 1), (3, 1), (3, 3)]
Queen in column 1 goes to row 2
Conflicts: 1
Board with 1 conflicts.
```



```
flag: 4
Encountered Local Optima
Number of Conflict in the Last Board is: 1
Last Board: [1 2 0 3]
```

Out[223]: 1

In [226…
```
n=4
stoch1(n, False)
```

Out[226]: 0

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.

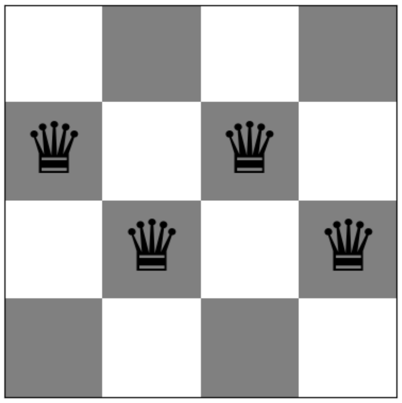
In [315…
```
def stoch2(n,Verbose):
    board = random_board(n)
    if Verbose==True:
        show_board(board)
    c_flag=[]

    while True:
        column_array = [k for k in range(0,n)]
        # Select the column randomly. Then we will
        # move the queen in that column
        selected_column=random.choice(column_array)
        if Verbose==True:
            print("Randomly selected column:",selected_column)
        conflict_original=conflicts(board)
        #row of selected queen
        queen_position=board[selected_column]
        action_array=[k for k in range(0,n)]
        #remove the current row of queen from action array
        action_array.remove(queen_position)
        if Verbose==True:
            print("Possible actions after removing current queen position",action_array)
        #take a random action from the list of rows.
        random_action=random.choice(action_array)

        if Verbose==True:
            print("Randomly selected action to test is the queen in column number",
              selected_column, "moves to row number", random_action)

        potential_board=np.copy(board)
        #potential board takes the action of
        #randomly selected row and column
        potential_board[selected_column]=random_action

        conflict_potential=conflicts(potential_board)

        if Verbose==True:
            print("Potential Action's number of conflict:", conflict_potential)

        #if conflict is smaller, accept the action
        if conflict_potential<conflict_original:
            board=np.copy(potential_board)
            if Verbose==True:
                print("Action selected:")
                show_board(board)
        else:
            board=board
            if Verbose==True:
                print("Action is not selected")


        c_flag.append(conflicts(board))
        last_5n = c_flag[-(15*n):]
        if Verbose==True:
            print("last 5n conflict values", last_5n)

            #check if last 15 conflict are same:
            # if yes, local optima
        result = all(element == last_5n[0] for element in last_5n)

            # This is an exception. If I do not put there
            # algorithm can break, thinking it is in local
            #optima earlier.
        if result==True and len(last_5n)>14*n:
            if Verbose==True:
                print("We are in local minimum. Stop the algorithm")
            break
            #break if zero conflicts
        if conflicts(board)==0:
            break

    if Verbose==True:
        print("Final Board:", board)

    return conflicts(board)
```

In [263…
```
stoch2(4, False)
```

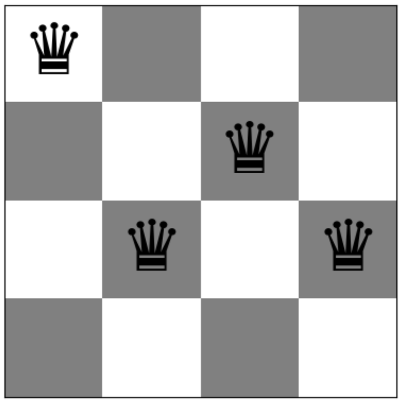
Out[263]:
```
0
```

In [265…
```
stoch2(4, True)
```

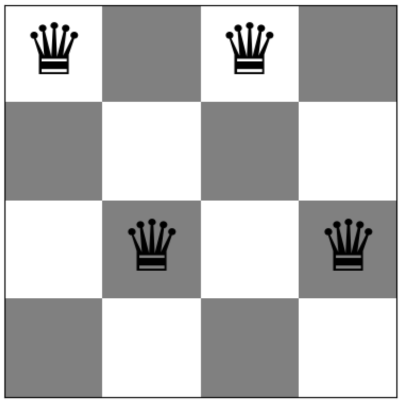
```
Board with 5 conflicts.
```



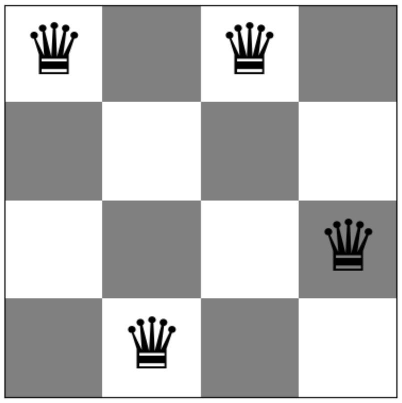
```
Randomly selected column: 0
Possible actions after removing current queen position [0, 2, 3]
Randomly selected action to test is the queen in column number 0 moves to row number 0
Potential Action's number of conflict: 3
Action selected:
Board with 3 conflicts.
```



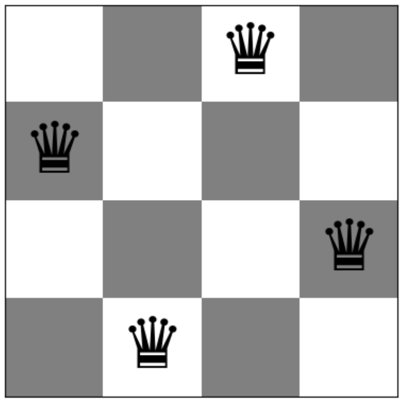
```
last 5n conflict values [3]
Randomly selected column: 2
Possible actions after removing current queen position [0, 2, 3]
Randomly selected action to test is the queen in column number 2 moves to row number 0
Potential Action's number of conflict: 2
Action selected:
Board with 2 conflicts.
```



```
last 5n conflict values [3, 2]
Randomly selected column: 0
Possible actions after removing current queen position [1, 2, 3]
Randomly selected action to test is the queen in column number 0 moves to row number 3
Potential Action's number of conflict: 2
Action is not selected
last 5n conflict values [3, 2, 2]
Randomly selected column: 0
Possible actions after removing current queen position [1, 2, 3]
Randomly selected action to test is the queen in column number 0 moves to row number 1
Potential Action's number of conflict: 2
Action is not selected
last 5n conflict values [3, 2, 2, 2]
Randomly selected column: 1
Possible actions after removing current queen position [0, 1, 3]
Randomly selected action to test is the queen in column number 1 moves to row number 3
Potential Action's number of conflict: 1
Action selected:
Board with 1 conflicts.
```



```
last 5n conflict values [3, 2, 2, 2, 1]
Randomly selected column: 1
Possible actions after removing current queen position [0, 1, 2]
Randomly selected action to test is the queen in column number 1 moves to row number 2
Potential Action's number of conflict: 2
Action is not selected
last 5n conflict values [3, 2, 2, 2, 1, 1]
Randomly selected column: 3
Possible actions after removing current queen position [0, 1, 3]
Randomly selected action to test is the queen in column number 3 moves to row number 3
Potential Action's number of conflict: 3
Action is not selected
last 5n conflict values [3, 2, 2, 2, 1, 1, 1]
Randomly selected column: 3
Possible actions after removing current queen position [0, 1, 3]
Randomly selected action to test is the queen in column number 3 moves to row number 3
Potential Action's number of conflict: 3
Action is not selected
last 5n conflict values [3, 2, 2, 2, 1, 1, 1, 1]
Randomly selected column: 2
Possible actions after removing current queen position [1, 2, 3]
Randomly selected action to test is the queen in column number 2 moves to row number 3
Potential Action's number of conflict: 2
Action is not selected
last 5n conflict values [3, 2, 2, 2, 1, 1, 1, 1, 1]
Randomly selected column: 3
Possible actions after removing current queen position [0, 1, 3]
Randomly selected action to test is the queen in column number 3 moves to row number 0
Potential Action's number of conflict: 3
Action is not selected
last 5n conflict values [3, 2, 2, 2, 1, 1, 1, 1, 1, 1]
Randomly selected column: 0
Possible actions after removing current queen position [1, 2, 3]
Randomly selected action to test is the queen in column number 0 moves to row number 3
Potential Action's number of conflict: 1
Action is not selected
last 5n conflict values [3, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1]
Randomly selected column: 1
Possible actions after removing current queen position [0, 1, 2]
Randomly selected action to test is the queen in column number 1 moves to row number 2
Potential Action's number of conflict: 2
Action is not selected
last 5n conflict values [3, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1]
Randomly selected column: 3
Possible actions after removing current queen position [0, 1, 3]
Randomly selected action to test is the queen in column number 3 moves to row number 3
Potential Action's number of conflict: 3
Action is not selected
last 5n conflict values [3, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Randomly selected column: 0
Possible actions after removing current queen position [1, 2, 3]
Randomly selected action to test is the queen in column number 0 moves to row number 2
Potential Action's number of conflict: 3
Action is not selected
last 5n conflict values [3, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Randomly selected column: 3
Possible actions after removing current queen position [0, 1, 3]
Randomly selected action to test is the queen in column number 3 moves to row number 1
Potential Action's number of conflict: 3
Action is not selected
last 5n conflict values [3, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Randomly selected column: 1
Possible actions after removing current queen position [0, 1, 2]
Randomly selected action to test is the queen in column number 1 moves to row number 1
Potential Action's number of conflict: 3
Action is not selected
last 5n conflict values [3, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Randomly selected column: 0
Possible actions after removing current queen position [1, 2, 3]
Randomly selected action to test is the queen in column number 0 moves to row number 1
Potential Action's number of conflict: 0
Action selected:
Board with 0 conflicts.
```



```
last 5n conflict values [3, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
Final Board: [1 3 0 2]
```

Out[265]: 0

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 [266…
```
from collections import Counter
```

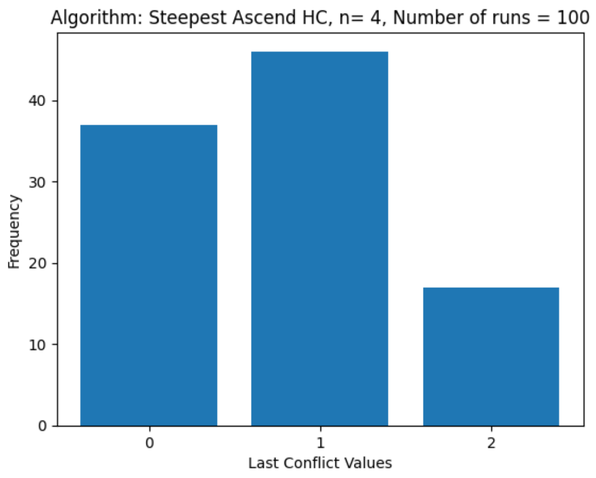
In [267…
```
import time
```

In [268…
```
def runsk(n,k):
    conflict_list_sahc=[]
    conflict_list_stoch1=[]
    conflict_list_stoch2=[]

    for i in range (k):
        conflict_list_sahc.append(sahc(n, False))

    for j in range (k):
        conflict_list_stoch1.append(stoch1(n, False))

    for l in range (k):
        conflict_list_stoch2.append(stoch2(n, False))



    return conflict_list_sahc, conflict_list_stoch1, conflict_list_stoch2
```

In [269…
```
k=100
n=4
conflict_list_sahc, conflict_list_stoch1, conflict_list_stoch2=runsk(n,k)
```

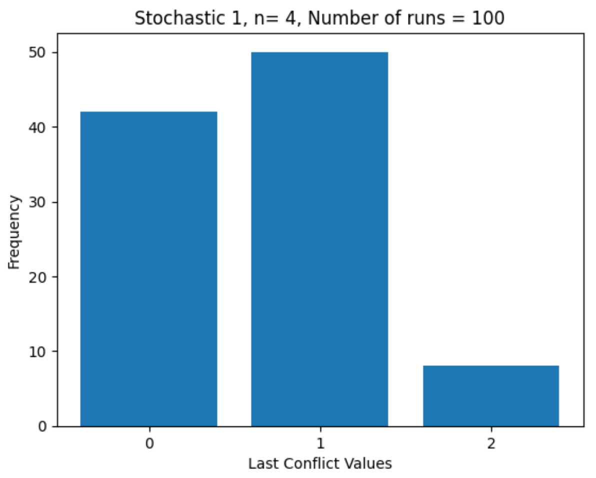
In [270…
```
values_sahc, counts_sahc = np.unique(conflict_list_sahc, return_counts=True)
plt.bar(values_sahc, counts_sahc)
plt.xticks(range(min(conflict_list_sahc),max(conflict_list_sahc)+1))
#plt.title("Steepest Ascend HC,  n=%i, number of runs: %i " %n %k )
plt.title('Algorithm: Steepest Ascend HC, n= '+str(n)+', Number of runs = '+str(k))
plt.xlabel("Last Conflict Values")
plt.ylabel("Frequency")
```

Out[270]:
```
Text(0, 0.5, 'Frequency')
```



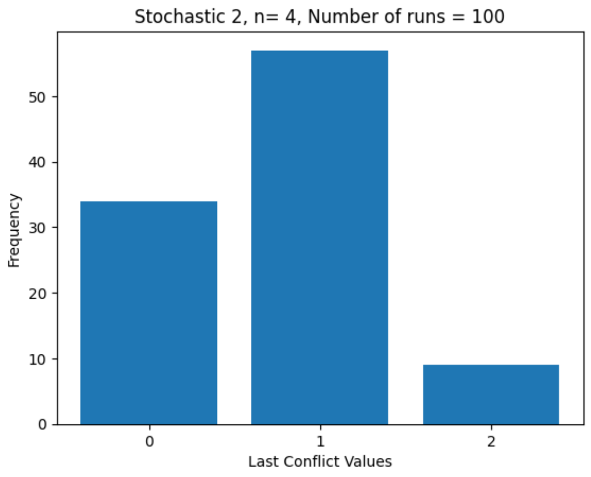
In [271…
```
values_stoch1, counts_stoch1 = np.unique(conflict_list_stoch1, return_counts=True)
plt.bar(values_stoch1, counts_stoch1)
plt.xticks(range(min(conflict_list_stoch1),max(conflict_list_stoch1)+1))
plt.title('Stochastic 1, n= '+str(n)+', Number of runs = '+str(k))
plt.xlabel("Last Conflict Values")
plt.ylabel("Frequency")
```

Out[271]:
```
Text(0, 0.5, 'Frequency')
```



In [272…
```
values_stoch1, counts_stoch2 = np.unique(conflict_list_stoch2, return_counts=True)
plt.bar(values_stoch1, counts_stoch2)
plt.xticks(range(min(conflict_list_stoch2),max(conflict_list_stoch2)+1))
plt.title('Stochastic 2, n= '+str(n)+', Number of runs = '+str(k))
plt.xlabel("Last Conflict Values")
plt.ylabel("Frequency")
```

```
Out[272]: Text(0, 0.5, 'Frequency')
```



As can be seen from the graphs, all of three implementation can achieve optimal solution.

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?

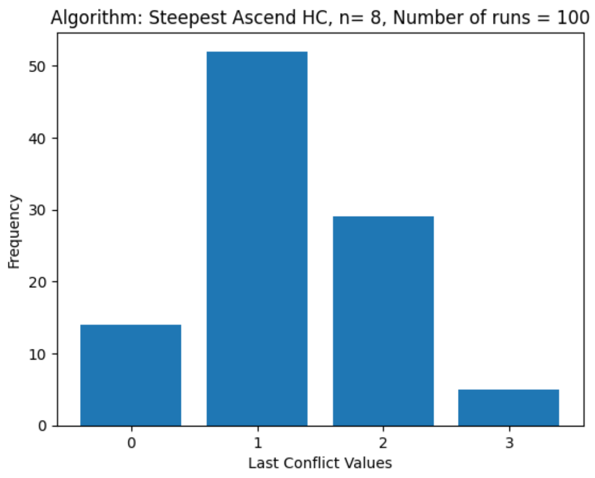
See Profiling Python Code for help about how to measure runtime in Python.

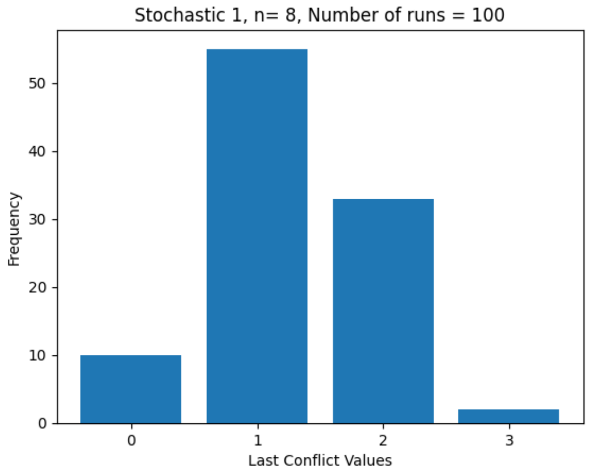
Plots for Trials for Different Size of Tables

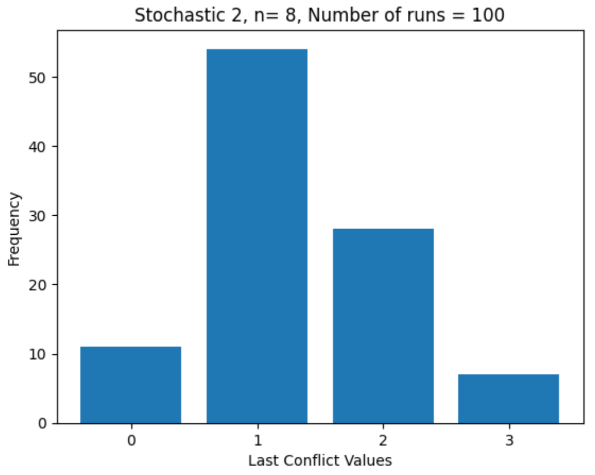
```
In [273…
def plot(n, k, conflict_list_sahc, conflict_list_stoch1, conflict_list_stoch2):
    values_sahc, counts_sahc = np.unique(conflict_list_sahc, return_counts=True)
    plt.bar(values_sahc, counts_sahc)
    plt.xticks(range(min(conflict_list_sahc),max(conflict_list_sahc)+1))
    #plt.title("Steepest Ascend HC,  n=%i, number of runs: %i " %n %k )
    plt.title('Algorithm: Steepest Ascend HC, n= '+str(n)+', Number of runs = '+str(k))
    plt.xlabel("Last Conflict Values")
    plt.ylabel("Frequency")
    plt.show()


    values_stoch1, counts_stoch1 = np.unique(conflict_list_stoch1, return_counts=True)
    plt.bar(values_stoch1, counts_stoch1)
    plt.xticks(range(min(conflict_list_stoch1),max(conflict_list_stoch1)+1))
    plt.title('Stochastic 1, n= '+str(n)+', Number of runs = '+str(k))
    plt.xlabel("Last Conflict Values")
    plt.ylabel("Frequency")
    plt.show()

    values_stoch1, counts_stoch2 = np.unique(conflict_list_stoch2, return_counts=True)
    plt.bar(values_stoch1, counts_stoch2)
    plt.xticks(range(min(conflict_list_stoch2),max(conflict_list_stoch2)+1))
    plt.title('Stochastic 2, n= '+str(n)+', Number of runs = '+str(k))
    plt.xlabel("Last Conflict Values")
    plt.ylabel("Frequency")
    plt.show()

    average_sahc=sum(conflict_list_sahc)/len(conflict_list_sahc)
    average_stoch1=sum(conflict_list_stoch1)/len(conflict_list_stoch1)
    average_stoch2=sum(conflict_list_stoch2)/len(conflict_list_stoch2)
    x_array=["SAHC", "STOCH 1", "STOCH 2"]
    y_array=[average_sahc,average_stoch1,average_stoch2]
    plt.bar(x_array,y_array )
    plt.title('Average last conflict values Board Size:'+str(n)+', Number of runs = '+str(k))
    plt.xlabel("Algorithms")
    plt.ylabel("Average Last Conflict Values")
    plt.show()
```

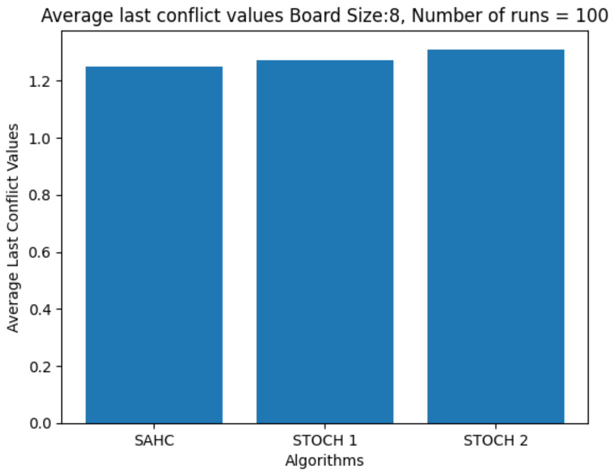
8 queens, 100 runs

```
In [275…
#8 Queens, 100 runs.
k=100
n=8
conflict_list_sahc, conflict_list_stoch1, conflict_list_stoch2=runsk(n,k)
plot(n, k, conflict_list_sahc, conflict_list_stoch1, conflict_list_stoch2)
```


Algorithm: Steepest Ascend HC, n= 8, Number of runs = 100
Frequency
Last Conflict Values
0
1
2
3
0
10
20
30
40
50

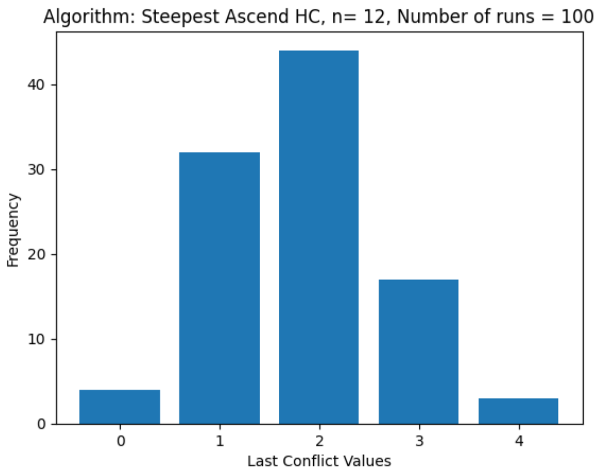

Stochastic 1, n= 8, Number of runs = 100
Frequency
Last Conflict Values
0
1
2
3
0
10
20
30
40
50

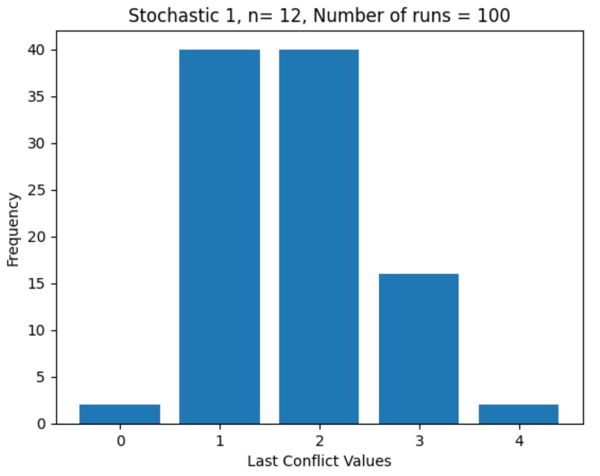

Stochastic 2, n= 8, Number of runs = 100
Frequency
Last Conflict Values
0
1
2
3
0
10
20
30
40
50

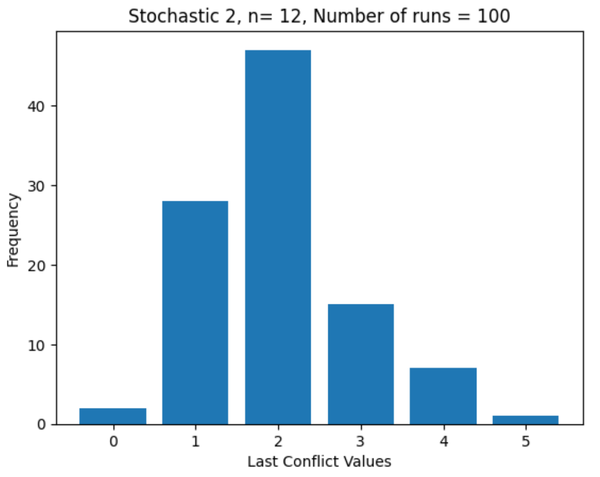

Average last conflict values Board Size:8, Number of runs = 100
Average Last Conflict Values
Algorithms
SAHC
STOCH 1
STOCH 2
0.0
0.2
0.4
0.6
0.8
1.0
1.2

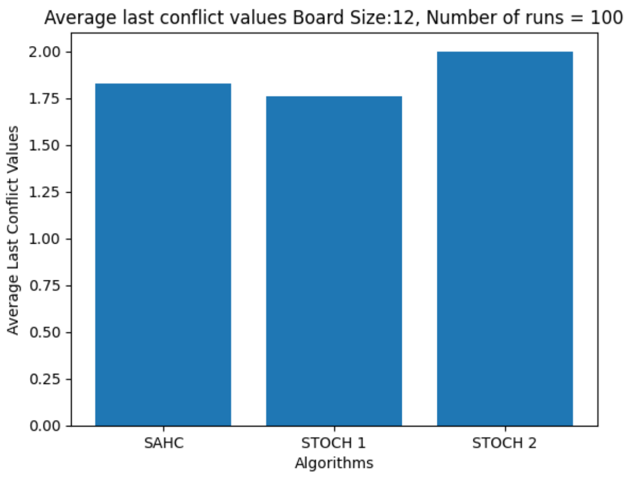
12 queens, 100 runs

```
In [276…
k=100 #number of iterations
n=12
conflict_list_sahc, conflict_list_stoch1, conflict_list_stoch2=runsk(n,k)
plot(n, k, conflict_list_sahc, conflict_list_stoch1, conflict_list_stoch2)
```



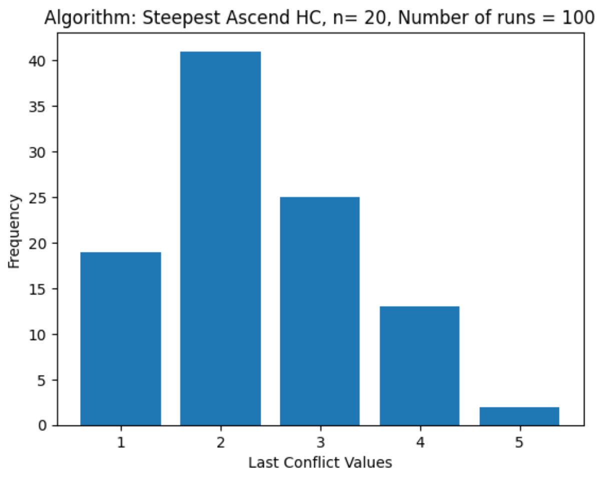


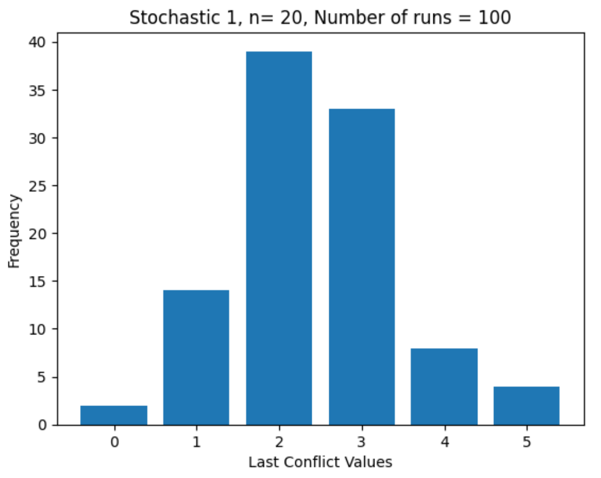


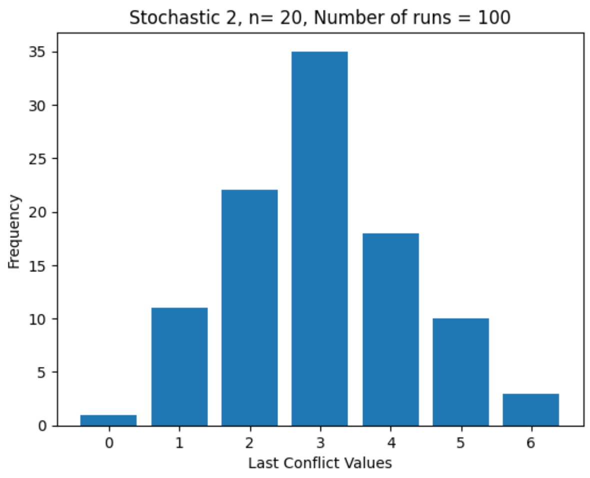


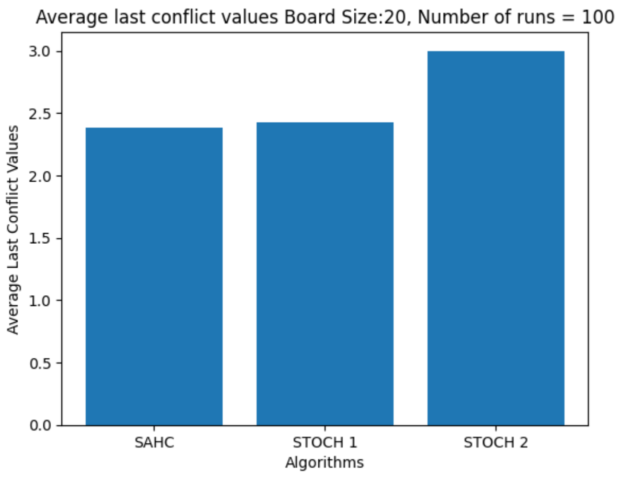
20 queens, 100 runs

```
In [278…
k=100 #number of iterations
n=20
conflict_list_sahc, conflict_list_stoch1, conflict_list_stoch2=runsk(n,k)
plot(n, k, conflict_list_sahc, conflict_list_stoch1, conflict_list_stoch2)
```



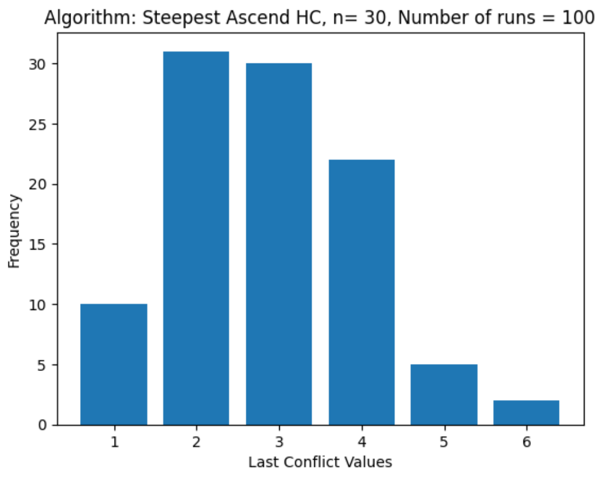


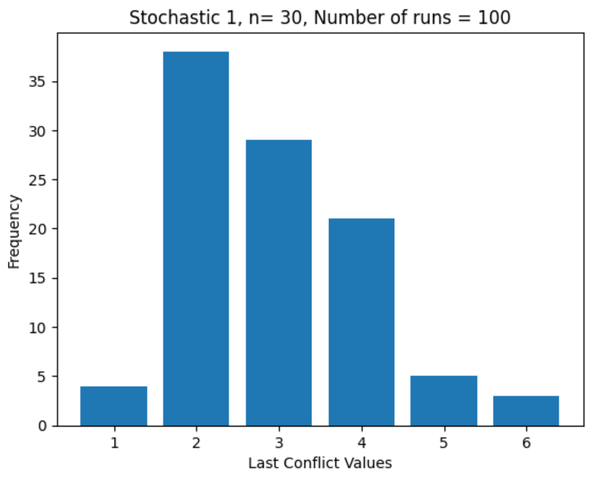


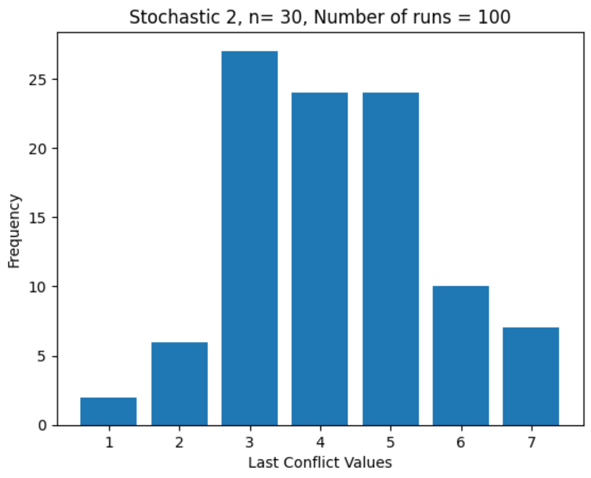


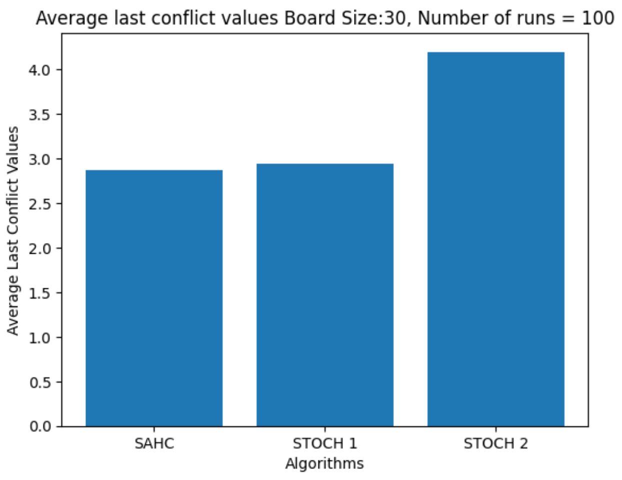
30 queens, 100 runs

```
In [279…
k=100 #number of iterations
n=30
conflict_list_sahc, conflict_list_stoch1, conflict_list_stoch2=runsk(n,k)
plot(n, k, conflict_list_sahc, conflict_list_stoch1, conflict_list_stoch2)
```









In 100 random runs for 30 queens, none of the algorithms found the solution.

With the increasing size of the board, Stochastic 2 algorithm perform worse than the other two. My Stochastic 2 algorithm looks at the last 15* n conflict of selected action. If they are same, it decides that it is in the local optima, and it breaks the loop. It is possible that rather than looking number of last elements proportional to n, I could have looked last n^2 elements. This would increase the success rate a little bit more. We expect Stochastic 1 and Stochastic 2 identical. The difference might be coming from not rightfully detecting the local optima for Stoch 2 algorithm.

After board size 20, it is rare that the algorithms find the global optimum.

Time Comparison

In [280…

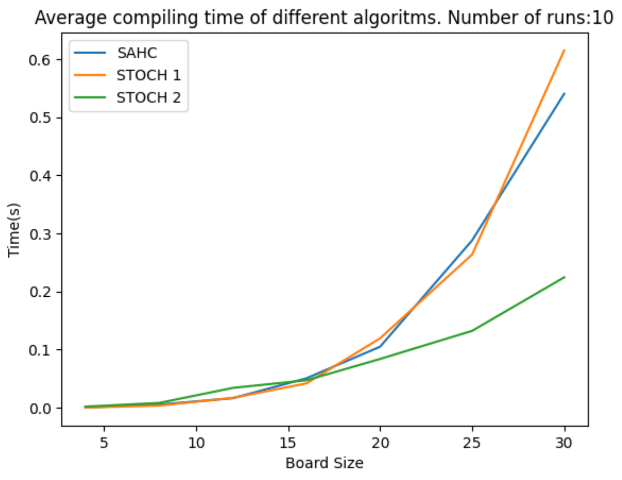
```python
def time_performance(k, VerboseSimulatedAnnealing, n_array):
    conflict_list_sahc=[]
    conflict_list_stoch1=[]
    conflict_list_stoch2=[]
    if VerboseSimulatedAnnealing==True:
        conflict_list_sa=[]




    time_list_sahc=[]
    time_list_stoch1=[]
    time_list_stoch2=[]
    if VerboseSimulatedAnnealing==True:
        time_list_sa=[]



    for n_iter in n_array:
        start_sahc = time.time()
        for i in range (k):
            conflict_list_sahc.append(sahc(n_iter, False))
        end_sahc = time.time()
        time_list_sahc.append(end_sahc-start_sahc)

    for n_iters1 in n_array:
        start_stoch1 = time.time()
        for i in range (k):
            conflict_list_stoch1.append(stoch1(n_iters1, False))
        end_stoch1 = time.time()
        time_list_stoch1.append(end_stoch1-start_stoch1)

    for n_iters2 in n_array:
        start_stoch2 = time.time()
        for i in range (k):
            conflict_list_stoch2.append(stoch2(n_iters2, False))
        end_stoch2 = time.time()
        time_list_stoch2.append(end_stoch2-start_stoch2)

    if VerboseSimulatedAnnealing==True:
        for n_iters3 in n_array:
            start_sa = time.time()
            for i in range (k):
                conflict_list_sa.append(stoch_simulated_annealing(n_iters3, False))
            end_sa = time.time()
            time_list_sa.append(end_sa-start_sa)

    average_time_sahc = []
    for x in time_list_sahc:
        average_time_sahc.append(x/k)

    average_time_stoch1=[]
    for y in time_list_stoch1:
        average_time_stoch1.append(y/k)

    average_time_stoch2=[]
    for z in time_list_stoch2:
        average_time_stoch2.append(z/k)

    if VerboseSimulatedAnnealing==True:
        average_time_sa=[]
        for r in time_list_sa:
            average_time_sa.append(r/k)

    plt.plot(n_array, average_time_sahc, label="SAHC")
    plt.plot(n_array, average_time_stoch1, label="STOCH 1")
    plt.plot(n_array, average_time_stoch2, label="STOCH 2")

    if VerboseSimulatedAnnealing==True:
        plt.plot(n_array, average_time_sa, label="Simulated Annealing")


    #plt.title('Average Time required over' +str(k) ' runs for different size of boards')
    plt.title('Average compiling time of different algoritms. Number of runs:'+str(k))

    plt.xlabel("Board Size")
    plt.ylabel("Time(s)")
    #plt.title("Board Size")
    plt.legend(loc="best")


    return average_time_sahc, average_time_stoch1, average_time_stoch2
```

In [283…

```
time_performance(10, False, n_array=[4, 8, 12, 16, 20, 25, 30])
```

Out[283]:

```
([0.0005995512008666992,
  0.005496954917907715,
  0.016391801834106445,
  0.05067100524902344,
  0.10503914356231689,
  0.2880323171615601,
  0.5401897668838501],
 [0.0004002332687377930,
  0.0036975860595703123,
  0.016990137100219727,
  0.0418757438659668,
  0.1194295883178711,
  0.26364798545837403,
  0.6146452903747559],
 [0.0019988298416137697,
  0.008296608924865723,
  0.034178853034973145,
  0.04727418422698974,
  0.08395154476165771,
  0.132323694229126,
  0.22437045574188233])
```



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.

1. Implementation

In [303…

```python
# Implementation
def stoch_simulated_annealing(n,Verbose):
    board = random_board(n)
    if Verbose==True:
        show_board(board)
    c_flag=[]
    init_temperature=n**5
    iters=0
    temp=[]
    while True:
        column_array = [k for k in range(0,n)]
        selected_column=random.choice(column_array)
        # Select the column randomly. Then we will
        # move the queen in that column
        if Verbose==True:
            print("Randomly selected column:",selected_column)
        conflict_original=conflicts(board)
        queen_position=board[selected_column]
        action_array=[k for k in range(0,n)]
        #remove the current row of queen from action array
        action_array.remove(queen_position)
        if Verbose==True:
            print("Possible actions after removing current queen position",action_array)
        #Randomly select the row which the queen will go
        random_action=random.choice(action_array)

        if Verbose==True:
            print("Randomly selected action to test is the queen in column number",
              selected_column, "moves to row number", random_action)

        potential_board=np.copy(board)
        potential_board[selected_column]=random_action
        #Check the conflicts of the board comes from random action we selected
        conflict_potential=conflicts(potential_board)

        if Verbose==True:
            print("Potential Action's number of conflict:", conflict_potential)
        #difference between potential board conflict and original board conflict
        diff= conflict_potential-conflict_original
        #define temperature for simulated annealing
        temperature=init_temperature/(iters+1)

        if Verbose==True:
            print("Current temp:", temperature)
        #define the Metropolis criterion for simulated annealing
        Metropolis = math.exp(-(diff)/temperature)

        if Verbose==True:
            print("Metropolis criterion:",Metropolis)
            print("difference",diff)
        #select a probability between 0 and 1
        probability = random.random()

        if Verbose==True:
            print("Probability:",probability)
        # if potential conflict is smaller than the original
        # or selected probability is higher than Metropolis
        #criterion, accept the action. If not, use the original
        #board
        if diff<0 or probability<Metropolis:
            board=np.copy(potential_board)
            if Verbose==True:
                print("Action selected:")
            if Verbose==True:
                show_board(board)

        else:
            board=board
            if Verbose==True:
                print("Action is not selected")

        iters=iters+1

        if Verbose==True:
            print("number of iteration:", iters)

        c_flag.append(conflicts(board))
        last_2n = c_flag[-(3*n):]
        if Verbose==True:
            print("last 3n conflicts", last_2n)
            #check if last 3*n conflict are same:
            # It stops the algorithm and says
            #it is in local optimum (never hapens for this algorithm)

        result = all(element == last_2n[0] for element in last_2n)
        if result==True and len(last_2n)>10*n:
            if Verbose==True:
                print("We are in local minimum. Stop the algorithm")
            break

        if conflicts(board)==0:
            break
    if Verbose==True:
        print("Final Board:", board)

    return conflicts(board)
```

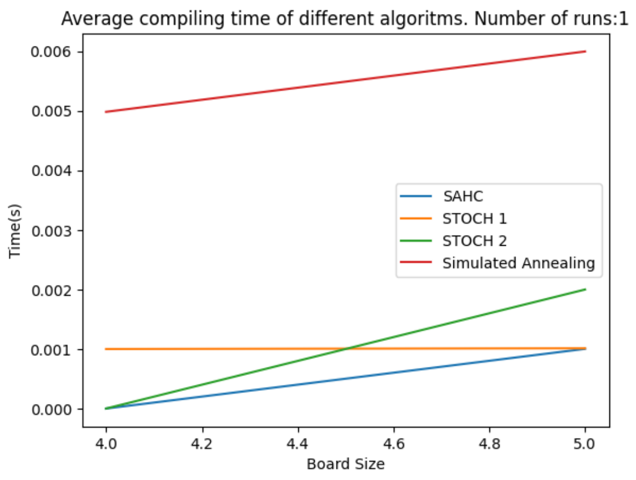
In [304…
```
stoch_simulated_annealing(8, False)
```

Out[304]:
```
0
```

2. Time and Objective Function Performance

In [286…
```
time_performance(1, True, n_array=[4, 5])
```

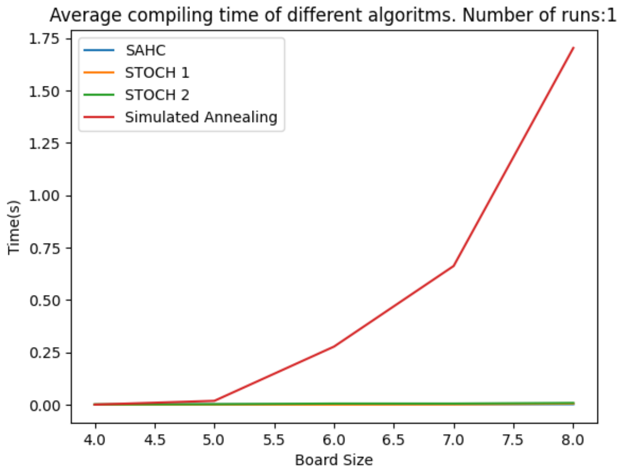
Out[286]:
```
([0.0, 0.0010020732879638672],
 [0.000999450683593775, 0.0010118484497070312],
 [0.0, 0.0019986629486083984])
```



I gave only n=5 and n=5 since simulated annealing takes a lot more time that first 3 algorithms

In [287…
```
time_performance(1, True, n_array=[4, 5, 6, 7, 8])
```

Out[287]:
```
([0.0009996891021728516,
  0.0020160675048828125,
  0.0019822120666503906,
  0.0029973983764648438,
  0.003015756607055664],
 [0.0009984970092773438,
  0.0009989738464355469,
  0.001003265380859375,
  0.0019795894622802734,
  0.006016731262207031],
 [0.001992940902709961,
  0.0029833316802978516,
  0.006012678146362305,
  0.005980253219604492,
  0.008995532989501953])
```



As seen from previous graphs using simulated annealing is way more costly than first three algorithms in terms of computation.

Let's do 10 iterations for every different number of board sizes. I will do it up to n=12 since it takes a lot of time to run simulated annealing algorithm.

In [301…
```
def runsksa(n,k):
    conflict_list_sa=[]
    for i in range (10):
        conflict_list_sa.append(stoch_simulated_annealing(n, False))
    print("Simulated annealing returns # of conflicts for every iteration:", conflict_list_sa ,
          "for board size:", n)
```

In [302…
```
runsksa(4,5)
runsksa(6,5)
runsksa(8,5)
runsksa(10,5)
runsksa(12,5)
```

```
Simulated annealing returns # of conflicts for every iteration: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0] for board size: 4
Simulated annealing returns # of conflicts for every iteration: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0] for board size: 6
Simulated annealing returns # of conflicts for every iteration: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0] for board size: 8
Simulated annealing returns # of conflicts for every iteration: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0] for board size: 10
Simulated annealing returns # of conflicts for every iteration: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0] for board size: 12
```

Simulated Annealing always returns 0 number of conflicts (finds global optimum). However, it is costly to use this algorithm in terms of computation.

3. Annealing Schedule

Temperature=Initial Temperature/(iteration+1)

Figure 1. Multiplicative cooling curves: (A) Exponential, (B) logarithmical, (C) linear, (D) quadratic

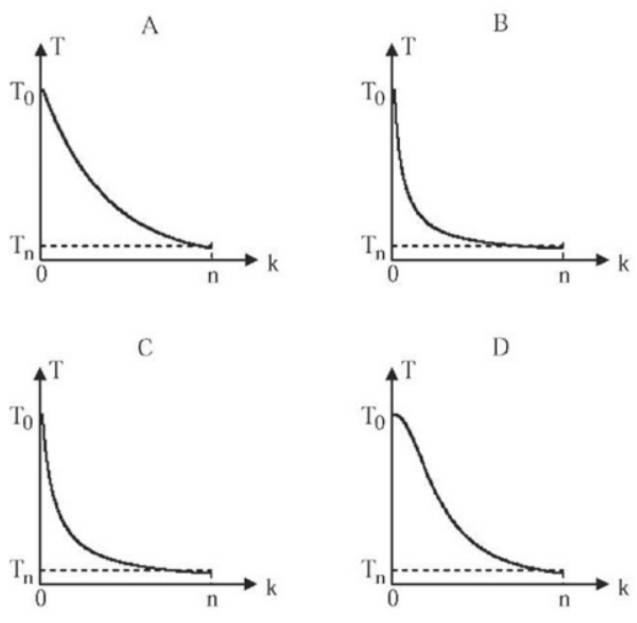
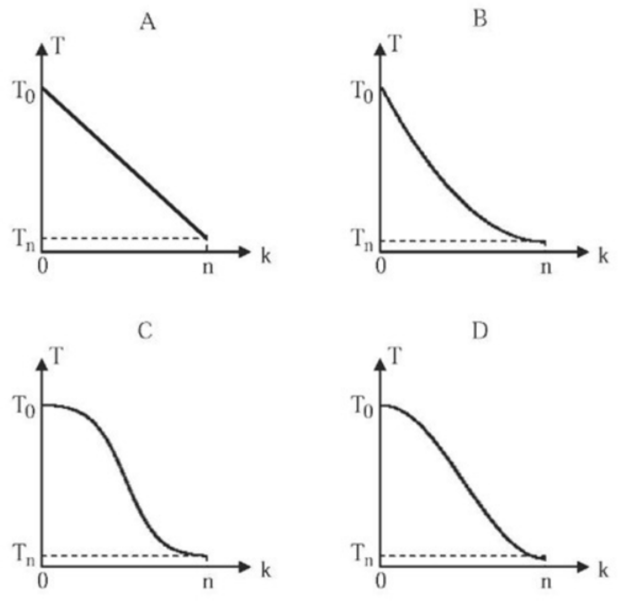


Figure 2: Additive cooling curves: (A) linear, (B) quadratic, (C) exponential, (D) trigonometric.



Reference: http://what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/

Temperature schedule is selected as above, Linear multiplicative cooling, A in figure above.

I used linear cooling since works excellent in terms of escaping local optima. I implemented it first and did not change it. We could use quadratic and exponential tails since they cancel out faster, returns the best standard error values (standard error of the number of iterations). It can be another task to compare these different cooling algorithms.

For the initial Temperature, I selected n^5. This value is selected emprically. Smaller values can return math error in python since number of digits increases dramatically when the temperature gets smaller.