Printing the Chess Board

This procedure is done with a function called **`print_board()`**. Using the sample configuration in Figure 1, the solution can be represented with a list as follows: **[0, 1, 2, 1, 6, 2, 7, 1]**; array indexing format is used, this means that the first value in the list: '0' can be interpreted as placing the queen in column 1 (index 0) in row 1 while the second value '1' is interpreted as the queen in column 2 (index 1) is placed in row 2, and so on. You should notice that we are looking at the chess board from a bottom-up view, but when we try to print on the console, we have to start from the top. This means the first view on the console is equivalent to the last view from our point of view. Hence the reason for the first for-loop starting from row 7 (last row) and iterating backwards to the first row (row 0), i.e. the following expression "`for i in range(7, -1, -1):`". The second for-loop expression "`for j in range(8):`" controls the column-order iteration. Therefore, checking if ***i*** is equal to ***val*** (`chess_board[j]`) tries to determine what will be printed in a particular column ('Q' or '_'). This means that for every row, we go through all the columns to determine the pattern that will be printed for that line (row). For example, looking at the last row (aka the first row, based on the console output), we have the Queen on the 7th column (column 6). This means that when **i = 7** (meaning we are currently on the last row; remember, we print from the back), the output on the console will be '_' until **j** becomes 6, then 'Q' is printed on the screen.

```
_ _ _ _ _ _ Q _
_ _ _ _ Q _ _ _
_ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _
_ _ Q _ _ Q _ _
_ Q _ Q _ _ _ Q
Q _ _ _ _ _ _ _
```

Figure 1: Sample initial state of the 8-queens problem

```
def print_board(chess_board):
    for i in range(7, -1, -1):
        for j in range(8):
            val = chess_board[j]
            if i == val:
                print('Q', end=" ")
            else:
                print('_', end=" ")
        print()
```

This makes sense because the queen at column 7 (index 6 of our list) is placed in row 7 (last row); the value at index 6 of the list is 7. Remember when i = 7 and j becomes 6, the expression i == val evaluates to true since **`i = 7`** is the same as the value of **`chess_board[6] = 7`** at that point in the loop. You can use the same

explanation to figure out the printing/placing of other queens on the chess board based on the content of the list.

The main logic in the main function

The while loop signals the entry point of the main Simulated Annealing implementation. Being conscious of the running time as well as possibility of the algorithm getting stuck, a time-based stopping criterion is used. Therefore the expression "`while not timeElapsed:`" implements the following line "`for t = 1 to ∞ do`" in the textbook. The **cur_sol** variable is a list that stores the current state/configuration of the problem being solved (i.e. the 8-queens problem). The code in lines 46-48 in the source code randomly initialises the initial state. Lines 60-64 of the source code implements the expression "`next ← a randomly selected successor of current`" in the textbook. The while loop construct in lines 62-63 was added to ensure a useless move is not made, i.e. randomly picking a row position that a queen currently occupies. It is therefore designed to loop as long as the random row selected is equal to the current one the queen in question occupies. The last three lines of the algorithm presented in the textbook are handled by the simulated annealing acceptance procedure (this version is from a paper, see the reference below) implemented from line 108 through to 127 of the source code. We stop running the algorithm if (1) the condition "`if eval_cur == 0:`" is fulfilled **OR** (2) The algorithm run times out after 2 minutes without finding a solution. The first condition checks if the evaluation value of the current solution is 0 (that is, no queen attacks another).

```
def main():
    global cur_sol, next_sol, start_time, elapsedTime, timeElapsed
    iter = 0
    cur_sol = []
    timeElapsed = False
    print("Simulated Annealing implementation for solving the 8-queens problem")

    for i in range(0, 8):
        position = random.randint(0, 7)
        cur_sol.append(position)

    # print the initial configuration
    print('Initial configuration printed below...')
    print_board(cur_sol)
    print()
```

```
    start_time = timer()

    while not timeElapsed:
        next_sol = cur_sol.copy()
        # make a random move in the next solution
        position = random.randint(0, 7)
        val = random.randint(0, 7)
        while val == next_sol[position]:
            val = random.randint(0, 7)
        next_sol[position] = val  # make the move
        eval_cur = compute_value(cur_sol)
        eval_next = compute_value(next_sol)
        if sa_accept(eval_next, eval_cur):
            cur_sol = next_sol.copy()
            eval_cur = eval_next
        iter += 1
        if eval_cur == 0:
            print('The Simulated Annealing algorithm has found a solution to the
8-queens problem')
            break
        cur_time = timer()
        elapsedTime = cur_time - start_time
        if elapsedTime >= execTime:
            timeElapsed = True

    print('Done after {0} iterations. Solution printed below'.format(iter))
    if eval_cur > 0:
        print('The current solution is not optimal, algorithm had to be aborted')
    print_board(cur_sol)
```

Computing the number of attacks on the diagonal

The procedure that handles this is **same_diagonal()**. It is a little bit technical but a simple matrix visualization and a little bit of arithmetic help to deduce the diagonal attacks. Let's start by observing the 8 × 8 matrix below.

7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7

Can you see a pattern on the highlighted diagonals using the position 2,0 as a reference point? $2-0=3-1=4-2=\cdots=7-5$. The expression "`abs_diff_a == abs_diff_b`" checks for that. This is tied to the first two lines of the function, which compute the difference between the row index and the column index of the positions of the two queens (**i** and **j**) to compare. The second pattern $2+0=1+1=0+2$ also flags diagonal attack. Once the summation of the row and column indices where a queen is currently placed is same as that of another queen, then they are in the same diagonal. Therefore, once the first pattern or the second pattern is detected for a pair **(i, j)**, then the function returns **True**. You can confirm if this idea works for other diagonals apart from the one used as an example.

```
def same_diagonal(chess_board, i, j):
    abs_diff_a = i - chess_board[i]
    abs_diff_b = j - chess_board[j]
    sum_a = i + chess_board[i]
    sum_b = j + chess_board[j]
    if abs_diff_a == abs_diff_b or sum_a == sum_b:
        return True
    else:
        return False
```

Computing the heuristic function value of a state

The function `compute_value()` computes the heuristic function value of a board (state). This function relies on the **`same_diagonal()`** function and also checks if two queens are on the same row (to detect row attack). It goes through a two-level nested for loop for row and column order iterations. The second

level for-loop starts from **i + 1** since we are not trying to (for example) check for 2,1 but 1,2. Why? Since 1,2 and 2,1 are the same combinations. In summary, any row attack or diagonal attack increments the value of **h** for every pair considered in the loop.

```
def compute_value(chess_board):
    h_value = 0
    for i in range(0, 7):
        for j in range(i + 1, 8):
            if chess_board[i] == chess_board[j]:  # same row attack
                h_value += 1
            elif same_diagonal(chess_board, i, j):  # same diagonal attack
                h_value += 1
    return h_value
```

Simulated Annealing Acceptance Procedure Credit

Adriaensen, S., Brys, T., & Nowé, A. (2014, July). Designing reusable metaheuristic methods: A semi-automated approach. In *2014 IEEE congress on evolutionary computation (CEC)* (pp. 2969-2976). IEEE.