Started on	Sunday, 13 September 2020, 6:33 PM
State	Finished
Completed on	Friday, 18 September 2020, 5:06 PM
Time taken	4 days 22 hours
Marks	4.85/5.00
Grade	**97.00** out of 100.00

Information

Optimisation

Optimisation has a wide range of applications. One of them is solving CSPs. In the next two questions you are required to write two functions: one objective (cost) function, and one function to find the neighbours of a given point in the search space. In later questions, you will implement greedy optimisation procedures.

n-queens puzzle

n queens puzzle is the mathematical generalisation of the well-known eight queens puzzle. Using non-chess terms, the objective is to place *n* objects on an $n \times n$ board (grid), such that no two objects are in the same row, column or diagonal. *n* is alway a positive integer.

Traditionally, the rows and columns of the board are numbered from 1 to n where the position (1,1) can be arbitrarily chosen to be any of the squares at the four corners of the board. Using this numbering scheme, for example, an assignment in which the positions (5,6) and (5,8) are occupied or an assignment in which the positions (3,7) and (7,7) are occupied, are not solutions because in these assignments, two objects share the same row or column. Also, for example, having an object at (2, 4) and another object at (5, 7) is illegal because the two objects share the same diagonal.

Finding solutions for an *n* queens puzzle can be computationally expensive but by using an appropriate representation and search method, the required computational effort can be reduced. In the following, you will solve this puzzle (for large values of *n*) using CSP and local search techniques.

Representation

There are many ways of representing a total assignment in this problem. We briefly review three possible options.

- **Option 1 (not so good)**: a very simplistic representation for this problem would be to have an array of length n^2 where each element can be either 0 or 1 indicating the absence of presence of an object (queen). A large proportion of possible assignments in this representation are not solutions.
- **Option 2 (good)**: Considering that in each column (or row) there can be at most one object (queen), a possible representation is to have an array (or tuple) of length *n* where each element can be a number from 1 to *n*. The number at index *i* of the array indicates the row number of the object at column *i* (or the column number of the object at row *i*). For example, if the second element of the array is 5, it indicates that the queen in the second column will be on row 5. This is the representation that was used in the lecture notes.
- **Option 3 (better)**: Option 2 can be further improved by not allowing duplicate numbers in the array (or tuple). A duplicate number means the presence of two queens on the same row which is illegal. Therefore we can avoid these assignments by only considering possible permutations of numbers from 1 to n. In this representation we do not need to check for row or column conflicts as they cannot happen. Thus we only need to check whether there is any diagonal conflict for an assignment. We use this representation in the following questions.

Question **1**

Correct

Mark 1.00 out of 1.00

Write a function `n_queens_neighbours(state)` that takes a state (total assignment) for an *n*-queen problem and returns a sorted list of states that are the neighbours of the current assignment. A neighbour is obtained by swapping the position of two numbers in the given permutation.

Like before, the state will be given in the form of a sequence (more specifically, a tuple). The state is a permutation of numbers from 1 to *n* (inclusive). The value of *n* must be inferred from the given state.

Because of the choice of representation (the permutation of numbers from 1 to *n*) the concept of neighbourhood in this question is different from that in the examples given in the lecture notes. The representation we use here does not allow repeated numbers in a sequence, therefore we define a neighbouring assignment to be one that can be obtained by swapping the position of two numbers in the current assignment.

Note the spelling of `neighbours`. Also note that the neighbours of an assignment do not include the assignment itself.

For example:

Test	Result
from student_answer import n_queens_neighbours as neighbours print(neighbours((1, 2)))	[(2, 1)]
from student_answer import n_queens_neighbours as neighbours print(neighbours((1, 3, 2)))	[(1, 2, 3), (2, 3, 1), (3, 1, 2)]
from student_answer import n_queens_neighbours as neighbours print(neighbours((1, 2, 3)))	[(1, 3, 2), (2, 1, 3), (3, 2, 1)]
from student_answer import n_queens_neighbours as neighbours print(neighbours((1,)))	[]
from student_answer import n_queens_neighbours as neighbours for neighbour in neighbours((1, 2, 3, 4, 5, 6, 7, 8)): print(neighbour)	(1, 2, 3, 4, 5, 6, 8, 7) (1, 2, 3, 4, 5, 7, 6, 8) (1, 2, 3, 4, 5, 8, 7, 6) (1, 2, 3, 4, 6, 5, 7, 8) (1, 2, 3, 4, 7, 6, 5, 8) (1, 2, 3, 4, 8, 6, 7, 5) (1, 2, 3, 5, 4, 6, 7, 8) (1, 2, 3, 6, 5, 4, 7, 8) (1, 2, 3, 7, 5, 6, 4, 8) (1, 2, 3, 8, 5, 6, 7, 4) (1, 2, 4, 3, 5, 6, 7, 8) (1, 2, 5, 4, 3, 6, 7, 8) (1, 2, 6, 4, 5, 3, 7, 8) (1, 2, 7, 4, 5, 6, 3, 8) (1, 2, 8, 4, 5, 6, 7, 3) (1, 3, 2, 4, 5, 6, 7, 8) (1, 4, 3, 2, 5, 6, 7, 8) (1, 5, 3, 4, 2, 6, 7, 8) (1, 6, 3, 4, 5, 2, 7, 8) (1, 7, 3, 4, 5, 6, 2, 8) (1, 8, 3, 4, 5, 6, 7, 2) (2, 1, 3, 4, 5, 6, 7, 8) (3, 2, 1, 4, 5, 6, 7, 8) (4, 2, 3, 1, 5, 6, 7, 8) (5, 2, 3, 4, 1, 6, 7, 8) (6, 2, 3, 4, 5, 1, 7, 8) (7, 2, 3, 4, 5, 6, 1, 8) (8, 2, 3, 4, 5, 6, 7, 1)
from student_answer import n_queens_neighbours as neighbours for neighbour in neighbours((2, 3, 1, 4)): print(neighbour)	(1, 3, 2, 4) (2, 1, 3, 4) (2, 3, 4, 1) (2, 4, 1, 3) (3, 2, 1, 4) (4, 3, 1, 2)

Answer: (penalty regime: 0, 15, ... %)

```
def n_queens_neighbours(state):
    possible_state = []
    for i in range(len(state)):
        #print(i)
        for j in range(len(state)):
            if i < j:
                original_state = list(state)
                original_state[i], original_state[j] = original_state[j], original_state[i]
                possible_state.append(tuple(original_state))
```

```
        else:
            pass
    return sorted(possible_state)
```

	Test	Expected	Got	
✔	from student_answer import n_queens_neighbours as neighbours print(neighbours((1, 2)))	[(2, 1)]	[(2, 1)]	✔
✔	from student_answer import n_queens_neighbours as neighbours print(neighbours((1, 3, 2)))	[(1, 2, 3), (2, 3, 1), (3, 1, 2)]	[(1, 2, 3), (2, 3, 1), (3, 1, 2)]	✔
✔	from student_answer import n_queens_neighbours as neighbours print(neighbours((1, 2, 3)))	[(1, 3, 2), (2, 1, 3), (3, 2, 1)]	[(1, 3, 2), (2, 1, 3), (3, 2, 1)]	✔
✔	from student_answer import n_queens_neighbours as neighbours print(neighbours((1,)))	[]	[]	✔

	Test	Expected	Got	

✔	`from student_answer import n_queens_neighbours as neighbours`
	
	`for neighbour in neighbours((1, 2, 3, 4, 5, 6, 7, 8)):`
	`    print(neighbour)`
		(1, 2, 3, 4, 5, 6, 8, 7)
		(1, 2, 3, 4, 5, 7, 6, 8)
		(1, 2, 3, 4, 5, 8, 7, 6)
		(1, 2, 3, 4, 6, 5, 7, 8)
		(1, 2, 3, 4, 7, 6, 5, 8)
		(1, 2, 3, 4, 8, 6, 7, 5)
		(1, 2, 3, 5, 4, 6, 7, 8)
		(1, 2, 3, 6, 5, 4, 7, 8)
		(1, 2, 3, 7, 5, 6, 4, 8)
		(1, 2, 3, 8, 5, 6, 7, 4)
		(1, 2, 4, 3, 5, 6, 7, 8)
		(1, 2, 5, 4, 3, 6, 7, 8)
		(1, 2, 6, 4, 5, 3, 7, 8)
		(1, 2, 7, 4, 5, 6, 3, 8)
		(1, 2, 8, 4, 5, 6, 7, 3)
		(1, 3, 2, 4, 5, 6, 7, 8)
		(1, 4, 3, 2, 5, 6, 7, 8)
		(1, 5, 3, 4, 2, 6, 7, 8)
		(1, 6, 3, 4, 5, 2, 7, 8)
		(1, 7, 3, 4, 5, 6, 2, 8)
		(1, 8, 3, 4, 5, 6, 7, 2)
		(2, 1, 3, 4, 5, 6, 7, 8)
		(3, 2, 1, 4, 5, 6, 7, 8)
		(4, 2, 3, 1, 5, 6, 7, 8)
		(5, 2, 3, 4, 1, 6, 7, 8)
		(6, 2, 3, 4, 5, 1, 7, 8)
		(7, 2, 3, 4, 5, 6, 1, 8)
		(8, 2, 3, 4, 5, 6, 7, 1)
			(1, 2, 3, 4, 5, 6, 8, 7)
			(1, 2, 3, 4, 5, 7, 6, 8)
			(1, 2, 3, 4, 5, 8, 7, 6)
			(1, 2, 3, 4, 6, 5, 7, 8)
			(1, 2, 3, 4, 7, 6, 5, 8)
			(1, 2, 3, 4, 8, 6, 7, 5)
			(1, 2, 3, 5, 4, 6, 7, 8)
			(1, 2, 3, 6, 5, 4, 7, 8)
			(1, 2, 3, 7, 5, 6, 4, 8)
			(1, 2, 3, 8, 5, 6, 7, 4)
			(1, 2, 4, 3, 5, 6, 7, 8)
			(1, 2, 5, 4, 3, 6, 7, 8)
			(1, 2, 6, 4, 5, 3, 7, 8)
			(1, 2, 7, 4, 5, 6, 3, 8)
			(1, 2, 8, 4, 5, 6, 7, 3)
			(1, 3, 2, 4, 5, 6, 7, 8)
			(1, 4, 3, 2, 5, 6, 7, 8)
			(1, 5, 3, 4, 2, 6, 7, 8)
			(1, 6, 3, 4, 5, 2, 7, 8)
			(1, 7, 3, 4, 5, 6, 2, 8)
			(1, 8, 3, 4, 5, 6, 7, 2)
			(2, 1, 3, 4, 5, 6, 7, 8)
			(3, 2, 1, 4, 5, 6, 7, 8)
			(4, 2, 3, 1, 5, 6, 7, 8)
			(5, 2, 3, 4, 1, 6, 7, 8)
			(6, 2, 3, 4, 5, 1, 7, 8)
			(7, 2, 3, 4, 5, 6, 1, 8)
			(8, 2, 3, 4, 5, 6, 7, 1)
				✔

✔	`from student_answer import n_queens_neighbours as neighbours`
	
	`for neighbour in neighbours((2, 3, 1, 4)):`
	`    print(neighbour)`
		(1, 3, 2, 4)
		(2, 1, 3, 4)
		(2, 3, 4, 1)
		(2, 4, 1, 3)
		(3, 2, 1, 4)
		(4, 3, 1, 2)
			(1, 3, 2, 4)
			(2, 1, 3, 4)
			(2, 3, 4, 1)
			(2, 4, 1, 3)
			(3, 2, 1, 4)
			(4, 3, 1, 2)
				✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **2**

Correct

Mark 1.00 out of 1.00

Write a function `n_queens_cost(state)` that takes a state (a total assignment) for an *n*-queen problem and returns the number conflicts for that state. We define the number of conflicts to be the number of unordered pairs of queens (objects) that threaten (attack) each other. The state will be given in the form of a sequence (tuple more specifically). The state is a permutation of numbers from 1 to *n* (inclusive). The value of *n* must be inferred from the given state.

Hint: diagonals have a slope of 1 or -1. You want to see if `abs(dx)==abs(dy)`.

Challenge: you can write this in one statement (one return statement in two lines). See `itertools.combinations`.

For example:

Test	Result

`from student_answer import n_queens_cost as cost`
 	
`print(cost((1, 2)))`
	1

`from student_answer import n_queens_cost as cost`
 	
`print(cost((1, 3, 2)))`
	1

`from student_answer import n_queens_cost as cost`
 	
`print(cost((1, 2, 3)))`
	3

`from student_answer import n_queens_cost as cost`
 	
`print(cost((1,)))`
	0

`from student_answer import n_queens_cost as cost`
 	
`print(cost((1, 2, 3, 4, 5, 6, 7, 8)))`
	28

`from student_answer import n_queens_cost as cost`
 	
`print(cost((2, 3, 1, 4)))`
	1

Answer: (penalty regime: 0, 15, ... %)

```
def n_queens_cost(state):
    possible_state = []
    count = 0
    for i in range(len(state)):
        #print(i)
        for j in range(len(state)):
            if i < j:
                #print(state[i])
                #print(state[j])
                #checking for diagonal
                #if diff in index is the same as the diff in state index means both are diagonal
                if (abs(i - j) == abs(state[i] - state[j])):
                    count += 1
            else:
                pass
    return count
```

	Test	Expected	Got	

✔	`from student_answer import n_queens_cost as cost`
	
	`print(cost((1, 2)))`
		1	1	✔

✔	`from student_answer import n_queens_cost as cost`
	
	`print(cost((1, 3, 2)))`
		1	1	✔

✔	`from student_answer import n_queens_cost as cost`
	
	`print(cost((1, 2, 3)))`
		3	3	✔

✔	`from student_answer import n_queens_cost as cost`
	
	`print(cost((1,)))`
		0	0	✔

	Test	Expected	Got	

✔	`from student_answer import n_queens_cost as cost`
	
	`print(cost((1, 2, 3, 4, 5, 6, 7, 8)))`
		28	28	✔

✔	`from student_answer import n_queens_cost as cost`
	
	`print(cost((2, 3, 1, 4)))`
		1	1	✔

✔	`from student_answer import n_queens_cost as cost`
	
	`print(cost((2, 4, 1, 3)))`
		0	0	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **3**

Correct

Mark 1.00 out of 1.00

Write a function `greedy_descent(initial_state, neighbours, cost)` that takes an initial state and two functions to compute the neighbours and cost of a state, and then iteratively improves the state until a local minimum (which may be global) is reached. The function must return the list of states it goes through (including the first and last one) in the order they are encountered. The algorithm should move to a new state only if the cost improves. If there is a tie between multiple states, the first one (in order they appear in the sequence returned by neighbours) must be used.

Arguments

- `initial_state`: the state from which the search starts
- `neighbours`: a function that takes a state and returns a list of neighbours
- `cost` a function that takes a state returns its cost (e.g. number of conflicts).

Notes

- consider using the `min` function in Python.
- you do not need to provide any other function or code.

For example:

<table>
<tr><th>Test</th><th>Result</th></tr>
<tr><td><pre>from student_answer import greedy_descent

def cost(x):
    return x**2

def neighbours(x):
    return [x - 1, x + 1]

for state in greedy_descent(4, neighbours, cost):
    print(state)</pre></td><td><pre>4
3
2
1
0</pre></td></tr>
<tr><td><pre>from student_answer import greedy_descent

def cost(x):
    return x**2

def neighbours(x):
    return [x - 1, x + 1]

for state in greedy_descent(-6.75, neighbours, cost):
    print(state)</pre></td><td><pre>-6.75
-5.75
-4.75
-3.75
-2.75
-1.75
-0.75
0.25</pre></td></tr>
</table>

Answer: (penalty regime: 0, 15, ... %)

```
def greedy_descent(initial_state, neighbours, cost):
        state_path = []
        state_path.append(initial_state)
        current_cost = cost(initial_state)
        current_neighbours = neighbours(initial_state)
        current_state =initial_state
        minimum = False
        while (minimum == False):
                state_to_change = None
                best_state = current_state
                #print(current_state)
                current_cost = cost(current_state)
                current_neighbours = neighbours(current_state)
                for neighbour in current_neighbours:
                        if (cost(neighbour) < current_cost):
                                #print(cost(neighbour))
                                #print(current_cost)
                                current_cost = cost(neighbour)
                                state_to_change = neighbour
                if (state_to_change == None):
                        minimum = True
                else:
                        current_state = state_to_change
                        state_path.append(current_state)
        return state_path
```

	Test	Expected	Got	

	Test	Expected	Got	

✔	`from student_answer import greedy_descent`
	
	`def cost(x):`
	`    return x**2`
	
	`def neighbours(x):`
	`    return [x - 1, x + 1]`
	
	`for state in greedy_descent(4, neighbours, cost):`
	`    print(state)`
		4
		3
		2
		1
		0
			4
			3
			2
			1
			0
				✔

✔	`from student_answer import greedy_descent`
	
	`def cost(x):`
	`    return x**2`
	
	`def neighbours(x):`
	`    return [x - 1, x + 1]`
	
	`for state in greedy_descent(-6.75, neighbours, cost):`
	`    print(state)`
		-6.75
		-5.75
		-4.75
		-3.75
		-2.75
		-1.75
		-0.75
		0.25
			-6.75
			-5.75
			-4.75
			-3.75
			-2.75
			-1.75
			-0.75
			0.25
				✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **4**

Correct

Mark 0.85 out of 1.00

Write a procedure `greedy_descent_with_random_restart(random_state, neighbours, cost)` that takes three functions, one to get a new random state and two to compute the neighbours or cost of a state and then uses `greedy_descent` (you wrote earlier) to find a solution. The first state in the search must be obtained by calling the function `random_state`. The procedure must print each state it goes through (including the first and last one) in the order they are encountered. When the search reaches a local minimum that is not global, the procedure must print RESTART and restart the search by calling `random_state`. Your procedure will be tested only with optimisation versions of CSP problems that have a solution.

Arguments

- `random_state`: a function that takes no argument and return a random state;
- `neighbours`: a function that takes a state and returns a list of neighbours;
- `cost` a function that takes a state returns its cost (e.g. number of conflicts).

Important: You must also provide your implementation of `n_queens_neighbours`, `n_queens_cost`, and `greedy_descent` from previous questions. You do not need to implement `random_state`; it is implemented in test cases.

For example:

Test	Result
from student_answer import n_queens_neighbours as neighbours, n_queens_cost as cost from student_answer import greedy_descent, greedy_descent_with_random_restart import random N = 6 random.seed(0) def random_state(): return tuple(random.sample(range(1,N+1), N)) greedy_descent_with_random_restart(random_state, neighbours, cost)	(4, 6, 1, 2, 3, 5) (4, 6, 1, 2, 5, 3) RESTART (3, 4, 6, 5, 1, 2) (3, 4, 6, 1, 5, 2) RESTART (3, 2, 1, 6, 5, 4) (3, 2, 6, 1, 5, 4) (3, 4, 6, 1, 5, 2) RESTART (3, 1, 5, 6, 2, 4) (3, 1, 6, 5, 2, 4) RESTART (5, 1, 3, 2, 4, 6) (3, 1, 5, 2, 4, 6) (1, 3, 5, 2, 4, 6) RESTART (5, 4, 6, 3, 2, 1) (2, 4, 6, 3, 5, 1) RESTART (5, 1, 6, 3, 2, 4) (3, 1, 6, 5, 2, 4) RESTART (5, 4, 3, 1, 2, 6) (5, 4, 1, 3, 2, 6) (2, 4, 1, 3, 5, 6) RESTART (2, 5, 6, 1, 3, 4) (2, 4, 6, 1, 3, 5)
from student_answer import n_queens_neighbours as neighbours, n_queens_cost as cost from student_answer import greedy_descent, greedy_descent_with_random_restart import random N = 8 random.seed(0) def random_state(): return tuple(random.sample(range(1,N+1), N)) greedy_descent_with_random_restart(random_state, neighbours, cost)	(7, 8, 4, 1, 3, 6, 2, 5) (5, 8, 4, 1, 3, 6, 2, 7)

Answer: (penalty regime: 0, 15, ... %)

```
import itertools
import random

def greedy_descent_with_random_restart(random_state, neighbours, cost):
    #start greedy decent with a random state
    initial_state = random_state()

    state_cost = cost(initial_state)

    result = greedy_descent(initial_state, neighbours, cost )

    #give the answers for each iteration
```

```
    for i in result:
        print(i)

    #if cost of current state is more than 0(termination condition) or
    #if cost of current state > calculated state cost
    #we need to renew local minimum
    if state_cost > 0 and state_cost > cost(result[-1]):
        #print(result[-1])
        state_cost = cost(result[-1])

    #as long as it is not a global minimum(in this case 0), we need to restart the greedy
    #decent from a new random state until it reaches zero and then we can terminate
    while state_cost > 0:
        print("RESTART")
        initial_state = random_state()
        result = greedy_descent(initial_state, neighbours, cost )

        for i in result:
            print(i)
        if state_cost > 0 and state_cost > cost(result[-1]):
            state_cost = cost(result[-1])





def n_queens_neighbours(state):
    possible_state = []
    for i in range(len(state)):
        #print(i)
        for j in range(len(state)):
            if i < j:
                original_state = list(state)
                original_state[i], original_state[j] = original_state[j], original_state[i]
                possible_state.append(tuple(original_state))
            else:
                pass
    return sorted(possible_state)

def n_queens_cost(state):
    possible_state = []
    count = 0
    for i in range(len(state)):
        #print(i)
        for j in range(len(state)):
            if i < j:
                #print(state[i])
                #print(state[j])
                #checking for diagonal
                #if diff in index is the same as the diff in state index means both are diagonal
                if (abs(i - j) == abs(state[i] - state[j])):
                    count += 1
            else:
                pass
    return count

def greedy_descent(initial_state, neighbours, cost):
        state_path = []
        state_path.append(initial_state)
        current_cost = cost(initial_state)
        current_neighbours = neighbours(initial_state)
        current_state =initial_state
        minimum = False
        while (minimum == False):
                state_to_change = None
                best_state = current_state
                #print(current_state)
                current_cost = cost(current_state)
                current_neighbours = neighbours(current_state)
                for neighbour in current_neighbours:
                        if (cost(neighbour) < current_cost):
                                #print(cost(neighbour))
                                #print(current_cost)
                                current_cost = cost(neighbour)
                                state_to_change = neighbour
                if (state_to_change == None):
                        minimum = True
                else:
                        current_state = state_to_change
                        state_path.append(current_state)
        return state_path
```

	Test	Expected	Got	
✔	from student_answer import n_queens_neighbours as neighbours, n_queens_cost as cost from student_answer import greedy_descent, greedy_descent_with_random_restart import random N = 6 random.seed(0) def random_state(): return tuple(random.sample(range(1,N+1), N)) greedy_descent_with_random_restart(random_state, neighbours, cost)	(4, 6, 1, 2, 3, 5) (4, 6, 1, 2, 5, 3) RESTART (3, 4, 6, 5, 1, 2) (3, 4, 6, 1, 5, 2) RESTART (3, 2, 1, 6, 5, 4) (3, 2, 6, 1, 5, 4) (3, 4, 6, 1, 5, 2) RESTART (3, 1, 5, 6, 2, 4) (3, 1, 6, 5, 2, 4) RESTART (5, 1, 3, 2, 4, 6) (3, 1, 5, 2, 4, 6) (1, 3, 5, 2, 4, 6) RESTART (5, 4, 6, 3, 2, 1) (2, 4, 6, 3, 5, 1) RESTART (5, 1, 6, 3, 2, 4) (3, 1, 6, 5, 2, 4) RESTART (5, 4, 3, 1, 2, 6) (5, 4, 1, 3, 2, 6) (2, 4, 1, 3, 5, 6) RESTART (2, 5, 6, 1, 3, 4) (2, 4, 6, 1, 3, 5)	(4, 6, 1, 2, 3, 5) (4, 6, 1, 2, 5, 3) RESTART (3, 4, 6, 5, 1, 2) (3, 4, 6, 1, 5, 2) RESTART (3, 2, 1, 6, 5, 4) (3, 2, 6, 1, 5, 4) (3, 4, 6, 1, 5, 2) RESTART (3, 1, 5, 6, 2, 4) (3, 1, 6, 5, 2, 4) RESTART (5, 1, 3, 2, 4, 6) (3, 1, 5, 2, 4, 6) (1, 3, 5, 2, 4, 6) RESTART (5, 4, 6, 3, 2, 1) (2, 4, 6, 3, 5, 1) RESTART (5, 1, 6, 3, 2, 4) (3, 1, 6, 5, 2, 4) RESTART (5, 4, 3, 1, 2, 6) (5, 4, 1, 3, 2, 6) (2, 4, 1, 3, 5, 6) RESTART (2, 5, 6, 1, 3, 4) (2, 4, 6, 1, 3, 5)	✔
✔	from student_answer import n_queens_neighbours as neighbours, n_queens_cost as cost from student_answer import greedy_descent, greedy_descent_with_random_restart import random N = 8 random.seed(0) def random_state(): return tuple(random.sample(range(1,N+1), N)) greedy_descent_with_random_restart(random_state, neighbours, cost)	(7, 8, 4, 1, 3, 6, 2, 5) (5, 8, 4, 1, 3, 6, 2, 7)	(7, 8, 4, 1, 3, 6, 2, 5) (5, 8, 4, 1, 3, 6, 2, 7)	✔
✔	from student_answer import n_queens_neighbours as neighbours, n_queens_cost as cost from student_answer import greedy_descent, greedy_descent_with_random_restart import random N = 20 random.seed(0) def random_state(): return tuple(random.sample(range(1,N+1), N)) greedy_descent_with_random_restart(random_state, neighbours, cost)	(13, 14, 2, 9, 16, 7, 20, 5, 8, 6, 10, 4, 3, 12, 18, 1, 15, 17, 19, 11) (13, 8, 2, 9, 16, 7, 20, 5, 14, 6, 10, 4, 3, 12, 18, 1, 15, 17, 19, 11) (10, 8, 2, 9, 16, 7, 20, 5, 14, 6, 13, 4, 3, 12, 18, 1, 15, 17, 19, 11) (10, 8, 2, 9, 16, 3, 20, 5, 14, 6, 13, 4, 7, 12, 18, 1, 15, 17, 19, 11) (10, 8, 2, 9, 16, 3, 20, 15, 14, 6,	(13, 14, 2, 9, 16, 7, 20, 5, 8, 6, 10, 4, 3, 12, 18, 1, 15, 17, 19, 11) (13, 8, 2, 9, 16, 7, 20, 5, 14, 6, 10, 4, 3, 12, 18, 1, 15, 17, 19, 11) (10, 8, 2, 9, 16, 7, 20, 5, 14, 6, 13, 4, 3, 12, 18, 1, 15, 17, 19, 11) (10, 8, 2, 9, 16, 3, 20, 5, 14, 6, 13, 4, 7, 12, 18, 1, 15, 17, 19, 11) (10, 8, 2, 9, 16, 3, 20, 15, 14, 6,	✔

	Test	Expected	Got	
		13, 4, 7, 12, 18, 1, 5, 17, 19, 11) (10, 8, 4, 9, 16, 3, 20, 15, 14, 6, 13, 2, 7, 12, 18, 1, 5, 17, 19, 11) (10, 8, 4, 9, 16, 3, 20, 15, 19, 6, 13, 2, 7, 12, 18, 1, 5, 17, 14, 11) RESTART (4, 3, 11, 16, 20, 6, 7, 15, 10, 18, 17, 9, 8, 12, 5, 19, 1, 13, 14, 2) (4, 3, 8, 16, 20, 6, 7, 15, 10, 18, 17, 9, 11, 12, 5, 19, 1, 13, 14, 2) (3, 4, 8, 16, 20, 6, 7, 15, 10, 18, 17, 9, 11, 12, 5, 19, 1, 13, 14, 2) (3, 4, 8, 16, 20, 6, 7, 15, 10, 18, 17, 9, 11, 13, 5, 19, 1, 12, 14, 2) (3, 4, 8, 16, 20, 2, 7, 15, 10, 18, 17, 9, 11, 13, 5, 19, 1, 12, 14, 6) (3, 11, 8, 16, 20, 2, 7, 15, 10, 18, 17, 9, 4, 13, 5, 19, 1, 12, 14, 6) (3, 11, 8, 2, 20, 16, 7, 15, 10, 18, 17, 9, 4, 13, 5, 19, 1, 12, 14, 6) RESTART (13, 1, 16, 11, 8, 12, 6, 15, 2, 4, 10, 17, 9, 7, 20, 5, 18, 19, 3, 14) (13, 1, 16, 11, 8, 12, 6, 15, 2, 4, 20, 17, 9, 7, 10, 5, 18, 19, 3, 14) (13, 1, 16, 11, 8, 12, 6, 15, 2, 4, 20, 17, 9, 14, 10, 5, 18, 19, 3, 7) (13, 1, 16, 6, 8, 12, 11, 15, 2, 4, 20, 17, 9, 14, 10, 5, 18, 19, 3, 7) (11, 1, 16, 6, 8, 12, 13, 15, 2, 4, 20, 17, 9, 14, 10, 5, 18, 19, 3, 7) (11, 1, 16, 6, 8, 12, 13, 15, 2, 4, 20, 3, 9, 14, 10, 5, 18, 19, 17, 7) (11, 1, 16, 6, 8, 18, 13, 15, 2, 4, 20, 3, 9, 14, 10, 5, 12, 19, 17, 7) RESTART (17, 16, 4, 10, 20, 12, 2, 9, 6, 13, 18, 11, 5, 19, 1, 8, 7, 14, 15, 3) (5, 16, 4, 10, 20, 12, 2, 9, 6, 13, 18, 11, 17, 19, 1, 8, 7, 14, 15, 3) (5, 16, 4, 10, 20, 12, 2, 9, 6, 13,	13, 4, 7, 12, 18, 1, 5, 17, 19, 11) (10, 8, 4, 9, 16, 3, 20, 15, 14, 6, 13, 2, 7, 12, 18, 1, 5, 17, 19, 11) (10, 8, 4, 9, 16, 3, 20, 15, 19, 6, 13, 2, 7, 12, 18, 1, 5, 17, 14, 11) RESTART (4, 3, 11, 16, 20, 6, 7, 15, 10, 18, 17, 9, 8, 12, 5, 19, 1, 13, 14, 2) (4, 3, 8, 16, 20, 6, 7, 15, 10, 18, 17, 9, 11, 12, 5, 19, 1, 13, 14, 2) (3, 4, 8, 16, 20, 6, 7, 15, 10, 18, 17, 9, 11, 12, 5, 19, 1, 13, 14, 2) (3, 4, 8, 16, 20, 6, 7, 15, 10, 18, 17, 9, 11, 13, 5, 19, 1, 12, 14, 2) (3, 4, 8, 16, 20, 2, 7, 15, 10, 18, 17, 9, 11, 13, 5, 19, 1, 12, 14, 6) (3, 11, 8, 16, 20, 2, 7, 15, 10, 18, 17, 9, 4, 13, 5, 19, 1, 12, 14, 6) (3, 11, 8, 2, 20, 16, 7, 15, 10, 18, 17, 9, 4, 13, 5, 19, 1, 12, 14, 6) RESTART (13, 1, 16, 11, 8, 12, 6, 15, 2, 4, 10, 17, 9, 7, 20, 5, 18, 19, 3, 14) (13, 1, 16, 11, 8, 12, 6, 15, 2, 4, 20, 17, 9, 7, 10, 5, 18, 19, 3, 14) (13, 1, 16, 11, 8, 12, 6, 15, 2, 4, 20, 17, 9, 14, 10, 5, 18, 19, 3, 7) (13, 1, 16, 6, 8, 12, 11, 15, 2, 4, 20, 17, 9, 14, 10, 5, 18, 19, 3, 7) (11, 1, 16, 6, 8, 12, 13, 15, 2, 4, 20, 17, 9, 14, 10, 5, 18, 19, 3, 7) (11, 1, 16, 6, 8, 12, 13, 15, 2, 4, 20, 3, 9, 14, 10, 5, 18, 19, 17, 7) (11, 1, 16, 6, 8, 18, 13, 15, 2, 4, 20, 3, 9, 14, 10, 5, 12, 19, 17, 7) RESTART (17, 16, 4, 10, 20, 12, 2, 9, 6, 13, 18, 11, 5, 19, 1, 8, 7, 14, 15, 3) (5, 16, 4, 10, 20, 12, 2, 9, 6, 13, 18, 11, 17, 19, 1,	

	Test	Expected	Got	
		18, 11, 17, 19, 1, 3, 7, 14, 15, 8) (5, 16, 4, 10, 20, 15, 2, 9, 6, 13, 18, 11, 17, 19, 1, 3, 7, 14, 12, 8) (4, 16, 5, 10, 20, 15, 2, 9, 6, 13, 18, 11, 17, 19, 1, 3, 7, 14, 12, 8) (4, 16, 5, 10, 20, 15, 2, 9, 6, 1, 18, 11, 17, 19, 13, 3, 7, 14, 12, 8) RESTART (6, 7, 20, 2, 9, 8, 17, 14, 11, 3, 12, 1, 13, 18, 5, 4, 10, 19, 16, 15) (6, 7, 20, 2, 9, 8, 17, 14, 16, 3, 12, 1, 13, 18, 5, 4, 10, 19, 11, 15) (6, 7, 20, 2, 3, 8, 17, 14, 16, 9, 12, 1, 13, 18, 5, 4, 10, 19, 11, 15) (6, 7, 20, 2, 3, 8, 17, 14, 16, 9, 4, 1, 13, 18, 5, 12, 10, 19, 11, 15) (4, 7, 20, 2, 3, 8, 17, 14, 16, 9, 6, 1, 13, 18, 5, 12, 10, 19, 11, 15) (4, 7, 20, 2, 3, 8, 17, 14, 16, 9, 5, 1, 13, 18, 6, 12, 10, 19, 11, 15) RESTART (19, 14, 9, 15, 16, 11, 17, 12, 6, 2, 13, 20, 8, 5, 10, 3, 18, 1, 7, 4) (19, 14, 9, 3, 16, 11, 17, 12, 6, 2, 13, 20, 8, 5, 10, 15, 18, 1, 7, 4) (19, 14, 9, 3, 16, 11, 17, 12, 6, 4, 13, 20, 8, 5, 10, 15, 18, 1, 7, 2) (19, 14, 9, 3, 1, 11, 17, 12, 6, 4, 13, 20, 8, 5, 10, 15, 18, 16, 7, 2) (3, 14, 9, 19, 1, 11, 17, 12, 6, 4, 13, 20, 8, 5, 10, 15, 18, 16, 7, 2) (3, 14, 9, 19, 1, 11, 17, 12, 6, 4, 2, 20, 8, 5, 10, 15, 18, 16, 7, 13) (3, 5, 9, 19, 1, 11, 17, 12, 6, 4, 2, 20, 8, 14, 10, 15, 18, 16, 7, 13) (8, 5, 9, 19, 1, 11, 17, 12, 6, 4, 2, 20, 3, 14, 10, 15, 18, 16, 7, 13)	8, 7, 14, 15, 3) (5, 16, 4, 10, 20, 12, 2, 9, 6, 13, 18, 11, 17, 19, 1, 3, 7, 14, 15, 8) (5, 16, 4, 10, 20, 15, 2, 9, 6, 13, 18, 11, 17, 19, 1, 3, 7, 14, 12, 8) (4, 16, 5, 10, 20, 15, 2, 9, 6, 13, 18, 11, 17, 19, 1, 3, 7, 14, 12, 8) (4, 16, 5, 10, 20, 15, 2, 9, 6, 1, 18, 11, 17, 19, 13, 3, 7, 14, 12, 8) RESTART (6, 7, 20, 2, 9, 8, 17, 14, 11, 3, 12, 1, 13, 18, 5, 4, 10, 19, 16, 15) (6, 7, 20, 2, 9, 8, 17, 14, 16, 3, 12, 1, 13, 18, 5, 4, 10, 19, 11, 15) (6, 7, 20, 2, 3, 8, 17, 14, 16, 9, 12, 1, 13, 18, 5, 4, 10, 19, 11, 15) (6, 7, 20, 2, 3, 8, 17, 14, 16, 9, 4, 1, 13, 18, 5, 12, 10, 19, 11, 15) (4, 7, 20, 2, 3, 8, 17, 14, 16, 9, 6, 1, 13, 18, 5, 12, 10, 19, 11, 15) (4, 7, 20, 2, 3, 8, 17, 14, 16, 9, 5, 1, 13, 18, 6, 12, 10, 19, 11, 15) RESTART (19, 14, 9, 15, 16, 11, 17, 12, 6, 2, 13, 20, 8, 5, 10, 3, 18, 1, 7, 4) (19, 14, 9, 3, 16, 11, 17, 12, 6, 2, 13, 20, 8, 5, 10, 15, 18, 1, 7, 4) (19, 14, 9, 3, 16, 11, 17, 12, 6, 4, 13, 20, 8, 5, 10, 15, 18, 1, 7, 2) (19, 14, 9, 3, 1, 11, 17, 12, 6, 4, 13, 20, 8, 5, 10, 15, 18, 16, 7, 2) (3, 14, 9, 19, 1, 11, 17, 12, 6, 4, 13, 20, 8, 5, 10, 15, 18, 16, 7, 2) (3, 14, 9, 19, 1, 11, 17, 12, 6, 4, 2, 20, 8, 5, 10, 15, 18, 16, 7, 13) (3, 5, 9, 19, 1, 11, 17, 12, 6, 4, 2, 20, 8, 14, 10, 15, 18, 16, 7, 13) (8, 5, 9, 19, 1,	

	Test	Expected	Got	
			`11, 17, 12, 6, 4, 2, 20, 3, 14, 10, 15, 18, 16, 7, 13)`	

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00. Accounting for previous tries, this gives **0.85/1.00**.

Question **5**

Correct

Mark 1.00 out of 1.00

Write a function `roulette_wheel_select(population, fitness, r)` that takes a list of individuals, a fitness function, and a floating-point random number *r* in the interval [0, 1), and selects and returns an individual from the population using the roulette wheel selection mechanism. The fitness function (which will be provided as an argument) takes an individual and returns a non-negative number as its fitness. The higher the fitness the better. When constructing the roulette wheel, do not change the order of individuals in the population.

For example:

Test	Result

`from student_answer import roulette_wheel_select`
 	
`population = ['a', 'b']`
 	
`def fitness(x):`
`    return 1 # everyone has the same fitness`
 	
`for r in [0, 0.33, 0.49999, 0.5, 0.75, 0.99999]:`
`    print(roulette_wheel_select(population, fitness, r))`
	a
	a
	a
	b
	b
	b

`from student_answer import roulette_wheel_select`
 	
`population = [0, 1, 2]`
 	
`def fitness(x):`
`    return x`
 	
`for r in [0, 0.33, 0.34, 0.5, 0.75, 0.99]:`
`    print(roulette_wheel_select(population, fitness, r))`
	1
	1
	2
	2
	2
	2

Answer: (penalty regime: 0, 15, ... %)

```python
def roulette_wheel_select(population, fitness, r):
    #Sum the fitness of all individuals, call it T
    fitness_sum = 0
    running_total = []
    for i in population:
        candidate_fitness = fitness(i)
        fitness_sum += candidate_fitness

    #for each member in the population, generate a list of the fitness level of
    #each member/ total population fitness (chance for each individual to be selected)
    for m in population:
        running_total.append(fitness(m)/fitness_sum)
    #
    for j in range(len(running_total)):
        if j == 0:
            running_total[j] = running_total[j]
        else:
            running_total[j] += running_total[j-1]
    #index is probability k is index and r is the factor that defines success or failure of roulette
    for index, k in enumerate(running_total):
        #if probability is better then number given then it is selected
        if k > r or index == len(running_total) - 1:
            #return selection in this case a or b
            return population[index]
```

	Test	Expected	Got	

✔	`from student_answer import roulette_wheel_select`
	
	`population = ['a', 'b']`
	
	`def fitness(x):`
	`    return 1 # everyone has the same fitness`
	
	`for r in [0, 0.33, 0.49999, 0.5, 0.75, 0.99999]:`
	`    print(roulette_wheel_select(population, fitness, r))`
		a
		a
		a
		b
		b
		b
			a
			a
			a
			b
			b
			b
				✔

	Test	Expected	Got	

✔	`from student_answer import roulette_wheel_select`
	
	`population = [0, 1, 2]`
	
	`def fitness(x):`
	`    return x`
	
	`for r in [0, 0.33, 0.34, 0.5, 0.75, 0.99]:`
	`    print(roulette_wheel_select(population, fitness, r))`
		1
		1
		2
		2
		2
		2
			1
			1
			2
			2
			2
			2
				✔

✔	`from student_answer import roulette_wheel_select`
	
	`population = ['cosc']`
	
	`def fitness(x):`
	`    return 50`
	
	`for r in [0, 0.33, 0.49999, 0.5, 0.75, 0.99999]:`
	`    print(roulette_wheel_select(population, fitness, r))`
		cosc
		cosc
		cosc
		cosc
		cosc
		cosc
			cosc
			cosc
			cosc
			cosc
			cosc
			cosc
				✔

✔	`from student_answer import roulette_wheel_select`
	
	`population = [0, 1, 2, 3, 4, 5]`
	
	`def fitness(x):`
	`    return x`
	
	`for r in range(13):`
	`    print(roulette_wheel_select(population, fitness, r/13))`
		1
		2
		2
		3
		3
		3
		4
		4
		4
		5
		5
		5
		5
			1
			2
			2
			3
			3
			3
			4
			4
			4
			5
			5
			5
			5
				✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

◄ 6. Constraint satisfaction problems

Jump to...

8. Belief networks and probabilistic inference ►