```
%matplotlib inline
```

Assignment 3

DUE: Tuesday, February 14th 2022 at 11:59pm

Turn in the assignment via Canvas.

To write legible answers you will need to be familiar with both Markdown and Latex

Before you turn this problem in, make sure everything runs as expected. To do so, restart the kernel and run all cells (in the menubar, select Runtime→→Restart and run all).

Show your work!

Whenever you are asked to find the solution to a problem, be sure to also **show how you arrived** at your answer.

Resources

[1] https://community.plotly.com/t/two-3d-surface-with-different-color-map/18931/2 (Different color plots plotly)

[2] https://plotly.com/python/3d-surface-plots/ [plotting surfaces]

[3] https://community.plotly.com/t/3d-scatter-plot-with-surface-plot/27556/5 [plotting scatter plots]

[4] https://community.plotly.com/t/what-colorscales-are-available-in-plotly-and-which-are-the-default/2079 [color scales]

```
%matplotlib inline
```

```
NAME = "Omkar Ghanekar"
STUDENT_ID = "1926974"
```

Problem 1: Local Search on the Ackley Surface

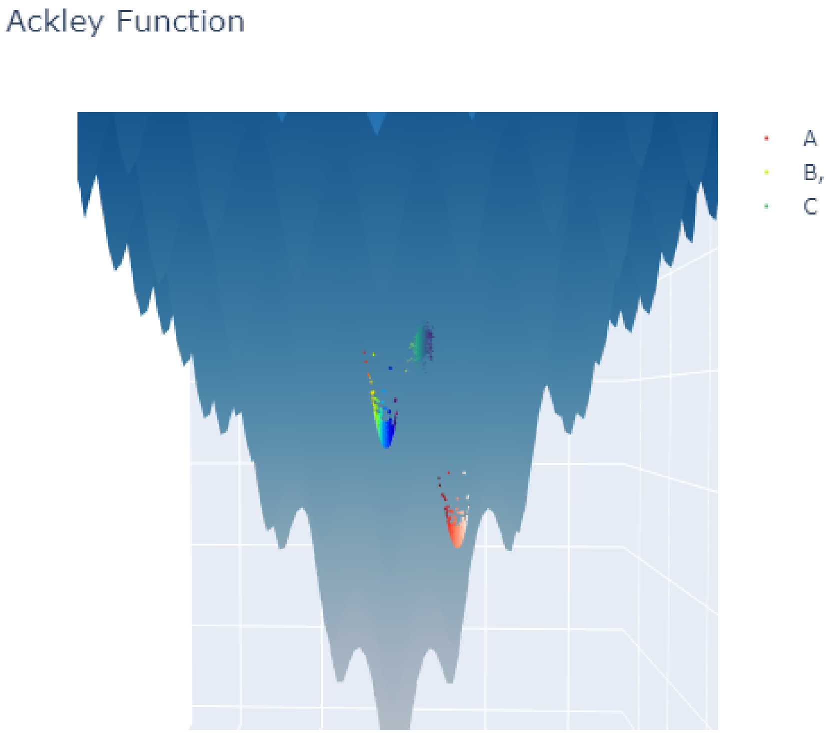
As discussed in class with Local Search problems the path to the goal is not as important as the goal itself. In this question we work with the Ackley function, a non-convex function typically used as a test for optimization algorithms.

The Achkey function has many local optima but only one global optimum that is $f(0,0) = 0$. Interestingly enough it might not be that easy to find this solution.

The Ackley function is defined as:

$$f(x, y) := -20exp[-0.2\sqrt{0.5(x^2 + y^2)}] - exp$$

The figure below illustrates the result of running three optimization methods on the Ackley Function. These solutions were found using methods A, B and C.



In this question, you will implement 3 optimization algorithms:

1. Stochastic Hill Climbing with Restarts (SHCR)
2. Simulated Annealing (SA)
3. Local Beam Search (LBS)

Deliverables

1. Complete this Notebook with implementation of SHCR, SA and LBS.

```
import numpy as np
from numpy import arange
from numpy import exp
from numpy import sqrt
from numpy import cos
from numpy import e
from numpy import pi
from numpy import meshgrid
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed
# hill climbing search of the ackley objecti
from numpy import asarray
import plotly.graph_objects as go

# objective function
def f(x, y):
    return -20.0 * exp(-0.2 * sqrt(0.5 * (x*

def create_ackley_figure(solution_points=np.
  """
  :param solution_points A numpy array of di
  where i is the number of solution points y
  number of steps used in estimating your so
  number of steps if you want to plot all so
```

```
    :return An interactive Ackley function fig
    """
    # define range for input
    r_min, r_max = -5.0, 5.0
    # sample input range uniformly at 0.1 incr
    xaxis = arange(r_min, r_max, 0.1) # (100,)
    yaxis = arange(r_min, r_max, 0.1) # (100,)
    # create a mesh from the axis
    x, y = meshgrid(xaxis, yaxis)

    # compute targets
    results = f(x, y)

    figure = go.Figure(data=[go.Surface(z=resu
    figure.update_layout(title='Ackley Functio
                    width=500, height=500,
                    margin=dict(l=65, r=50, b=

    if solution_points.size:
      soln_colors = ['Reds', 'Rainbow', 'Virid
      for i in range(len(solution_points)):
        x_sln = solution_points[i][:,0]
        y_sln = solution_points[i][:,1]
        zdata = f(x_sln, y_sln)
        figure.add_scatter3d(name=soln_names[i
        marker=dict(size=1, color=x_sln.flatte

        figure.update_layout(showlegend=True)
```

```
  return figure

### RUN THIS ###
"""
This is the Ackley Function.
"""
ack_figure = create_ackley_figure()
ack_figure
```

Ackley Function

General Function Definitions

```
# objective function
def objective(v):
  """
  This function defines the Ackley surface,
  can be used to test if we have found point
  :param v, a tuple representing a 2D point

  returns a value in the range [-5.0, 5.0]
  """
  x, y = v
  return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2


# check if a point is within the bounds of t
def in_bounds(point, bounds=asarray([[-5.0,
  """
  It is possible our optimzation method coul
  space, so it is helpful to check.
  :param point a tuple representing a 2D poi
```

```
    :param a 2D array, that describes the boun
    returns a Boolean of whether the point lie
    """
      # enumerate all dimensions of the point
    for d in range(len(bounds)):
      # check if out of bounds for this dimens
      if point[d] < bounds[d, 0] or point[d] >
        return False
    return True
```

Part 1) Stochastic Hill Climbing with Restarts

1. Implement SHCR in the cell below.
2. Visualize and comment on the candidate points visited.

```
import random
# stochastic hill climbing with restarts alg
def stochastic_hillclimbing(objective, bound
    """
    :param objective, the function we are tryi
    :param bounds, the boundaries of the probl
    :param n_iterations number of times to rep
    :param step_size how much we should move
    :param start_pt the point we start optimiz
```

```
    returns [solution, solution_value, candida
    """
    ### YOUR CODE HERE ###

    solution, solution_value = start_pt, objec
    candidates = []
    for i in range(n_iterations):
        # take a step and evaluate candidate p
        candidate = start_pt + randn(len(bound
        candidates.append(candidate)
        candidate_eval = objective(candidate)
        if candidate_eval <= solution_value:
              solution, solution_value = candi
                  # store the new point and pr
    return [solution, solution_value, candidat

def random_restarts(objective, bounds, n_ite
    """
    :param objective, the function we are tryi
    :param bounds, the boundaries of the probl
    :param n_iter number of times to repeat st
    :param step_size how much we should move
    :param n_restarts, the number of times we
    returns [best solution, best solution valu
    """
    ### YOUR CODE HERE ###

    #state = problem.initial()
    count = 0
    best_sol, best_sol_val, best_sol_points =
```

```
    for iter in range(n_restarts):
        x = random.randrange(-5.0, 5.0)
        y = random.randrange(-5.0, 5.0)
        sol, sol_val, sol_points  = stochastic
        if sol_val < best_sol_val:
            best_sol, best_sol_val, best_sol_p
    return best_sol, best_sol_val, best_sol_po
    #return -1, -1, -1

# seed the pseudorandom number generator
seed(240)
# define range for input
bounds = asarray([[-5.0, 5.0], [-5.0, 5.0]])
# define the total iterations
n_iter = 1000
# define the maximum step size
step_size = 0.01
# total number of random restarts
n_restarts = 30
# perform the hill climbing search
best, score, shcr_candidates = random_restar
print('Done!')
print('f(%s) = %f' % (best, score))
```

```
Done!
f((0, 0)) = 0.000000
```

```
# An example of plotting solutions from 3 op
```

```
# all_method_candidates has shape [3, 1000,
# the second dim is the number of steps take
# shcr_candidates = np.random.rand(100,2) #
#shcr_candidates = shcr_candidates.reshape(-
all_method_candidates = np.array([shcr_candi
ack_figure = create_ackley_figure(all_method
ack_figure
```

Ackley Function

The restart random method helps the algorithm to reach the global minima at point(0.00, 0.00) with a cost of 0.0. It starts from 30 random points with a step size of 0.01 which help it reach very close to the global minima.

Part 2) Simulated Annealing

1. Implement SA in the cell below.
2. Visualize and comment on the candidate points visited.

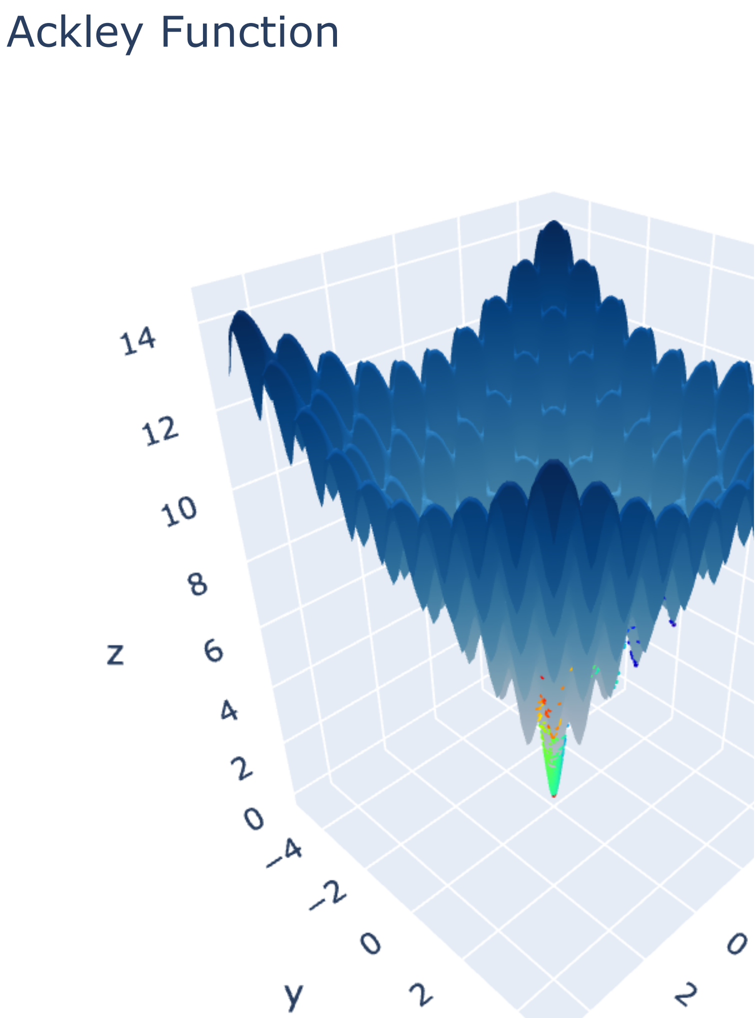
```
def get_temperature_schedule(epoch, temp, di
  """
  :param temp Temperature
  :param epoch Iterations elapsed
  :param diff Difference between value of ca
  """
  # calculate temperature for current epoch
  t = temp / float(epoch + 1)
  # calculate acceptance criterion
```

```
    # calculate acceptance criterion
    criterion = exp(-diff / t)
    return criterion

# simulated annealing algorithm
def simulated_annealing(objective, bounds, n
    """
    :param objective, the function we are tryi
    :param bounds, the boundaries of the probl
    :param n_iterations number of times to rep
    :param step_size how much we should move
    :param temp temperature for each epoch
    returns [solution, solution_value, candida
    """
    ### YOUR CODE HERE ###
    candidates = []
    best = (random.randrange(-5.0, 5.0), rando
    best_eval = objective(best)
    curr, curr_eval = best, best_eval
    for iter in range(n_iterations):
        candidate = curr + randn(len(bounds))
        candidates.append(candidate)

        # take a step and evaluate candidate p
        candidate_eval = objective(candidate)
        if candidate_eval < best_eval:
            best, best_eval = candidate, candi
        diff = candidate_eval - curr_eval

        criterion = get_temperature_schedule(i
        if diff < 0 or rand() < criterion:
```

```
            curr, curr_eval = candidate, candi
    return [best, best_eval, candidates]

# seed the pseudorandom number generator
seed(240)
# define range for input
bounds = asarray([[-5.0, 5.0], [-5.0, 5.0]])
# define the total iterations
n_iterations = 1000
# define the maximum step size
step_size = 0.1
# initial temperature
temp = 100
# perform the simulated annealing search
best, score, sa_candidates = simulated_annea
print('Done!')
print('f(%s) = %f' % (best, score))
```

```
Done!
f([ 0.00099391 -0.00211901]) = 0.00676
```

```
# sa_candidates = np.random.rand(40,2)
# sa_candidates = sa_candidates.reshape(-1,2
all_method_candidates = np.array([shcr_candi
ack_figure = create_ackley_figure(all_method
ack_figure
```



The simulated annealing helps the algorithm to reach the global minima at point(-0.0001002 0.00013996) with a cost of 0.000488. It has a step size of 0.1 and runs for 1000 epochs, with the temprature of 100 which helps it reach very close to the global minima.

Part 3) Local Beam Search

1. Implement LBS in the cell below.
2. Visualize and comment on the candidate points visited.
3. Compare all 3 implementations by commenting on the distribution of points on the Ackley surface and the empirical run time of each method.

```
# generate the neighbors based on step size
def generate_neighbors(point, step_size=0.1)
    point = point.reshape(-1,1)
    possible_steps = possible_steps = np.arr
                              [step_size*i for
    neighbors = point + possible_steps
    return neighbors


def local_beam_search(objective, bounds, ste
  """
  :param objective, the function we are tryi
  :param bounds, the boundaries of the probl
```

```
    :param k, how many candidates to consider
    :param n_iterations, how long to search fo
    returns [solution, solution_value, candida
    """
    ### YOUR CODE HERE ###
    return -1, -1, -1

# seed the pseudorandom number generator
seed(240)
# define range for input
bounds = asarray([[-5.0, 5.0], [-5.0, 5.0]])
# define the total iterations
n_iterations = 10
# define the maximum step size
step_size = 0.1
# candidates to consider
k = 1
# perform the local beam search
sequences = local_beam_search(objective, bou
```

[YOUR ANSWER HERE]

Problem 2: CSP

Consider the following constraint satisfaction problem. A linear graph has nodes of the following colors:

- Red
- Yellow
- Green
- Blue
- Violet

Each node has a domain of {1, 2, ..., 9}.

Each node type has the following constraints on its value:

- Red - No contraints
- Yellow - equals the rightmost digit of of the product of all its neighbors
- Green - equals the rightmost digit of the sum of all its neighbors
- Blue - equals the leftmost digit of the sum of all its neighbors
- Violet - equals the leftmost digit of the product of all of its neighbors

As a reminder here is the pseudo code for the Min-Conflicts search algorithm:

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
     if current is a solution for csp then return current
     var ← a randomly chosen conflicted variable from csp.VARIABLES
     value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
     set var = value in current
  return failure
```

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

Notes:

- It's possible that you won't converge to a solution in a single run. Try a few runs to see if you get to a solution.
- The testcases are to show you what a problem looks like, we will test/grade your program on different examples.

Part 1) Implementation

Complete the function `solve_csp` defined below. You may find some helper functions useful.

```
import random
import copy
```

```
def create_arc(node_graph, source, dest, nod
    if nodes[source] in node_graph:
      node_graph[nodes[source]].append(nodes
    else:
      node_graph[nodes[source]] = [nodes[des

    return node_graph

def find_rightmost_digit(number):    # to fi
    #print(number%10)
    return number%10

def find_leftmost_digit(number):     # to fi
    #print(int(str(number)[0]))
    return int(str(number)[0])

def number_of_conflicts(node_graph, current,

    conflicts = []
    for node in node_graph:
      if not is_node_consistent(node_graph,
        conflicts.append(node)
    return conflicts



def is_node_consistent(node_graph, node, cur
    neighbours = node_graph[node]
    # print('in is consistent', node, neighb
    if node in ['Y','V']:
```

```
        neighbour_prod =1
        for neighbor in neighbours:
          neighbour_prod *= current[node_key
        if node == 'Y' and find_rightmost_di
            return True
        elif find_leftmost_digit(neighbour_p
            return True
        return False
    elif node in ['G','B']:
        neighbour_sum =0
        for neighbor in neighbours:
            neighbour_sum += current[node_ke
        if node == 'G' and find_rightmost_di
            return True
        elif find_leftmost_digit(neighbour_s
            return True
        return False
    return True



def assign_node_value(node, current, node_ke
    # print('in number of conflicts', node,
    if node in ['Y','V']:
        neighbour_prod =1
        for neighbor in neighbours:
          neighbour_prod *= current[node_key
        #print(node, neighbour_prod, current
        if node == 'Y':
            return find_rightmost_digit(neig
        else:
```

```
            return find_leftmost_digit(neigh
    elif node in ['G','B']:
        neighbour_sum =0
        for neighbor in neighbours:
            neighbour_sum += current[node_ke
        #print(node, neighbour_sum, current,
        if node == 'G':
            return find_rightmost_digit(neig
        else:
            return find_leftmost_digit(neigh


def min_conflicts(node_graph, node_domains,
    current = [1] *len(node_key_value)
    #current = [random.randint(1,9) for i in
    for i in range(max_steps):
        # number of conflicts in the current
        conflicts = number_of_conflicts(node
        #print(i,len(conflicts),conflicts)
        # Check if we have a valid solution
        if not conflicts:
            return current

        conflict_cnt= {}

        # Tried with the minimum constraint

        # make copy of current assignment to
        # for node in conflicts:
        #     current_copy = copy.deepcopy(c
```

```
        #     current_copy[node_key_value[no

        #     conflicts_after_assignment = n
        #     conflict_cnt[node] = len(confl

        #print('conflicts after assignment',

        #scores = {key: value for key, value
        #current[node_key_value[next(iter(sc
        #print('sorted conflict cnt', scores

        # # Select a cell in conflict at ran
        random_node =  random.choice(conflic
        current[node_key_value[random_node]]


    return []

def solve_csp(nodes, arcs, max_steps):
    """
    This function solves the csp using the M

    :param nodes, a list of letters that ind
                  the index of the node in t
                  letters = {R, Y, G, B, V}


    :param arcs,  a list of tuples that cont
                  IDs of the nodes the arc c


    :param max_steps, max number of steps to
```

```
    returns a list of values for the solutio
            index of the value corresponds
    """
    node_values = []
    ### YOUR CODE HERE ###

    # create a map of nodes and neighbours
    node_graph = {}
    for item in arcs:
      node_graph = create_arc(node_graph, it
      node_graph = create_arc(node_graph, it

    node_key_value = {}
    # node_domains = {}
    node_domains = [1,2,3,4,5,6,7,8,9]
    for index,node_value in enumerate(nodes)
      node_key_value[node_value] = index
      # node_domains[node_value] = [1,2,3,4,


    node_values = min_conflicts(node_graph,
    return node_values
```

Test Cases

Below we've included 4 test cases to help you debug your code. Your submitted code will be tested on other cases as well, but if your implementation of the above Min-Conflicts search algorithm is able to solve these problems, you should be good to go.

```
# test Case 1

nodes = 'YGVRB'
arcs = [(0,1), (0,2), (1,2), (1,3), (1,4), (
max_steps = 1000

for _ in range(max_steps):
    sol = solve_csp(nodes, arcs, max_steps)
    if sol != []:
        break

all_solutions = [[1, 1, 1, 7, 2],[2, 1, 2, 4
                 [3, 3, 1, 5, 4],[6, 2, 8, 7

if sol == []:
    print('No solution')
else:
    if sol in all_solutions:
        print('Solution found:', sol)
    else:
        print('ERROR: False solution found:'
```

```
ERROR: False solution found: [4, 2, 7,
```

```
# test Case 2

nodes = 'YVBGR'
arcs = [(0,1), (0,2), (1,3), (2,4)]
max_steps = 1000

for _ in range(max_steps):
    sol = solve_csp(nodes, arcs, max_steps)
    if sol != []:
        print(nodes)
        break

all_solutions = [[1, 1, 1, 1, 9], [1, 3, 7,

if sol == []:
    print('No solution')
else:
    if sol in all_solutions:
        print('Solution found:', sol)
    else:
        print('ERROR: False solution found:'
```

```
YVBGR
ERROR: False solution found: [0, 0, 1,
```

```
# test Case 3

nodes = 'VYGBR'
arcs = [(0,1), (1,2), (2,3), (3,4)]
```

```
max_steps = 1000

for _ in range(max_steps):
    sol = solve_csp(nodes, arcs, max_steps)
    if sol != []:
        print(nodes)
        break

all_solutions = [[2, 2, 1, 9, 8],[3, 3, 1, 8
                 [6, 6, 1, 5, 4],[7, 7, 1, 4

if sol == []:
    print('No solution')
else:
    if sol in all_solutions:
        print('Solution found:', sol)
    else:
        print('ERROR: False solution found:'
```

```
VYGBR
Solution found: [9, 9, 1, 2, 1]
```

```
# test Case 4

nodes = 'YGVBR'
arcs = [(0,1), (0,2), (1,3), (2,3), (3,4), (
max_steps = 1000

for _ in range(max_steps):
```

```
    sol = solve_csp(nodes, arcs, max_steps)
    if sol != []:
        print(nodes)
        break

all_solutions = [[4, 4, 1, 9, 4],[4, 7, 2, 1
                 [4, 7, 2, 1, 5],[4, 7, 2, 1
                 [4, 8, 3, 1, 1],[4, 8, 3, 1
                 [4, 8, 3, 1, 6],[4, 8, 3, 1
                 [5, 1, 5, 1, 6],[5, 1, 5, 1

if sol == []:
    print('No solution')
else:
    if sol in all_solutions:
        print('Solution found:', sol)
    else:
        print('ERROR: False solution found:'
```

```
YGVBR
Solution found: [4, 7, 2, 1, 1]
```

Problem 3: The N-Rooks Problem

Rooks can move any number of squares horizontally or vertically on a chess board. The n rooks problem is to arrange rooks on an $n \times n$

board in such a way that none of the rooks could bump into another by making any of its possible horizontal or vertical moves.

For this problem, the variables are each column (labeled 0, 1, ... , $n-1$), the the domain consists of each possible row (also labeled 0, 1, ... , $n-1$). In each column we place a rook on row 0, row 1, ... , row $n-1$.

For example, if $n=2$ we have only two solutions to this problem:

```
R @
@ R
```

or

```
@ R
R @
```

Part 1)

How many possible solutions are there to this CSP, in terms of n? Also, give a simple proof that your answer is correct.

There are n! solutions possible to the n-rook problem. The simplest solution is placing the first rook in cell[1][1] and the corresponding ith rook at cell[i][i]. Thus, ith rook can be placed in (n-i) positions in the row, which results in total permutations as n*(n-1)*(n-2)..2*1 which is n!

Python Library to solve CSPs

One useful Python module for solving these types of problems is called *constraint*. In the next cell you'll see how to load this module into a Jupyter notebook running in Colab.

We'll use this module to solve the following simple CSP. Suppose our variables are x and y, and the values they are allowed to assume are numbers in the domain {1, 2, ... , 100}, and with constraints be that $x^2 = y$ and that x is odd.

```
# The following line imports the constraint
# (This only needs to be done once per sessi

!pip install python-constraint

# Let's load all the functions available in
# (This only needs to be done once per sessi

from constraint import *

# Now we initialize a new problem and add th
problem = Problem()
problem.addVariables(["x", "y"], list(range(

# So far there are no constraints. If we sol
# then every possible combination of x and y
solutions = problem.getSolutions()

# Total number of solutions
print( len(solutions) )
```

```
Requirement already satisfied: python-c
10000
```

```
# Add the given constraints:
problem.addConstraint(lambda a, b: a**2 == b

# There are only 5 solutions that satisfy th
```

```
solutions = problem.getSolutions()
print( len(solutions) )
```

```
5
```

```
# Here are all 5 solutions:
print(solutions)
```

```
[{'x': 9, 'y': 81}, {'x': 7, 'y': 49},
```

Notice that the solutions consist of a list of dictionaries, where each dictionary represents a solution. For example, the first solution is {'x': 9, 'y': 81}, since with $x = 9$ and $y = 81$ it's true that $x^2 = y$.

Part 2) The n-rooks problem revisited

Modify the example before to solve the n-rooks problem.

```
### YOUR CODE HERE ###

n_rook_problem = Problem()
num_of_rooks = 4       # update the number of
cols = range(num_of_rooks)
```

```
rows = range(num_of_rooks)
n_rook_problem.addVariables(cols, rows)
for col1 in cols:
  for col2 in cols:
    if col1 < col2:
      n_rook_problem.addConstraint(lambda ro
solutions = n_rook_problem.getSolutions()
print(solutions)
print(len(solutions))
```

```
[{0: 3, 1: 2, 2: 1, 3: 0}, {0: 3, 1: 2,
24
```

Part 3)

- How many ways are there of arranging 8 rooks on an 8×8 board so that none impede the others?
- How many ways are there arranging 11 rooks on an 11×11 board so that none impede the others?

1. 8! = 40,320
2. 11! = 39,916,800

0s completed at 11:34 PM