Intro

My matricola is s333044 In this file , i'll show off all i've done during computational intelligence course. The usual pattern for all course weeks was , going to the lecture ,taking notes . After the lecture study my notes about the lecture and study the slides.

When doing reviews, i cloned the repository of the reviewed and checked if there were errors. Each suggestion done was tested before adding it to the review.

Lab0

LAB0

I wrote a funny joke about the first lesson as required , this is the joke:

> "Probably in a cemetery more students would catch the references of the first lesson than in our classroom."

Reviews0

I also gave remarks about my colleagues' jokes:

Joke by GiorgioBongiovanni:

> "Artificial intelligence is advancing so fast that soon it'll be doing our homework... but who will teach AI to miss deadlines like we do? If AI could learn like us, it would probably forget half of the lecture before the break, just to fit in. After all, we're teaching machines to be smart while we're still struggling to remember the Wi-Fi password! But let's be real: the one thing AI can't do yet is take notes during class... or maybe by the end of the course, it'll figure that out too. We're here building AI to solve complex problems, yet somehow, surviving lab deadlines remains the biggest challenge!"

My remark:

> The solution for these issues that students endure everyday . it's to embrace a world without time where yesterday and tomorrow are contemporary.

Joke by GDennis01:

> "What is intelligence?" "Well, to correctly make a cup of coffee at Teacher Giovanni Squillero's House!"

My remark:

> I guess his coffee machine was intelligent the way his students are intelligent.

Lab1

My LAB_1

In this laboratory i tried to solve Set Cover problem in an efficient way .

In the end solution i used random mutation hill climbing with tweak4 + tweak5 . The solution was enveloped into a .ipymb file , before each code section there is an explanation of what the code does , and there are comments throughout the code sections. At the end there is a table showing off the best results i got using my solution.

my sol

Set Cover problem

```
from random import random, seed
from itertools import product
import numpy as np
import timeit
from typing import Tuple
from icecream import ic
from tqdm.auto import tqdm
from matplotlib import pyplot as plt
```

```
from itertools import accumulate
from itertools import compress
```

Reproducible Initialization

If you want to get reproducible results, use `rng` (and restart the kernel); for non-reproducible ones, use `np.random`.

```
UNIVERSE_SIZE = 10000  #useless
NUM_SETS = 1000

DENSITY = 0.2
rng = np.random.Generator(np.random.PCG64([UNIVERSE_SIZE, NUM_SETS, int(10_000 * DENSITY)]))
```

```
#### DON'T EDIT THESE LINES!

SETS = np.random.random((NUM_SETS, UNIVERSE_SIZE)) < DENSITY
for s in range(UNIVERSE_SIZE):
    if not np.any(SETS[:, s]):
        SETS[np.random.randint(NUM_SETS), s] = True
COSTS = np.pow(SETS.sum(axis=1), 1.1)
```

Helper Functions

```
def valid(solution,sets):
    """Checks wether solution is valid (ie. covers all universe)"""
    return np.all(np.logical_or.reduce(sets[solution]))


def valid2(solution,sets):   #faster valid implentation
    """Checks whether solution is valid (i.e., covers all universe)"""

    covered_universe = np.zeros(sets.shape[1], dtype=bool)


    for i in np.where(solution)[0]:  # Only loop through 'True' elements of thee solution
        covered_universe |= sets[i]   #sets element of the covered universe to true if they are taken in the solution
        # Exit early if the solution is fully covers the universe , no need to check the rest of the solution if the
universe is already covered
        if np.all(covered_universe):
            return True

    # Check if universe is fully covered at the end
    return np.all(covered_universe)

def cost(solution,costs):
    """Returns the cost of a solution (to be minimized)"""
    return costs[solution].sum()
```

Have Fun!

```
NCALLS=0
listTest=[(100,10,0.2),
          (1000,100,0.2),
          (10000,1000,0.2),
          (100000,10000,0.2),
          (100000,10000,0.1),
          (100000,10000,0.3)]
```

```
def fitness(solution : np.array,costs: np.array)-> int:
    costo=-cost(solution,costs)
```

```
    return costo

def fitness2(solution : np.array,costs,sets)-> Tuple[bool,int]:  #returns also if a solution is valid
    global NCALLS
    costo=-cost(solution,costs)
    NCALLS+=1

    validity=False
    if valid2(solution,sets):


       validity=True



    return (validity,costo)
```

tweak :Simple tweak only swap a random value of the solution , results aren't great

```
def tweak(solution : np.array,nsets)-> np.array:
    new_solution = solution.copy()
    pos = rng.integers(0,nsets)


    new_solution[pos] = not new_solution[pos]
    return new_solution
```

tweak3 : it flips multiple elemts in the solution . If i just started exploring it changes a lot of elemnts , if i already explored enough it changes only few elements. Works well if i increse number of steps, getting better results but it takes more time then my main solution (tweak5 + tweak 6)

```
def tweak3(solution : np.array,step: int,max_step :int,nsets)-> np.array:

    size_change=(20*(max_step -step))//max_step
    new_solution = solution.copy()
    rints = rng.integers(low=0, high=nsets, size=size_change)
    for el in rints:
        new_solution[el] = not new_solution[el]

    return new_solution
```

tweak4 : checks if the current solution is alredy valid , if it's valid and i'm near the end , i try to flip an element(small step) and see if the solution get better , if it's valid and i'm not near the end i try to flip elemt from true to false to decreadse the cost (i don't do it near the end or i risk getting an invalid solution). If the current solution is not valid , i try to flip some elements , results are similar to tweak3

```
def tweak4(solution : np.array,step: int,max_step :int,nsets,sets)-> np.array:

    size_change=(20*(max_step -step))//max_step

    new_solution = solution.copy()

    if valid2(new_solution,sets):
        if ((max_step -step)/max_step)<0.05 :

            rints = rng.integers(low=0, high=nsets, size=size_change)
            for el in rints:
                    new_solution[el] = not new_solution[el]


        else:
            index_true= np.where(new_solution == True)[0]
            rints = rng.choice(index_true, size=size_change, replace=False)
            for el in rints:
                new_solution[el] = False

    else:
         rints = rng.integers(low=0, high=nsets, size=size_change)
         for el in rints:
                new_solution[el] = not new_solution[el]
```

```
    return new_solution
```

tweak 5 : i change a n(size_change) values of the solution, if the solution was already valid i flip n values from true to false , otherwise i flip them from false to true. If there aren't enough elemnt to change n elemnts , i change max 20% of them . Using this tweak i need around 1/100 of the step of tweak 4 to get a bit worse(20%) result then tweak4

```
def tweak5(solution : np.array,step: int,max_step :int,valid : bool,usize)-> np.array:

    size_change = int(np.log10(usize)) * 2*(max_step - step) // max_step
    new_solution = solution.copy()

    if valid:

            index_true=np.where(new_solution==True)[0]
            if( size_change>len(index_true)):
                size_change=len(index_true)*20//100
            rints = rng.choice(index_true, size=size_change, replace=False)

            new_solution[rints] = False

    else:


            index_true=np.where(new_solution==False)[0]
            if( size_change>len(index_true)):
                size_change=len(index_true)*20//100
            rints = rng.choice(index_true, size=size_change, replace=False)

            new_solution[rints] = True

    return new_solution
```

tweak 6 : It changes 70% of true elements in false and 20% of false elements into true. It's useful to better explore the solution space when it get stuck in a mesa. Used combined with tweak 5 when it gets stuck without improvemnts for too long , it improves solution by 20% without needing more time or steps .

```
def tweak6(solution : np.array)-> np.array:

    new_solution = solution.copy()

    index_true=np.where(new_solution==True)[0]

    size_change=len(index_true)*70//100
    rints1 = rng.choice(index_true, size=size_change, replace=False)

    index_true=np.where(new_solution==False)[0]

    size_change=len(index_true)*20//100
    rints = rng.choice(index_true, size=size_change, replace=False)

    new_solution[rints] = True

    new_solution[rints1] = False


    return new_solution
```

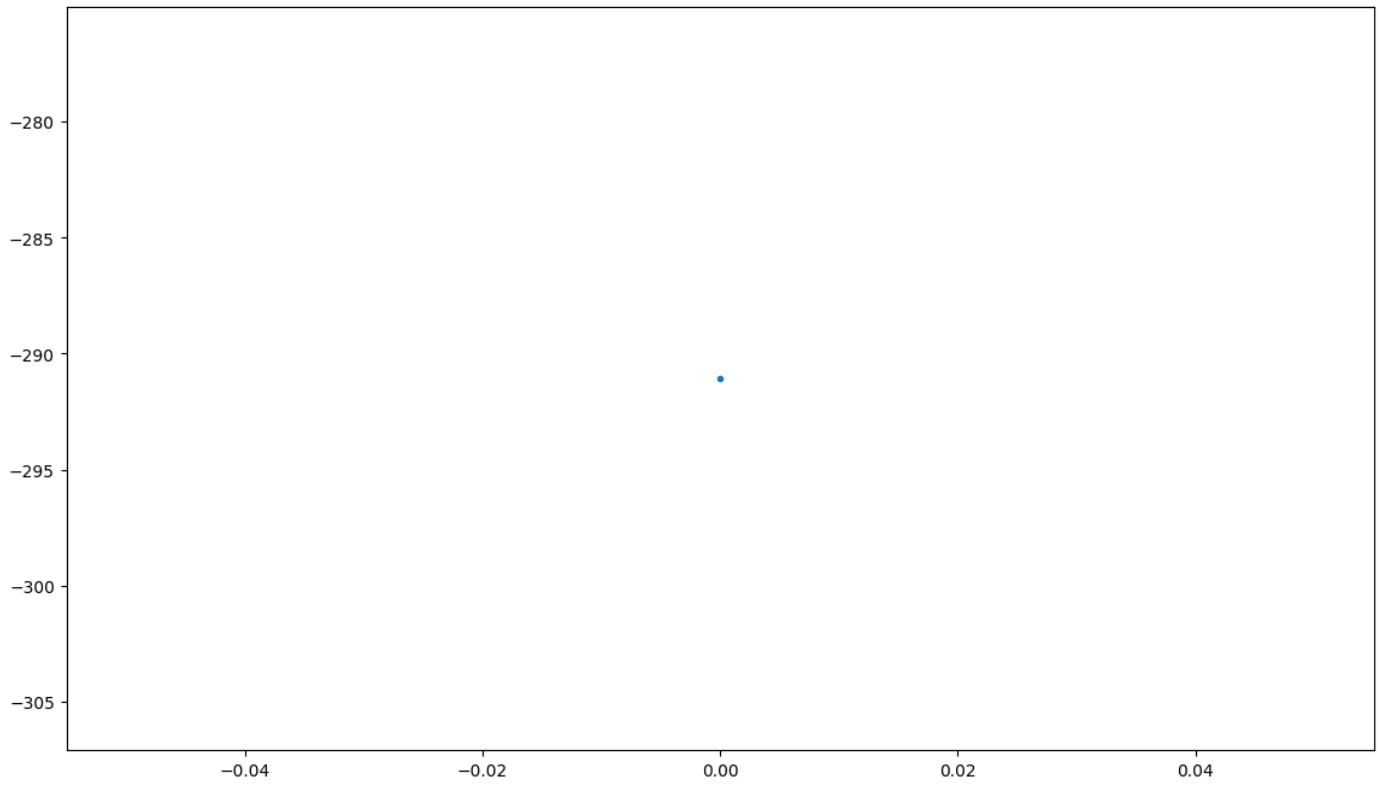
It's used RMHC (Random Mutation Hill-Climbing)

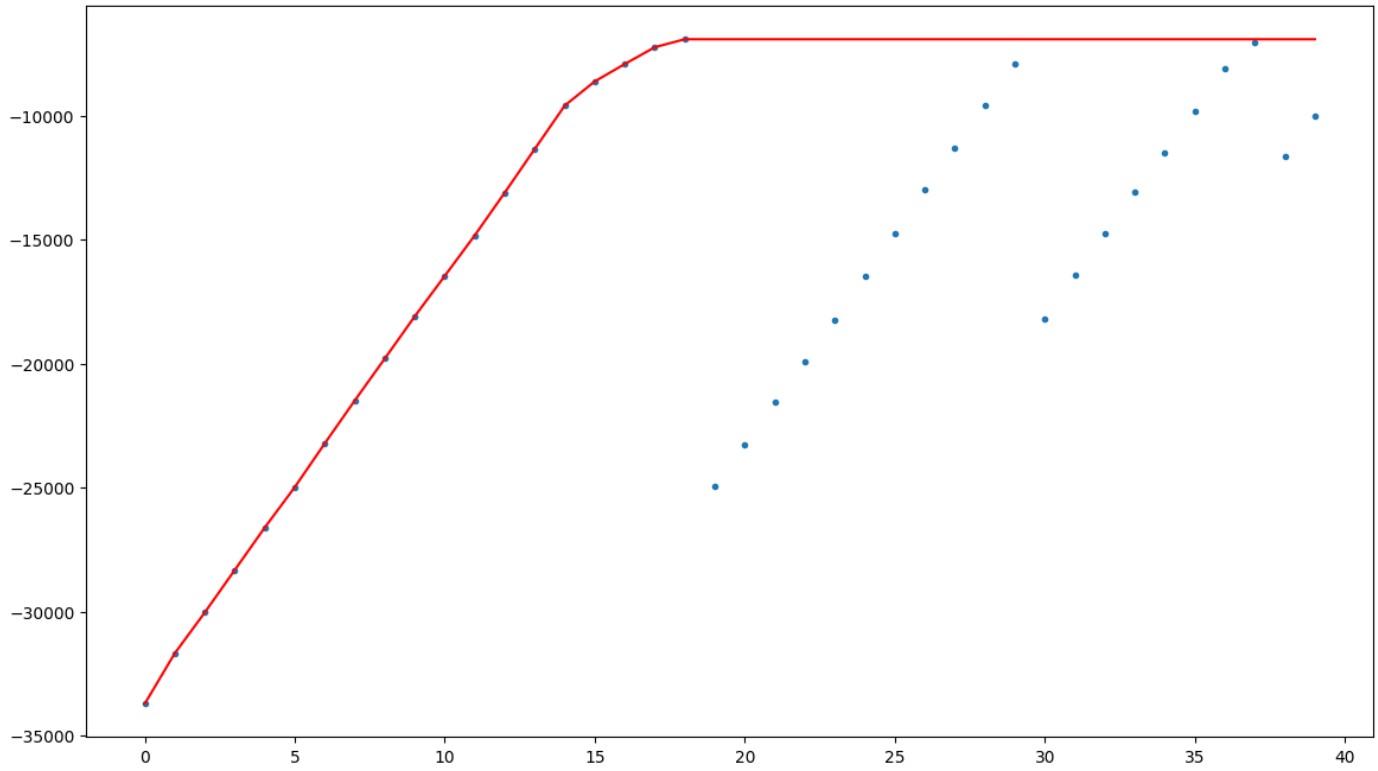
```
def resolve(uni_size,nsets,dens):

    ### generate sets and costs, this party is really slow
    sets = np.random.random((nsets, uni_size)) < dens
    for s in range(uni_size):
        if not np.any(sets[:, s]):
            sets[np.random.randint(nsets), s] = True
    costs = np.pow(sets.sum(axis=1), 1.1)
    ###


    current_solution =rng.random(nsets)  #inizialize current solution




    max_steps=uni_size//10  # no significant improvement in increasing

    n_restart=5 # slightly improves results


    history = []


    for nres in tqdm(range(n_restart)):
        i=0
        num_true=int((100-(nres)*(100/n_restart))*nsets/100)


        current_solution = np.zeros(nsets, dtype=bool)  # Initialize with all False
        true_indices = rng.choice(nsets, num_true, replace=False)  # Randomly pick indices to set to True
        current_solution[true_indices] = True  # Set those indices to True

        current_fit=fitness2(current_solution,costs,sets)
        if(nres==0):  #the first solution is always right because it's all true
            defsol=current_solution[:]
            defit=fitness2(defsol,costs,sets)
            history.append(defit[1])
        for step in range(max_steps):
            i+=1  #counts steps until the last improvement
            if(i%(3*max_steps//10)==0 ):  # if it's stuck for the 30% of steps , i try to use tweak 6
                solution = tweak6(current_solution)
                i=0
                current_solution=solution[:]
                current_fit=fitness2(solution,costs,sets)
            else:
                solution = tweak5(current_solution,step,max_steps,current_fit[0],uni_size)

            sol_fit=fitness2(solution,costs,sets)

            if sol_fit[0] and sol_fit[1]>current_fit[1] :  # if it's valid and it's better then the previus current
solution , update teh current solution
                history.append(sol_fit[1])
                current_solution = solution
                current_fit=sol_fit
                i=0
                if sol_fit[1]>defit[1]:  #if it's better then the previus global solution  update global solution with
the solution
                    defsol = solution[:]
                    defit=sol_fit

    finalcost=cost(defsol,costs)
    ic(finalcost,valid(defsol,sets))  #show cost and if it's valid or not
    ic(history.index(defit[1])) #last improvement index



    plt.figure(figsize=(14, 8))
    plt.plot(
        range(len(history)),
        list(accumulate(history, max)),
        color="red",
    )
    _ = plt.scatter(range(len(history)), history, marker=".")
    #plt.ylim(-100167,-170167)


    with open('log.txt', 'a') as file:   # log to track how well the algorithm performs
```

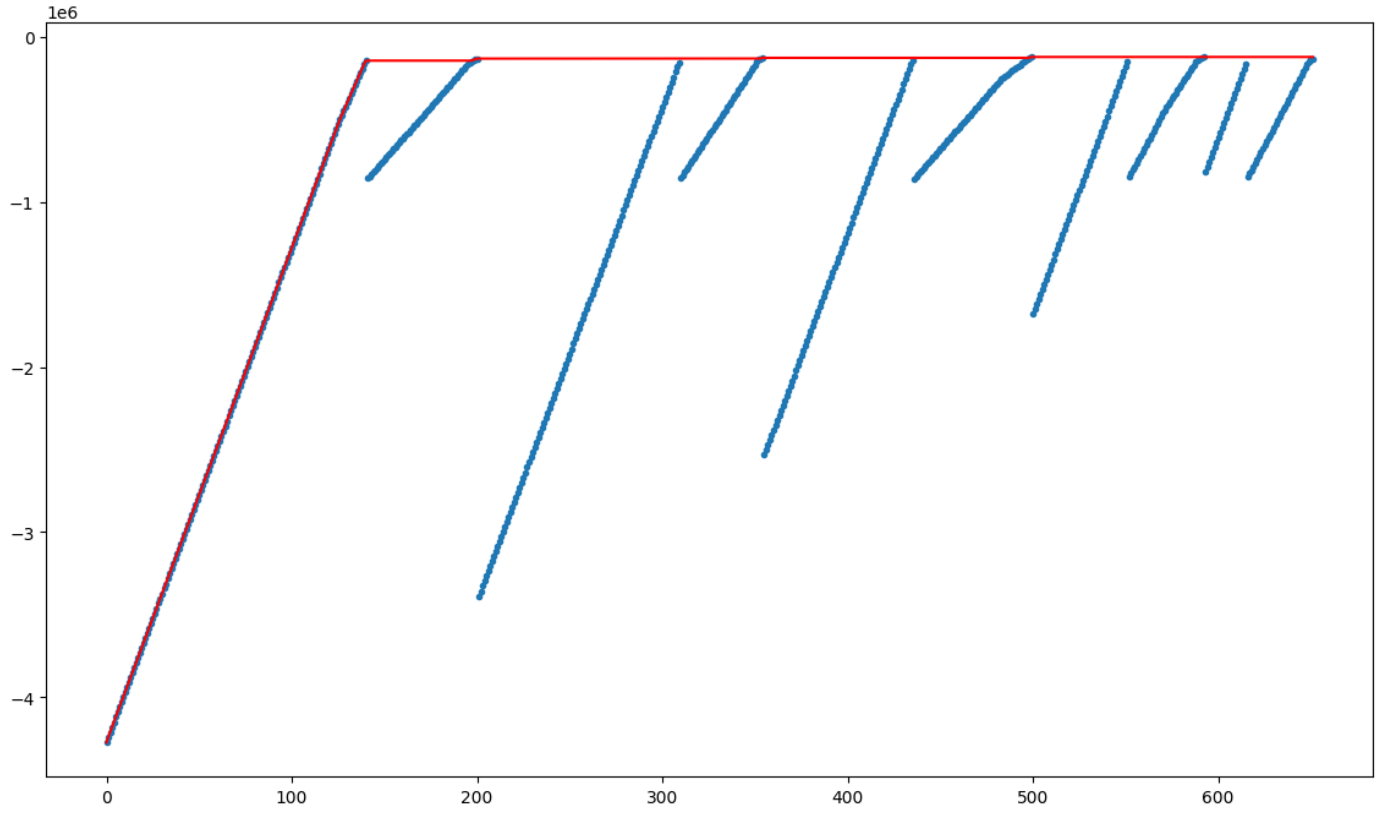
```
        file.write(f"{uni_size} , {nsets} , {dens} , {max_steps} , {n_restart} , {finalcost} \n")
    return finalcost




for test in listTest:
    NCALLS=0
    resolve(test[0],test[1],test[2])
    print(NCALLS)
#pythonoptimize=1     and then start program
```

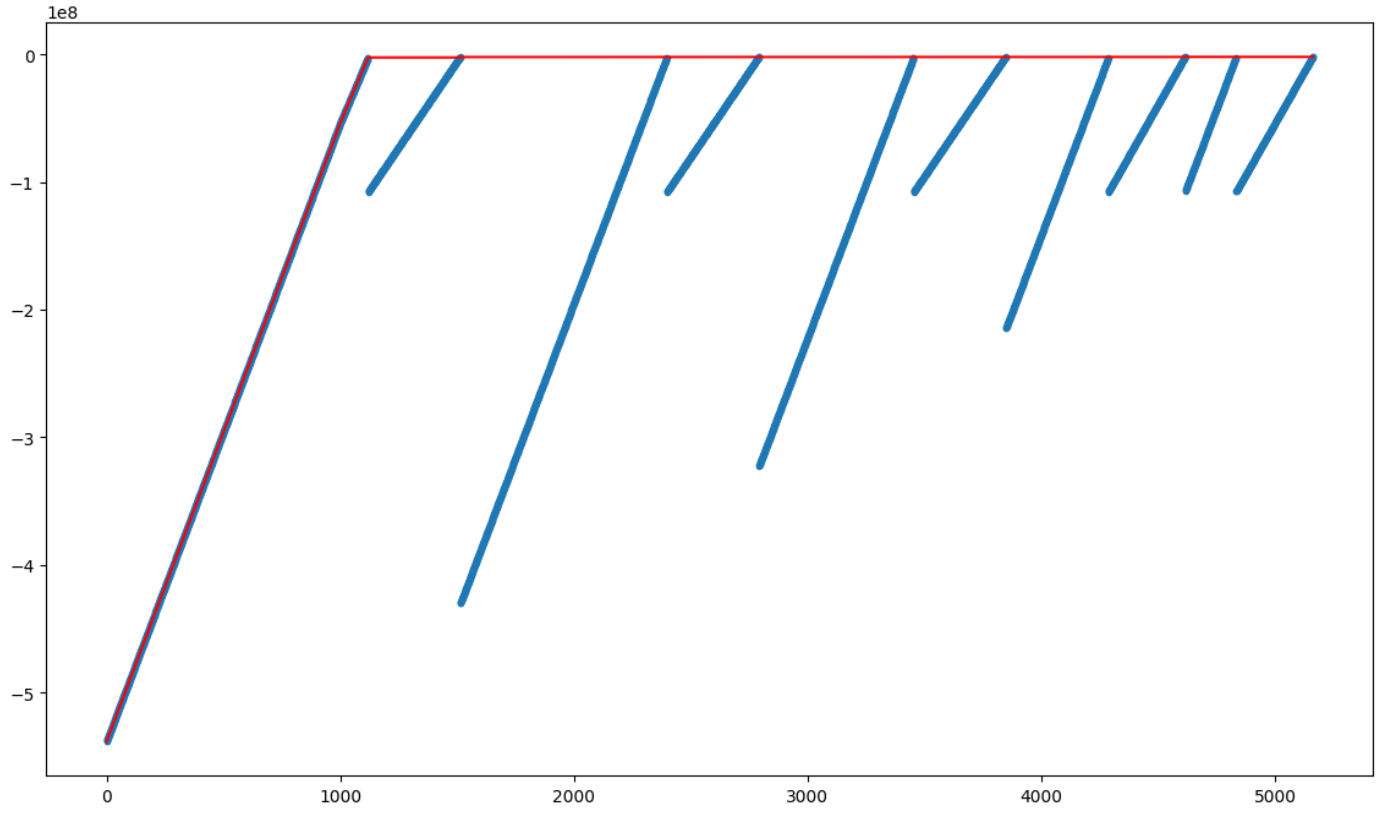
```
  0%|          | 0/5 [00:00<?, ?it/s]


ic| finalcost: np.float64(291.08030135804086)
    valid(defsol,sets): np.True_
ic| history.index(defit[1]): 0


71



  0%|          | 0/5 [00:00<?, ?it/s]


ic| finalcost: np.float64(6921.1793577301)
    valid(defsol,sets): np.True_
ic| history.index(defit[1]): 18


516



  0%|          | 0/5 [00:00<?, ?it/s]


ic| finalcost: np.float64(122827.22872652378)
    valid(defsol,sets): np.True_
ic| history.index(defit[1]): 499


5011



  0%|          | 0/5 [00:00<?, ?it/s]


ic| finalcost: np.float64(2044864.9450448642)
    valid(defsol,sets): np.True_
ic| history.index(defit[1]): 4615


50011



  0%|          | 0/5 [00:00<?, ?it/s]


ic| finalcost: np.float64(1884422.0296513021)
    valid(defsol,sets): np.True_
ic| history.index(defit[1]): 4364


50011



  0%|          | 0/5 [00:00<?, ?it/s]
```

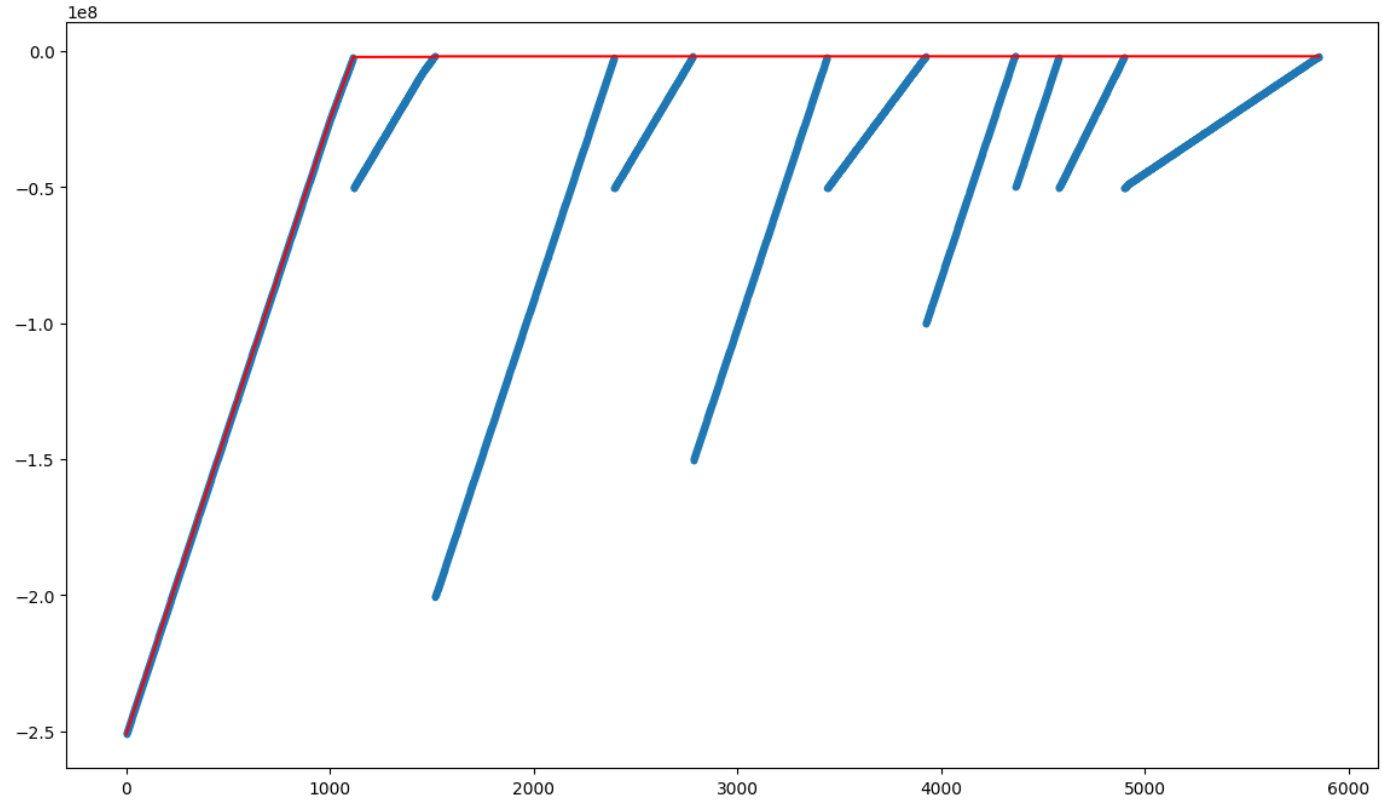
```
ic| finalcost: np.float64(2104142.4876685557)
    valid(defsol,sets): np.True_
ic| history.index(defit[1]): 5319


50014
```

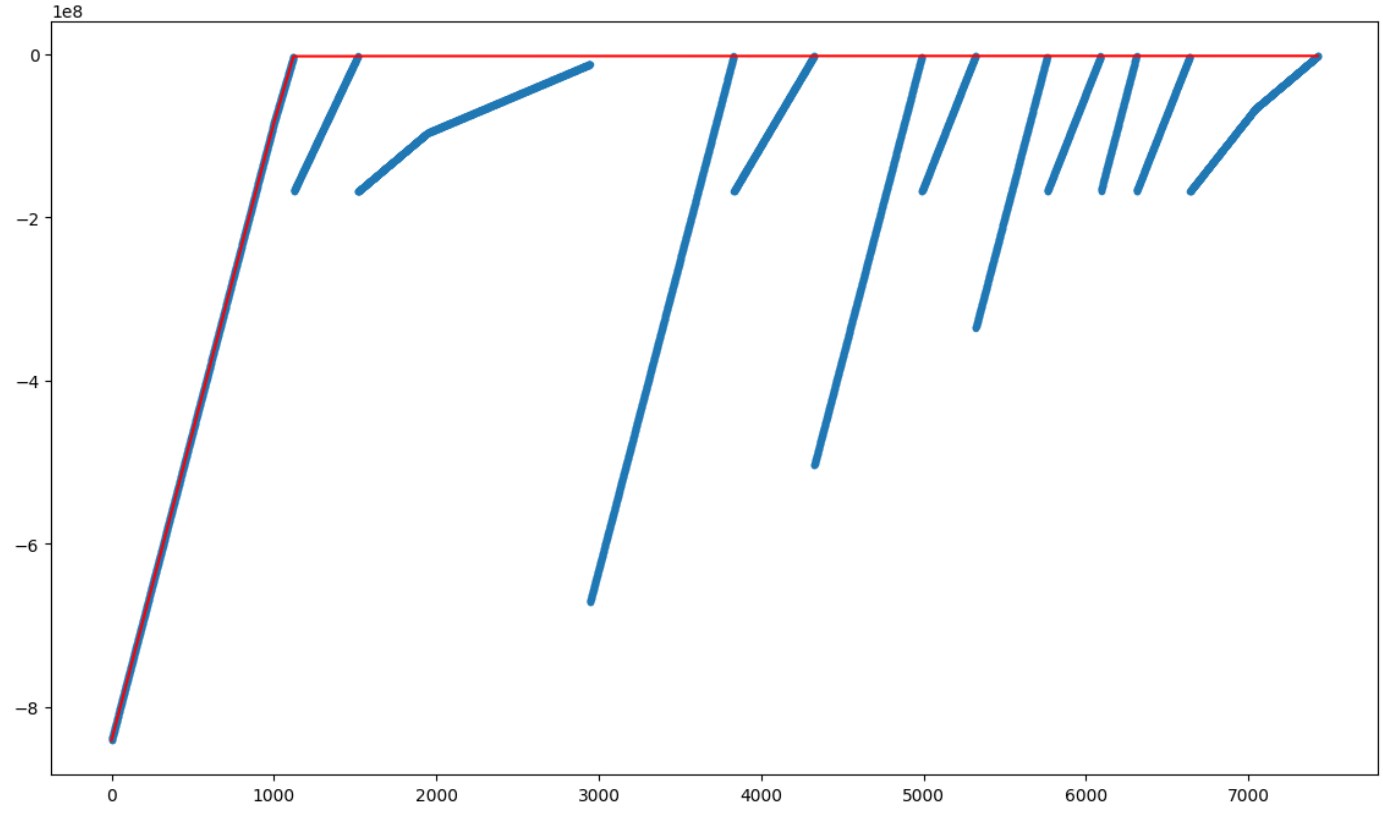





1e6
0
−1
−2
−3
−4
0
100
200
300
400
500
600


1e8
0
−1
−2
−3
−4
−5
0
1000
2000
3000
4000
5000





Results

Universe Size	Number of Sets	Density	Cost	Number of Times I Call Fitness Function
100	10	0.2	277.6620635309684	71
1000	100	0.2	7510.8060872504675	518
10000	1000	0.2	123795.99497518649	5010
100000	10000	0.1	2102128.359116447	50012
100000	10000	0.2	1910697.4551600877	50011
100000	10000	0.3	2183831.17137303	50014

Reviews1

I also reviewed and gave advice about my colleagues' solutions ,before giving a possible advice i tested on their code if it would work too:

About carlopantax solution:

> "Great job! Your algorithm is a correct implementation of simulated annealing for the set cover problem.
>
> Here are some suggestions for further improving: You can use a different function to check if a solution is valid or not . Iterate over each taken element of the solution and stop early if you see that all sets are covered, you can slash execution time this way. (Although that doesn't improve number of calls to fitness function or lower the minimum cost found, it's useful to do faster test in future). Looking at your graphs, it looks like that for small universe sizes your function gets rapidly to a plateaux , for big ones it never gets there (as you also correctly observed in README) . A way to change that is by changing maximum number of steps(max_iter) using the universe size information (bigger universe, more steps and smaller universe, less steps) and dropping the early exit for low temperature. Only decreasing the min_temp stop condition in the last three cases(as you suggested in the README) won't work because you'll get anyway to the maximum number of steps(max_iter=20_000) ."

About michepaolo solution:

> Great job! Your algorithm is a correct implementation of a SELF-ADAPTIVE algorithm for the set cover problem. Your project could benefit from better documentation , for example by adding some comments that explain what you are doing. A possible improvement is to avoid always doing 10000 steps, on smaller universe size you get same results with less steps, an idea is to change max number of steps based on the universe size (smaller universe less steps , bigger ones more steps).

Lab2

My LAB_2

In this laboratory i tried to solve travel salesman problem using EA . I spent a lot of time trying to find a better greedy algorithm then the default one (the one that build a path taking always the nearest city(called startgreedy2 in the code)).

My improved version of the greedy algorithm (startgreedy2) works differently. Instead of starting from a single point, it begins by selecting two endpoints: a starting city and the city farthest from it. For each of these endpoints, I identify their two nearest neighbors. The algorithm builds the path by repeatedly connecting to the closest neighbor. If the two growing branches of the path eventually meet from opposite ends. I restart the process using any remaining disconnected cities as new start and end points, and continue until all cities are included. Two branches meeting from the same side it's avoided otherwise there would be a cycle.

EAlgoithm creates initial population by using both greedy alghoritms , apply twice tournament selection to decide the two parents to crossover (using partially_mapped_crossover), and always mutate the child using inversion mutation.

The tsp doesn't improve when using EA alghoritm most of the times , sometimes it gets stuck too credits: partially_mapped_crossover is an implementation from ruta-tamosiunaite

my sol

```

```

```
import pandas as pd
from icecream import ic
import numpy as np
import random
from itertools import combinations
from geopy.distance import geodesic
from dataclasses import dataclass
#from itertools import iterrows
from matplotlib import pyplot as plt
from itertools import accumulate
from tqdm.auto import tqdm

cities= pd.read_csv('cities/vanuatu.csv', header=None, names=['name', 'lat', 'lon'])
cities
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	name	lat	lon
0	Isangel	-19.53	169.28
1	Lakatoro	-16.09	167.40
2	Longana	-15.30	168.00
3	Luganville	-15.51	167.15
4	Norsup	-16.07	167.39
5	Port Olry	-15.05	167.05
6	Sola	-13.87	167.55
7	Vila	-17.74	168.31

```

```

```
DIST_MATRIX= np.zeros((len(cities), len(cities)))
for c1, c2 in combinations(cities.itertuples(), 2):
    DIST_MATRIX[c1.Index, c2.Index] = DIST_MATRIX[c2.Index, c1.Index] = geodesic(
        (c1.lat, c1.lon), (c2.lat, c2.lon)
    ).km


for c1 in range(len(cities)):
    DIST_MATRIX[c1,c1]=np.inf



masked_dist_matrix = np.ma.masked_equal(DIST_MATRIX, np.inf)
min_index = np.ma.argmin(masked_dist_matrix[0])



#np.argmin(DIST_MATRIX[0])

cities.head()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	name	lat	lon
0	Isangel	-19.53	169.28
1	Lakatoro	-16.09	167.40
2	Longana	-15.30	168.00
3	Luganville	-15.51	167.15
4	Norsup	-16.07	167.39

```
def tsp_cost(tsp):
    assert tsp[0] == tsp[-1]
    assert set(tsp) == set(range(len(cities)))

    tot_cost = 0
    for c1, c2 in zip(tsp, tsp[1:]):
        tot_cost += DIST_MATRIX[c1, c2]
    return tot_cost
```

```

```

```
def startgreedy(start):   #greedy approach that takes always the nearest city
   giro=[]
   costi=[]
   startpoint=globalstart=posto= start #np.int64(np.random.randint(len(cities)))
   giro.append(startpoint)
   copydist=DIST_MATRIX.copy()
   distances = DIST_MATRIX.copy()
   distances[:, posto] = np.inf

   for i in range(len(cities)):
      if(i==(len(cities))-1):
         posto=globalstart
      else:
         posto=np.argmin(distances[startpoint])

      costo=copydist[posto][startpoint]
      costi.append(costo)

      giro.append(posto)
      distances[:, posto] = np.inf
      startpoint=posto



   return(giro)
```

```
def startexp3(startpoint,distances,ended,near,giro):   #trying to start in 2 different point so that alghorithm will
up in the middle
    nearest_index=0
    if(ended==True):
        nearest_index=2

    second_near_index=nearest_index+1
    near[nearest_index]=np.ma.argmin(distances[startpoint]) #nearest
    distances[startpoint,near[nearest_index]] = np.ma.masked  #nearest to origin
    distances[near[nearest_index],startpoint] = np.ma.masked
    near[second_near_index]=np.ma.argmin(distances[:,startpoint])



    distances[startpoint] = np.ma.masked
    distances[:, startpoint] = np.ma.masked
    distances[startpoint, near[second_near_index]] = np.ma.masked
    distances[near[second_near_index], startpoint] = np.ma.masked
    giro[near[second_near_index]] = startpoint
    giro[startpoint] = near[nearest_index]
    distances[near[second_near_index]]=np.ma.masked  # none can start from the second near
    distances[:,near[nearest_index]]=np.ma.masked  # none can get to the  nearest
```

```
def findNearToNear(near,distances,giro):   #each open end of the path looks for the nearest element
   togo=0
   count =len(giro)
   for e in giro:
      if e ==-1:
         togo+=1
   blocked=[]
   startvalue=-1-len(giro)

   neartonear=[startvalue,startvalue,startvalue,startvalue]
   n=0
```

```
    while n<3:
       onlybranch=0
       count=0
       if(n==2):
          count = np.count_nonzero(giro != -1)
       if(near[0+n]>=0 and near[1+n] >=0 ):

          distances[near[0+n],near[1+n]] = distances[near[1+n],near[0+n]]= np.ma.masked

       if near[0+n] >=0 and (not distances[near[0+n]].mask.all()):

          neartonear[0+n]=np.ma.argmin(distances[near[0+n]])


       else:
          onlybranch+=1


       if   near[1+n] >=0 and  not distances[:,near[1+n]].mask.all():
          neartonear[1+n]=np.ma.argmin(distances[:,near[1+n]])

       else:
          onlybranch+=2

       val=0
       if(onlybranch==0 and neartonear[1+n]==neartonear[n]):
          dist2=distances[near[1+n],neartonear[1+n]]
          dist1=distances[near[n],neartonear[n]]
          if(dist2>dist1):
             val=0
          else:
             val=1
       if(onlybranch!= 3 and onlybranch!=1):
          if (distances.mask[neartonear[n+val],near[n+val]]==False):

             giro[neartonear[n+val]]=near[n+val]
             if (checkCycle(giro,neartonear[val+n])==True):
                giro[neartonear[val+n]]=-1
                distances[neartonear[val+n],near[val+n]]=np.ma.masked
                neartonear[val+n]=np.ma.argmin(distances[:,near[val+n]])
                giro[neartonear[val+n]]=near[val+n]


             distances[neartonear[n+val]]=np.ma.masked
             distances[:,near[n+val]]=np.ma.masked

          elif (distances.mask[near[n+val],neartonear[n+val]]==False):
             giro[near[n+val]]=neartonear[n+val]
             if (checkCycle(giro,near[val+n])==True):
                giro[near[val+n]]=-1

                distances[near[val+n],neartonear[val+n]]=np.ma.masked
                neartonear[val+n]=np.ma.argmin(distances[near[val+n]])
                giro[near[val+n]]=neartonear[val+n]


             distances[near[n+val]]=np.ma.masked
             distances[:,neartonear[n+val]]=np.ma.masked




          distances[neartonear[n+val],near[n+val]]=np.ma.masked
          distances[near[n+val],neartonear[n+val]]=np.ma.masked

       if(onlybranch==0 and neartonear[1+n]==neartonear[n]):
          if not distances[near[1-val+n]].mask.all():
             neartonear[1-val+n]=np.ma.argmin(distances[near[1-val+n]])

       if(onlybranch!= 3 and onlybranch!=2):
          if (distances.mask[neartonear[1-val+n],near[1-val+n]]==False):

             giro[neartonear[1-val+n]]=near[1-val+n]

             if (checkCycle(giro,neartonear[1-val+n])==True):
                giro[neartonear[1-val+n]]=-1
                distances[neartonear[1-val+n],near[1-val+n]]=np.ma.masked
                neartonear[1-val+n]=np.ma.argmin(distances[:,near[1-val+n]])
                giro[neartonear[1-val+n]]=near[1-val+n]
```

```
            distances[neartonear[1-val+n]]=np.ma.masked
            distances[:,near[1-val+n]]=np.ma.masked

         elif (distances.mask[near[1-val+n],neartonear[1-val+n]]==False):

            giro[near[1-val+n]]=neartonear[1-val+n]
            if (checkCycle(giro,near[1-val+n])==True):
               giro[near[1-val+n]]=-1
               distances[near[1-val+n],neartonear[1-val+n]]=np.ma.masked
               neartonear[1-val+n]=np.ma.argmin(distances[near[1-val+n]])
               giro[near[1-val+n]]=neartonear[1-val+n]


            distances[near[1-val+n]]=np.ma.masked
            distances[:,neartonear[1-val+n]]=np.ma.masked

         distances[neartonear[1-val+n],near[1-val+n]]=np.ma.masked
         distances[near[1-val+n],neartonear[1-val+n]]=np.ma.masked




      n+=2

   return neartonear



def checkCycle(giro,start):  # check if there s a cycle , if there is a cycle and we haven't covered all vertex gives
true otherwise gives false
   togo=0
   step=giro[start]
   i=0
   while (step!=start and step>=0):
      i+=1
      step=giro[step]
      if step==-1:
         return False

   if(i<len(giro)-1):
      return True

   return False






def startgreedy2(start=-1):   # an other greedy approch , that down't work well ( but for small populations it's
better then the other one )
   giro=[-1] * len(cities)

   distances = np.ma.masked_equal(DIST_MATRIX, np.inf)
   start_flat= np.argmax(distances)
   if( start==-1): #select as start and end the 2 furthest elements
      start_2d = np.unravel_index(start_flat, distances.shape)
      start=start_2d[0]
      end=start_2d[1]
   else:   #select as  start the given one and as end the furthest from start
      end=np.argmax(distances[start])


   #np.argmax(distances[start])

   startpoint=start #np.int64(np.random.randint(len(cities)))



   startvalue=-len(cities)
   near=[startvalue,startvalue,startvalue,startvalue]


   #end=np.ma.argmax(distances[startpoint])


   startexp3(startpoint,distances,False,near,giro)
```

```
   startexp3(end,distances,True,near,giro)

   count =giro.count(-1)
   while count >0:
      neartonear=findNearToNear(near,distances,giro)
      near=neartonear.copy()


      count =giro.count(-1)

   return giro
```

```
def devround(path,start):   #changes way in which the path is displayed , so it's the same as the one done using
greedy1
    newpath=[]
    newpath.append(start)
    for e in range(len(path)):
        start=path[start]
        newpath.append(start)


    return newpath
```

Greedy part Here i show the results of both greedy approches used, for more cities startgreedy would perform better then startgreedy2

```
totalcost=0

totalcostbasic=0

for i in range(1): #greedy approch starting from 1

    totalcost+=tsp_cost(devround(startgreedy2(i),i))
    totalcostbasic+=tsp_cost(startgreedy(i))

ic(totalcost,totalcostbasic)
```

```
ic| totalcost: np.float64(1353.0818278087431)
    totalcostbasic: np.float64(1475.528091104531)



(np.float64(1353.0818278087431), np.float64(1475.528091104531))
```

Evolutionary part

```
def tournament_sel(start,n=2): # tournament selection for deciding the nearest element    it's unused


    n = min(n, DIST_MATRIX.shape[1])

    # Choose n elements at random from the specified row
    random_elements_indexes = np.random.choice(DIST_MATRIX.shape[1], size=n, replace=True)
    random_elements_values= DIST_MATRIX[start][random_elements_indexes]



    bestIndex=np.argmin(random_elements_values)
    bestValue=random_elements_values[bestIndex]

    chosen=random_elements_indexes[bestIndex]

    return (start,chosen)
```

```
def tournament_sel_array(population,n=2): # tournament selection to decide which individual to crossover



    # Choose n elements at random from the specified row
    fitness_list=[]
    random_elements_indexes = np.random.choice(len(population), size=n, replace=True)
    for el in random_elements_indexes:
        fitness_list.append((population[el]).fitness)


    bestIndex=np.argmin(fitness_list)
    bestValue=fitness_list[bestIndex]

    chosen=population[random_elements_indexes[bestIndex]]

    return chosen




tournament_sel(0)
```

```
(0, np.int64(2))
```

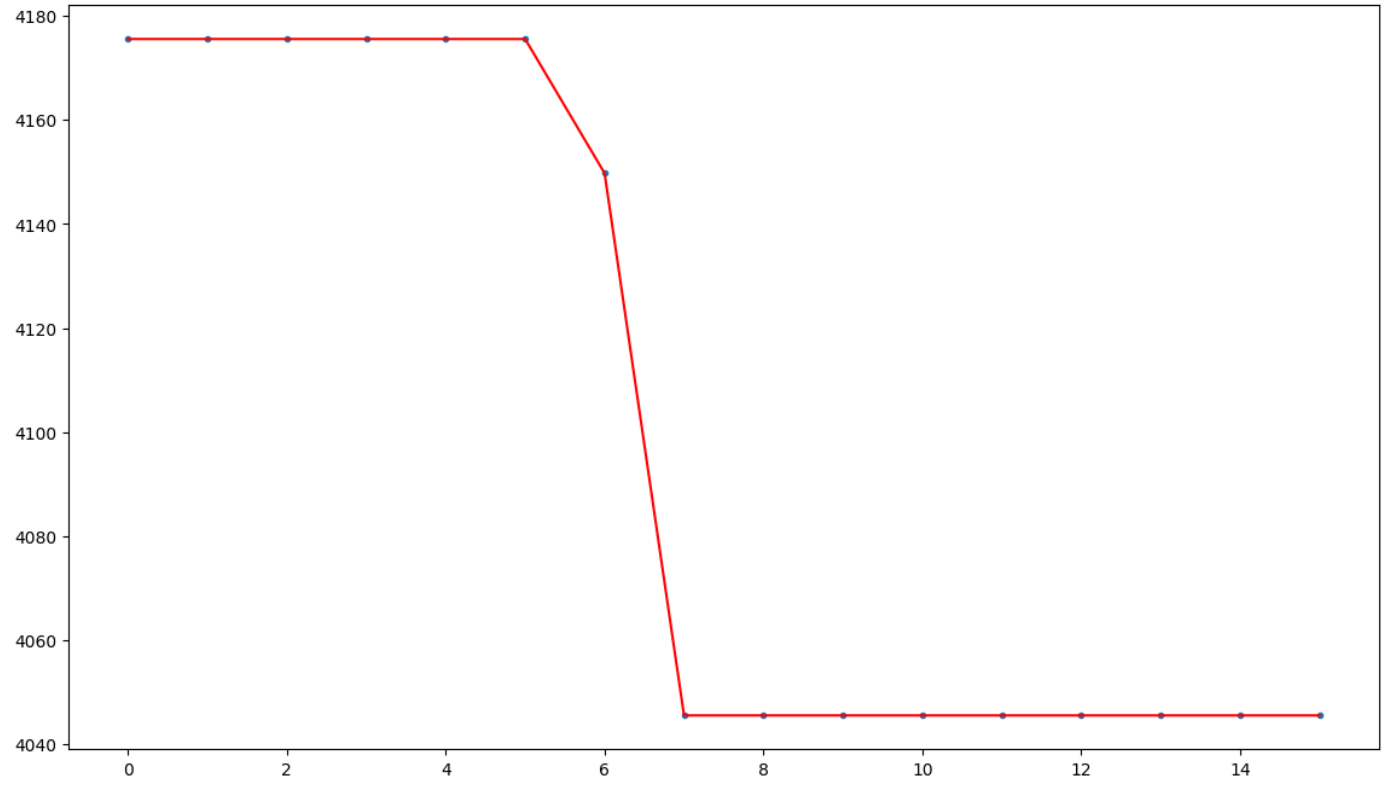
```
def partially_mapped_crossover(parent1, parent2):
    size = len(parent1)
    # Step 1: Select crossover range at random
    start, end = sorted(random.sample(range(1, size - 2), 2))  # Avoid the first and last gene (the hive) (Last
element of the list is (length - 1). Thus, it is (length - 2) to avoid the last gene)

    # Step 2: Create offspring by exchanging the selected range
    child1 = parent1[:start] + parent2[start:end] + parent1[end:]
    child2 = parent2[:start] + parent1[start:end] + parent2[end:]

    # Step 3: Determine the mapping relationship to legalize offspring
    mapping1 = {parent2[i]: parent1[i] for i in range(start, end)}
    mapping2 = {parent1[i]: parent2[i] for i in range(start, end)}

    # Step 4: Legalize children with the mapping relationship
    for i in list(range(start)) + list(range(end, size)):
        if child1[i] in mapping1:
            while child1[i] in mapping1:
                child1[i] = mapping1[child1[i]]
        if child2[i] in mapping2:
            while child2[i] in mapping2:
                child2[i] = mapping2[child2[i]]

    if random.random() < 0.5:

            return child1
    else:
            return child2
```

```
@dataclass
class Individual:
    genome: np.ndarray
    fitness: float = None




def fitness(individual):  # lower fitness the better
```

```
    fit=0
    highvalue=300
    if(individual[0]!=individual[1]):
        fit+=len(individual)*highvalue

    if (set(individual) != set(range(len(cities)))):
        fit+=len(individual)*highvalue

    tot_cost = 0
    for c1, c2 in zip(individual, individual[1:]):
        fit += DIST_MATRIX[c1, c2]



    return fit

def popInizialize(): # inizialise the population using both tried greedy alghorithms
    pop=[]


    dim=DIST_MATRIX.shape[1]
    nppl1=dim//3
    random_integers = np.random.choice(dim, size=nppl1, replace=False)
    for el in random_integers:
        pop.append(Individual(genome=startgreedy(el)))


    nppl2=dim//8
    random_integers = np.random.choice(dim, size=nppl2, replace=False)

    for el in random_integers:
        pop.append(Individual(genome=devround(startgreedy2(el),el)))


    for el in pop:
        el.fitness=fitness(el.genome)



    return pop

def inversion_mut(individual):   #Inversion Mutation implemntation

    new_path = individual.genome.copy()

    # Randomly select two points
    idx1, idx2 = sorted(np.random.choice(len(new_path), 2, replace=False))
    # Invert the segment between idx1 and idx2
    new_path[idx1:idx2+1] = new_path[idx1:idx2+1][::-1]


    return Individual(genome=new_path,fitness=fitness(new_path))




history=[]

def EAlgoithm():

    mutrate=1
    pop=popInizialize()
    nstep=len(cities)*2
    lowest_fitness_individual = min(pop, key=lambda x: x.fitness)
    #ic(lowest_fitness_individual.fitness)

    for el in tqdm(range(nstep)):

        if min(pop, key=lambda x: x.fitness).fitness==max(pop, key=lambda ind: ind.fitness).fitness:  #reached steady
state
            break
        ris1=tournament_sel_array(pop,2)   # uses tournament selection to select 2 parent for the crossover
        ris2=tournament_sel_array(pop,2)


        child = partially_mapped_crossover(ris1.genome,ris2.genome)  # uses partially mapped crossover
        child=Individual(genome=child,fitness=fitness(child))

        if( np.random.rand()<mutrate):  # has a chance of mutating the child using inversion mutation
```

```
            child=inversion_mut(child)



        pop.append(child)  # add child to population



        lowest_fitness_individual = min(pop, key=lambda x: x.fitness)
        #ic(lowest_fitness_individual.fitness)
        history.append(lowest_fitness_individual.fitness)

        maxfit_ind=(max(pop, key=lambda ind: ind.fitness))
        #ic(maxfit_ind.fitness)
        pop.remove(maxfit_ind) # delete from population the wrost individual


    return pop


pop=EAlgoithm()
minfit=min(pop, key=lambda x: x.fitness)
ic(minfit)


plt.figure(figsize=(14, 8))
plt.plot(
        range(len(history)),
        list(accumulate(history, min)),
        color="red",
    )
_ = plt.scatter(range(len(history)), history, marker=".")
```

```
  0%|          | 0/16 [00:00<?, ?it/s]


ic| minfit: Individual(genome=[np.int64(7),
                               np.int64(1),
                               np.int64(4),
                               np.int64(3),
                               np.int64(5),
                               np.int64(6),
                               np.int64(2),
                               np.int64(0),
                               np.int64(7)],
                      fitness=np.float64(4045.5449564733112))
```



Reviews2

I also reviewed and gave advice about my colleagues' solutions ,before giving a possible advice i cloned their repository and tried their code:

About Pios15 solution:

> Sorry , your Evolutionary Algorithm doesn't work , there is no improvement after the greedy step, the first elite found it's always the best one , it gets stuck in a local minimum (the best greedy generated). Nice try anyway ,Your greedy algorithm works pretty well The main problem of your program is that your inver over function, takes genes only from first parent (so you have no actual crossover) , you should implement a proper inver_over crossover to achieve some improvement over the greedy approach . the function inver_over you implemented is not really an inver_over crossover function , it implements a swap mutation half the times and a reverse mutation the other half, it can be adapted to be your mutate function in this way though : def mutate(path):
>
> ```
> if random.random() < mutation_rate:
>     i, j = random.sample(range(1, len(path) - 1), 2)
>
>     if random.random() < 0.5:
>             path[i:j+1] = reversed(path[i:j+1])
>     else:
>             path[i], path[j] = path[j], path[i]
>
> return path
> ```
>
> An other idea worth exploring is to generate more candidate individuals for your population and then keeping only top population_size individuals in your population. In this way you should get more selective pressure An other minor thing I would suggest is when using greedy heuristic to decide the half of the starting population, it may happen that the same path is generated more than once due to starting from the same point twice with the same nearest neighbor algorithm. To improve diversity it would be better to avoid adding duplicate paths by keeping track of starting points already selected and avoiding starting from them.

About GiovanniTagliaferri solution:

> Congrats! You implemented a working evolutionary algorithm! Your code is easily readable and well-explained. I have only a few minor comments about it. Applying elitism and mutating the elite can lead to losing the best individuals in your population. I would suggest not mutating the elite . We are mostly interested in the vicinity trait between cities, scramble mutation isn't ideal for preserving that trait. You could try to implement also reverse mutation to better maintain that trait. Using an adaptive mutation rate that decreases with each generation could also be beneficial, making your algorithm more focused on exploitation rather than exploration toward the end of the evolution cycle.

Lab3

My LAB3

thanks to Emanumele Carelli for the help , he gave me the the base fuction to calculate linear conflict distance

To solve the n-puzzle problem , it's implemented A* alghorithm using as heuristic for the cost the linear conflict distance.

Sometimes it took to long to run and vscode would crash before a good solution was found. To impove that for big puzzle , the alghorithm first finds the moves to get to first row(or column) in current state equal to the solution , then it resizes the problem by not moving in that row (or column) anymore.

As you can see in the table this Firstcut alghorithm maneged to find the solution where normal A* didn't . Where both alghoritm worked , traditional A* found a solution with less moves but with more evaluated states then using Firstcut .

Cost is the number of evaluated states and Quality is the number of moves needed to solve the puzzle.

results

Dimension	Number of Random Steps	Seed	Firstcut	Quality	Cost
3x3	100003	43242	no	24	679
4x4	100003	43242	yes	69	230221
4x4	100003	43242	no	x	x
4x4	100003	4	no	47	1003783
4x4	100003	4	yes	51	63216

x = VSCode crashed before finding a solution

With higher dimensions , it works only if Number of Random Steps is low . It also works for non quadratic puzzles for examples 4x3 or 3x4.

my sol

Contains commented code telling what's is doing

```
from collections import namedtuple
from random import choice,seed,randint
from tqdm.auto import tqdm
import numpy as np
from icecream import ic
import heapq
import math


SEED = 43242
#SEED = 4
#SEED= randint(0, 2**32 - 1)
seed(SEED)
ic(SEED)
```

```
ic| SEED: 43242




43242
```

```
PUZZLE_DIM = 4
FIRSTCUT=1
action = namedtuple('Action', ['pos1', 'pos2'])

objective=[]
Y=PUZZLE_DIM
X=PUZZLE_DIM
for i in range(Y):
    locobjective=[]
    for k in range(X):
        locobjective.append(k+1+i*X)

    objective.append(locobjective)
objective[-1][-1]=0


objective=np.array(objective, dtype=np.int8)
objective
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15,  0]], dtype=int8)
```

```
def available_actions(state: np.ndarray,width,hight) -> list['Action']:
    x, y = [int(_[0]) for _ in np.where(state == 0)]
    actions = list()
    if x > 0:
        actions.append(action((x, y), (x - 1, y)))
    if x < hight - 1:
        actions.append(action((x, y), (x + 1, y)))
    if y > 0:
        actions.append(action((x, y), (x, y - 1)))
    if y < width - 1:
        actions.append(action((x, y), (x, y + 1)))
    return actions



def do_action(state: np.ndarray, action: 'Action') -> np.ndarray:
    new_state = state.copy()
    new_state[action.pos1], new_state[action.pos2] = new_state[action.pos2], new_state[action.pos1]
    return new_state
```

```
RANDOMIZE_STEPS =100003

state = np.empty_like(objective)
solfinal=np.empty_like(objective)
k=0
state=objective[:]
for r in tqdm(range(RANDOMIZE_STEPS), desc='Randomizing'):
    state = do_action(state, choice(available_actions(state,X,Y)))
    k+=1
state
```

```
Randomizing:   0%|          | 0/100003 [00:00<?, ?it/s]
```

```
array([[15, 11, 14, 12],
       [ 2,  7,  6,  5],
       [13,  0, 10,  9],
       [ 8,  4,  1,  3]], dtype=int8)
```

```
ic(state)
```

```
def linear_conflict_distance_fast(curr, goal,y,x,start=0,xstart=0,ystart=0):  #start select the modality , if start =0 it's regular   linear_conflict_distance , otherwise it calculate distances only from first row and column of the objective fuction
    distance = 0
    linear_conflict = 0
    goodfirstline=[]
    if(start==1):
        for i in goal[0,:]:  #goodfirst line must contain first row and first column elements , so if i have a big puzzle i try first only to complete first row and first column
            goodfirstline.append(i)
        for j in goal[1:,0]:

            goodfirstline.append(j)
```

```
    for i in range(y):
        for j in range(x):
          #start use is explained above
            if (start==1 and curr[i][j] in goodfirstline) or (curr[i][j] != 0 and start==0):

                goal_i, goal_j = np.where(goal == curr[i][j])   #  Manhattan distance
                distance += abs(goal_i - i) + abs(goal_j - j)
                # Check for linear conflicts in the row
                if goal_i == i:
                    for k in range(j + 1, x):
                        if curr[i][k] != 0:
                            goal_k_i, goal_k_j = np.where(goal == curr[i][k])
                            if goal_k_i == i and goal_j > goal_k_j:
                                linear_conflict += 2  # if we have a conflict we must do at least 2 moves 1 to
resiolve conflict and one resolve conflict , one to put the second tile in correct position
                # same as above but for col
                if goal_j == j:
                    for k in range(i + 1, y):  # Compare with other tiles in the column
                        if curr[k][j] != 0:
                            goal_k_i, goal_k_j = np.where(goal == curr[k][j])
                            if goal_k_j == j and goal_i > goal_k_i:
                                linear_conflict += 2

    return (distance[0] + linear_conflict)


def discard_n_worst(open_list, n):  #not used currently, useful to avoid getting your pc unresponsive for too much
memory use , it can greatly decrease perfonces and someyimes it never reaches a solution
    if n <= 0:
        return open_list
    if len(open_list) <= n:
        return []


    all_elements = sorted(open_list, key=lambda x: x[0])  # Sort by `f` value

    remaining_elements = all_elements[:-n]  # Remove the last n elements

    # Rebuild the preority queue
    heapq.heapify(remaining_elements)
    return remaining_elements

def check_if_row_or_col_eq(state, objective):
    completed=(0,0)

    if np.array_equal(state[0], objective[0]): # check if first row it's equal to the solution
        completed=(completed[0]+1,completed[1])
    if np.array_equal(state[:,0], objective[:,0]): # check if first column it's equal to the solution
        completed=(completed[0],completed[1]+1)


    return completed


def astar(state,globobjective,distancefun):
    xsolfinal=0
    ysolfinal=0

    #maxsize=1000000  #maxsize of open_list if we are using discard_n_worst
    open_list = []

    hashable_state = tuple(tuple(arr) for arr in state)

    heapq.heappush(open_list, (0,hashable_state,0))  #push in the preority que , with heuristic cost , hashable state
and , real cost

    visited = set()
    target_size=math.factorial(X*Y) #at worst we have X*Y! states (it's a  simple disposition of X*Y elements)

    heuristic_cache = {}
    objective=globobjective[:]
    hight=Y
    w=X
    if(X>=4 and Y>=4):  #if we have a big table try first to complete first row or columnt , once one of those are
completed we search for a solutiin in a semplified state with that row or column cutoff
        firstcut=FIRSTCUT
```

```
    else:
        firstcut=0

    heuristic_cache[hashable_state] = distancefun(state, objective,hight,w,firstcut) # we save the heuristic distance
of hashable_state from the solution

    analized=0
    analized+=1  # number of actions evaluated




    open_list_dict = {}
    with tqdm(total=target_size, desc="Filling the set", unit="item") as pbar:

        lenopen=len(open_list)

        while  lenopen>0:   #

            h_cost,current_state,current_cost  = heapq.heappop(open_list)
            lenopen-=1
            if current_state in visited:

                continue

            listcurrentstate=np.array(current_state, dtype=np.int8)

            if(lenopen%100000==0): #print the state evry now and then
               ic(listcurrentstate)

            if (hight>3 and w>3 and firstcut==1):  #bif puzzle it's considered if it has height and width of more then
3


                to_del=check_if_row_or_col_eq(listcurrentstate,objective)
                if to_del!=(0,0):   #to_del different from (0,0) means that or the first row it's equal to the
solution or the first column

                    if(to_del[0]==1):

                            objective=objective[1:,:] #we cut the first row from the objective
                            solfinal[ysolfinal, xsolfinal:xsolfinal + listcurrentstate.shape[1]] = listcurrentstate[0,
:] #we save that row in the solution because it's already the right one
                            ysolfinal += 1

                            listcurrentstate=listcurrentstate[1:,:] #we cut the first row from the state , from this
point on we won't change that row anymore
                            hight=hight-1   #our new state has one less row

                    if(to_del[1]==1): #same as above but with columns

                            objective=objective[:,1:]
                            solfinal[ysolfinal:ysolfinal + listcurrentstate.shape[0], xsolfinal] = listcurrentstate[:,
0]
                            xsolfinal += 1

                            listcurrentstate=listcurrentstate[:,1:]
                            w-=1


                    ic("Modified shape!!!")  # if the shape got modified we can delete the support data of the previus
shape
                    open_list = []
                    hashable_state = tuple(tuple(arr) for arr in listcurrentstate)

                    visited = set()
                    lenopen=0
                    if (hight>3 and w>3):

                        firstcut=1
                    else:
                        firstcut=0
                    heuristic_cache = {}
                    heuristic_cache[hashable_state] = distancefun(listcurrentstate,
objective,hight,w,firstcut,xsolfinal,ysolfinal)
                    analized+=1
                    current_cost=current_cost
                    current_state=hashable_state

                    ic(listcurrentstate)
```

```
            if np.array_equal(listcurrentstate, objective) == True:  # we found the final solution
                ic(analized)

                solfinal[ysolfinal:ysolfinal + listcurrentstate.shape[0],xsolfinal: xsolfinal+
listcurrentstate.shape[1]] = listcurrentstate[:,:]  #recompose the solution adding to solfinal the current state ,
(previusly if we reshaped in solfinal there were cutoff rows and column)

                return solfinal, current_cost,analized



            """  if len(open_list) > maxsize//10: #not used currently, useful to avoid getting your pc unresponsive
for too much memory use , it can greatly decrease perfonces and someyimes it never reaches a solution
                open_list = discard_n_worst(open_list, 1)  """

            visited.add(current_state) #add the state to the visited set , that means it's no longer on the frontier

            actions=available_actions(listcurrentstate,w,hight)  #provides a list of avaible action(it works for a
puzzle nxm also)

            for ia in range(len(actions)):

                newstate=do_action(listcurrentstate,actions[ia])

                hashable_new_state = tuple(tuple(arr) for arr in newstate)
                if hashable_new_state not in visited:
                    g = current_cost + 1  # Assume cost per step is 1; adjust if needed

                    if hashable_new_state not in heuristic_cache:
                        heuristic_cache[hashable_new_state] = distancefun(newstate,
objective,hight,w,firstcut,xsolfinal,ysolfinal) #calc euristic cost and save it
                        analized+=1
                        pbar.update(1)  #tracks analized states
                    h = heuristic_cache[hashable_new_state]  #euristic cost
                    f = float(g + h)

                    if hashable_new_state in open_list_dict and open_list_dict[hashable_new_state] <= f:

                        continue  # Skip adding this state if there is already a better one

                    #otherwise update the best f value for this state
                    open_list_dict[hashable_new_state] = f

                    heapq.heappush(open_list, (f, hashable_new_state,g)) # add the new evaluated state to the priority
queue

                    lenopen+=1







#we pass the name of the distance fuction to use in A*
state,moves,analized=astar(state,objective,linear_conflict_distance_fast)  # xe uso euclidea non funziona perche non è
ammissibile (penso)

ic(moves)  #number of steps to reach the fuction
ic(state)  #final state
ic(analized) #evaluated_states
```

```
ic| state: array([[15, 11, 14, 12],
                  [ 2,  7,  6,  5],
                  [13,  0, 10,  9],
                  [ 8,  4,  1,  3]], dtype=int8)



Filling the set:   0%|          | 0/20922789888000 [00:00<?, ?item/s]


ic| listcurrentstate: array([[15, 11, 14, 12],
                            [ 2,  7,  6,  5],
                            [13,  0, 10,  9],
                            [ 8,  4,  1,  3]], dtype=int8)
ic| 'Modified shape!!!'
```

```
ic| listcurrentstate: array([[ 2, 15, 14],
                             [ 0, 11, 12],
                             [ 7,  6,  3],
                             [ 8, 10,  4]], dtype=int8)
ic| analized: 230221
ic| moves: 69
ic| state: array([[ 1,  2,  3,  4],
                  [ 5,  6,  7,  8],
                  [ 9, 10, 11, 12],
                  [13, 14, 15,  0]], dtype=int8)
ic| analized: 230221




230221
```

Reviews3

I also reviewed and gave advice about my colleagues' solutions ,before giving a possible advice i cloned their repository and tried implementing my suggestions:

About UtkuKepir solution:

> I like your approach of trying multiple ways to solve this n-puzzle problem. You have a good solution for the problem. Results are presented in a clear manner in the readme. A small complaint , it 's that i would suggest you to comment more your code , some part are difficult to understand . An other small complaint is that your algorithm doesn't deal with the edge case where the starting state is identical to the goal state, but it's only a problem of how you structured the if statement to print .
>
> This is the culprit
> `final_state_a_star_linear_conflict,path_a_star_linear_conflict,quality_a_star_linear_conflict,cost_a_star_linear_conflict = a_star(state, h_linear_conflict)`
>
> `if path_a_star_linear_conflict:`
>
> ```
> print(f"Steps (quality): {quality_a_star_linear_conflict}, Total states evaluated (cost)
> {cost_a_star_linear_conflict},Finalstate: {final_state_a_star_linear_conflict}, A* solution:
> {path_a_star_linear_conflict}")
> ```
>
> `else:`
>
> ```
> print("A* linear conflict solution not found")`
> ```
>
> You get an empty path_a_star_linear_conflict in this case and so your algorithm tells "A* linear conflict solution not found"

About anilbayramgogebakan solution:

> If the initial state it's already the objective , your algorithm tells that the best solution is doing 2 moves . Maybe by checking first if you are in the objective state already you can give a solution telling to do 0 moves. Using manhattan distance as heuristic is a good choice , to get some improvements you can also try using linear conflict distance.
>
> When you find a new_state , you check if that new state it's not in the non_visited list but you don't check if it isn't also in the visited list. In your code half the time you find a "new" state that already have in the visited list and so fully explored it.
>
> By changing the if statement from `if new_state not in non_visited` into `if new_state not in non_visited and new_state not in visited` ,the cost (so the numbers of explored states) is halved , with no loss in the quality of the solution. In this way your algorithm is also faster.
>
> Overall a good job in solving the n puzzle, i would advice some more comments to make your code more understandable.

Project

Here is the github link to the project: (https://github.com/Scrivane/CI2024_project-work)

I worked on this algorithm alone. The objective of this program is to discover mathematical functions that best minimize the Mean Squared Error (MSE) on the given datasets. The algorithm is based on a modern Genetic Algorithm (GA).

Genotype representation

Each individual in the population has an expression tree as its genotype. Each node within a tree can be a constant, a variable, or a mathematical function. The algorithm can evolve formulas by choosing from many NumPy functions (hoping to always find a possible good solution, even if that means running for more epochs), but only a few constants (3 constants: pi, Euler's number, and the Euler–Mascheroni constant). The available variables are provided by each problem.

Each node serves as the root of its respective subtrees, and each node has a list of its subtrees and its successors. Each node can have a maximum of 2 successors

Fitness

Each individual has a fitness field. The fitness combines the Mean Squared Error (MSE) with a parsimony pressure term (a penalty for lengthy individuals). This penalty is scaled based on the maximum values observed in the target (y) data and the number of data points in the problem. The maximum length of an individual's genome is limited to 500 nodes (including operations, constants, and variables) to avoid excessive bloat, which would have caused Python errors.

Optimized Individual Evaluation

The program leverages the gxgp library by Professor Squillero, extending its capabilities with custom functions and deleting unused functions. A significant performance enhancement involves a custom evaluation function, which converts a tree into its equivalent NumPy formula string. This string is then executed using Python's eval() function, passing the input x (a NumPy array) as a parameter. This approach allows for faster evaluations.

Evolutionary set-up

The algorithm employs a steady-state evolutionary model; during each epoch, a new individual replaces the worst-fitting one.

Parent selection for crossover is performed using tournament selection (size 2, with low selective pressure to maintain diversity). Crossover is always applied. To increase the selective pressure during parent selection in the large population (1500 individuals), Over-selection is utilized. Specifically, 80% of parent selections are made exclusively from the top 24% of individuals in the population. This helps promote faster convergence while maintaining enough diversity from the rest of the population.

The crossover function used is xover_swap_subtree, which was already present in the gxgp library. It selects a random subtree from parent one and replaces it with a random subtree from parent two.

The resulting offspring may mutate at a low rate. Here are the possible mutation functions that can be employed:

- Mutation Point: Alters a node; if it's a leaf (constant or variable), it's replaced by another constant or variable. If it's an internal node, it's replaced by another operand with the same arity.
- Mutation Hoist: Selects a random subtree and makes it the new root of the tree, effectively "cutting" the tree.
- Mutation Permutations: Changes the order of operands for binary function nodes, avoiding commutative functions.
- Mutation Collapse: Replaces an entire subtree with a single constant or variable.
- Mutation Delete Unary: Removes a unary function node, promoting its child directly to its parent's position.
- Mutation Add Unary: Inserts a new unary function node at a random point in the tree, adding depth.

Survivor selection is deterministic. The worst-fitting individual in the population is purged in each epoch. To maintain genetic diversity and prevent premature convergence, the algorithm explicitly avoids adding individuals with formulas identical to those already present in the population.

Refinement Strategies

For some problems, specific optimization strategies are applied to get better results:

- Restart Mechanism: The algorithm performs 6 independent restarts. Each restart initializes a new population and runs the GA until a maximum number of steps is reached.
- Final Refinement Run: After the restarts, a final refinement run is done. The starting population for this run contains the best two individuals from each previous restart, their mutated versions, and individuals generated through crossover between them, alongside a proportion of randomly initialized individuals. This approach aims to refine the "best individuals" discovered across multiple independent searches. During this run , it's increased the selective pressure using tournament selection of size 4 (instead of 2 as previous runs)
- "Run Until Plateau": Instead of a fixed number of steps, the algorithm continues as long as there are improvements in fitness over the last nstep/3 epochs, detecting when the search has stagnated in a local optimum. This is a computationally intensive strategy, so it's not done for every program.
- Long Refinement: The Final Refinement Run has x10 steps. The idea is that the initial 6 restarts efficiently locate promising local minimums, and the final run can perform a more extensive search.
- Rescaling : For problem 5 , the objective values of y were with a really low scale , so they got rescaled into the range [0,5] evolution began.After the formula was evolved, the y values were scaled back to their original range. This mechanism significantly improved the results.

Results project

Here are written best results obtained across multiple runs

Problem Number	Nsteps	Nrestarts	Rescaling	Run Until Plateau	Final Refinement Run	Long Refinement	MSE	Formula
1	10000	1	No	No	No	No	0	np.sin(x[0])

Problem Number	Nsteps	Nrestarts	Rescaling	Run Until Plateau	Final Refinement Run	Long Refinement	MSE	Formula
2	50000	6	No	No	Yes	Yes	1548510021431.2053	np.multiply(np.tanh(np.add(np.multiply(x[0], np.lo np.multiply(np.tanh(np.maximum(np.maximum(np np.add(np.sin(np.exp(np.log1p(np.exp(np.maximu np.log1p(np.sin(np.sin(np.add(np.multiply(x[0], np np.log1p(np.tanh(np.maximum(np.tanh(np.log1p(np.arctan(np.add(x[2], x[0]))), x[1]))))), np.add(np.e np.sin(x[0])), x[1]), np.add(np.log1p(np.e), np.tanh np.sin(np.log1p(np.e)))))), np.add(np.log1p(np.sin(x[0])))))), np.add(x[1], np.arctan(np.sin(np.exp(np.lo np.arctan(np.add(np.add(np.maximum(np.add(np np.log2(np.absolute(np.exp(np.log1p(np.e))))))), np np.add(np.tanh(np.sin(np.arctan(np.sin(x[0]))))), x[2 np.add(np.log1p(np.tanh(np.log1p(np.tanh(np.tan np.maximum(np.sinh(np.add(np.tanh(np.add(np.ta np.sin(np.maximum(np.sin(x[0]), np.maximum(np. np.maximum(np.sin(np.arctan(np.sin(np.add(np.si np.sin(np.maximum(np.arctan(np.log1p(np.tanh(n np.log2(np.absolute(np.add(np.maximum(np.add(np.add(np.sin(np.sin(np.arctan(np.sin(np.arctan(np np.sin(np.exp(np.arctan(np.add(x[2], x[1])))))), np.s np.log1p(np.sin(np.log1p(np.tanh(np.sin(np.arctan np.sin(np.maximum(np.maximum(np.exp(np.log1p np.log2(np.absolute(np.add(np.maximum(np.add(x[1])))))))), np.add(np.exp2(np.sin(x[0])), np.exp2(n np.sin(np.sin(x[0])))))))), np.tanh(x[2])))))), np.sin(np.log1p(np.sin(np.log1p(np.tanh(np.sin(np.arctan np.arctan(np.add(x[2], x[1]))), x[0])))))), np.sinh(np.
3	50000	6	No	No	Yes	Yes	0.4122990064737059	np.add(np.add(np.add(np.add(np.add(np.exp2(np np.euler_gamma)))), np.negative(x[2])), np.negativ
4	50000	6	No	Yes	Yes	No	0.09592352523610304	np.multiply(np.sinc(np.multiply(x[1], np.multiply(n np.euler_gamma))))))), np.hypot(np.multiply(x[1],
5	15000	6	Yes	Yes	Yes	No	5.512898055429115e-19	np.add(np.divide(np.add(np.sin(np.hypot(np.powe np.arctan(x[0])), np.negative(np.sin(x[1]))), np.arct -2.8520706810421616e-08)
6	50000	6	No	No	Yes	No	7.145589969138924e-07	np.add(np.multiply(np.arctan(np.cbrt(np.euler_ga
7	50000	6	No	No	Yes	Yes	54.844555000111505	np.hypot(np.multiply(np.add(np.add(np.add(np.hy np.cosh(np.remainder(x[1], x[0]))), np.multiply(np. np.remainder(np.cbrt(np.multiply(np.hypot(x[0], n np.exp2(np.hypot(x[1], np.log10(np.euler_gamma np.remainder(x[0], np.multiply(x[1], np.cosh(np.eu np.multiply(np.remainder(x[1], np.multiply(x[0], n np.log10(np.hypot(np.remainder(x[1], x[0]), np.rer
8	50000	6	No	No	Yes	No	166579.13617635533	np.add(np.multiply(np.add(np.add(x[5], np.round(np.cbrt(np.negative(np.exp2(np.cbrt(np np.add(x[5], x[5]))), np.e), np.cbrt(np.remainder(np np.negative(np.add(x[5], x[5])))))))), np.multiply(np np.add(x[5], x[5]))))), np.add(np.multiply(np.square np.round(x[5]))))))

Code

Main program

```
import numpy as np

from gxgp import Node
import gxgp
from gxgp import *
from tqdm import tqdm
from dataclasses import dataclass
from matplotlib import pyplot as plt
from itertools import accumulate
```

```
i=1

history=[]

import math


@dataclass
class Individual:  #internally generates a init for this dataclass.
    genome: Node
    fitness: float = None



def length_penalty(length, threshold=400, scale=1e6):
    if length <= threshold:

        return   0.001*length*scale#scale * 0.001 * length**2
    else:
        base = scale * 0.001 * threshold**2
        blowup = np.exp((length - threshold) / 10) * scale
        return base + blowup

def fitness(tree,nodefun:Node,x, y):
    #start_time = time.perf_counter()


    with np.errstate(all="raise"):          # only for the code inside
        try:
            mse_val = tree.mse(nodefun, x.T, y)
        except (FloatingPointError, ZeroDivisionError, ValueError):
            return -math.inf                # error= worst fitness


    if not np.isfinite(mse_val) :  #infinite result or nan result  = worst fitness
        return -math.inf




    if numproblem==3: #increase penalty for lengthy individuals
        div=0.8
    elif numproblem==2:  #increase penalty for lengthy individuals
        div=0.8
    else:
        div=1


    return -mse_val-((length_penalty(len(nodefun),1000,maxy*sizey/(5000*2))) /div)  #+ penalty_coeff * (tree_length **
2)



def error(tree,nodefun:Node,x, y):


    with np.errstate(all="raise"):          # only for the code inside
        try:
            mse_val = tree.mse(nodefun, x.T, y)
        except (FloatingPointError, ZeroDivisionError, ValueError):
            return math.inf                 # can't be compute


    if not np.isfinite(mse_val):
        return math.inf

    return mse_val


def popInizialize(gpTree,nelements,x,y):
    population=[]
    unique_string_genomes = set()

    while len(population)<nelements:
```

```
        locel=gpTree.create_individual(10)

        fitnesslocel=fitness(gpTree,locel,x,y)
        if (str(locel) not in unique_string_genomes):
            unique_string_genomes.add(str(locel))
            population.append(Individual(genome=locel,fitness=fitnesslocel))

    return population

def tournament_sel_array(population,n=2): # tournament selection to decide which individual to crossover


    fitness_list=[]
    random_elements_indexes = np.random.choice(len(population), size=n, replace=True)
    for el in random_elements_indexes:
        fitness_list.append((population[el]).fitness)


    bestIndex=np.argmax(fitness_list)

    chosen=population[random_elements_indexes[bestIndex]]

    return chosen



def EA_Run(nstep,pop,crossOverRate,mutrate,MAX_TREE_LENGTH,gptree,extra_el_sel=0):  #tournament selection with size 2
, in all runs except in the last one when it's size 4 (increased selective pressure )
        run_until_plateaux=False

        unique_string_genomes=set()
        for el in pop:
            localGenome=el.genome
            unique_string_genomes.add(str(localGenome))

        best_individual = max(pop, key=lambda x: x.fitness)
        lastImprouvement=-1
        originalnstep=nstep
        while lastImprouvement<originalnstep/3 and (lastImprouvement==-1 or run_until_plateaux==True):  #if
run_until_plateaux is true , it runs as long as a plateux is not detected
            if lastImprouvement==-1:
                lastImprouvement=0
            print(lastImprouvement)
            print(best_individual.fitness)

            for el in tqdm(range(nstep)):
                lastImprouvement+=1


                topPoptreshold=0.24  #1500   #pop 1000 =0.16    #pop 2000  0.32
                getting_From_Top=0.8




                if(crossOverRate==1 or  np.random.rand()<crossOverRate):
                    sorted_pop = sorted(pop, key=lambda x: x.fitness,reverse=True)
                    best_pop=sorted_pop[:int(topPoptreshold*len(sorted_pop))]
                    worst_pop=sorted_pop[int(topPoptreshold*len(sorted_pop)):]

                    if np.random.rand()<getting_From_Top:

                        ris1=tournament_sel_array(best_pop,2+extra_el_sel)   # uses tournament selection to select 2
parent for the crossover
                    else:
                        ris1=tournament_sel_array(worst_pop,2+extra_el_sel)   # uses tournament selection to select 2
parent for the crossover


                    if np.random.rand()<getting_From_Top:

                        ris2=tournament_sel_array(best_pop,2+extra_el_sel)   # uses tournament selection to select 2
parent for the crossover
                    else:
                        ris2=tournament_sel_array(worst_pop,2+extra_el_sel)

                    genome_child = gxgp.xover_swap_subtree(ris1.genome,ris2.genome)
                    if len(genome_child) > MAX_TREE_LENGTH:
                        #generated new one because old one was too long
                        listonepop=popInizialize(gptree,1,x,y)
```

```
                        randomNewIndividual=listonepop[0]
                        genome_child = randomNewIndividual.genome


                else:
                    genome_child=(tournament_sel_array(pop,1)).genome  #select a random individual that later can be
mutated
                if len(genome_child) > 2300:
                    print("aqui")

                    continue


                mutation_functions = [
                lambda genome: gxgp.mutation_point(genome, gptree),
                lambda genome: gxgp.mutation_hoist(genome),
                lambda genome: gxgp.mutation_permutations(genome),
                lambda genome: gxgp.mutation_collapse(genome, gptree),
                lambda genome: gxgp.mutation_delete_unary(genome),
                lambda genome: gxgp.mutation_add_unary(genome,gptree)
                ]

                if( np.random.rand()<mutrate):  # has a chance of mutating the child using a mutation
                    mutation_fn = np.random.choice(mutation_functions)
                    if len(genome_child) > 2300:
                        print("si")
                        continue

                    genome_child = mutation_fn(genome_child)


                if (str(genome_child) not in unique_string_genomes):
                    unique_string_genomes.add(str(genome_child))
                    child=Individual(genome=genome_child, fitness=fitness(gptree, genome_child, x, y))
                    pop.append(child)  # add child to population
                    newIndividual=child


                else:  # if it  isn't new

                    listonepop=popInizialize(gptree,1,x,y)
                    randomNewIndividual=listonepop[0]
                    while str(randomNewIndividual.genome) in unique_string_genomes:
                        listonepop=popInizialize(gptree,1,x,y)
                        randomNewIndividual=listonepop[0]

                    unique_string_genomes.add(str(randomNewIndividual.genome))
                    pop.append(randomNewIndividual)
                    newIndividual=randomNewIndividual




                if(newIndividual.fitness>best_individual.fitness):
                    best_individual=newIndividual
                    lastImprouvement=0

                localbestfit=best_individual.fitness

                history.append(localbestfit)

                worst_fit_ind=(min(pop, key=lambda ind: ind.fitness))
                pop.remove(worst_fit_ind) # delete from population the wrost individual
                unique_string_genomes.discard(str(worst_fit_ind.genome))









        sorted_pop = sorted(pop, key=lambda x: x.fitness,reverse=True)  #reverse true means desc order(first the one
with higher fitness)

        # Get the best and second individuals
        lowest_fitness_individual = sorted_pop[0]
        formula=lowest_fitness_individual.genome.to_np_formula()
        print(formula)
        print("fitness:")
```

```
        print(lowest_fitness_individual.fitness)
        second_lowest_fitness_individual = sorted_pop[1]
        formula=second_lowest_fitness_individual.genome.to_np_formula()

        return [lowest_fitness_individual,second_lowest_fitness_individual]




def EAlgoithm(gptree,x,y):

    final_run_long=True
    MAX_TREE_LENGTH=500
    mutrate=0.055
    nrestarts=6
    nelemets=1500
    crossOverRate=1     #trying dynamically adjusting crossover rate and mutation rate did't improved the solutions

    nstep=50000

    best_individuals_each_restart=[]
    for _ in range(nrestarts):
        pop=popInizialize(gptree,nelemets,x,y)  #generates nelemets individuals each restart

        top_2_individual=EA_Run(nstep,pop,crossOverRate,mutrate,MAX_TREE_LENGTH,gptree) #return  an array with the
top2 individuals
        best_individuals_each_restart.extend(top_2_individual)  #adds those 2 individuals to the pool of all previus
generated champions for each restart



    if final_run_long==True:  #if it's true , does an other run using previus champions, their mutation , their
crossover a


        unique_string_genomes=set()
        pop=[]
        for individual in best_individuals_each_restart:
            genome_child=individual.genome


            if (str(genome_child) not in unique_string_genomes):
                        unique_string_genomes.add(str(genome_child))
                        child=Individual(genome=genome_child, fitness=fitness(gptree, genome_child, x, y))
                        pop.append(child)  # add child to population


            mutation_functions = [
                lambda genome: gxgp.mutation_point(genome, gptree),
                lambda genome: gxgp.mutation_hoist(genome),
                lambda genome: gxgp.mutation_permutations(genome),
                lambda genome: gxgp.mutation_collapse(genome, gptree),
                lambda genome: gxgp.mutation_delete_unary(genome),
                lambda genome: gxgp.mutation_add_unary(genome,gptree)
                ]

            for i in range(int(0.4*nelemets/len(best_individuals_each_restart))):
                mutation_fn = np.random.choice(mutation_functions) #select a random mutation and applies it
                mutated_genome = mutation_fn(genome_child)

                if str(mutated_genome) not in unique_string_genomes:
                    unique_string_genomes.add(str(mutated_genome))
                    mutated_child = Individual(genome=mutated_genome, fitness=fitness(gptree, mutated_genome, x, y))
                    pop.append(mutated_child)
        nlong=0
        for _ in range(int(0.15*nelemets)):
                p1=tournament_sel_array(best_individuals_each_restart,2)   # uses tournament selection to select 2
parent for the crossover
                p2=tournament_sel_array(best_individuals_each_restart,2)

                crossed = gxgp.xover_swap_subtree(p1.genome, p2.genome)

                if str(crossed) not in unique_string_genomes and len(crossed) <= MAX_TREE_LENGTH :
                    nlong+=1
                    unique_string_genomes.add(str(crossed))
                    child = Individual(genome=crossed, fitness=fitness(gptree, crossed, x, y))
                    pop.append(child)


        print(nlong)
```

```
        missing_individuals=nelemets-len(pop)
        poploc=popInizialize(gptree,missing_individuals,x,y)
        pop.extend(poploc)
        if numproblem==2:  #do more step in the final run
            nstep=nstep*10
        elif numproblem==3:  #it worked
            nstep=nstep*10
        elif numproblem==7: #it worked
            nstep=nstep*10



        top_2_individual=EA_Run(nstep,pop,crossOverRate,mutrate,MAX_TREE_LENGTH,gptree,2) #last run puts more
selective pressure when doing parent selection (tournament selection size 4)
        best_individuals_each_restart.extend(top_2_individual)


    best_one=min(best_individuals_each_restart,key=lambda el: error(gptree,el.genome,x,y))
    formula=best_one.genome.to_np_formula()




    return formula,error(gptree,best_one.genome,x,y)



def normalization(y,range=(0,5)):  #scales the y into the passed range doing min-max scaling
    y_min = np.min(y)
    y_max = np.max(y)


    offset=y_min
    mulCoefficient=(range[1] -range[0])/(y_max-y_min)

    yscaled=(y-offset)*mulCoefficient+range[0]

    return yscaled,mulCoefficient,offset,range[0]



numproblem=7
scaling=False
problem = np.load('../data/problem_{}.npz'.format(numproblem)) #depends on where you are running the program , if in
the project directory only one dot at the start ,, if in the src directory use two dots
x = problem['x']  #3 righe 5000 colonne

y=problem['y']


if(scaling==True):

    y_scaled,mulCof,offset,_=normalization(y)
    y=y_scaled
sizey=np.size(y)
maxy=np.max(y)


if numproblem==2:  #there is maximum function instead of sinc one (we have high values of y here , i belive it's not
useful sinc )
            operators=
[np.mod,np.arctan,np.maximum,np.sqrt,np.cbrt,np.add,np.negative,np.multiply,np.sin,np.tanh,np.reciprocal,np.exp,np.exp
2,np.pow,np.sinh,np.round,np.cosh,np.i0,np.hypot,np.absolute,np.square,np.log1p,np.log2,np.log10,np.log]
else:
        operators=
[np.mod,np.arctan,np.sinc,np.sqrt,np.cbrt,np.add,np.negative,np.multiply,np.sin,np.tanh,np.reciprocal,np.exp,np.exp2,n
p.pow,np.sinh,np.round,np.cosh,np.hypot,np.i0,np.absolute,np.square,np.log1p,np.log2,np.log10,np.log]  #np.acos],
#ignore_op#np.round],  #np.round #np.pow, #np.ldexp],


dag = gxgp.DagGP(
    operators=operators,
    variables=x.shape[0],
    constants=[np.pi,np.e,np.euler_gamma],
)

formula,fit=EAlgoithm(dag,x,y)
print("THe formula is :")
print(formula)
ygen = eval(formula, {"np": np, "x": x})
if (ygen.size==1):
```

```
    ygen = ygen * np.ones(np.size(y))

ris= np.mean((y.astype(np.float64) - ygen.astype(np.float64)) ** 2)

print("min_mse no scaling")
print(fit)
if ris==fit:
    print("good")
else:  #there is an error if they don't match
    print("wrong")

if scaling==True:
    denormalizedFromula=f"np.add(np.divide({formula}, {mulCof.item()}), {offset.item()})"  #denormalizing by inverting
the previus minamx scaling directly in the string formula
    print(denormalizedFromula)
    y_denormalized=eval(denormalizedFromula, {"np": np, "x": x})
    if (y_denormalized.size==1):
        y_denormalized = y_denormalized *np.ones(np.size(y))
    y=problem['y']
    ris= np.mean((y.astype(np.float64) - y_denormalized.astype(np.float64)) ** 2)

    ris= sum((a - b) ** 2 for a, b in zip(y_denormalized, y)) / len(y)

    print("min_mse using scaling: ")
    print(ris)



results_file = "gp_results_log.txt"   #writes all result in a log file
with open(results_file, "a") as f:
    f.write("="*60 + "\n")

    f.write(f"Problem Number: {numproblem}\n")
    f.write(f"Scaling Used: {scaling}\n")


    if scaling:
        f.write(f"Formula: {denormalizedFromula}\n")
    else:
        f.write(f"Formula: {formula}\n")


    f.write(f"MSE: {ris}\n")
    f.write("="*60 + "\n\n")



plt.figure(figsize=(14, 8))
plt.plot(
            range(len(history)),
            list(accumulate(history, max)),
            color="red",
        )
_ = plt.scatter(range(len(history)), history, marker=".")
plt.show()
```

Changes in the library gxgp

Node.py

```
    def to_np_formula(self):  #custom function to go from string representation , to numpy string representation
(useful for evaluating the function)
        stringa=self.long_name
        newstringa=""
        openbracket=0
        for i in range(0,len(stringa)):
            char=stringa[i]

            newstringa=newstringa+stringa[i]
            if openbracket==1:
                newstringa=newstringa+"]"
                openbracket=0

            if( char=="x" and stringa[i+1].isdigit()):
                newstringa=newstringa+"["
                openbracket=1
```

```
        isfun=0
        stringa=newstringa
        newstringa=""
        for i in range(0,len(stringa)):
            char=stringa[i]
            if( char.isalpha() and isfun==0 and stringa[i+1]!="["):
                isfun=1
                newstringa=newstringa+"np."
            elif(char.isalpha()==False and ( ( i!=len(stringa)-1 and stringa[i+1]!='p') or ( i<len(stringa)-2 and
stringa[i+1]=='p' and stringa[i+2]=='o'))  ):
                isfun=0
            newstringa=newstringa+stringa[i]


        newstringa = newstringa.replace("3.14159", "np.pi")
        newstringa = newstringa.replace("0.577216", "np.euler_gamma")
        newstringa = newstringa.replace("2.71828", "np.e")


        return newstringa
```

utils.py

```
import  numpy as np
def arity(f: Callable) -> int:
    if isinstance(f, np.ufunc):
        return f.nin
    try:
        if inspect.getfullargspec(f).varargs is not None:
            return None
        else:
            return len(inspect.getfullargspec(f).args)

    except TypeError:
        ris=inspect.signature(f)
        parameters=ris.parameters.values()
        todelete=0
        for p  in parameters:
            type=p.kind
            default=p.default


            if default==None or default==0:  #checks how many parameters are optional (so they have a default value of
1 or none)
                todelete+=1

        return len(parameters)-todelete # i return the number of  non optional parameters
```

gp_dag.py

```
@staticmethod
def evaluate2(individual: Node, X, variable_names=None):



        formula=individual.to_np_formula()


        y_pred = eval(formula, {"np": np, "x": X.T})
        if np.isscalar(y_pred) or  y_pred.shape == ():
            y_pred = y_pred * np.ones(X.shape[0])




        return y_pred
```

gp_common.py

```
def find_parent(root, target):
        for node in root.subtree:
            for i, child in enumerate(node.successors):
                if child is target:
                    return node, i
        return None, None




def mutation_hoist(tree1: Node) -> Node:
    offspring = deepcopy(tree1)
    internal_nodes = [node for node in offspring.subtree if not node.is_leaf and node is not offspring]
    successors = None
    if not internal_nodes:
        return offspring

    # Pick random internal node
    node = gxgp_random.choice(internal_nodes)

    return node




def mutation_collapse(tree1: Node,gptree:DagGP) -> Node:
    offspring = deepcopy(tree1)
    internal_nodes = [node for node in offspring.subtree if not node.is_leaf]
    successors = None
    if not internal_nodes:
        return offspring



    # Pick random internal node
    target = gxgp_random.choice(internal_nodes)
    possibilities= gptree._variables + gptree._constants
    new_node = deepcopy(gxgp_random.choice(possibilities))

    parent, idx = find_parent(offspring, target)
    if parent is not None:
        children = parent.successors
        children[idx] = new_node
        parent.successors = children

    return offspring




def mutation_delete_unary(tree1: Node) -> Node:
    offspring = deepcopy(tree1)
    internal_nodes_arity_1 = [node for node in offspring.subtree if not node.is_leaf and node.arity==1]
    if not internal_nodes_arity_1:
        return offspring



    # Pick random internal node
    target = gxgp_random.choice(internal_nodes_arity_1)


    parent, idx = find_parent(offspring, target)

    if parent is not None:
        child = target.successors[0]  #it's unary so there is only the first
        succs = parent.successors  # This is a copy of the list done in successors , i need later to replace it whole
otherwise no update
        succs[idx] = child
        parent.successors = succs

    else:  # i'm the root and i return the child
        return deepcopy(target.successors[0])
    return offspring
```

```
def mutation_add_unary(tree1: Node, gptree: DagGP) -> Node:
    offspring = deepcopy(tree1)

    all_nodes = list(offspring.subtree)
    unary_ops = [op for op in gptree._operators if arity(op) == 1]

    if not unary_ops or not all_nodes:
        return offspring

    target = gxgp_random.choice(all_nodes)
    op = gxgp_random.choice(unary_ops)
    up_node = Node(op, [target])


    if target is offspring:

        return up_node

    parent, idx = find_parent(offspring, target)
    if parent is not None:
        succs = parent.successors
        succs[idx] = up_node
        parent.successors = succs



    return offspring

def mutation_point(tree1: Node,gptree:DagGP) -> Node:
    offspring = deepcopy(tree1)  # single deepcopy

    all_nodes = list(offspring.subtree)

    target = gxgp_random.choice(all_nodes)

    if target.is_leaf:

        possibilities=[e for e in (gptree._variables + gptree._constants) if str(e)!=str(target)]
        new_node = deepcopy(gxgp_random.choice(possibilities))

    else:

        possible_ops = [op for op in gptree._operators if arity(op) == target.arity and target.short_name!=op.__name__
]
        if len(possible_ops)>0:
            new_op = gxgp_random.choice(possible_ops)

        new_node = Node(new_op, target.successors)


    if target is offspring:

        return new_node

    parent, idx = find_parent(offspring, target)
    if parent is not None:
        children = parent.successors
        children[idx] = new_node
        parent.successors = children



    return new_node



commutative_func=["multiply","add","hypot","maximum"]

def mutation_permutations(tree1: Node) -> Node:
    offspring = deepcopy(tree1)  # single deepcopy


    internal_nodes = [node for node in offspring.subtree if not node.is_leaf and node.arity>=2 and node.short_name not
in commutative_func]  # uses non cummutative functions only
    if (len(internal_nodes)>=1):


        target = gxgp_random.choice(internal_nodes)
        successors = target.successors
        succ1=deepcopy(successors[0])
        succ2=deepcopy(successors[1])
```

```
        target.successors=[succ2,succ1]

    return offspring
```