Laborator 3 Ai Algoritmi evolutivi

```
def genereazaSol(obiecte):
    n = len(obiecte)
    sol = np.random.randint(2, size=n)
    return sol

# functia de fitness
def fctFitness(obiecte, sol, greutateTotala):
    greutate, valoare = evaluare(obiecte, sol)
    return greutate <= greutateTotala, valoare


# calculeaza greutatea si costul unei solutii
def evaluare(obiecte, sol):
    greutate = 0
    valoare = 0
    l = len(obiecte)
    for i in range(l - 1):
        greutate = greutate + sol[i] * obiecte[i][1]
        valoare = valoare + sol[i] * obiecte[i][0]
    return greutate, valoare
```

Functiile luat din laboratoarele anterioare folosite pentru generarea unei sol aleatorii, validarea ei si gasirea fitnessului

```
def initializarePop(pop_size, obiecte, greutateTotala):
    pop = []
    for i in range(pop_size):
        pop.append(solutieValida(obiecte, greutateTotala))
    return pop
```

Functie de initializare populatie

```
def selectieParinti(pop_size, pop, obiecte, greutateTotala):
    sample = np.random.default_rng().choice(pop_size, size=5, replace=False)
    best = pop[sample[0]].copy()
    for i in sample:
        if fctFitness(obiecte, pop[i], greutateTotala) > fctFitness(obiecte, best,
greutateTotala):
            best = pop[i].copy()
    return best
```

Functie folosita pntru selectia parintilor

```
def incrucisare1(parinte1, parinte2, obiecte, greutateTotala):
    taietura = np.random.randint(1, len(parinte1)- 1)
    copil1 = [0] * len(parinte1)
    copil2 = [0] * len(parinte2)
    for i in range(taietura):
        copil1[i] = parinte1[i]
```

```
        copil2[i] = parinte2[i]
    for i in range(taietura, len(parinte1)):
        copil1[i] = parinte2[i]
        copil2[i] = parinte1[i]
    copilValid(copil1, obiecte, greutateTotala)
    copilValid(copil2, obiecte, greutateTotala)
    return copil1, copil2
```

Functie de incrucisare cu o taietura unde formam copii

```
def mutatie( copil, obiecte, greutateTotala):
    mutant = copil.copy()
    for i in range(len(copil)):
        if np.random.random() < 0.4:
            mutant[i] = 1 - mutant[i]
    copilValid(mutant, obiecte, greutateTotala)
    return mutant
```

Functia de mutatie tare

```
def rucsac(k,obiecte, greutateTotala, nr_gen, pop_size):
    i = 0
    best = []
    worst = []
    average = []
    allAverage = []

    while i < k:
        pop = initializarePop(pop_size, obiecte, greutateTotala)
        t = 0
        while t < nr_gen:
            copii = []
            for i in range(pop_size // 2):
                parinte1 = selectieParinti(pop_size, pop, obiecte, greutateTotala)
                parinte2 = selectieParinti(pop_size, pop, obiecte, greutateTotala)
                copil1, copil2 = incrucisare1(parinte1, parinte2, obiecte,
greutateTotala)
                copii.append(copil1)
                copii.append(copil2)
            copiiMutanti = []
            for i in range(len(copii)):
                mutant = mutatie(copii[i], obiecte, greutateTotala)
                copiiMutanti.append(mutant)
            pop = bestOfGenerations(pop, copii, copiiMutanti, pop_size, obiecte,
greutateTotala)
            best.append(pop[0])
            worst.append(pop[-1])
            average.append(mediePop(obiecte, pop, greutateTotala))
            t = t+1
        medie = 0
        print("best", best)
```

```python
        print("worst", worst)
        for i in range(0, len(average)):
            medie = medie + average[i]
        medie = medie / len(average)
        allAverage.append(medie)
        allBest = bestOfAll(obiecte, best, greutateTotala)
        print("all best", allBest)
        fctFList = []
        for i in range(len(allBest)):
            fctFList.append(fctFitness(obiecte, allBest[i], greutateTotala))
        print("fctList", fctFList)
        print("fitness all best 1", fctFitness(obiecte, allBest[0], greutateTotala)[1])
        print("fitness all best 2", fctFitness(obiecte, allBest[len(allBest)-1],
greutateTotala)[1])
        print("len all best", len(allBest))
        allBest1 = allBest[0]
        allWorst = allBest[len(allBest)-1]
        print("all best", allBest)
        print("all worst", allWorst)
        bestSol = fctFitness(obiecte, allBest1, greutateTotala)[1]
        worstSol = fctFitness(obiecte, allBest[len(allBest)-1], greutateTotala)[1]
        print("best sol", bestSol)
        print("medie", medie)
        print("worst sol", worstSol)
        print("best", best)
        print("worst", worst)

        plots(obiecte, best, worst, greutateTotala)
        with open('solutiiRucsac.txt', 'a') as f:
            f.write(str(bestSol))
            f.write(" ")
            f.write(str(worstSol))
            f.write(" ")
            f.write(str(medie))
            f.write(" ")
            f.write(str(nr_gen))
            f.write(" ")
            f.write(str(pop_size))
            f.write(" ")
            f.write("\n")
```

Functia care ruleaza algoritmul evolutiv. Alegem o solutie valida. Se aleg 2 parinti, din care vor rezulta 2 copii prin incrucisarea cu o taitura care prin mutatie vom mai obtine 2 copii mutanti. Urmatoarea generatie v-a fi aleasa dintre cei mai buni indivizi copii, copii mutanti sau parinti

nr generatii	populatii	best	average	timp
50	100	654	547.7922	57.701672077178955
100	100	693	570.14515	181.9085943698883
200	100	698	574.178225	356.9071922302246
50	100	132628	129515.328299999	83.61578369140625
100	100	133545	130102.09774999994	178.4215178489685
200	100	132889	129888.60612500006	616.1670157909393

Observam ca cu cat creste numarul de genratii avem rezultate din ce in ce mai bune. creste si valoarea average pe solutii, dar si timpul de rulare

Instanța problemei	k	Valoarea medie	Cea mai buna valoare	Tabu nr	Nr executii	Timp mediu de executie
rucsac20.txt	100	141.3	503	5	10	7.8797643184661865
	300	194.6	581	5	10	9.098579168319702
	1000	270	588	5	10	9.922972917556763
	100	285	570	10	10	5.581806421279907
	300	345	657	10	10	9.147357702255249
	1000	167.8	615	10	10	{6.833710432052612
rucsac200.txt	100	39496.7	131735	5	10	6.392694711685181
	300	65681.9	132835	5	10	6.909295558929443
	1000	39340.9	131824	5	10	6.846782207489014
	100	52761.6	132219	10	10	8.499313592910767
	300	78620.9	132003	10	10	9.671289920806885
	1000	26338.2	132393	10	10	6.113893032073975

Dupa tabelul cu rezultate obtinute in laboratorul 2 observam ca avem rezultate mult mai bune fata de tabu search.

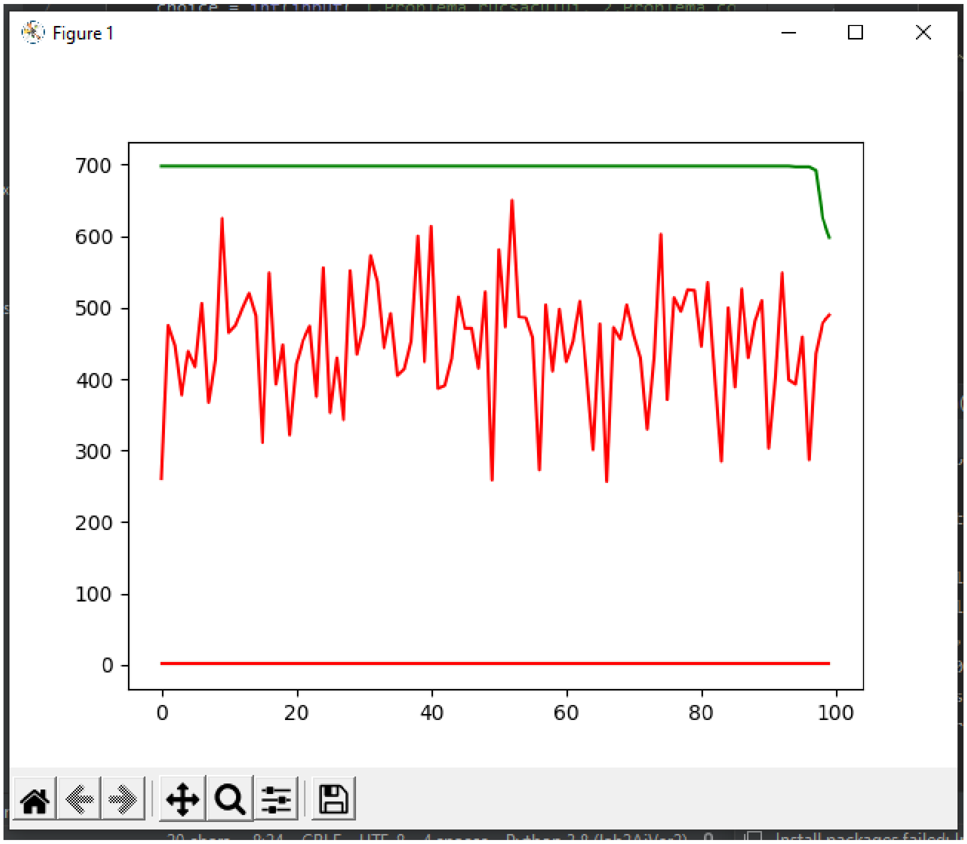


Figure 1
700
600
500
400
300
200
100
0
0
20
40
60
80
100

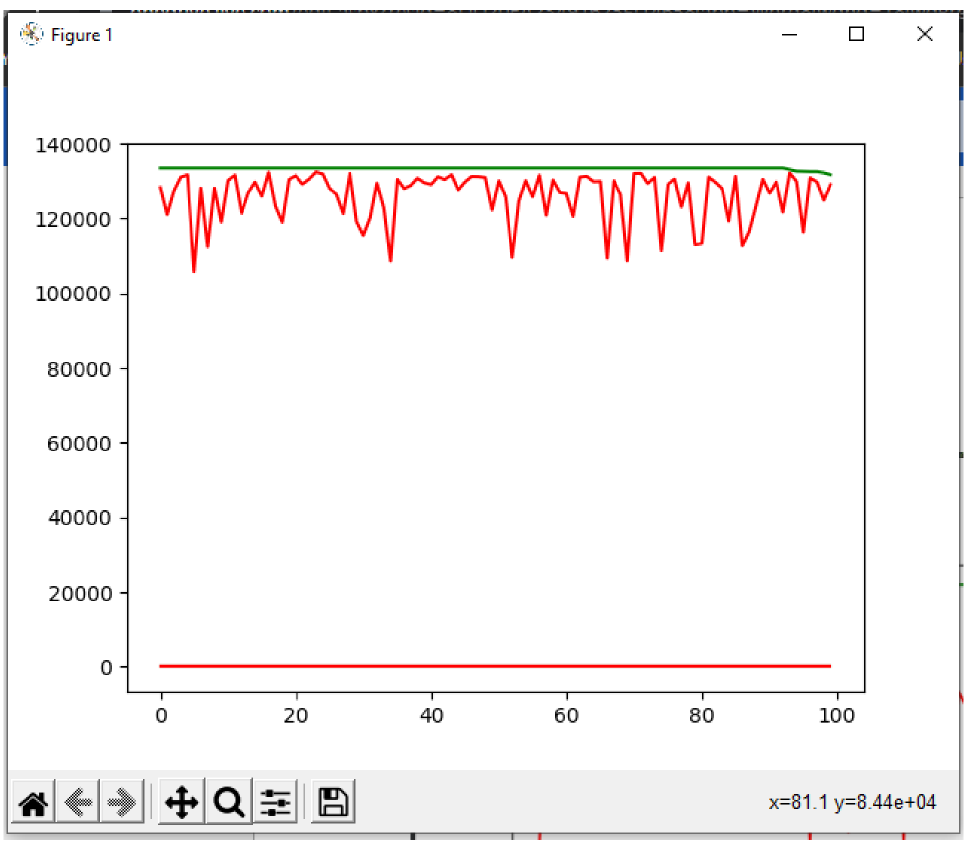


Figure 1
140000
120000
100000
80000
60000
40000
20000
0
0
20
40
60
80
100
x=81.1 y=8.44e+04

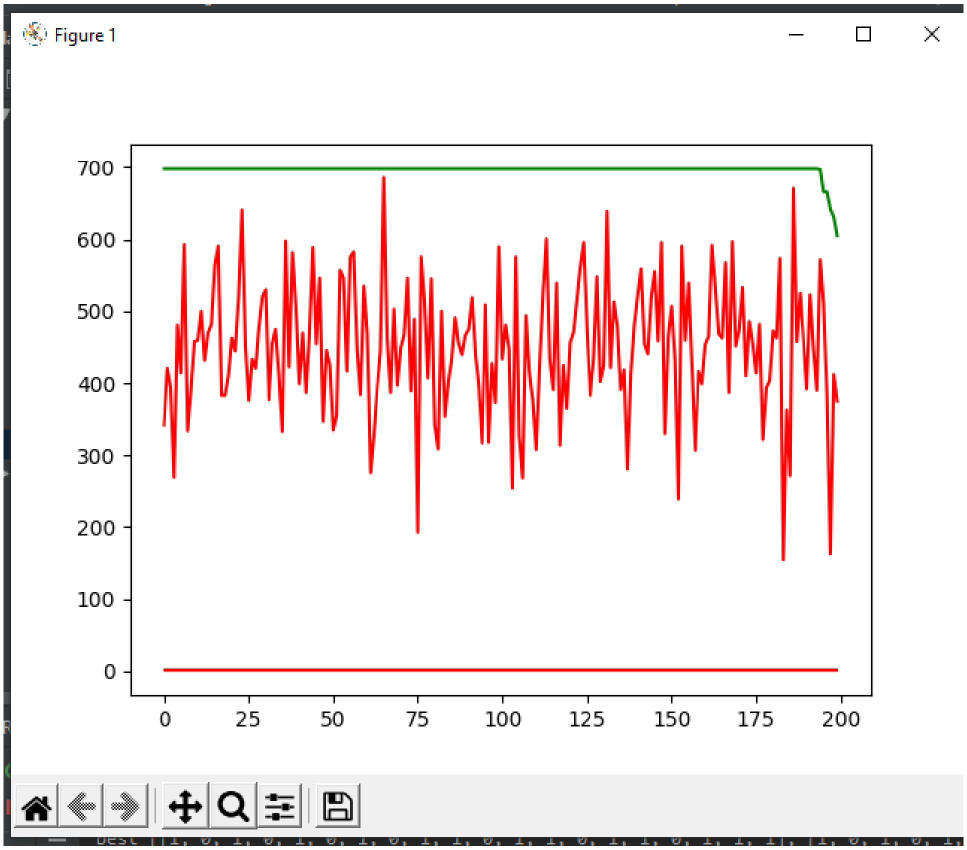
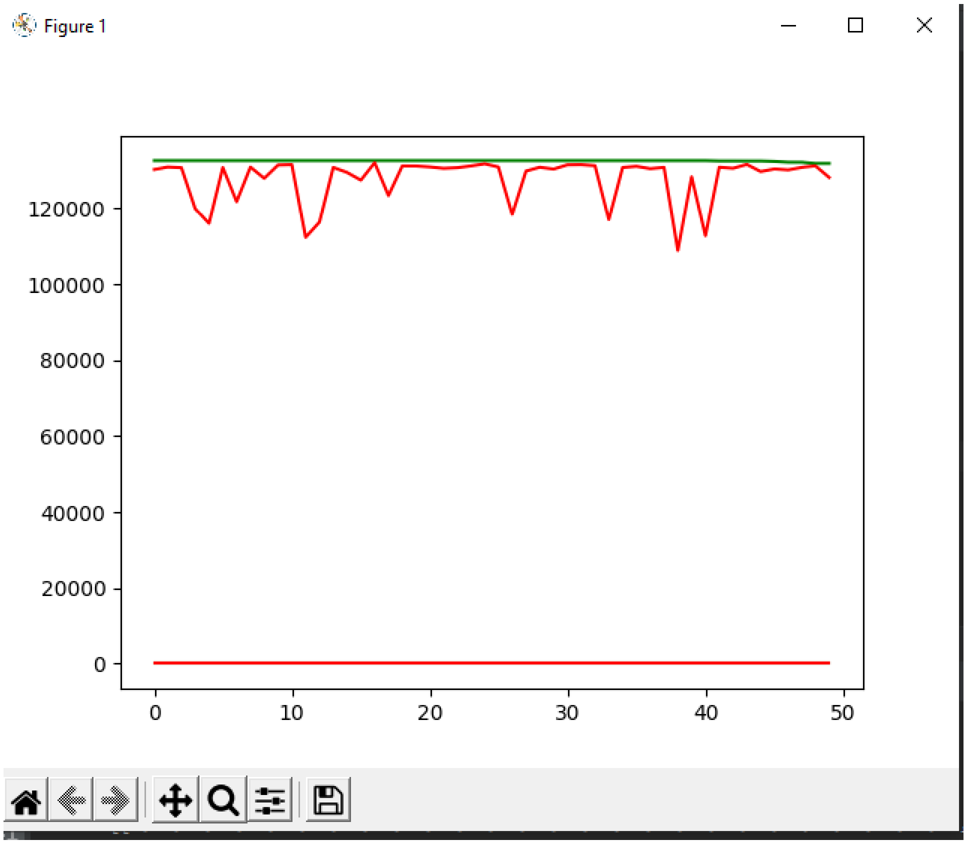


Figure 1
700
600
500
400
300
200
100
0
0
25
50
75
100
125
150
175
200



The traveling salesman problem

```
def distantaOrase(x1, x2, y1, y2):
    dist = sqrt((x2 - x1)*(x2-x1)+(y2-y1)*(y2-y1))
```

```
    return dist


def fitness(orase, permutare):
    sum = 0
    listaDistante = []
   # permutare = list(np.random.permutation(len(orase)))
    print(permutare)
    for i in range(len(orase)-1):
        coord1 = orase[int(permutare[i])][1]
        coord2 = orase[int(permutare[i])][2]
        coord3 = orase[int(permutare[i+1])][1]
        coord4 = orase[int(permutare[i+1])][2]
        dist = distantaOrase(coord1, coord2, coord3, coord4)
        sum = sum + dist

    coord1 = orase[int(permutare[0])][1]
    coord2 = orase[int(permutare[0])][2]
    dist = distantaOrase(coord1, coord2, coord3, coord4)
    sum = sum + dist


    return sum
```

Avem functiile de calculare distanta si calculare fitness din laboratorul anterior.

```
def initializare(pop_size, n):
    pop = []
    for i in range(pop_size):
        pop.append(np.random.permutation(n))
    print("pop din initializare", pop)

    return pop
```

Functia d initializare a populatiei

```
def sel_turnir(pop, pop_size, orase):
    print("pop init", pop)
    sample = np.random.default_rng().choice(pop_size, size=5, replace=False)
    best = pop[sample[0]].copy()
    print("pop", best)
    for i in sample:
        print("pop 1", pop[i])
        if fitness(orase, pop[i]) < fitness(orase, best):

            best = pop[i].copy()
    return list(best)
```

Functia de selectie a parintilor

```python
def incrucisareParintiOx(parinte1, parinte2, taietura1, taietura2):
    copil1 = []
    copil2 = []
    print("parinte 1")
    i = 0
    while i < len(parinte1):
        while taietura1 <=i <= taietura2:
            copil1.append(parinte2[i])
            copil2.append(parinte1[i])
            i = i+1
        copil1.append(0)
        copil2.append(0)
        i = i + 1
    print("copil1", copil1)
    print("copil2", copil2)
    p1 = []
    p2 = []
    i = taietura2 + 1
    while i < len(parinte1)-1:
        print("while1")
        p1.append(parinte1[i])
        p2.append(parinte2[i])
        i = i + 1

    i = 0
    while i <= taietura2:
        print("while2")
        p1.append(parinte1[i])
        p2.append(parinte2[i])
        i = i + 1
    i = taietura2+1
    j = 0
    while i < len(copil1):
        print("while3")
        copil1[i] = p2[j]
        copil2[i] = p1[j]
        i = i + 1
        j = j + 1

    i = 0
    while i < taietura2:
        print("while4")
        copil1[i] = p2[j]
        copil2[i] = p1[j]
        i = i+1
        j = j+1
    print("copil1 final", copil1)
    print("copil2 final", copi2)
    return copil1, copil2
```

Functia de incrucisare ox

```
def twoSwap(permutare):
    print("permutare", permutare)
    index = twoRandomNumbers(0, len(permutare))
    print("index", index)
    aux = permutare[index[0]]
    print("aux", aux)
    permutare[index[0]] = permutare[index[1]]
    print("permutare", permutare[index[0]])
    permutare[index[1]] = aux

    return permutare
```

Functia de mutatie two swap care interschimba elemente de pe pozitii aleatorii

```
def tsp(k,orase, nr_gen, pop_size):
    i = 0
    best = []
    worst = []
    average = []
    allAverage = []

    while i < k:
        pop = initializare(pop_size, len(orase))
        g = 0
        while g < nr_gen:
            copii = []
            for i in range(pop_size // 2):
                parinte1 = sel_turnir(pop, pop_size, orase)
                parinte2 =sel_turnir(pop, pop_size, orase)
                print("a trcut pe aici")
                index = []
                t = random.sample(range(0, len(pop)), 2)
                print("trece de random?")
                if t[0] > t[1]:
                    index.append(t[1])
                    index.append(t[0])
                else:
                    index.append(t[0])
                    index.append(t[1])
                print("index", index)
                t1 = index[0]
                t2 = index[1]
                print("t1 t2", t1, t2)
                copil1, copil2 = incrucisareParintiOx(parinte1, parinte2, t1, t2)
                print("copil 1 copil 2")
                copii.append(copil1)
               # copii.append(copil2)
                print("copii")
            copiiMutanti = []
            for i in range(len(copii)):
```

```
            print("for", i)
            print("copii total", copii)
            print("copil ce urmeaza sa fi mutat", copii[i])
            mutant = twoSwap(copii[i])
            print("mutant")
            copiiMutanti.append(mutant)
        pop = bestOfGenerations(copii, copiiMutanti, orase)
        best.append(pop[0])
        worst.append(pop[-1])
        average.append(mediePop(orase ,pop))
        g = g+1
    medie = 0
    print("best", best)
    print("worst", worst)
    for i in range(0, len(average)):
        medie = medie + average[i]
    medie = medie / len(average)
    allAverage.append(medie)
    allBest = bestOfAll(orase, best)
    print("all best", allBest)
    fctFList = []
    for i in range(len(allBest)):
        fctFList.append(fitness(orase, allBest[i]))
    print("fctList", fctFList)
    print("fitness all best 1", fitness(orase, allBest[0]))
    print("fitness all best 2", fitness(orase, allBest[len(allBest)-1]))
    print("len all best", len(allBest))
    allBest1 = allBest[0]
    allWorst = allBest[len(allBest)-1]
    print("all best", allBest)
    print("all worst", allWorst)
    bestSol = fitness(orase, allBest1)
    worstSol = fitness(orase, allBest[len(allBest)-1])
    print("best sol", bestSol)
    print("medie", medie)
    print("worst sol", worstSol)
    print("best", best)
    print("worst", worst)
```

Functia care ruleaza algoritmul evolutiv, este o functie care urmeaza aceeasi pasi ca si cea de la problema rucsacului

Nr generatii	populatie	best	averge	Timp 10 rulari
10	10	187884.88060505534	189799.16542862015	235.5416660308838
50	50	181920.3216793993	185963.1629723184	316.39731764793396
100	100	52592.75131682556	111894.28779982787	727.2747991085052
150	100	12430.418434781557	72702.67150054792	1072.6649754047394

Observam ca o data cu cresterea nr de generatii si a populatii avem o valoare best din ce in ce mai buna. Timpul de asemenea creste

Instanta problemei	Nr k	t	T min	alfa	Val medie	Val best	Nr executii	Timp executie
KroC100	100	0.00001	100000	0.9999	194867.66189047275	193561.52827596	10	{136.564089298 2483}
	300	0.00001	100000	0.9999	194008.14667173388	192951.49276824328	10	382.6438672542572
	500	0.00001	100000	0.9999	194260.26352420152	191926.1666265798	10	623.8953146934509}
	700	0.00001	100000	0.9999	194342.31077250073	191498.46251482685	10	991.3936157226562

Dupa cum vedem dupa valoriile obtinute cu Simulated anealing in laboratorul trecut, cu algoritmi evolutivi avem niste solutii mult mai bune

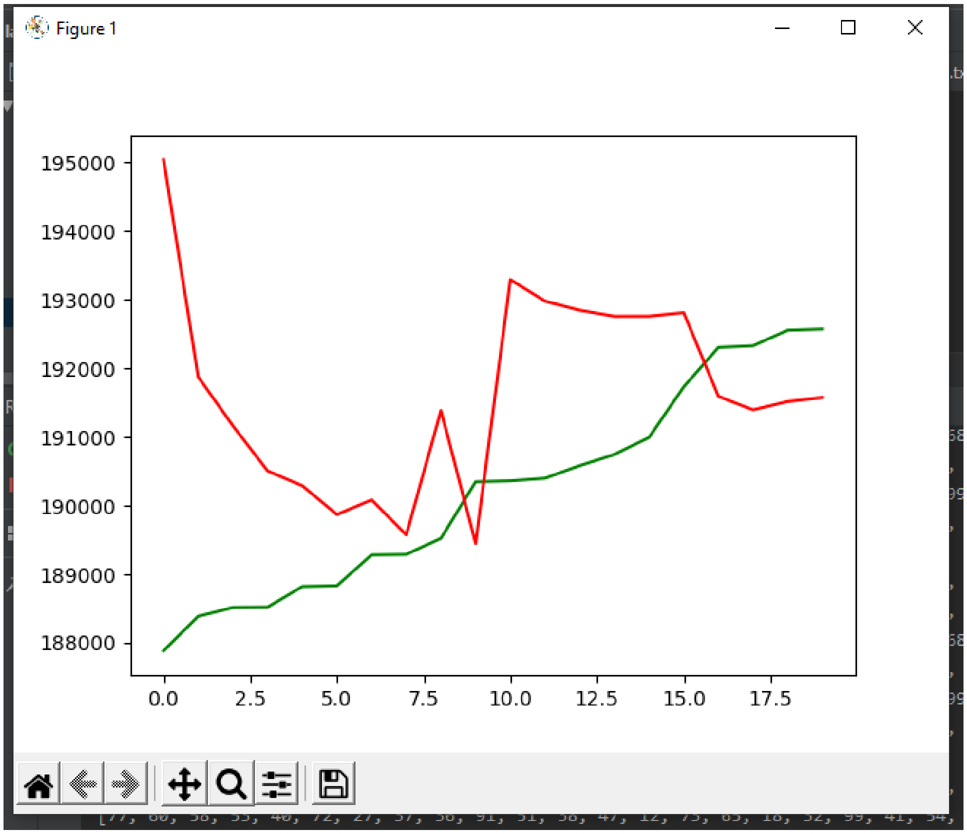


Figure 1
195000
194000
193000
192000
191000
190000
189000
188000
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

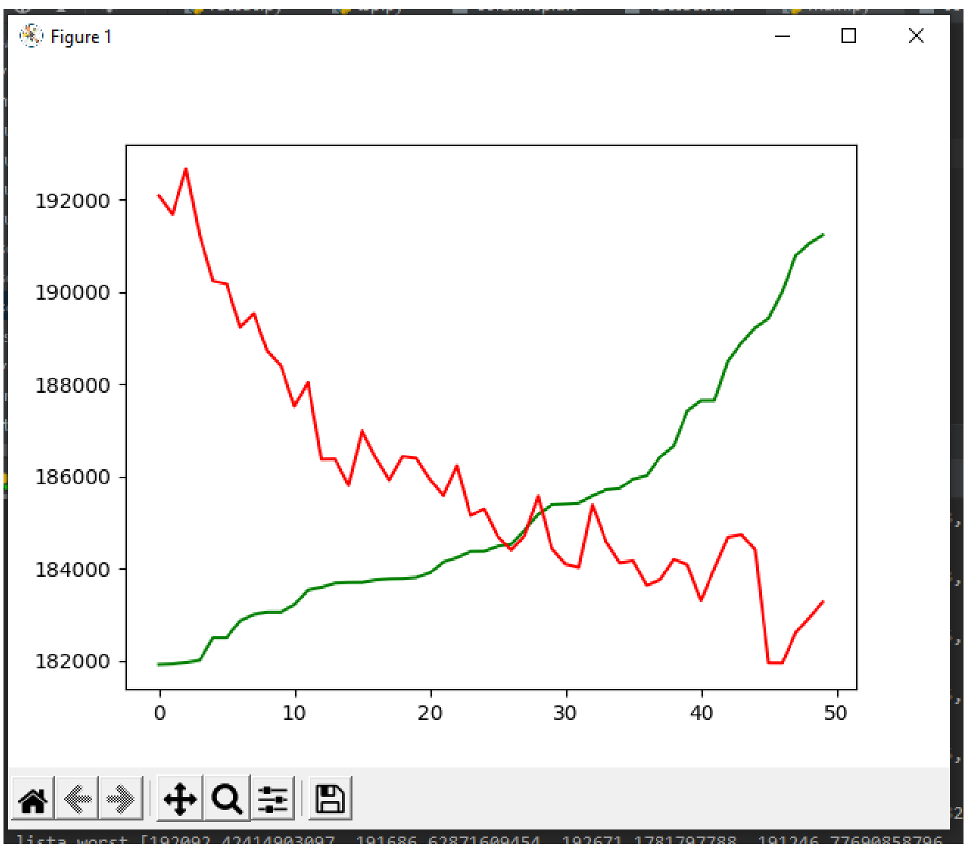


Figure 1
192000
190000
188000
186000
184000
182000
0
10
20
30
40
50

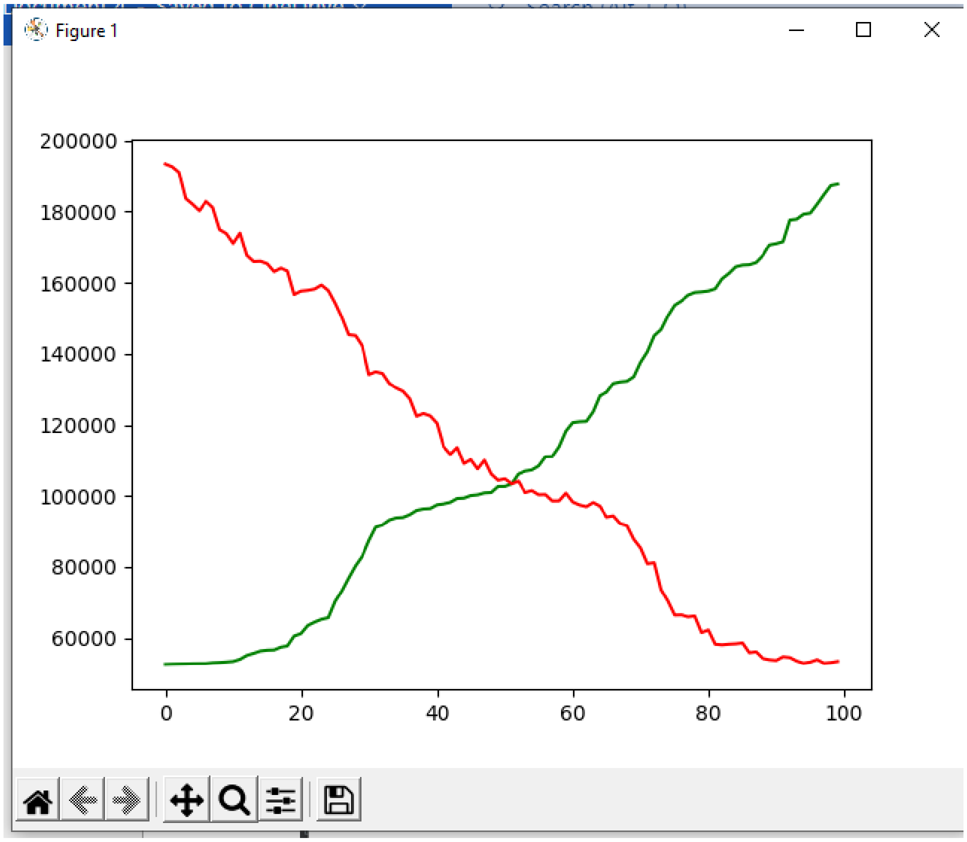


Figure 1
200000
180000
160000
140000
120000
100000
80000
60000
0
20
40
60
80
100