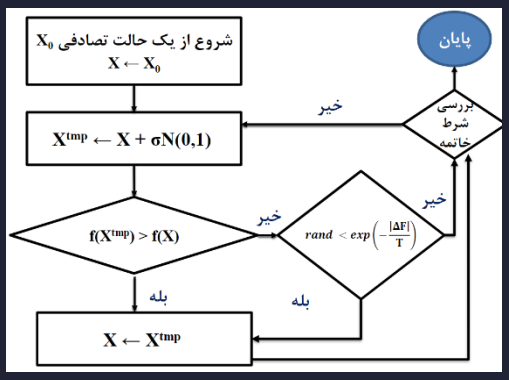
عرفان عابدی

تابع هدف: تعداد کلاز هایی که در CNF ترو میشود را برمیگرداند. مطلوب ما ماکسیمم کردن این تابع است.

```python
def countTrueClauses(clauses, variables):
    count = 0
    for clause in clauses:
        for literal in clause:
            if literal == variables[abs(literal) - 1]:
                count += 1
                break
    return count
```

الگوریتم simulated annealing:

فلوچارت الگوریتم به این گونه است:



کد:

```python
def simulatedAnnealing(clauses, varNum, T = 10000, Tmin = 0.00001, Alpha = 0.998):
    variables = [(random.choice([-1, 1]) * i) for i in range(1, varNum+1)]


    E = countTrueClauses(clauses, variables)
    Eth = len(clauses)

    Elist = [E]
    Tlist = [T]

    while (T > Tmin and E < Eth):
        randIndex = random.randint(0, varNum-1)

        newVariables = variables.copy()
        newVariables[randIndex] = -newVariables[randIndex]

        newE = countTrueClauses(clauses, newVariables)

        if (simulatedAnnealingAccept(newE - E, T)):
            variables = newVariables
            E = newE
            Elist.append(E)
            Tlist.append(T)

        T *= Alpha

    print("True clauses: " + str(E), "T: " + str(Tlist[0]), "Alpha: " + str(Alpha))

    return variables, Elist

```

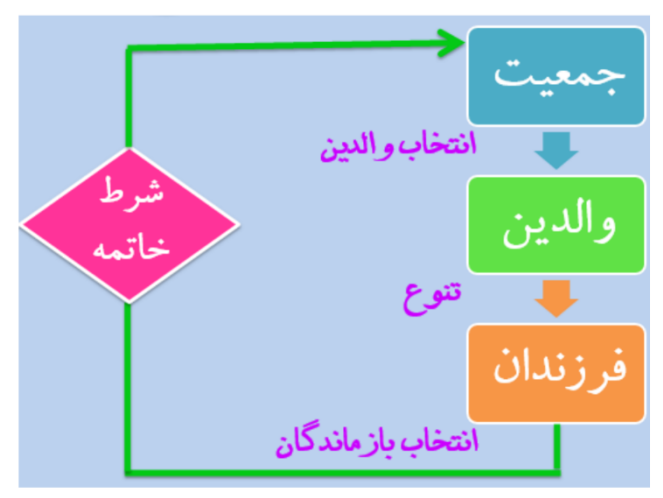
```python
def simulatedAnnealingAccept(deltaE, T):
    if deltaE > 0:
        return True
    else:
        return random.random() < math.exp(-abs(deltaE)/T)

```

عرفان عابدی

در این الگوریتم مقادیر T و alpha را تعیین میکنیم و سپس ابتدا یک variables با مقادیر رندوم درنظر میگیریم و مقدار E را برای آن ذخیره میکنیم.

حال یک فرزند از variables تولید میکنیم به طوری که یکی از متغیر ها نقیض شود و مقدار E را برای فرزند محاسبه میکنیم. سپس با استفاده از تابع accept تعیین میکنیم که این فرزند را نگه داریم یا نه. در صورتی که E آن بیشتر باشد قبولش میکنیم و درصورتی که نباشد با یک احتمالی ممکن است قبولش کنیم(برای فرار از local optimum).

الگوریتم genetic:



در این الگوریتم با استفاده از جمعیتی که در اختیار داریم چندین والد انتخاب میکنیم و سپس فرزندان آنهارا به جمعیت جدید اضافه میکنیم و در انتها در جمعیت جدید بازماندگان را انتخاب میکنیم. و اگر شرط خاتمه satisfy نشد به iteration بعدی میرویم.

کد:

در اینجا انتخاب والدین را بر اساس رتبه بندی و انتخاب بازماندگان را بر اساس شایسته سالاری انجام میدهیم.

```
def crossOver(parent1, parent2, Pc):
    randIndex = random.randint(0, len(parent1)-1)

    child1 = list(parent1).copy()
    child2 = list(parent2).copy()

    if random.random() < Pc:
        child1 = list(parent1[:randIndex] + parent2[randIndex:])
        child2 = list(parent2[:randIndex] + parent1[randIndex:])

    return child1, child2

def Mutation(parent, Pm):
    randIndex = random.randint(0, len(parent)-1)

    child = parent.copy()

    if random.random() < Pm:
        child[randIndex] = -child[randIndex]

    return tuple(child)
```

```python
def genetic(clauses, varNum, MaxIteration = 2000, PopulationSize = 10, Pc = 0.8, Pm = 0.1):
    population = {}

    while len(population) < PopulationSize:
        variables = [(random.choice([-1, 1]) * i) for i in range(1, varNum+1)]
        E = countTrueClauses(clauses, variables)
        population[tuple(variables)] = E

    population = dict(sorted(population.items(), key=lambda item: item[1]))

    Emax = list(population.values())[-1]
    Eth = len(clauses)
    Elist = [Emax]

    iteration = 0
    while (iteration < MaxIteration and Emax < Eth):
        newPopulation = {}

        for i in range(PopulationSize//2):
            parents = random.choices(list(population.keys()),
                                     range(PopulationSize, 0, -1),
                                     k = 2)

            child1, child2 = crossOver(parents[0], parents[1], Pc)

            child1 = Mutation(child1, Pm)
            child2 = Mutation(child2, Pm)

            E1 = countTrueClauses(clauses, child1)
            E2 = countTrueClauses(clauses, child2)

            newPopulation[child1] = E1
            newPopulation[child2] = E2

        newPopulation.update(population)

        population = dict(sorted(newPopulation.items(),
                                 key=lambda item: item[1], reverse=True)[:PopulationSize])


        Emax = list(population.values())[-1]
        Elist.append(Emax)

        iteration += 1

    print("True clauses: " + str(Emax), "MaxIteration: " + str(MaxIteration),
          "PopulationSize: " + str(PopulationSize),
          "Pc: " + str(Pc), "Pm: " + str(Pm))

    variables = list(list(population.keys())[-1])

    return variables, Elist
```

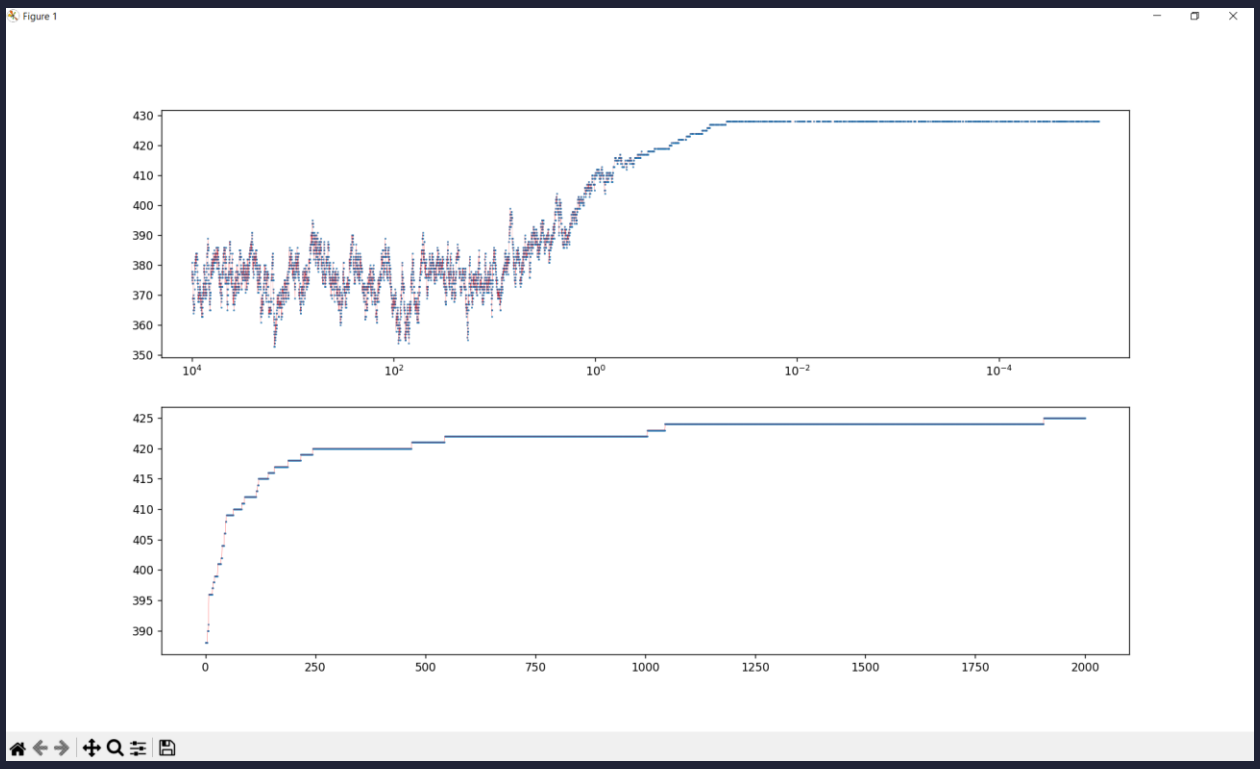
عرفان عابدی

مقایسه دو الگوریتم:

در الگوریتم simulated annealing چون در ابتدا T بزرگ است شرط $random < e^{(\frac{-deltaE}{T})}$ با احتمال زیادی True میشود درنتیجه الگوریتم اجازه میدهد به Eهای بدتر هم برود اما هنگامی که T کمتر از ۱۰ میشود Eهایی که بیشتر میشود را قبول میکند.

اما در الگورتیم genetic به خاطر داشتن جمعیت، تنوع بهتر حفظ میشود و با چندبار اجرا با انتخاب شایسه سالار به نتیجه مطلوب نزدیک میشویم.

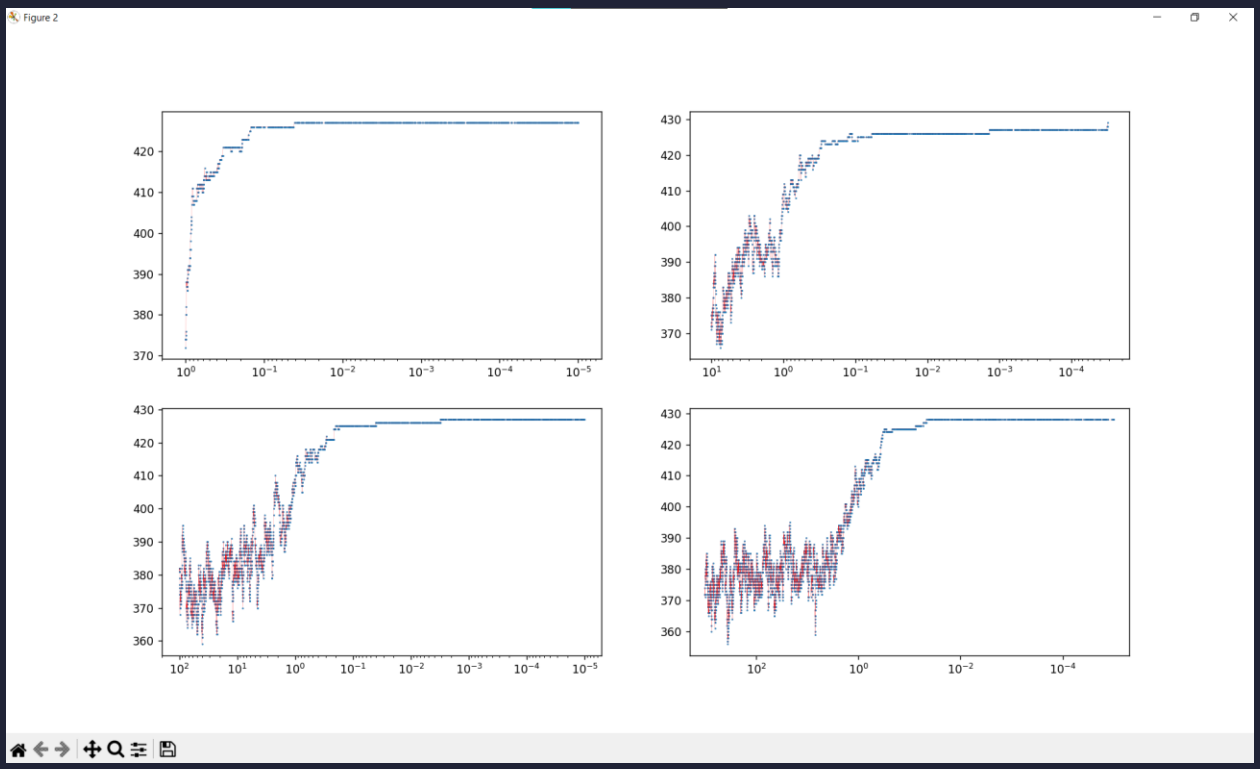
```
SimulatedAnnealing  T: 10000 Alpha: 0.998
True clauses: 429
variables: [-1, -2, 3, 4, -5, 6, -7, 8, 9, 10, 11, -12, -13, -14, 15, 16, -17, -18, -19, -20, 21, -22, -23, -24, -25, -26, -27, 28, 29, -30, 31, -32, 33, -34, 3
5, -36, -37, 38, -39, 40, -41, 42, 43, 44, 45, 46, -47, -48, 49, 50, -51, 52, -53, -54, -55, -56, -57, -58, 59, 60, -61, 62, 63, -64, 65, 66, 67, -68, 69, -70,
-71, 72, -73, -74, -75, -76, 77, 78, 79, -80, -81, -82, -83, 84, -85, 86, -87, -88, -89, 90, -91, -92, 93, 94, 95, 96, 97, -98, 99, 100]
Genetic  MaxIteration: 2000 PopulationSize: 10 Pc: 0.8 Pm: 0.1
True clauses: 426
variables: [-1, -2, 3, 4, -5, 6, 7, 8, -9, 10, -11, -12, 13, 14, 15, 16, -17, -18, -19, 20, 21, -22, 23, 24, -25, -26, -27, 28, 29, 30, 31, -32, -33, -34, -35,
-36, -37, -38, 39, 40, 41, 42, 43, 44, 45, 46, -47, -48, -49, -50, -51, 52, -53, -54, 55, -56, -57, -58, 59, 60, 61, -62, -63, 64, -65, -66, 67, -68, 69, -70, 7
1, 72, -73, -74, -75, 76, 77, 78, 79, 80, 81, -82, -83, 84, 85, -86, 87, -88, -89, 90, -91, -92, -93, 94, 95, -96, 97, -98, -99, 100]
```



عرفان عابدی

مقایسه های داخل الگوریتمی:

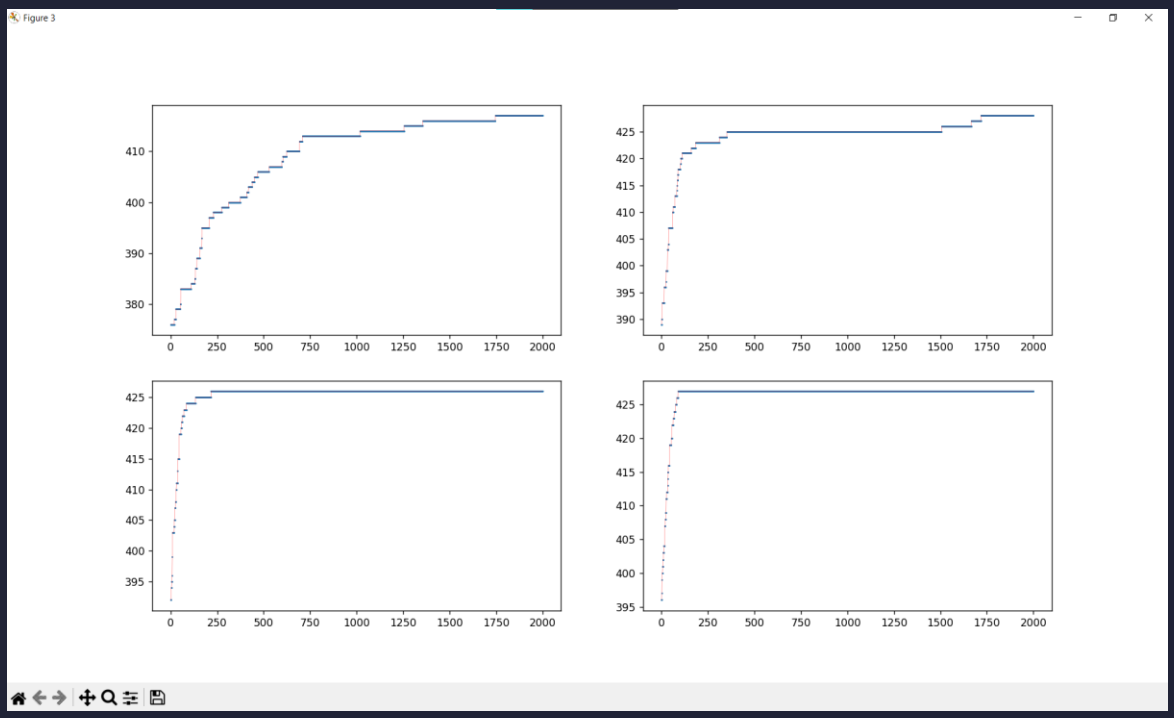
در الگوریتم simulated annealing با زیاد کردن T در ابتدای الگوریتم بیشتر به Eهای کمتر شانس میدهد.



T=1, T=10, T=100, T=1000

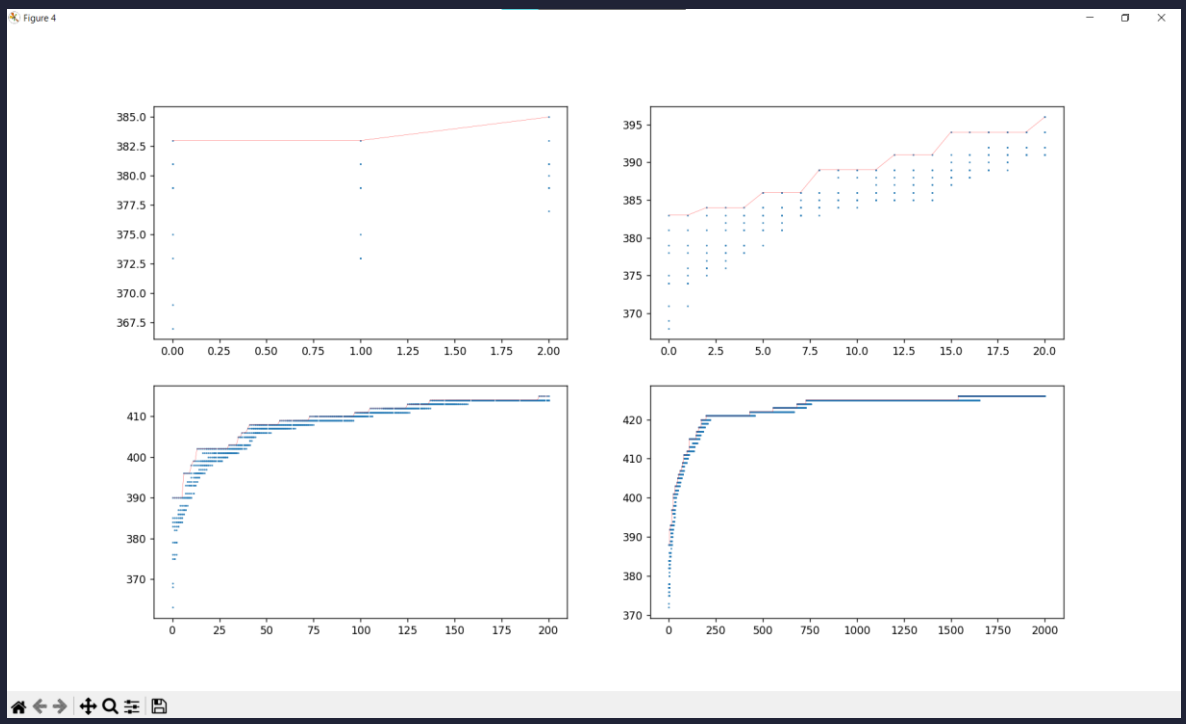
عرفان عابدی

در الگوریتم genetic هنگامی که جمعیت را افزایش میدهیم چون تنوع بیشتر میشود زودتر به حالت مطلوب میرسد



PopulationSize=2, PopulationSize=20, PopulationSize=200, PopulationSize=2000

هرچه iteration بیشتر میشود تنوع به سمت حالت مطلوب همگرا تر میشود.



Iteration=2, iteration=20, iteration=200, iteration=2000