• Simulated Annealing

در این روش ابتدا کلاس Individual را تعریف میکنیم که در آن ویژگی های literals که به معنای گزاره های اتمیک ما هستند، chromosome که آرایه ای را نگه میدارد که مجموعه ای از درست بودن و یا غلط بودن گزاره اتمیک iام را نگه میدارد که در واقع ژن ما را ما تشکیل میدهند، همچنینبا توجه به کروموزوم ما در این کلاس، fitness که همان تابع هدف ما است با استفاده از تابع cal_fitness ساخته میشود.

```
class Individual(object):

    def __init__(self, chromosome):
        self.literals = []
        for i in range(formula.nv):
            self.literals.append((i + 1))
        # Generate random list with positive or negative integers
        if chromosome is not None:
            self.chromosome = chromosome
        else:
            self.chromosome = [num if random.choice([True, False])
                               else -num for num in self.literals]

        self.fitness = self.cal_fitness()
```

در تابع cal_fitness بر روی جملات اتمیک موجود در کلاز ها ایتریت میکنیم و اگر در کروموزوم ما موجود بود هر گزاره اتمیک، یکی به امتیاز اضافه میشود. دلیل آن این است که موجود بودن آن در کروموزم نشان دهنده صحت آن گزاره اتمیک است و اگر حتی یکی از جملات ترکیب فصلی درست شود، آن جمله درست میگردد. اگر مقدار بازگشتی

این تابع برابر با تعداد ترکیب فصلی های موجود در CNF باشد پس CNF ما valid میشود.

```
def cal_fitness(self):

    fitness = 0
    for gs in formula.clauses:
        for lits in gs:
            if lits in self.chromosome:
                fitness += 1
                break
    return fitness
```

تابع اصلی که simulated_annealing است ورودی های آن فرمول cnf، ماکسیمم تعداد ایتریشن که شرط پایان است، دمای اولیه و نرخ سرد شدن دما میباشند و خروجی های این تابع سه چیز است: یکی جوابی که الگوریتم ما پیدا کرده است، یکی تعداد ایتریشن هایی که طول کشیده است تا جواب پیدا شود و امتیازاتی که توانسته است پیدا کند.

```
def simulated_annealing(formula, max_iterations,
                        initial_temperature, cooling_rate):
```

الگوریتم به این صورت کار میکند که در ابتدا فردی رندوم را میسازیم، امتیاز او و ژنش را در متغیر ها ذخیره میکنیم و فرض میکنیم امتیاز او بهترین امتیاز است و بهترین جواب همین ژن فرد ماست.

```
# Initialize the current solution

current_individual = Individual(None)
current_cost = current_individual.fitness
current_solution = current_individual.chromosome

# Initialize the best solution
best_solution = current_solution.copy()
best_cost = current_cost

# Initialize the temperature
temperature = initial_temperature
itr = 1
```

حال وارد ایتریشن اصلی میشویم و در آنجا ابتدا از فرد خودمان همسایه را میابیم و روش پیدا کردن همسایه این گونه است که عددی رندوم بین صفر تا تعداد گزاره های اتمیک موجود در فرمول CNF پیدا میکنیم و سپس اگر عدد پیدا شده j باشد، بیت jام خودمان

را فلیپ میکنیم تا همسایه ای ساخته شود، حالا امتیاز همسایه را نیز پیدا میکنیم و اختلاف این دو را در میابیم.

```
scores = []
# Iterate for the specified number of iterations
for iteration in range(max_iterations):
    # Generate a neighbor solution by flipping a random variable
    neighbor_solution = current_solution.copy()

    index = random.randint(0, formula.nv - 1)
    neighbor_solution[index] = -neighbor_solution[index]

    neighbor_cost = Individual(neighbor_solution).fitness

    # Calculate the cost difference between the current and neighbor solutions
    cost_difference = neighbor_cost - current_cost
```

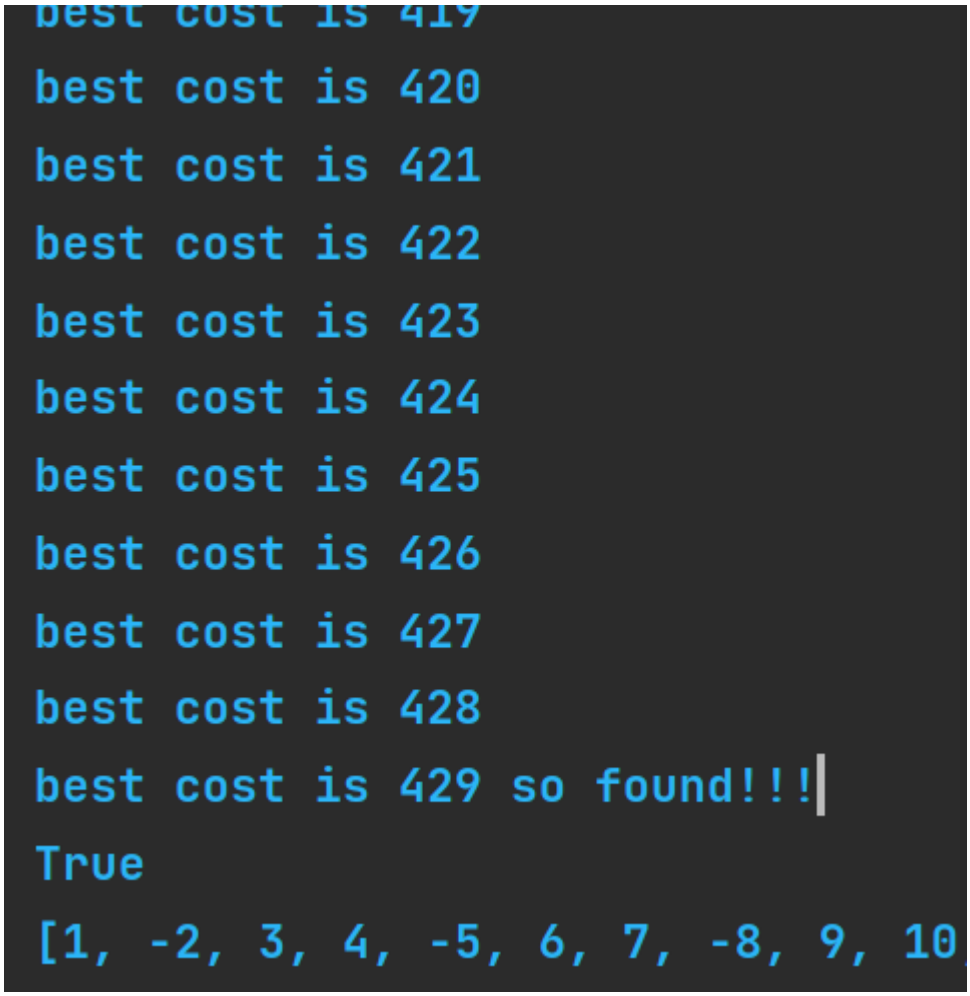
حال اگر این اختلاف مثبت بود و یا شرط simulated annealing برقرار بود برای شانس دوباره، بهترین جواب ما xای است که الان با نویز شبیه سازی درست شده است، در غیر این صورت این x را قبول نمیکنیم. در اخر هم دما را سردتر میکنیم و امتیاز به دست امده در این ایتریشن را به scores اضافه میکنیم.

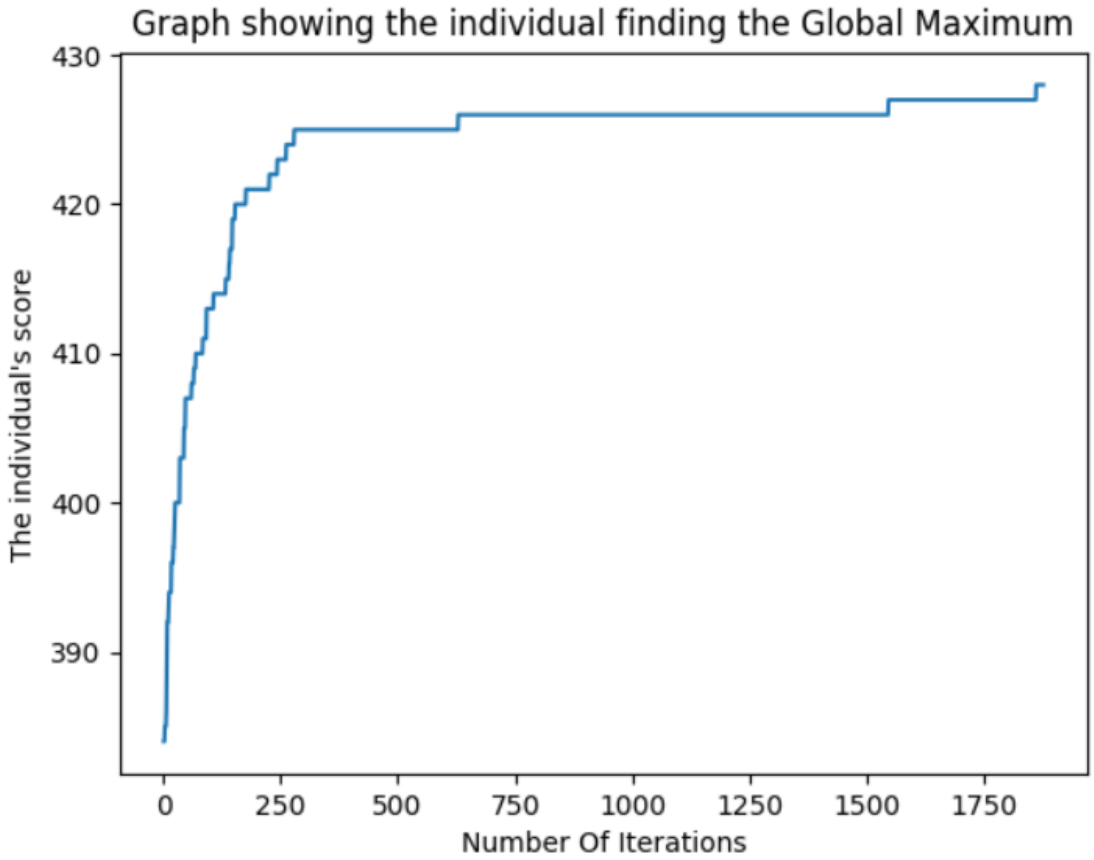
```
if cost_difference > 0 or \
        random.random() < math.exp(-abs(cost_difference) / temperature):
    current_solution = neighbor_solution
    current_cost = neighbor_cost

# Update the best solution
if current_cost > best_cost:
    best_solution = current_solution.copy()
    best_cost = current_cost
    if best_cost == len(formula.clauses):
        print("best cost is " + str(best_cost) + " so found!!!")
        break
    print("best cost is " + str(best_cost))

# Decrease the temperature
temperature *= cooling_rate
itr += 1
scores.append(best_cost)
```

حال در main تابع را صدا زده و نتایج را ذخیره میکنیم در آخر هم بر روی نمودار نشان میدهیم که در شکل های زیر اورده شده است. اگر دقت کنید در ابتدا الگوریتم به خوبی امتیازات بالا را پیدا کرده است و در اخر به همگرایی رسیده است که خوشبختانه در این شبیه سازی به نتیجه هم رسیدیم:)

```
result = simulated_annealing(formula,
                             max_iterations=50000,
                             initial_temperature=2.0,
                             cooling_rate=0.80)
solution = result[0]
itr = result[1]
scores = result[2]
solver = Solver()
solver.append_formula(formula.clauses)
print(solver.solve(assumptions=solution))
# Print the solution
print(solution)

plt.plot([i + 1 for i in range(itr - 1)], scores)
plt.xlabel('Number Of Iterations')
plt.ylabel("The individual's score")
plt.title('Graph showing the individual finding the Global Maximum')
plt.show()
```

طرز کار الگوریتم (درآخر مجموعه جواب هم چاپ میشود).





- **Genetic**

در این تابع هم کلاس individual با اسم Gene موجود هستو تابع cal_fitness نیز همان قبلی میباشد.
تابع create_child بدین صورت کار میکند که بر روی کروموزوم های پدر و مادر ایتریت میکنیم و در هر ایتریشن عددی رندوم را انتخاب میکنیم که اگر این احتمال از 3/1 کمتر بود از والد اول و اگر بیشتر بود از والد دوم انتخاب میکنیم(به گونه ای از روش خط کش احتمال و میانگیری در اسلاید ها استفاده کردم:]) حال بعد از ساخته شدن فرزند یکی از بیت های او را فلیپ میکنیم که به گونه ای جهش را نشان میدهد، البته جهش در

صورتی انجام میشود که اگر عدد رندومی که انتخاب کردیم از MUTATION_RATE کمتر باشد. حال این تابع فرزند جدید را بازمیگرداند.

```
def create_child(self, par2):

    child_chromosome = []
    i = 0
    for gp1, gp2 in zip(self.chromosome, par2.chromosome):
        parent_chooser = random.random()

        child_chromosome.append(gp1 if parent_chooser < 1 / 3
                                else gp2)

        i += 1

    mutated_chrome = (self.mutated_genes(child_chromosome)
                      if random.random() < MUTATION_RATE
                      else child_chromosome)
    return Gene(mutated_chrome)
```

تابع mutated_genes هم بدین صورت است عددی رندوم از بین صفر تا تعداد گزاره های اتمیک پیدا میکنیم و سپس اگر i انتخاب شد، بیت iام را فلیپ میکنیم.

```
def mutated_genes(self, chromosome):
    mutated_idx = random.randint(0, formula.nv - 1)
    chromosome[mutated_idx] = -chromosome[mutated_idx]
    return chromosome
```

حال در تابع اصلی ابتدا جمعیتی شانسی میسازیم در فوری به اندازه هایپرپارامتر تعداد جمعیت. سپس در فور اصلی که هر نسل را نشان میدهد ابتدا به صورت نزولی جمعیت را بر حسب امتیازشان سورت میکنیم و سپس چک میکنیم که اگر نفر اول جمعیت با بیشترین امتیاز، امتیازش برابر تعداد ترکیب های فصلی شده است(همه جملات را درست کرده است) پس الگوریتم تمام میشود.

```
generation = 1
population = []

for _ in range(POPULATION_SIZE):
    population.append(Gene(None))

scores = []
avg = []
for _ in range(NUM_GENERATIONS):
    population = sorted(population, key=lambda x: -x.fitness)

    if population[0].fitness == len(formula.clauses):
        break
```

در غیر این صورت این فرد برتر را به نسل بعدی میبریم(شایسته سالاری حالت پادشاهی:] و سپس فرزندان را از 100 تای اول جمعیت میسازیم که باز هم شایسته سالاری است ولی در این مسله همگرایی بسیار خوبی میدهد) ولی با احتمال MATE_RATE که در ژنتیکبالا میباشد، در اخر هم با احتمال جهش، فرد ساخته شده را جهش میدهیم:

```
new_generation.extend(population[:1])

for _ in range(7999):
    parent1 = random.choice(population[:100])
    parent2 = random.choice(population[:100])
    if random.random() < MATE_RATE:
        child = parent1.create_child(parent2)
    else:
        child = (parent1 if random.random() < 0.5 else parent2)
    if random.random() < MUTATION_RATE:
        child = (Gene(child.mutated_genes(child.chromosome))
                 if random.random() < MUTATION_RATE
                 else child)
    new_generation.append(child)
MUTATION_RATE -= 0.02
```

حال در اخر جمعیت را اپدیت میکنیم و در آخر امتیاز افراد ان جمعیت را باهم جمع و تقسیم بر تعداد جمعیت میکنیم تا در نمودار استفاده شود(حاصل در avg ذخیره میگردد) در آخر جواب را چاپ میکنیم که اگر پیدا شد True چاپ میشود و در غیراین صورت False.

```
    print("Generation: " + str(generation) +
          "  Fitness: " + str(population[0].fitness))
    avg.append(sum(scores) / len(scores))
    scores = []
    generation += 1
if population[0].fitness < len(formula.clauses):
    print(False)
    return

print("Generation: " + str(generation)
      + "  Fitness: " + str(population[0].fitness))

solver = Solver()
solver.append_formula(formula.clauses)
print(solver.solve(assumptions=population[0].chromosome))
print(population[0].chromosome)
```

```
Generation: 1  Fitness: 399
Generation: 2  Fitness: 405
Generation: 3  Fitness: 412
Generation: 4  Fitness: 416
Generation: 5  Fitness: 419
Generation: 6  Fitness: 423
Generation: 7  Fitness: 425
Generation: 8  Fitness: 427
Generation: 9  Fitness: 427
Generation: 10  Fitness: 428
Generation: 11  Fitness: 429
True
[-1, -2, 3, 4, 5, 6, -7, 8, 9, 10,
```

اگر به نمودار دقت کنیم، نشان دهنده این است که همگرایی بسیار خوبی داریم به طوری که نمودار شبه خطی شده است و همینطور در عین تنوع(5000نفر جمعیت) تکامل هم داریم و از میانگین امتیازات جایی کم نشده است.