Computational Intelligence
Dr. Mozayani
Fall 2022
Hoorieh Sabzevari - 98412004
HW4

۱. دو مورچه را فرض کنید که در حال حرکت از آشیانه به منبع غذایی، از طریق دو مسیر کاملا متفاوت از هم هستند. مورچه‌ها در ضمن حرکت خود به سمت منبع غذایی، ردی از «فرومون» (Pheromone) در محیط منتشر می‌کنند که به‌طور طبیعی و با گذر زمان متلاشی می‌شود. مورچه‌ای که (به‌طور تصادفی) کوتاهترین مسیر به سمت منبع غذایی را انتخاب کرده، سفر برگشتی به سمت آشیانه را زودتر از دیگر مورچه‌ها آغاز می‌کند. در چنین حالتی، این مورچه در مسیر بازگشت به آشیانه، دوباره شروع به منتشر کردن فرومون در محیط می‌کند و از این طریق، رد فرومون به جا گذاشته در کوتاهترین مسیر را تقویت می‌کند. مورچه‌های دیگر، به‌طور غریزی، قوی‌ترین مسیر فرومون موجود در محیط را دنبال و رد فرومون در این مسیر را تقویت می‌کنند. پس از گذشت مدت زمان مشخصی، نه تنها رد فرومون موجود در کوتاه‌ترین مسیر متلاشی نمی‌شود، بلکه، با انباشته شدن رد فرومون دیگر مورچه‌ها، بیش از پیش تقویت می‌شود. مسیری که قوی‌ترین رد فرومون در آن به جا گذاشته شده باشد، به مسیر پیش فرض برای حرکت مورچه‌ها از کلونی به منبع غذایی و برعکس تبدیل می‌شود.

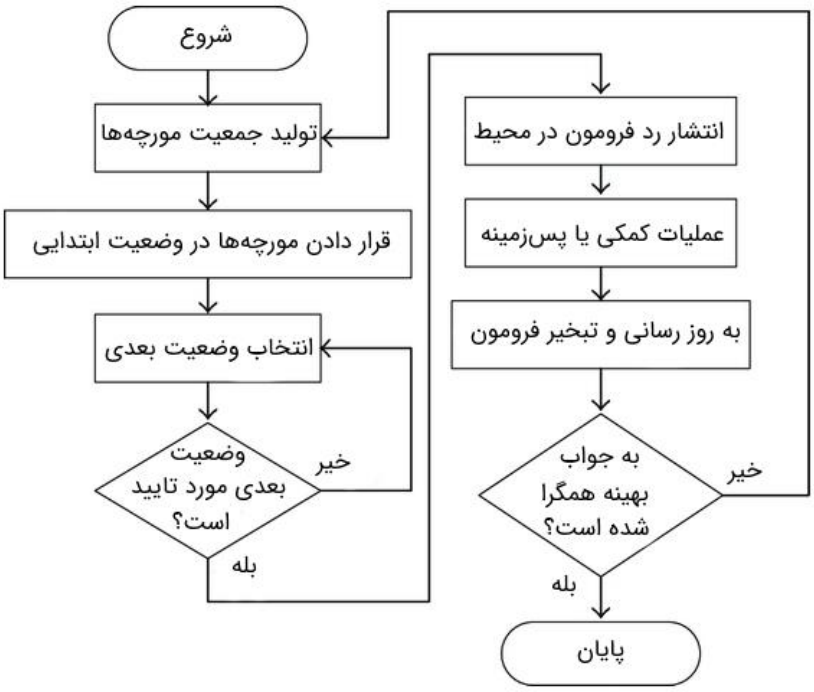
فرومون به مرور تبخیر می‌شود که از سه جهت مفید است:

باعث می‌شود مسیر جذابیت کمتری برای مورچه‌های بعدی داشته باشد. از آنجا که یک مورچه در زمان دراز راه‌های کوتاه‌تر را بیش تر می‌پیماید و تقویت می‌کند هر راهی بین خانه و غذا که کوتاه‌تر (بهتر) باشد بیشتر تقویت می‌شود و آنکه دورتر است کمتر.

اگر فرومون اصلاً تبخیر نمی‌شد، مسیرهایی که چند بار طی می‌شدند، چنان بیش از حد جذّاب می‌شدند که جستجوی تصادفی برای غذا را بسیار محدود می‌کردند.

وقتی غذای انتهای یک مسیر جذاب تمام می‌شد رد باقی می‌ماند.

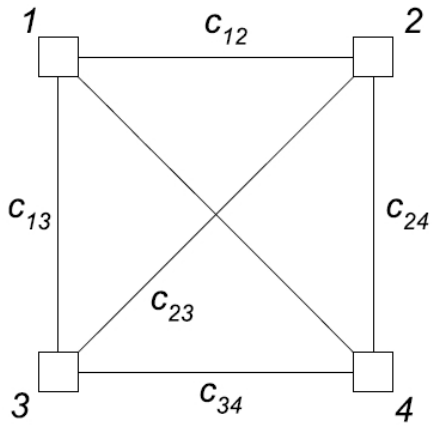
فلوچارت الگوریتم:



یکی از محبوب‌ترین روش‌ها برای نمایش چگونگی عملکرد روش فرا اکتشافی الگوریتم کلونی مورچگان، استفاده از آن در حل مسأله فروشنده دوره‌گرد است. این مسأله، از مجموعه‌ای از شهرها (مکان‌ها) و یک فروشنده دوره‌گرد تشکیل شده است. این فروشنده اجازه دارد از هر شهر تنها یک‌بار عبور کند. فاصله میان شهرها داده شده است و هدف پیدا کردن «تور همیلتونی» (Hamilton Cycle) با طول کمینه است. پیچیدگی این مسأله برابر با (NP-hard) است.

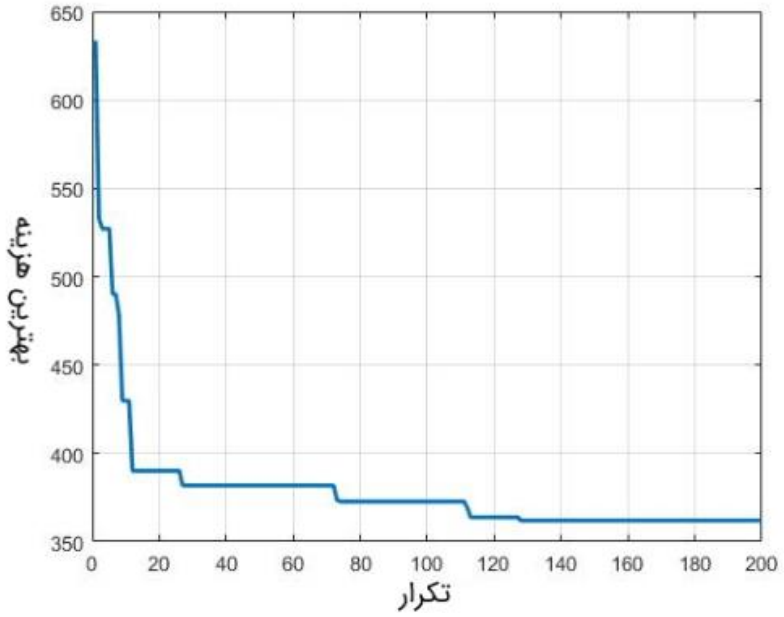
کاربرد بهینه‌سازی کلونی مورچگان برای حل مسأله فروشنده دوره‌گرد ساده و صریح است. حرکت میان شهرها (مکان‌ها)، مؤلفه‌های جواب کاندید است؛ یعنی، حرکت از شهر i به شهر j، مولفه جواب کاندید $C_{ij} = C_{ji}$ مسئله خواهد بود. گراف ساختاری $G_c = (V, E)$ از طریق ایجاد تناظر میان مجموعه شهرها با مجموعه رأس‌ها V در گراف ساختاری تعریف می‌شود.

از آنجا که هیچ محدودیتی در امکان حرکت از یک شهر به هر شهر دیگری وجود ندارد، گراف ساختاری تشکیل شده یک گراف کاملا متصل است و تعداد رأس‌های موجود در گراف برابر تعداد شهرهای تعریف شده در مسأله خواهد بود. همچنین، اندازه یال‌های گراف متناسب با فاصله شهرها (با رأس‌ها نمایش داده می‌شوند) از یکدیگر است. فرومون نیز متناظر با مجموعه یال‌ها E در گراف ساختاری خواهد بود. نمونه‌ای از یک گراف کامل متصل برای مسأله فروشنده دوره‌گرد با تعداد چهار شهر در شکل زیر آمده است.



مورچه‌ها به شکلی که در ادامه بیان شده است، اقدام به تولید جواب‌های مسأله می‌کنند. هر کدام از مورچه‌ها، از یک شهر (یک رأس در گراف) کاملا تصادفی شروع می‌کنند. سپس، در هر گام از فرآیند تولید جواب، در راستای یال‌های گراف به حرکت می‌پردازند. هر مورچه، مسیر پیموده شده در گراف را به خاطر می‌سپارد و در گام‌های بعدی، یال‌هایی را برای حرکت در گراف انتخاب می‌کند که به مکان‌های (رأس‌های) از پیش پیموده شده منتهی نشوند. به محض اینکه تمامی رأس‌های گراف توسط یک مورچه پیمایش شد، یک جواب کاندید تولید می‌شود.

در هر گام از فرآیند تولید جواب، مورچه‌ها به طور احتمالی، از میان یال‌های در دسترس (یال‌های پیموده نشده و منتهی به رأس‌هایی که از آن‌ها گذرنکرده)، یک یال را برای پیمایش انتخاب می‌کنند. نحوه محاسبه احتمال انتخاب یال‌ها، به پیاده‌سازی انجام شده از الگوریتم کلونی مورچگان بستگی دارد. پس از اینکه تمامی مورچه‌ها یک جواب کاندید تولید کردند، فرومون روی یال‌ها، براساس «قانون به روز رسانی فرومون» به روز رسانی می‌شود.



الگوریتم کلونی مورچگان، در تکرار ۱۲۸ به جواب بهینه سراسری (مقدار هزینه برابر با ۳۶۲.۰۳۸) همگرا شده است. شکل بالا، نحوه همگرایی به هزینه بهینه را نشان می‌دهد. (لینک کمکی)

۲. ابتدا تابع fitness خود را به صورت زیر تعریف می‌کنیم.

$$fitness = \sum_{i=1}^{n} c_i v_i; if \sum_{i=1}^{n} c_i w_i \leq kw$$

$$fitness = 0; otherwise$$

n = chromosome length, c_i *= ith gene,* v_i *= ith value,* w_i *= ith weight, kw = knapsack weight*

```
def cal_fitness(weight, value, population, threshold):
    fitness = np.empty(population.shape[0])
    for i in range(population.shape[0]):
        S1 = np.sum(population[i] * value)
        S2 = np.sum(population[i] * weight)
        if S2 <= threshold:
            fitness[i] = S1
        else :
            fitness[i] = 0
    return fitness.astype(int)
```

تابعی برای انتخاب individual های مناسب تعریف می‌کنیم.

```
def selection(fitness, num_parents, population):
    fitness = list(fitness)
    parents = np.empty((num_parents, population.shape[1]))
    for i in range(num_parents):
        max_fitness_idx = np.where(fitness == np.max(fitness))
        parents[i,:] = population[max_fitness_idx[0][0], :]
        fitness[max_fitness_idx[0][0]] = -999999
    return parents
```

از cross-over تک‌نقطه‌ای استفاده کرده و نرخ cross-over را روی یک مقدار بالا تنظیم خواهیم کرد تا اطمینان حاصل شود که تعداد بیشتری از بهترین نسل‌ها تحت این عمل قرار می‌گیرند.

```python
def crossover(parents, num_offsprings):
    offsprings = np.empty((num_offsprings, parents.shape[1]))
    crossover_point = int(parents.shape[1]/2)
    crossover_rate = 0.8
    i=0
    while (parents.shape[0] < num_offsprings):
        parent1_index = i%parents.shape[0]
        parent2_index = (i+1)%parents.shape[0]
        x = rd.random()
        if x > crossover_rate:
            continue
        parent1_index = i%parents.shape[0]
        parent2_index = (i+1)%parents.shape[0]
        offsprings[i,0:crossover_point] = parents[parent1_index,0:crossover_point]
        offsprings[i,crossover_point:] = parents[parent2_index,crossover_point:]
        i=+1
    return offsprings
```

در جهش اینکه کدام کروموزوم دچار جهش می‌شود، به صورت تصادفی انجام می‌شود. برای ایجاد جهش، از تکنیک bit-flip استفاده خواهیم کرد، یعنی اگر ژن انتخابی که قرار است جهش پیدا کند ۱ باشد، آن را به ۰ تغییر دهیم و بالعکس.

```python
def mutation(offsprings):
    mutants = np.empty((offsprings.shape))
    mutation_rate = 0.4
    for i in range(mutants.shape[0]):
        random_value = rd.random()
        mutants[i,:] = offsprings[i,:]
        if random_value > mutation_rate:
            continue
        int_random_value = randint(0,offsprings.shape[1]-1)
        if mutants[i,int_random_value] == 0 :
            mutants[i,int_random_value] = 1
        else :
            mutants[i,int_random_value] = 0
    return mutants
```

همانطور که تمام توابع لازم تعریف شده‌اند، اکنون آنها را به ترتیب نمودار جریان فراخوانی می‌کنیم تا پارامترهای مورد نیاز را پیدا کرده و تمام مقداردهی‌های اولیه لازم را انجام دهیم.

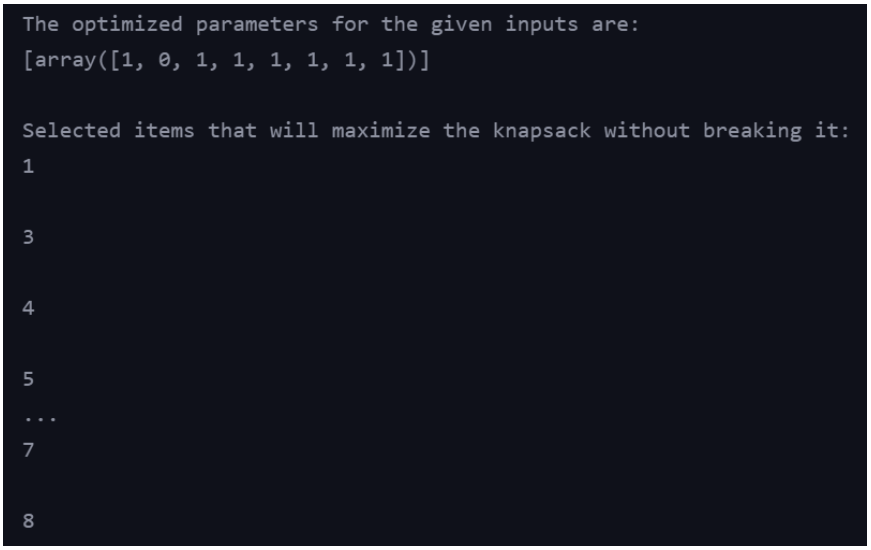
```
def optimize(weight, value, population, pop_size, num_generations, threshold):
    parameters, fitness_history = [], []
    num_parents = int(pop_size[0]/2)
    num_offsprings = pop_size[0] - num_parents
    for i in range(num_generations):
        fitness = cal_fitness(weight, value, population, threshold)
        fitness_history.append(fitness)
        parents = selection(fitness, num_parents, population)
        offsprings = crossover(parents, num_offsprings)
        mutants = mutation(offsprings)
        population[0:parents.shape[0], :] = parents
        population[parents.shape[0]:, :] = mutants

    print('Last generation: \n{}\n'.format(population))
    fitness_last_gen = cal_fitness(weight, value, population, threshold)
    print('Fitness of the last generation: \n{}\n'.format(fitness_last_gen))
    max_fitness = np.where(fitness_last_gen == np.max(fitness_last_gen))
    parameters.append(population[max_fitness[0][0],:])
    return parameters, fitness_history
```

حال ماتریس وزن‌ها و ارزش‌ها را تعریف کرده و جمعیت اولیه را مشخص می‌کنیم. در این مسئله، ایده رمزگذاری کروموزوم این است که کروموزومی متشکل از تعداد ژن‌های موجود باشد، به طوری که هر ایندکس ژن با شاخص آیتم در لیست مطابقت داشته باشد. هر ژن دارای مقدار ۱ یا ۰ است که نشان می دهد آیتم مربوطه وجود دارد یا نه.

```
solutions_per_pop = 8
pop_size = (solutions_per_pop, item_number.shape[0])
print('Population size = {}'.format(pop_size))
initial_population = np.random.randint(2, size = pop_size)
initial_population = initial_population.astype(int)
num_generations = 50
print('Initial population: \n{}'.format(initial_population))
```

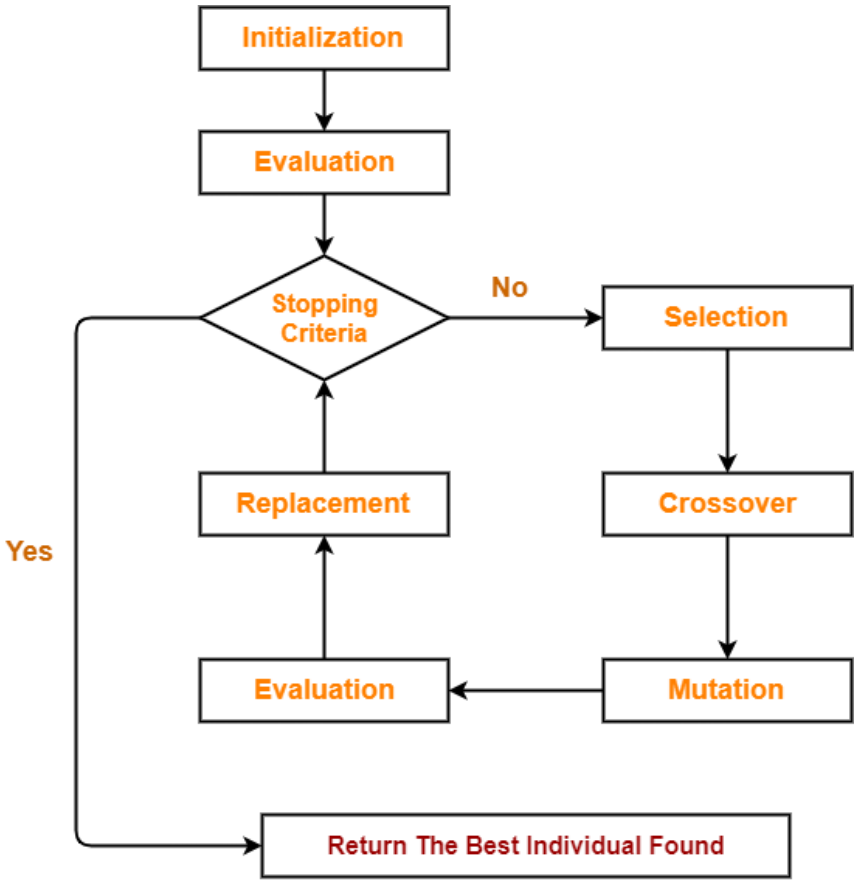
در انتها خروجی الگوریتم برای این مسئله به صورت زیر خواهد بود.

```
parameters, fitness_history = optimize(weight, value, initial_population, pop_size, num_generations, knapsack_threshold)
print('The optimized parameters for the given inputs are: \n{}'.format(parameters))
selected_items = item_number * parameters
print('\nSelected items that will maximize the knapsack without breaking it:')
for i in range(selected_items.shape[1]):
  if selected_items[0][i] != 0:
    print('{}\n'.format(selected_items[0][i]))
```



آیتم های مربوط به پارامترهای آرایه item_number مواردی هستند که نشان می‌دهد بهتر است تمامی کالاها را بجز آیتم شماره۲ که دارای کمترین ارزش است را برداریم. (لینک کمکی)

.٣



f(x) = x^2 (maximize) over {0, 1, 2, …, 50}

Encoding: to find the maximum f(x) = x^2 over the interval 0-50, we can represent each possible solution as a binary string of 5 bits. Also, we can use 6 bits to represent but we must note that the first 3 bits shouldn't become 1 in each one.

Initialization: begin with a random population of (say) 4 strings, each with 5 bits (e.g., 01101, 11000, 01000, 10011).

Evaluation: determine the fitness of each string x in the population by evaluating $F(x)=x^2$. In this case, the fitness values are 169, 576, 64, 361, with an average value of 293 and a maximum of 576.

Reproduction: sum the fitness functions (1170) and reproduce each string in the new population in proportion to its contribution to this sum (e.g., 14.4%, 49.2%, 5.5%, 30.9%).

- Choose a random number r between 1 and 1000.
- If r <= 144 select 01101 else if 145 <= r <= 636 select 11000
  else if 637 <= r <= 692 select 01000 else select 10011.
- Repeat 4 times, or once for each individual in the population.

Using random numbers, we selected 01101, 11000, 11000, and 10011. The string 01000 perished.

Crossover: choose two strings in the new population at random, choose a crossover point within the strings at random, and exchange the upper parts of the two strings.

At crossover point 4, strings 01101 and 11000 mate and produce 01100 and 11001. At crossover point 2, strings 11000 and 10011 mate and produce 11011 and 10000.

Mutation: with small probability (.001), on a bit-by-bit basis, change each bit in the new population. We expect 0.02 bits to change (20 bits * .001); none do in our example.

Evaluation: the fitness function for the new strings is 144, 625, 729, 256, with an average of 439 and a maximum of 729.

(Link)