局部搜索算法

赵耀

爬山算法

```python
def hill_climbing(problem):
    """From the initial node, keep choosing the neighbor with highest value,
    stopping when no neighbor is better. [Figure 4.2]"""
    current = Node(problem.initial)
    while True:
        neighbors = current.expand(problem)
        if not neighbors:
            break
        neighbor = argmax_random_tie(neighbors,
                                     key=lambda node: problem.value(node.state))
        if problem.value(neighbor.state) <= problem.value(current.state):
            break
        current = neighbor
    return current.state
```

爬山算法

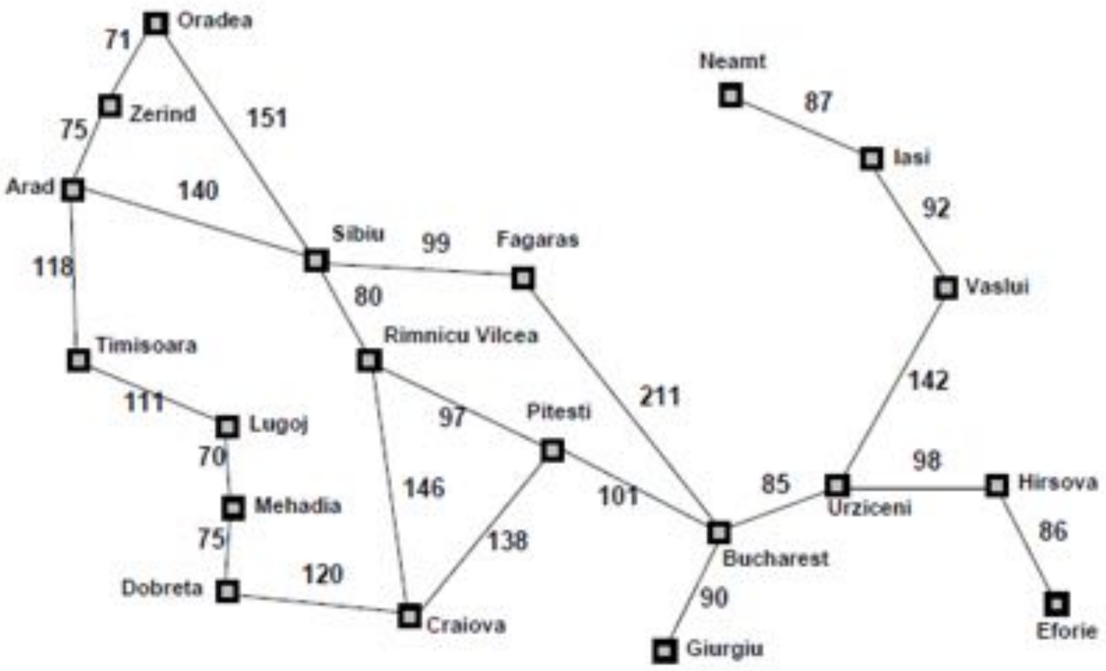
- 其实爬山算法就是应用了贪心的思想
- 每次都找当前节点的领区中最大的节点进行扩展
- 因此具有贪心算法的一切优缺点。

优点：简单，易实现

缺点：往往只能找到一个局部最优解，无法得到全局最优解。

爬山算法求路径

Romania with step costs in km



Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

爬山算法求路径

1、Arad   ->SiBiu
2、Sibiu   -> Fagaras
3、Fagaras ->Bucharest

爬山算法---邻域构造

- 八皇后问题的邻域构造

比如如果初始位置为：[5,3,0,5,7,1,6,0]，那么邻域可以构造为，每次只变化一位的位置，其他的位置数值不变。

比如：

1、[5,3,0,5,7,1,6,0]--->[0,3,0,5,7,1,6,0], [1,3,0,5,7,1,6,0]

[2,3,0,5,7,1,6,0], [3,3,0,5,7,1,6,0]

[4,3,0,5,7,1,6,0], [6,3,0,5,7,1,6,0]

[7,3,0,5,7,1,6,0]

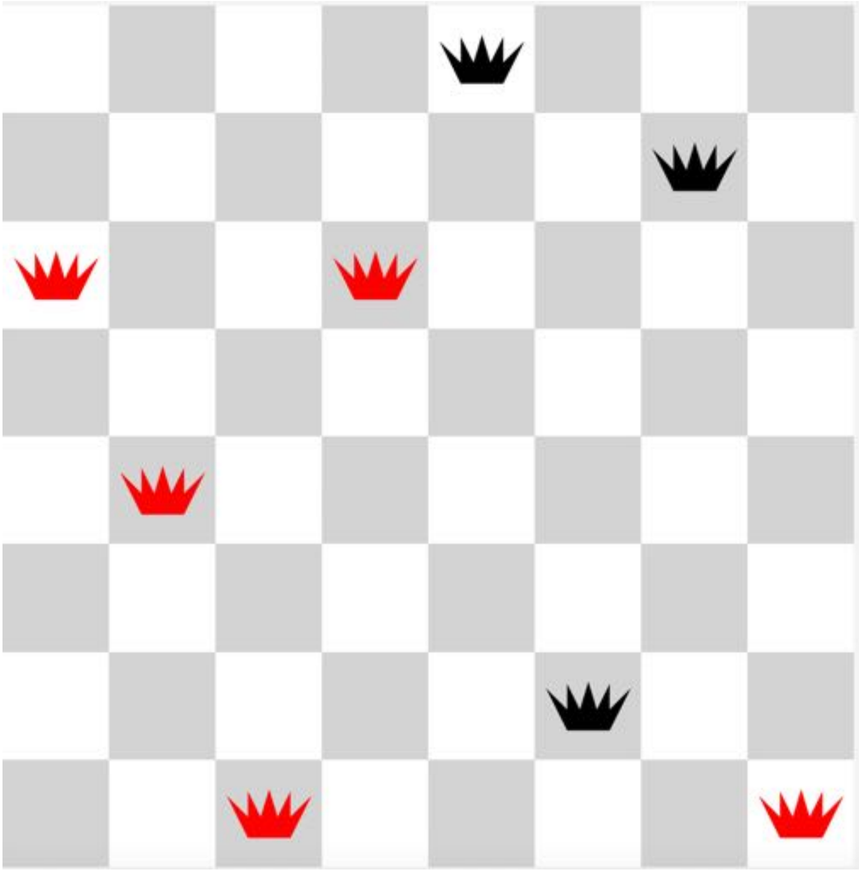
2、[5,3,0,5,7,1,6,0]--->[5,0,0,5,7,1,6,0], [5,1,0,5,7,1,6,0]

[5,2,0,5,7,1,6,0], [5,4,0,5,7,1,6,0]

[5,5,0,5,7,1,6,0], [5,6,0,5,7,1,6,0]

[5,7,0,5,7,1,6,0]

3、……依次类推



模拟退火

```python
def exp_schedule(k=20, lam=0.005, limit=100):
    """One possible schedule function for simulated annealing"""
    return lambda t: (k * math.exp(-lam * t) if t < limit else 0)


def simulated_annealing(problem, schedule=exp_schedule()):
    """[Figure 4.5] CAUTION: This differs from the pseudocode as it
    returns a state instead of a Node."""
    current = Node(problem.initial)
    for t in range(sys.maxsize):
        T = schedule(t)
        if T == 0:
            return current.state
        neighbors = current.expand(problem)
        if not neighbors:
            return current.state
        next = random.choice(neighbors)
        delta_e = problem.value(next.state) - problem.value(current.state)
        if delta_e > 0 or probability(math.exp(delta_e / T)):
            current = next
```

```python
def probability(p):
    """Return true with probability p."""
    return p > random.uniform(0.0, 1.0)
```

$$p=\begin{cases} 1 & if \quad f(x_i)<f(x_i') \\ \exp\left(-\dfrac{f(x_i)-f(x_i')}{T}\right) & if \quad f(x_i)\geq f(x_i') \end{cases}$$

模拟退火

```python
def exp_schedule(k=20, lam=0.005, limit=100):
    """One possible schedule function for simulated annealing"""
    return lambda t: (k * math.exp(-lam * t) if t < limit else 0)


def simulated_annealing(problem, schedule=exp_schedule()):
    """[Figure 4.5] CAUTION: This differs from the pseudocode as it
    returns a state instead of a Node."""
    current = Node(problem.initial)
    for t in range(sys.maxsize):
        T = schedule(t)
        if T == 0:
            return current.state
        neighbors = current.expand(problem)
        if not neighbors:
            return current.state
        next = random.choice(neighbors)
        delta_e = problem.value(next.state) - problem.value(current.state)
        if delta_e > 0 or probability(math.exp(delta_e / T)):
            current = next
```

初温及退温函数

状态产生函数

状态接受函数

$$p = \begin{cases} 1 & if \quad f(x_i) < f(x_i^{'}) \\ \exp\left(-\dfrac{f(x_i) - f(x_i^{'})}{T}\right) & if \quad f(x_i) \geq f(x_i^{'}) \end{cases}$$

```python
def probability(p):
    """Return true with probability p."""
    return p > random.uniform(0.0, 1.0)
```

影响模拟退火的主要因素

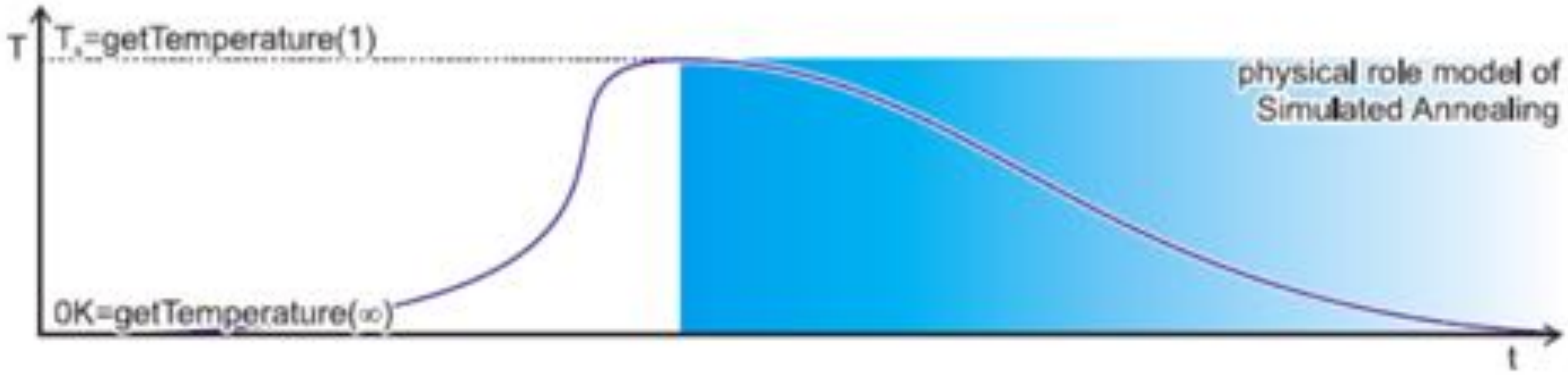
三函数两准则

- 状态产生函数（邻域函数的设计）
- 状态接受函数（尽可能接受最优解，常用Metropolis接受准则）
- 退温函数（降温速度慢一些，解的质量高一些）
- 抽样稳定准则
- 退火结束准则

初温（初温值只要选择充分大，获得高质量解的概率就大）

退温函数

本质上是模拟物理上降温的过程



退温函数-参数对退温过程的控制

```
schedule=exp_schedule(20,0.005,100)
for t in range(5):
    T = schedule(t)
    print(T)
```

```
20.0
19.900249583853647
19.800996674983363
19.70223879206125
19.603973466135105
```

lam值变大

```
schedule=exp_schedule(20,0.1,100)
for t in range(5):
    T = schedule(t)
    print(T)
```

```
20.0
18.09674836071919
16.374615061559638
14.816364413634357
13.406400920712787
```

K值变小

```
schedule=exp_schedule(10,0.1,100)
for t in range(5):
    T = schedule(t)
    print(T)
```

```
10.0
9.048374180359595
8.187307530779819
7.408182206817785
6.703200460356394
```

limit值变小

```
schedule=exp_schedule(10,0.1,5)
for t in range(6):
    T = schedule(t)
    print(T)
```

```
10.0
9.048374180359595
8.187307530779819
7.408182206817785
6.703200460356394
0
```

Metropolis接受准则

- p=exp[-(Ej-Ei)/KT]：

在高温下，可接受与当前状态能量差较大的新状态；在低温下，只接受与当前状态能量差较小的新状态。

模拟退火是对爬山算法的改进

- 算法从某一个初始状态开始后，每一步状态转移均是在当前状态的邻域中随机产生新状态，然后以一定概率进行接受的。
- 接受概率仅依赖于新状态和当前状态，并由温度加以控制。

模拟退火的应用

- 参见另一个PPT《模拟退火》

遗传算法

```python
def genetic_algorithm(population, fitness_fn, gene_pool=[0, 1], f_thres=None, ngen=1000, pmut=0.1):
    """[Figure 4.8]"""
    for i in range(ngen):
        new_population = []
        random_selection = selection_chances(fitness_fn, population)
        for j in range(len(population)):
            x = random_selection()
            y = random_selection()
            child = reproduce(x, y)
            if random.uniform(0, 1) < pmut:
                child = mutate(child, gene_pool)
            new_population.append(child)

        population = new_population

        if f_thres:
            fittest_individual = argmax(population, key=fitness_fn)
            if fitness_fn(fittest_individual) >= f_thres:
                return fittest_individual

    return argmax(population, key=fitness_fn)
```

传入初始化种群

选择

交叉

变异

新种群产生完毕

入参

- population:初始种群
- fitness_fn: 适应度函数
- gene_pool: 个体单条基因的取值范围. 默认为0，1
- f_thres: 适应度门限，如果个体达到了门限，迭代终止，如果这个为空，则需要一直将ngen的迭代次数走完
- ngen: 产生子代的迭代次数
- pmut: 变异的概率

整体流程

开始循环：

1.评估每条染色体所对应个体的适应度（适应度函数： fitness_fn ）。

2.遵照适应度越高，选择概率越大的原则，从种群中选择两个个体作为父方和母方（加权随机算法）。

3.抽取父母双方的染色体，进行交叉，产生子代。

4.对子代的染色体进行变异（不是每次循环都执行，遵循变异概率： pmut ；变异时需要知道基因的取值范围： gene_pool ）。

5.重复2，3，4步骤，直到新种群的产生。

如果找到满意的解（适应度门限： f_thres）或者达到迭代次数上限（ ngen ），结束，否则回到1。

遗传算法的组成

- 产生初始种群
- 适应度函数
- 产生子代
  - 选择
  - 交叉
  - 变异

产生初始种群

```python
def init_population(pop_number, gene_pool, state_length):
    """Initializes population for genetic algorithm
    pop_number  :  Number of individuals in population
    gene_pool   :  List of possible values for individuals
    state_length:  The length of each individual"""
    g = len(gene_pool)
    population = []
    for i in range(pop_number):
        new_individual = [gene_pool[random.randrange(0, g)] for j in range(state_length)]
        population.append(new_individual)

    return population
```

比如八皇后，初始种群可以为：population = init_population(5, range(8), 8)

适应度函数

- 需要根据具体问题自定义
- 比如八皇后，适应度函数可以定义如下：

```python
def fitness(q):
    non_attacking = 0
    for row1 in range(len(q)):
        for row2 in range(row1+1, len(q)):
            col1 = int(q[row1])
            col2 = int(q[row2])
            row_diff = row1 - row2
            col_diff = col1 - col2

            if col1 != col2 and row_diff != col_diff and row_diff != -col_diff:
                non_attacking += 1

    return non_attacking
```

最优解，就是八个皇后互不攻击的情况为28.（每个都是往后比，即7+6+5+......+1 = 28）

八皇后问题的初始值及适应度运行示例

[3, 6, 0, 3, 7, 3, 7, 7] ------------------ 19

[0, 2, 6, 0, 7, 3, 5, 2] ------------------ 23

[4, 3, 3, 5, 0, 6, 4, 5] ------------------ 19

[7, 7, 0, 7, 3, 7, 0, 2] ------------------ 20

[7, 0, 2, 2, 7, 7, 6, 5] ------------------ 19

选择

```python
def selection_chances(fitness_fn, population):
    fitnesses = map(fitness_fn, population)
    return weighted_sampler(population, fitnesses)
```

```python
def weighted_sampler(seq, weights):
    """Return a random-sample function that picks from seq weighted by weights."""
    totals = []
    for w in weights:
        totals.append(w + totals[-1] if totals else w)

    return lambda: seq[bisect.bisect(totals, random.uniform(0, totals[-1]))]
```

有趣的选择：加权随机算法，适应度越高的个体越有可能被选中

八皇后的一个选择示例，接前面的例子

[3, 6, 0, 3, 7, 3, 7, 7] ------------------ 19 -------→ 19%

[0, 2, 6, 0, 7, 3, 5, 2] ------------------ 23 -------→ 23%

[4, 3, 3, 5, 0, 6, 4, 5] ------------------ 19 -------→ 19%

[7, 7, 0, 7, 3, 7, 0, 2] ------------------ 20 -------→ 20%

[7, 0, 2, 2, 7, 7, 6, 5] ------------------ 19 -------→ 19%

sum = 100

假设种群数目,某个个体其适应度为f_i,则其被选中的概率为P_i:

$$P_i = \frac{f_i}{\sum_{i=1}^{n} f_i}$$

交叉

```python
def reproduce(x, y):
    n = len(x)
    c = random.randrange(1, n)
    return x[:c] + y[c:]
```

变异

```python
def mutate(x, gene_pool):
    n = len(x)
    g = len(gene_pool)
    c = random.randrange(0, n)
    r = random.randrange(0, g)

    new_gene = gene_pool[r]
    return x[:c] + [new_gene] + x[c+1:]
```

注意：pmut参数，只有一定的概率可能变异

```python
if random.uniform(0, 1) < pmut:
    child = mutate(child, gene_pool)
```

八皇后的交叉，变异示例，接前面的例子

[3, 6, 0, 3, 7, 3, 7, 7] ------------------ 19 -------→ 19%

[0, 2, 6, 0, 7, 3, 5, 2] ------------------ 23 -------→ 23%

[4, 3, 3, 5, 0, 6, 4, 5] ------------------ 19 -------→ 19%

[7, 7, 0, 7, 3, 7, 0, 2] ------------------ 20 -------→ 20%

[7, 0, 2, 2, 7, 7, 6, 5] ------------------ 19 -------→ 19%

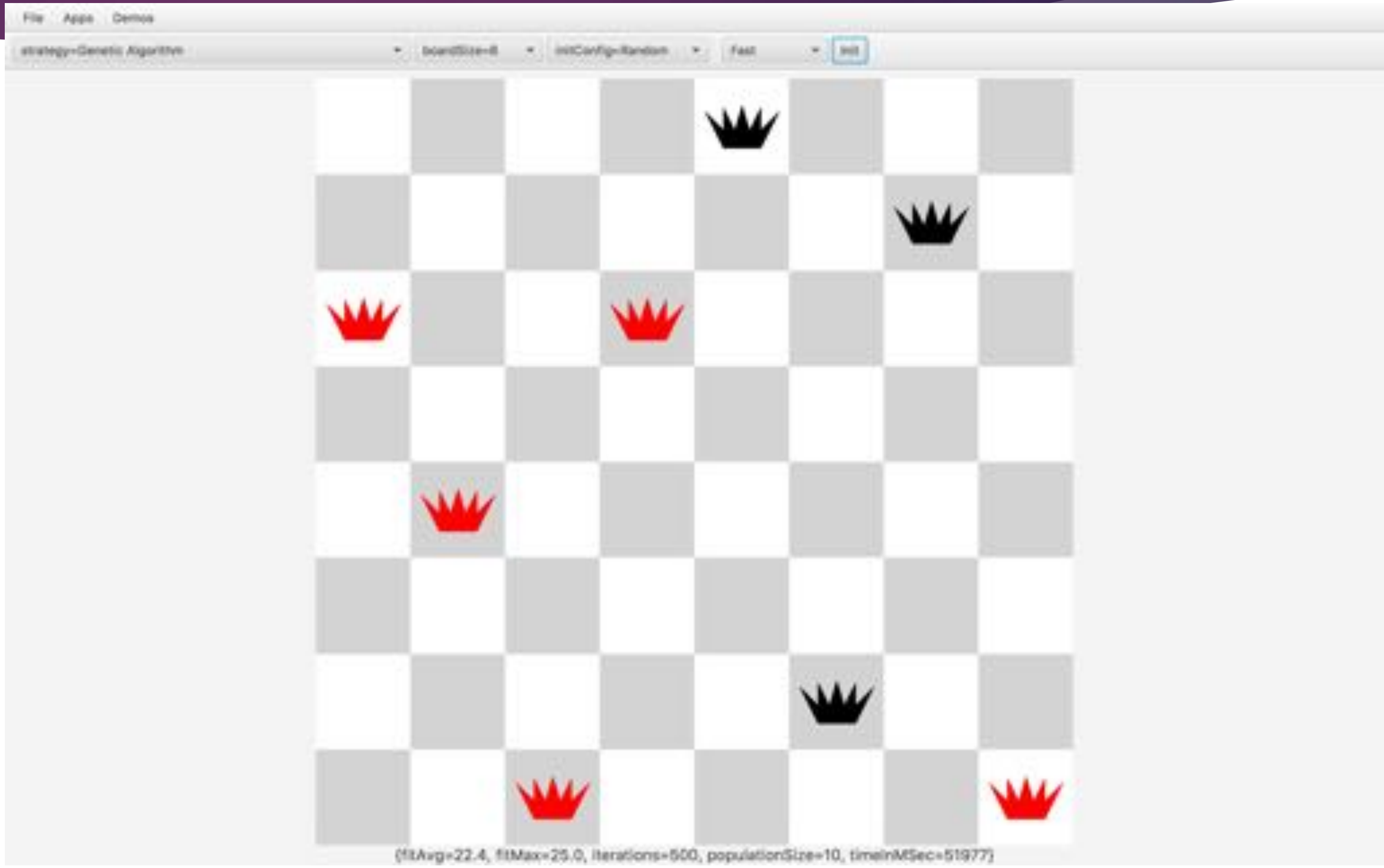
假设交叉产生的随机数c = 5，则新个体：
[0,2,6,0,7,7,6,5]

假设变异发生的位置c = 3， 变异后的值为7，新个体进一步变成:

[0,2,6,7,7,7,6,5]

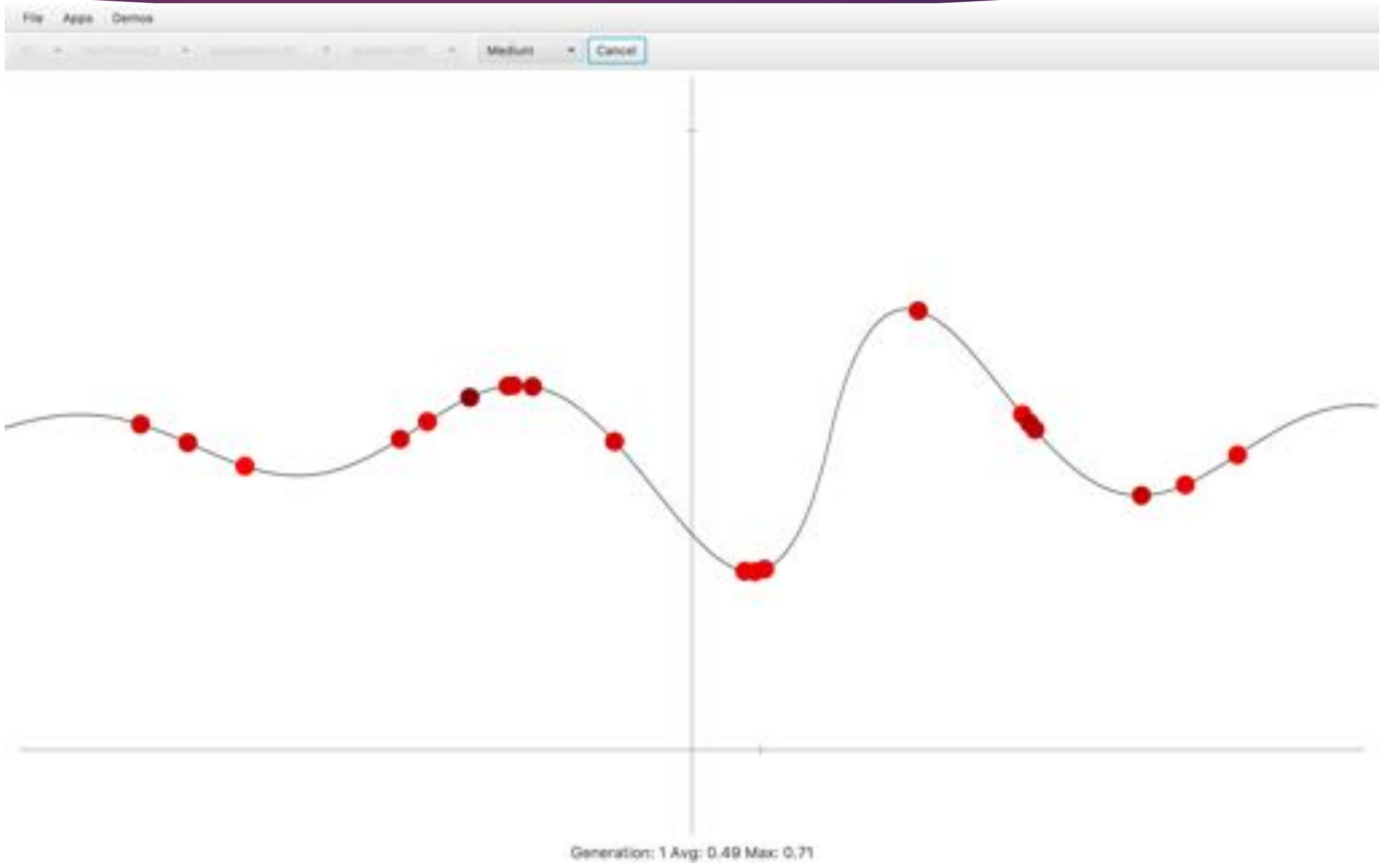
简单遗传算法不是很适合解决八皇后的问题

- 几乎每次运行都得不到最优解
- 为什么……
- 改进

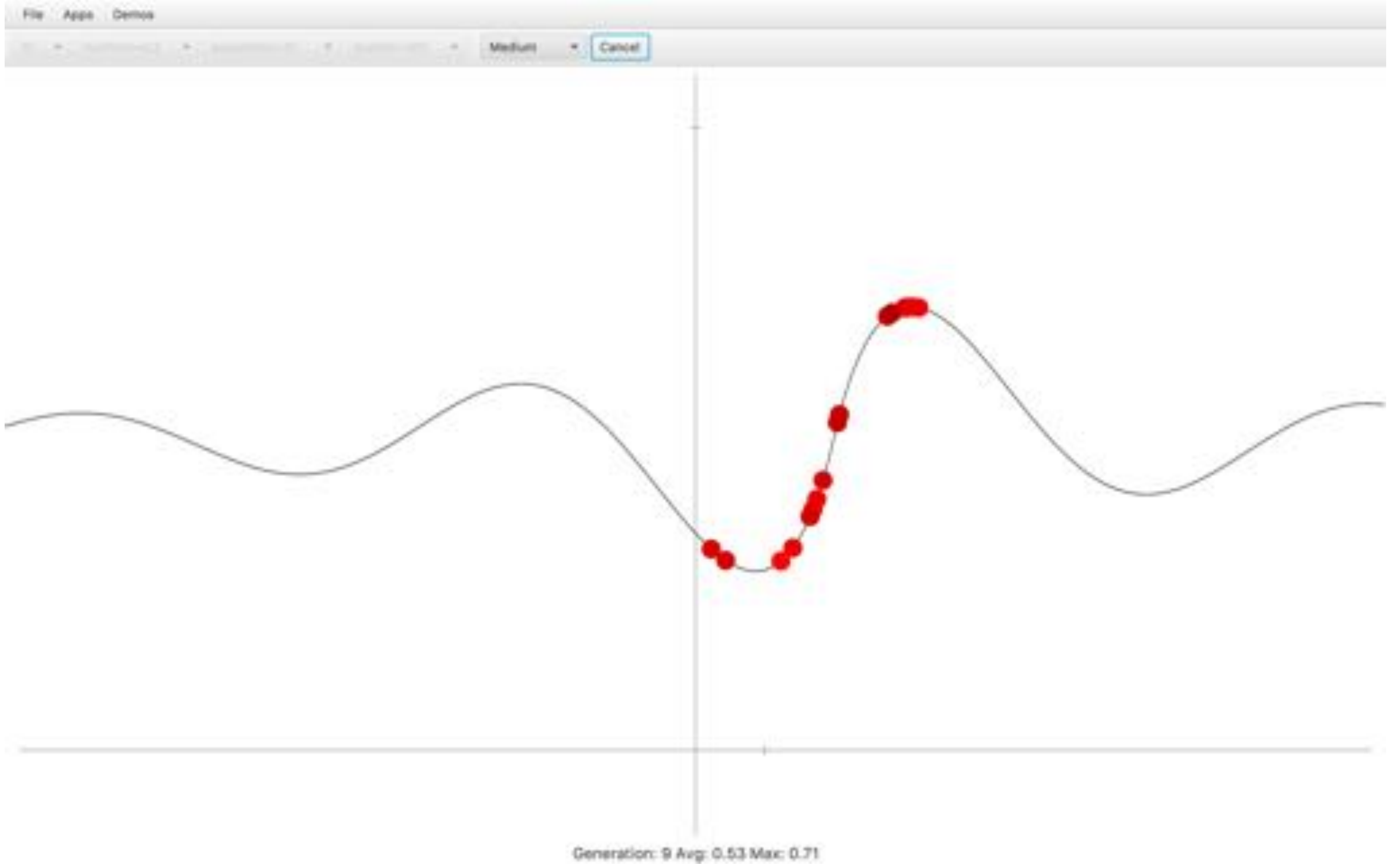


遗传算法在函数求最大值中的应用

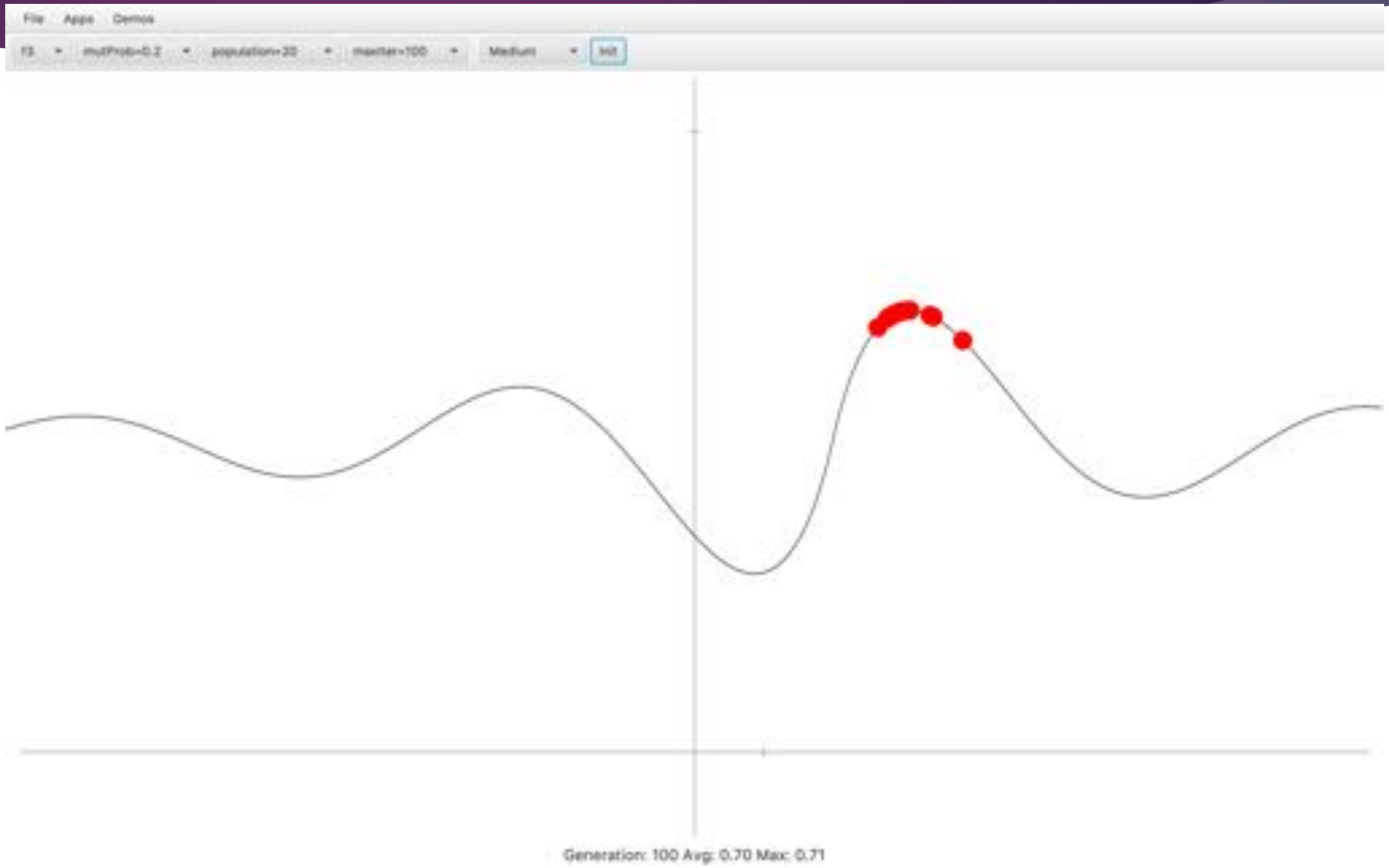
- 初期



- 中期

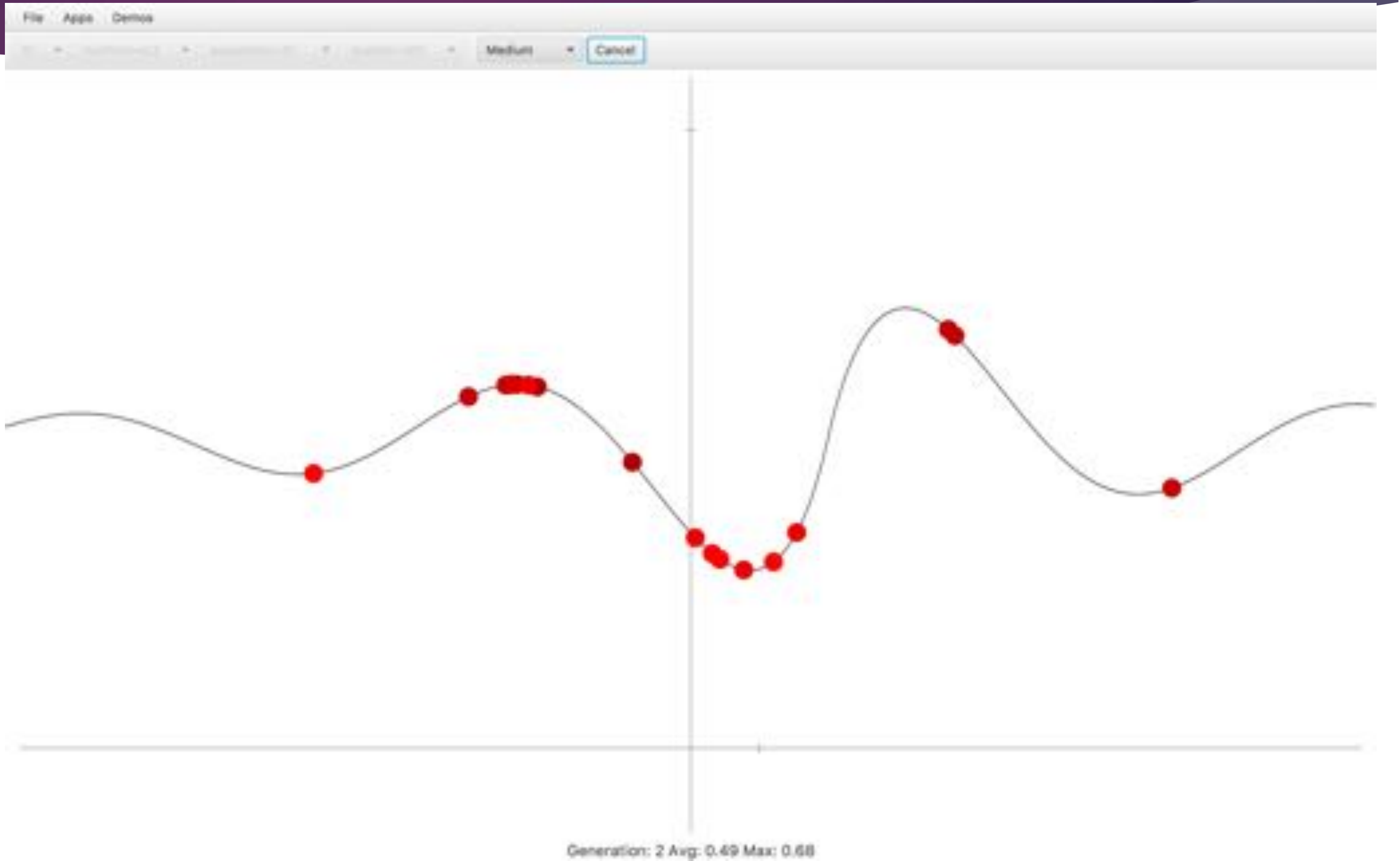


- 最终

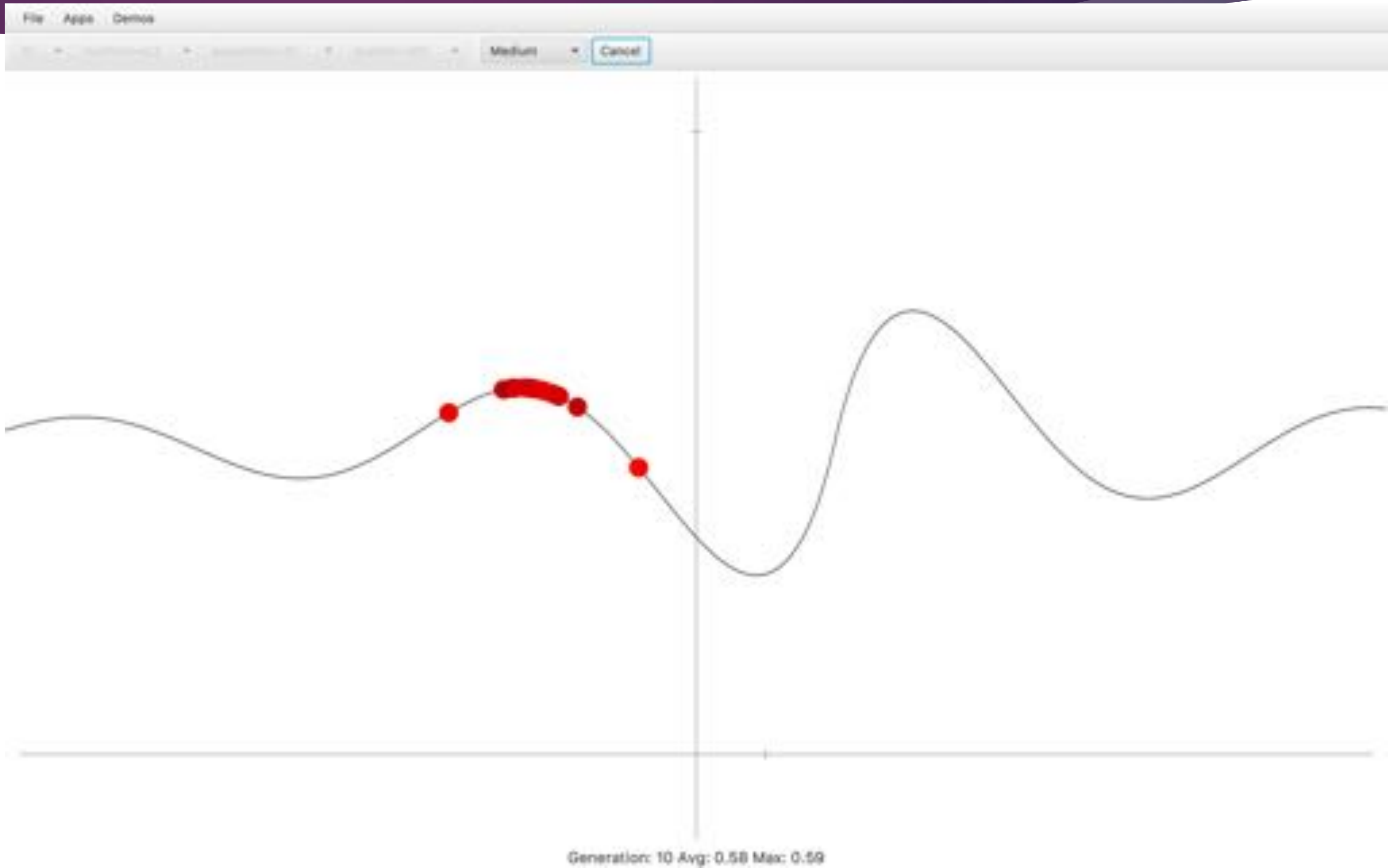


然而，并不是每次都可以得到最优解

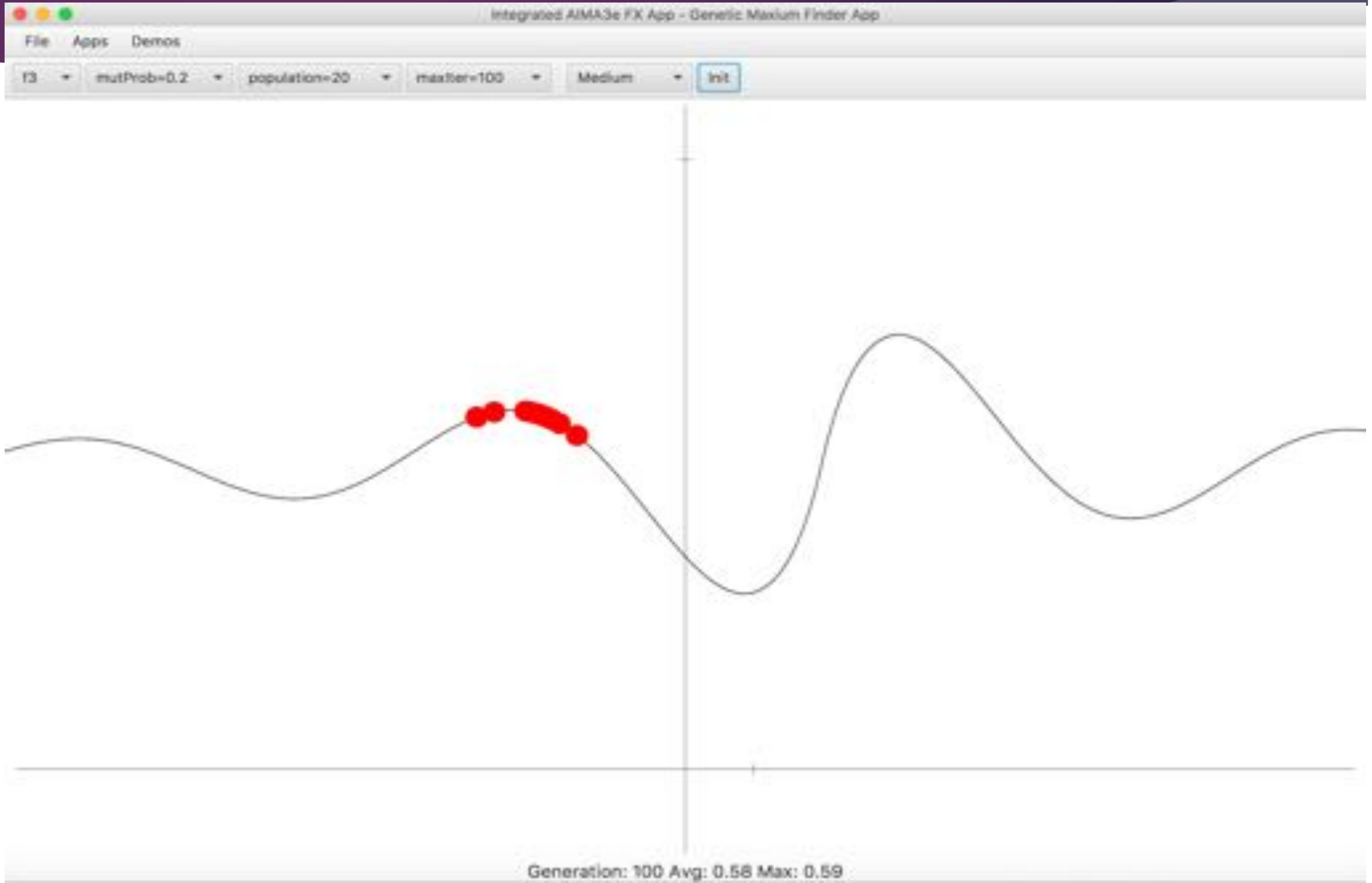
- 另一次运行，早期



- 另一次运行，中期



- 另一次运行，最终
- 对于这次的运行过程，称之为早熟收敛：当通过染色体的交叉和变异，种群已经很难产生优于亲本的子代个体时，就会发生早熟收敛。所有标准形式的交叉简单地重复产生当前亲本，进一步优化将完全依赖于变异，而且可能非常缓慢。
- 同时也很好的说明了遗传算法具有随机性



遗传算法更多

- 参考《简单遗传算法》

CARP问题

- urban waste collection
- post delivery
- sanding or salting the streets