Local Search

YAO ZHAO

Hill Climbing

```
hillClimbing(startNode){
    x ← startNode
    while (halt condition is not satisfied ) {
        Generate solution x' based on x
        Evaluate the f (x')
        If f(x')>f(x) {
            x = x'
        }
    }
}
```

Hill-climbing algorithm

- Hill-climbing algorithm use the idea of greedy algorithm:
  - Each time it selects the best node in the neighborhood of current node to compare
  - If the best neighbor is better than the current point, it will move to the neighbor. Otherwise, the process will be ended, and it returns the current point.
- So, it has all the advantages and disadvantages of greedy algorithms:
  - Advantages: Simple and easy to implement
  - Disadvantages: depends on the initial point, in most cases, can only find **a local optimal solution**, can not get the global optimal solution.

Hill-climbing for route search

Start: Arad
Target: Bucharest

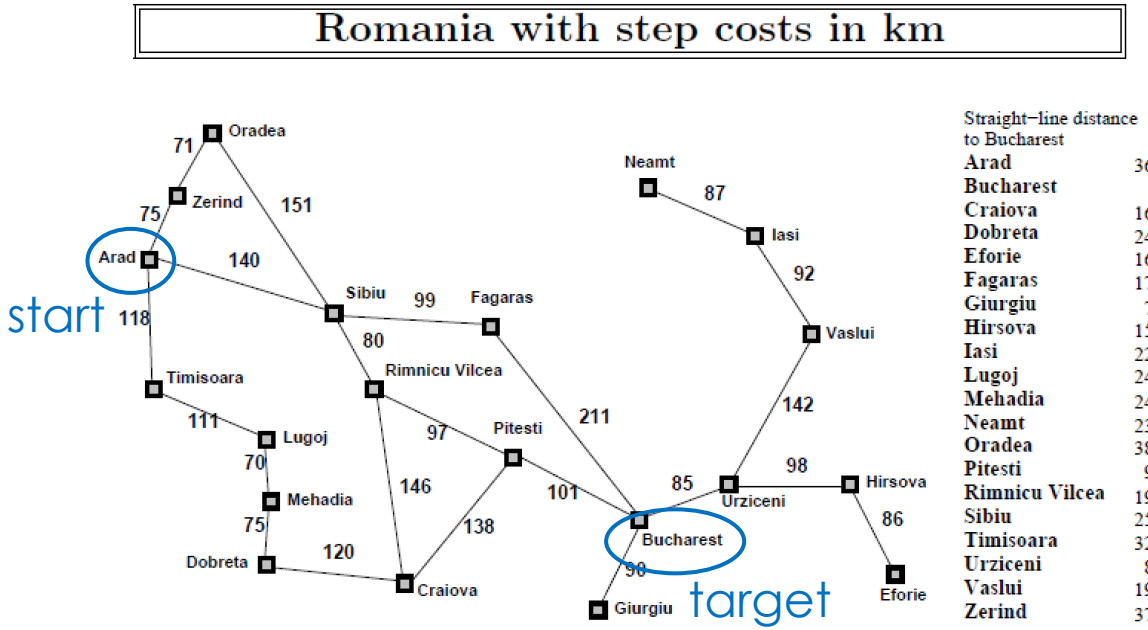
How to evaluating which neighborhood is best?

For simplicity, **the straight-line distance to the target city** is used to evaluate neighbor nodes

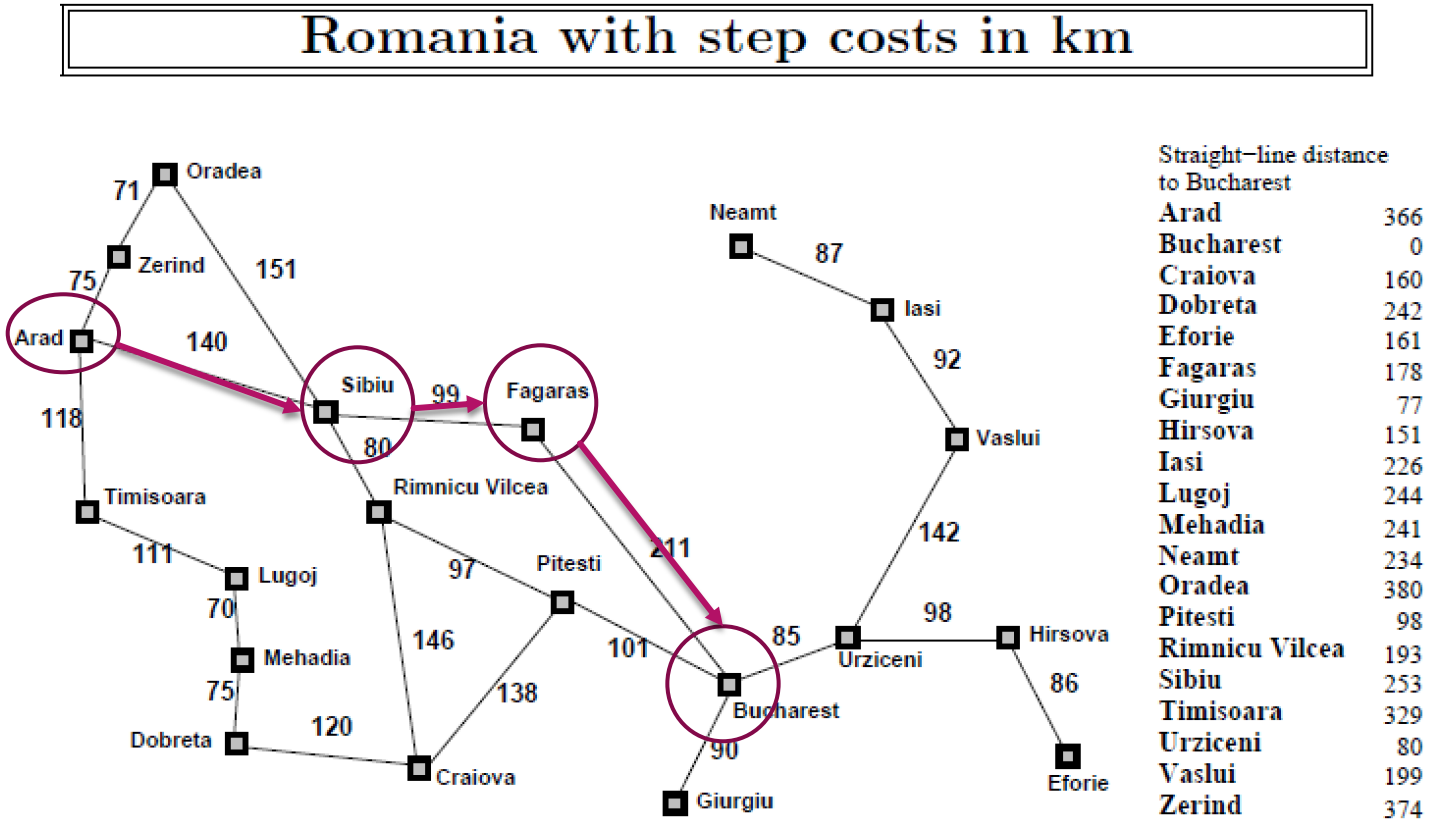
$f(x)$ = straight-line distance to Bucharest

How to generate x' based on x?

Randomly select the adjacency node of x



One possible execution process of a hill climbing algorithm for route search



Hill-climbing algorithm: construction of neighbors

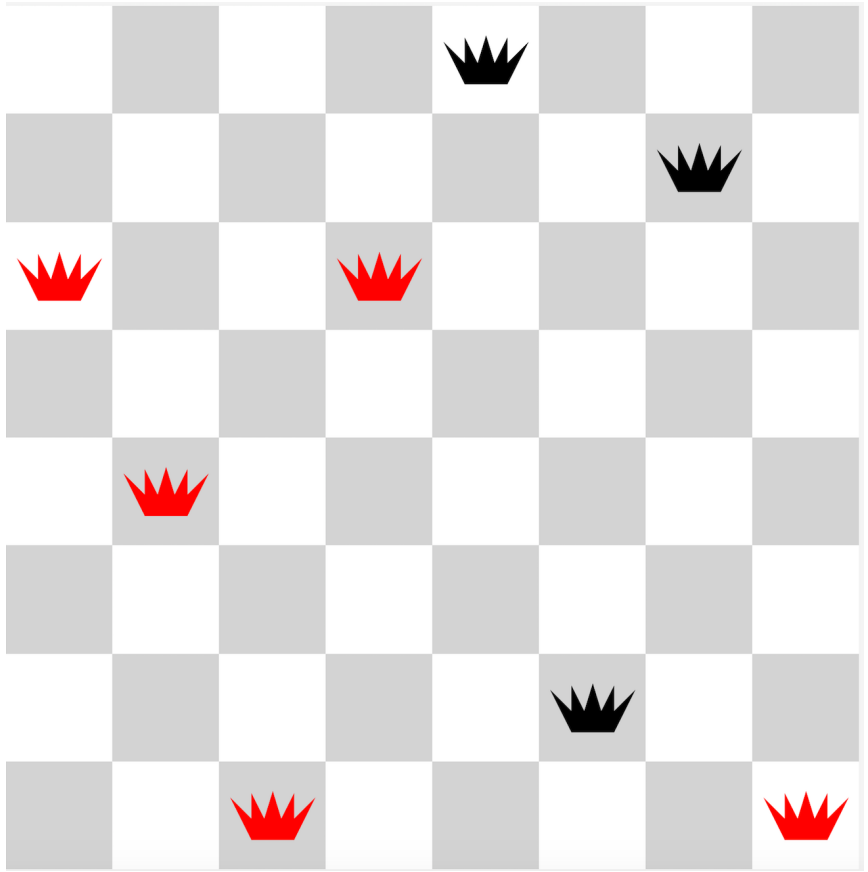
- Neighbor construction for eight-queen problem:

If the initial position is [5, 3, 0, 5, 7, 1, 6, 0], the neighbor can be constructed as the positions differing from the current position by a value on a single bit

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]

And so on.



Hill-climbing algorithm: continuous optimization problem

- Neighbor construction for continuous optimization problem:

$$\max\ f(x_1, x_2, x_3, x_4, x_5) = \sum_{i=1}^{5} x_i^2, \qquad x_1 \in [0, 10]$$

- If the initial position is [0, 0, 0, 0, 0], the neighbor can be constructed by adding a gaussian noise to each variable.

$[0, 0, 0, 0, 0]$

$\rightarrow\ [0+\delta, 0, 0, 0, 0], [0, 0+\delta, 0, 0, 0], [0, 0, 0+\delta, 0, 0], [0, 0, 0, 0+\delta, 0], [0, 0, 0, 0, 0+\delta]$

Simulated annealing

Simulated annealing is a probabilistic method proposed in Kirkpatrick, Gelett and Vecchi (1983) and Cerny (1985) for finding the global minimum of a cost function that may possess several local minima. It works by emulating the physical process whereby a solid is slowly cooled so that when eventually its structure is "frozen," this happens at a minimum energy configuration.

Simulated Annealing

- It is an **improvement** over Hill-climbing
  - From the initial state, every state is generated randomly from the neighborhood of the current state, and is **accepted with some probability**
  - The **acceptance probability** $\boldsymbol{P}$ only depends on the current state (x_i) and new state (x_i'), and is controlled by **temperature** $\boldsymbol{T}$:

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

- Simulated annealing combines random search and greedy search

Simulated annealing VS Hill Climbing

```
hillClimbing(startNode){
    x ← startNode
    while (halt condition is not satisfied ) {
        Generate solution x' based on x
        Evaluate the f (x')
        If f(x')>f(x) {
            x = x'
        }
    }
}
```

Replace this line with a probability p

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

```
//one possible cooling schedule function for simulated   annealing
expSchedule(k, lam, limit, t){
        if (t < limit){
              return k*Math.exp(-lam*t)
        }
         return   0
}
```

Each reduction rule reduces the temperature at a different rate and each method is better at optimizing a different type of model. For the 3rd rule, *beta* is an arbitrary constant.

1. **Linear Reduction Rule:** $t = t - \alpha$
2. **Geometric Reduction Rule:** $t = t * \alpha$
3. **Slow-Decrease Rule:** $t = \frac{t}{1+\beta t}$

```
simulatedAnnealing(startNode, t0, schedule=expSchedule()){
     x ← startNode
     initial k, lam, limit
     t ← t0
     while (halt condition is not satisfied ) {
            T = schedule(k, lam, limit, t)
            Generate solution x' based on x
             Evaluate the f (x')
             If (f(x')>f(x)) or (Math.exp((f(x')-f(x))/T)>random(0.0, 1.0)) {
                        x = x'
             }//end if
             update(t)
        }//end whhile
}//end simulatedAnnealing
```

Initial temperature

Cooling schedule function

Metropolis:

At high temperatures, a new state that is much worse than the current state can be accepted; at low temperatures, only a new state that is little bit worse than the current state can be accepted.

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

Cooling schedule function: parameter controlling annealing process

```
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
```

Increase lam value

```
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
```

Decrease k value

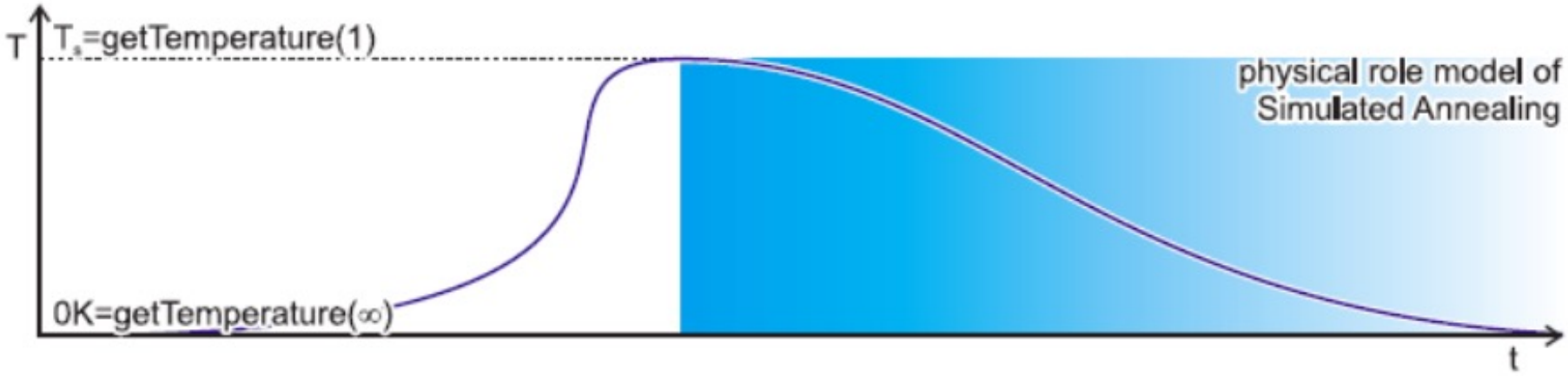
```
schedule=exp_schedule(10,0.1,100)
for t in range(5):
    T = schedule(t)
    print(T)
```

```
10.0
9.048374180359595
8.187307530779819
7.4081822068171785
6.703200460356394
```

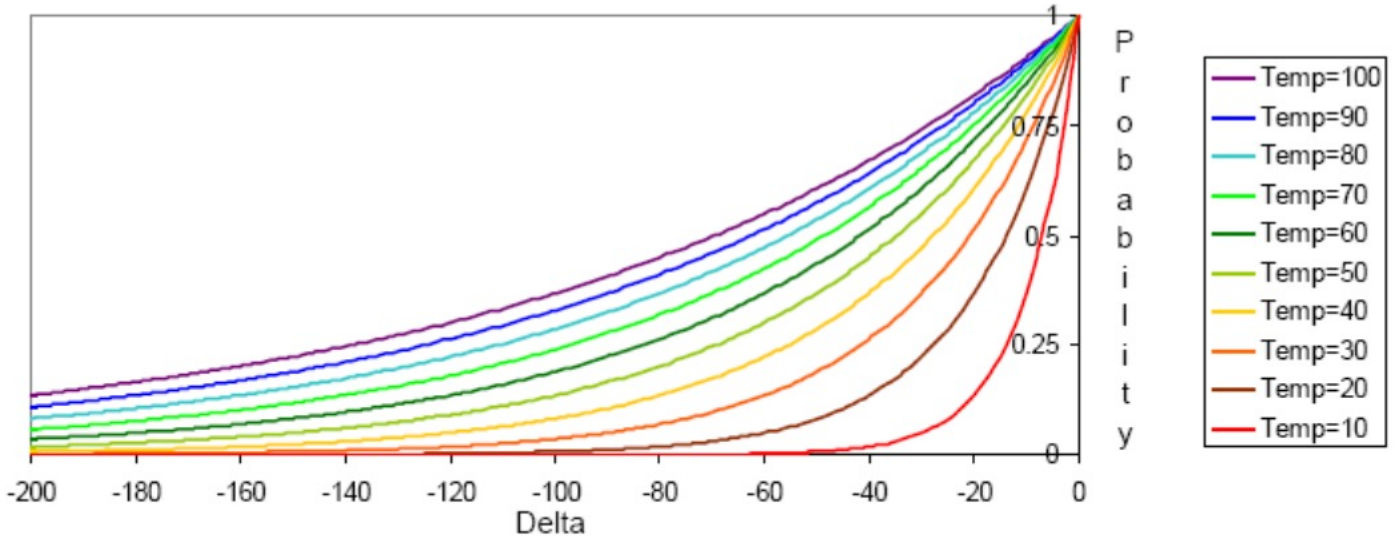
Decrease limit value

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

```
10.0
9.048374180359595
8.187307530779819
7.4081822068171785
6.703200460356394
0
```



Acceptance criterion and cooling schedule



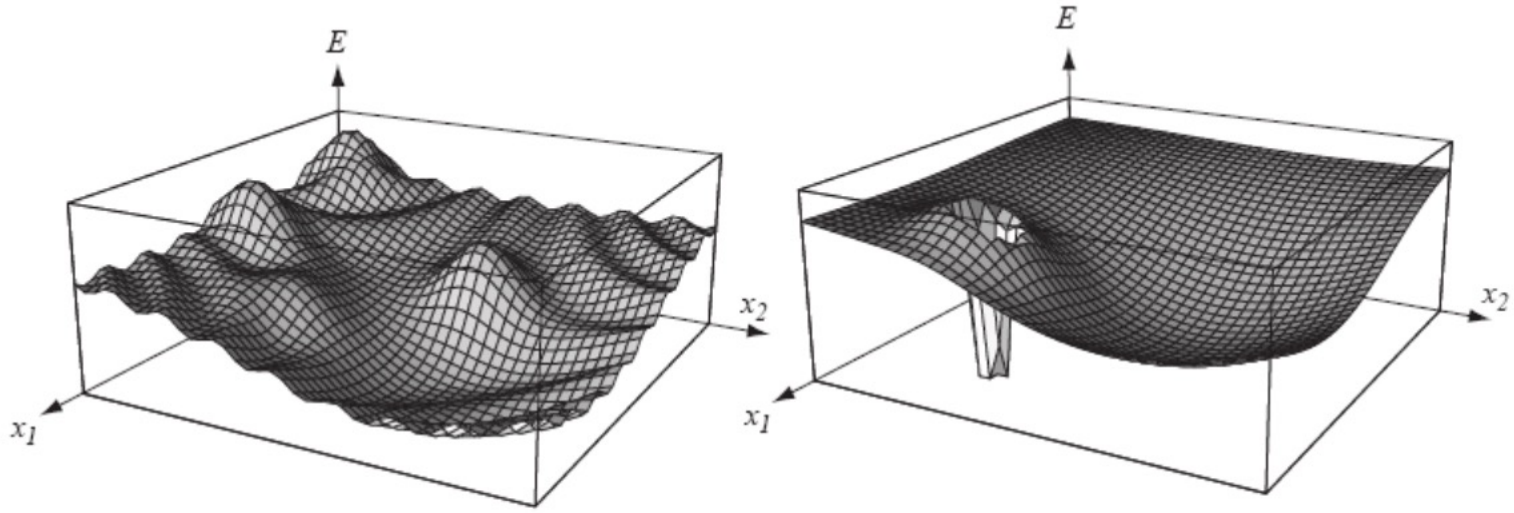
Initially temperature is very high (most bad moves accepted)

Temp slowly goes to 0, with multiple moves attempted at each temperature

Final runs with temp=0 (always reject bad moves) greedily “quench” the system

Practical Issues with simulated annealing

Cost function must be carefully developed, it must be "fractal and smooth". The energy function of the left would work with SA while the one of the right would fail.



Practical Issues with simulated annealing

- The cost function should be fast. It is going to be called “millions” of times.

- The best is if we just have to calculate the deltas produced by the modification instead of traversing through all the state.

- This is dependent on the application.

Practical Issues with simulated annealing

- In asymptotic convergence simulated annealing converges to globally optimal solutions.
- In practice, the convergence of the algorithm depends on the cooling schedule.
- There are some suggestion about the cooling schedule, but it stills requires a lot of testing and it usually depends on the application.
  - Start at a temperature where 50%(or higher) of bad moves are accepted.
  - Each cooling step reduces the temperature by 10%(or smaller).
  - The number of iterations at each temperature should attempt to move between 1-10 times each "element" of the state.
  - The final temperature should not accept bad moves; this step is known as the quenching step.

Simulated annealing: Steps for solving Sudoku

1. Generate initial state

	2	4			7			
6								
		3	6	8		4	1	5
4	3	1			5			
5							3	2
7	9						6	
2		9	7	1		8		
	4			9	3			
3	1				4	7	5	

1	**2**	**4**	*1*	*5*	**7**	*6*	*9*	*2*
6	*5*	*9*	*3*	*4*	*2*	*3*	*8*	*7*
8	*7*	**3**	**6**	**8**	*9*	**4**	**1**	**5**
4	**3**	**1**	*6*	*8*	**5**	*4*	*7*	*9*
5	*2*	*6*	*7*	*1*	*9*	*1*	**3**	**2**
7	**9**	*8*	*4*	*3*	*2*	*8*	**6**	*5*
2	*7*	**9**	**7**	**1**	*8*	**8**	*2*	*1*
8	**4**	*6*	*2*	**9**	**3**	*9*	*3*	*4*
3	**1**	*5*	*5*	*6*	**4**	**7**	**5**	*6*

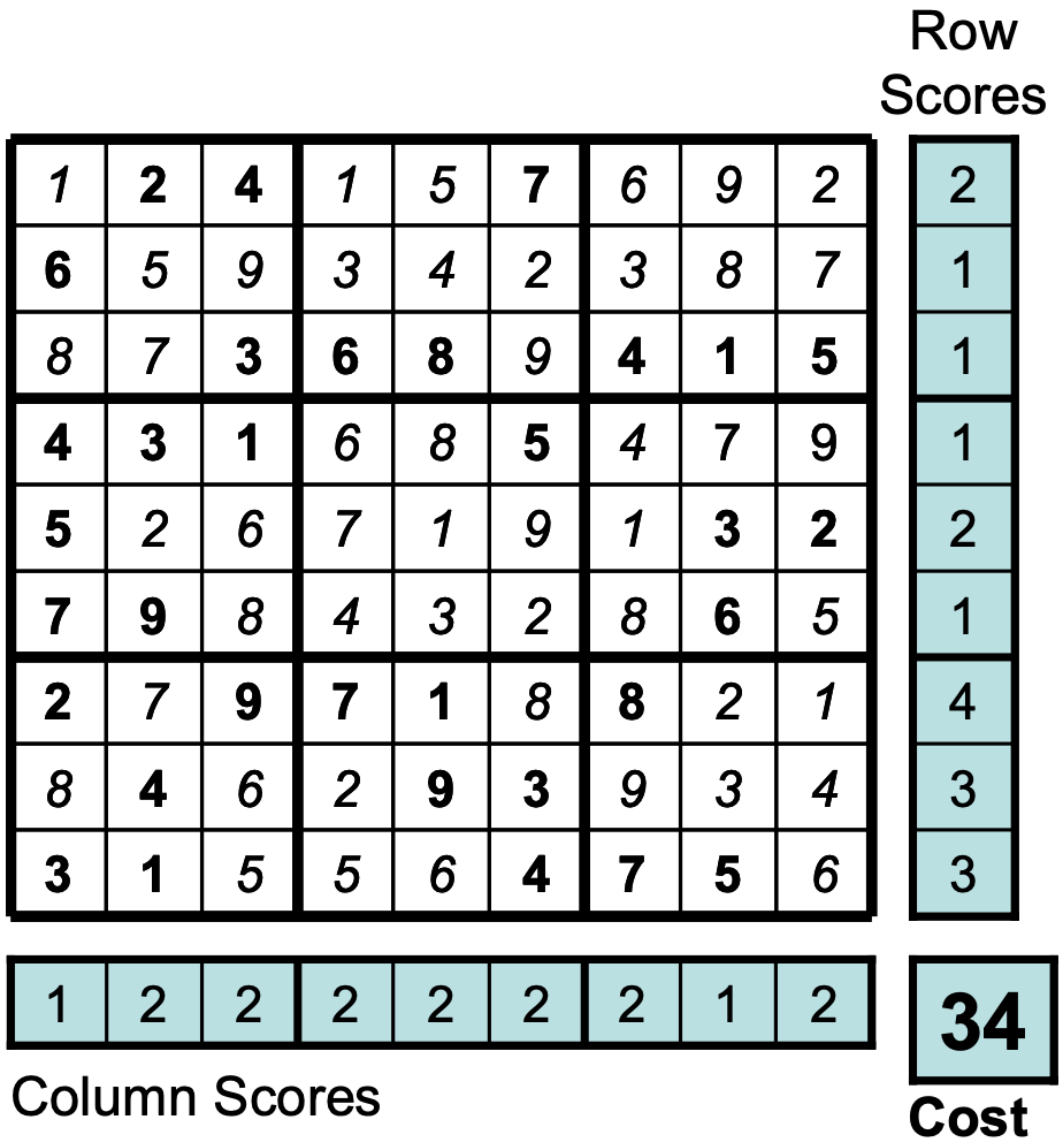
Fill the grid by assigning each non-fixed cell in the grid a value. This is done randomly, but make sure every square contains the values 1 to n^2 **exactly once** when the grid is full.

Simulated annealing: Steps for solving Sudoku

1. Generate random states
2. Design a cost function:

Scan each row individually and calculates the number of values, 1 through to n^2 that are not present.

The same is then done for each column, and the cost is simply the total of these values.

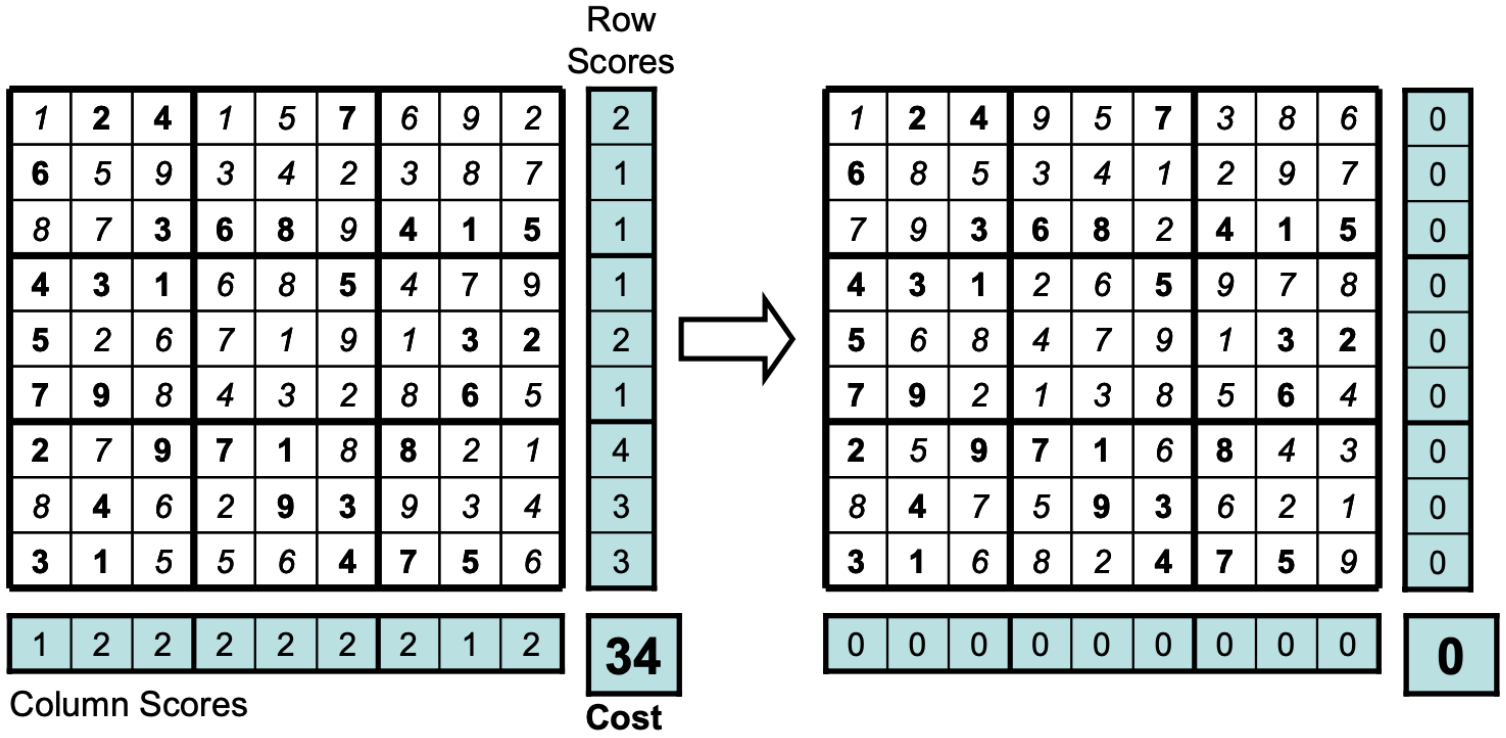


Simulated annealing: Steps for solving Sudoku

1. Generate random states
2. Design a cost function
3. How to generate next states
4. Determine the initial temperature
5. Determine the cooling schedule function

You can read the paper below for more detailed information:

https://www.researchgate.net/publication/220403361_Metaheuristics_can_solve_Sudoku_puzzles



Genetic Algorithm

```
664 def genetic_algorithm(population, fitness_fn, gene_pool=[0, 1], f_thres=None, ngen=1000, pmut=0.1):
665     """[Figure 4.8]"""
666     for i in range(ngen):
667         new_population = []
668         random_selection = selection_chances(fitness_fn, population)
669         for j in range(len(population)):
670             x = random_selection()
671             y = random_selection()
672             child = reproduce(x, y)
673             if random.uniform(0, 1) < pmut:
674                 child = mutate(child, gene_pool)
675             new_population.append(child)
676
677         population = new_population
678
679         if f_thres:
680             fittest_individual = argmax(population, key=fitness_fn)
681             if fitness_fn(fittest_individual) >= f_thres:
682                 return fittest_individual
683
684     return argmax(population, key=fitness_fn)
```

传入初始化种群
Initial population

选择
Selection process

交叉 crossover

变异mutation

新种群产生完毕 Already generate a new population

Parameters of Genetic Algorithm

- **population**: Initial population
- **fitness_fn**: fitness evaluation function
- **gene_pool**: the value range of single gene, default to be {0, 1}
- **f_thres**: threshold value for fitness value. Algorithm will terminate if we find a solution with fitness value larger than f_thres. If set to 0, then algorithm will terminate after ngen iterations
- **ngen**: number of generations
- **pmut**: probability of permutation

Overall Flowchart of Genetic Algorithm

Flowchart:

1. Evaluate the fitness of individuals corresponding to each chromosome (fitness function: fitness_fn)

2. According to the principle that individuals with higher fitness gain greater probability of selection, two individuals are selected from the population as the father and mother (**weighted random algorithm**)

3. Extract chromosomes of both parents and **crossover** them to **produce offspring**

4. Perform **mutation** to the chromosomes of offsprings (not every cycle, follow **the mutation probability: pmut**; the mutation needs to know **the range of genes: gene pool**)

5. **Repeat steps 2, 3, and 4 until a new population is produced**

If a satisfactory solution is found (**fitness threshold: f_thres**) or the upper limit of the number of iterations (**ngen**) is reached, then terminates the algorithm, otherwise returns to 1.

Key Components of Genetic Algorithm

- Initialize population
- Fitness function
- Generating offspring
  - selection
  - crossover
  - mutation

Initialize Population

```
687 def init_population(pop_number, gene_pool, state_length):
688     """Initializes population for genetic algorithm
689     pop_number  :  Number of individuals in population
690     gene_pool   :  List of possible values for individuals
691     state_length:  The length of each individual"""
692     g = len(gene_pool)
693     population = []
694     for i in range(pop_number):
695         new_individual = [gene_pool[random.randrange(0, g)] for j in range(state_length)]
696         population.append(new_individual)
697 
698     return population
```

Consider Eight Queen for example, we can initialize population by:
population = init_population(5, range(8), 8)

Fitness Function

- This should be designed case by case
- If we consider Eight Queens, the fitness function can be defined as follows:

```
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
```

Optimal solution: the situation in which the eight queens do not attack each other gets the fitness value 28 (Every time we consider comparison with later rows, i.e., 7 + 6 + 5 + . . . + 1 = 28)

Eight Queen: Example of Initialize Population and Fitness

Population	Fitness
[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

Selection

```
701 def selection_chances(fitness_fn, population):
702     fitnesses = map(fitness_fn, population)
703     return weighted_sampler(population, fitnesses)
```

```
207 def weighted_sampler(seq, weights):
208     """Return a random-sample function that picks from seq weighted by weights."""
209     totals = []
210     for w in weights:
211         totals.append(w + totals[-1] if totals else w)
212 
213     return lambda: seq[bisect.bisect(totals, random.uniform(0, totals[-1]))]
```

Interesting choice: weighted random algorithm, individuals with high adaptivity are more likely to be selected

Eight Queen Example: Selection

- Given a population, if an individual has a fitness f_i , then the probability of being selected is: p_i

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

[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

Crossover

```
706 ▼ def reproduce(x, y):
707       n = len(x)
708       c = random.randrange(1, n)
709       return x[:c] + y[c:]
```

Mutation

```
712 def mutate(x, gene_pool):
713     n = len(x)
714     g = len(gene_pool)
715     c = random.randrange(0, n)
716     r = random.randrange(0, g)
717
718     new_gene = gene_pool[r]
719     return x[:c] + [new_gene] + x[c+1:]
```

Note: pmut is a parameter indicating only a likelihood for mutation

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

Eight Queen Example: Crossover and Mutation

[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%

If the random variable indicating crossover is c = 5, then the new individual is ?

If the random variable indicating mutation is c = 3 and the value after mutation is 7, then the new individual is ?

Eight Queen Example: Crossover and Mutation

[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%

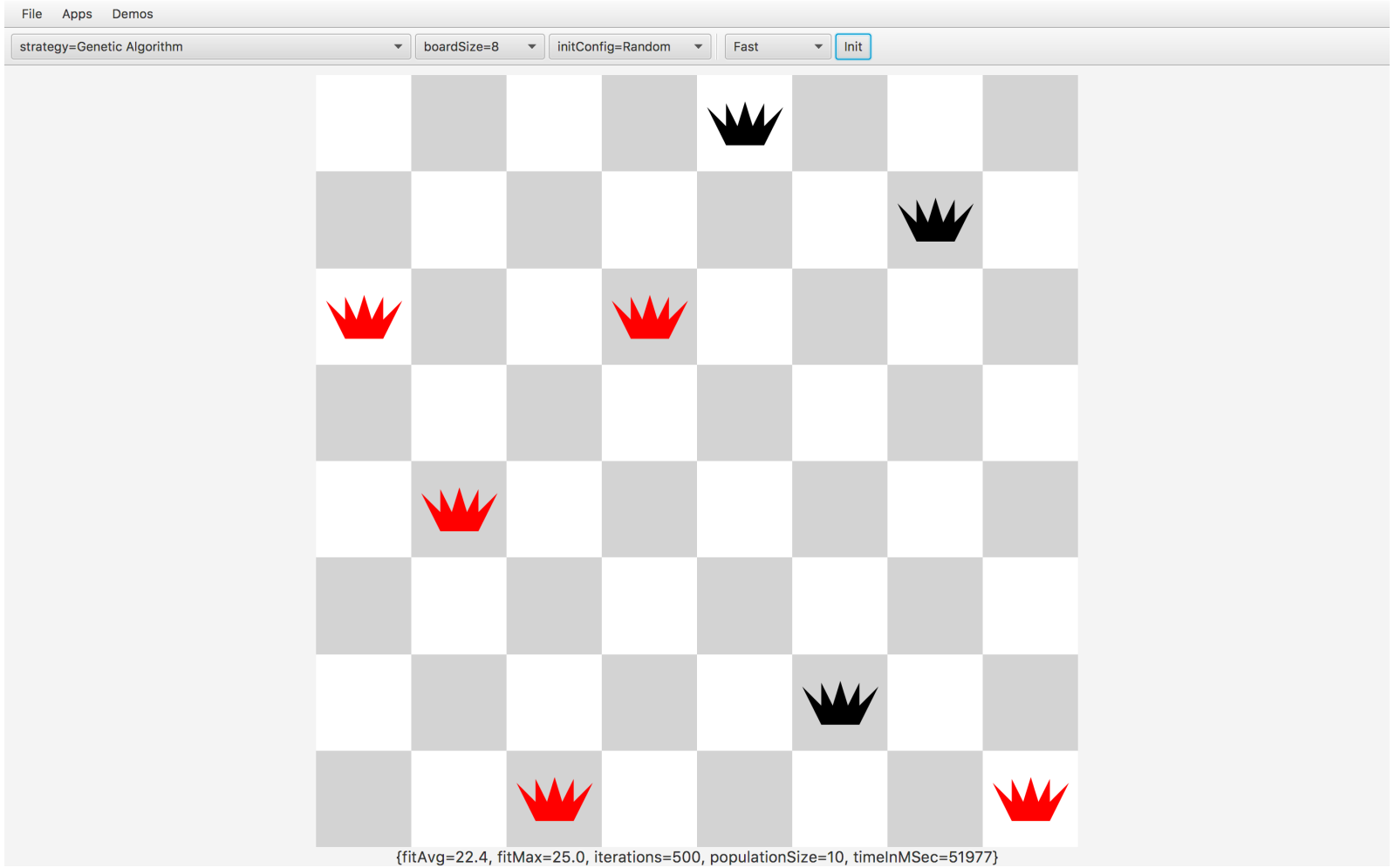
[7, 0, 2, 2, 7, 7, 6, 5] ---------- 19 -------→ 19%

If the random variable indicating crossover is c = 5, then the new individual is :
[0,2,6,0,7,7,6,5]

If the random variable indicating mutation is c = 3 and the value after mutation is 7, then the new individual is :
[0,2,6,7,7,7,6,5]

Simple GA Not Suitable for Eight Queen

- Almost every execution can not get optimal solution
- Why?
- Improvement

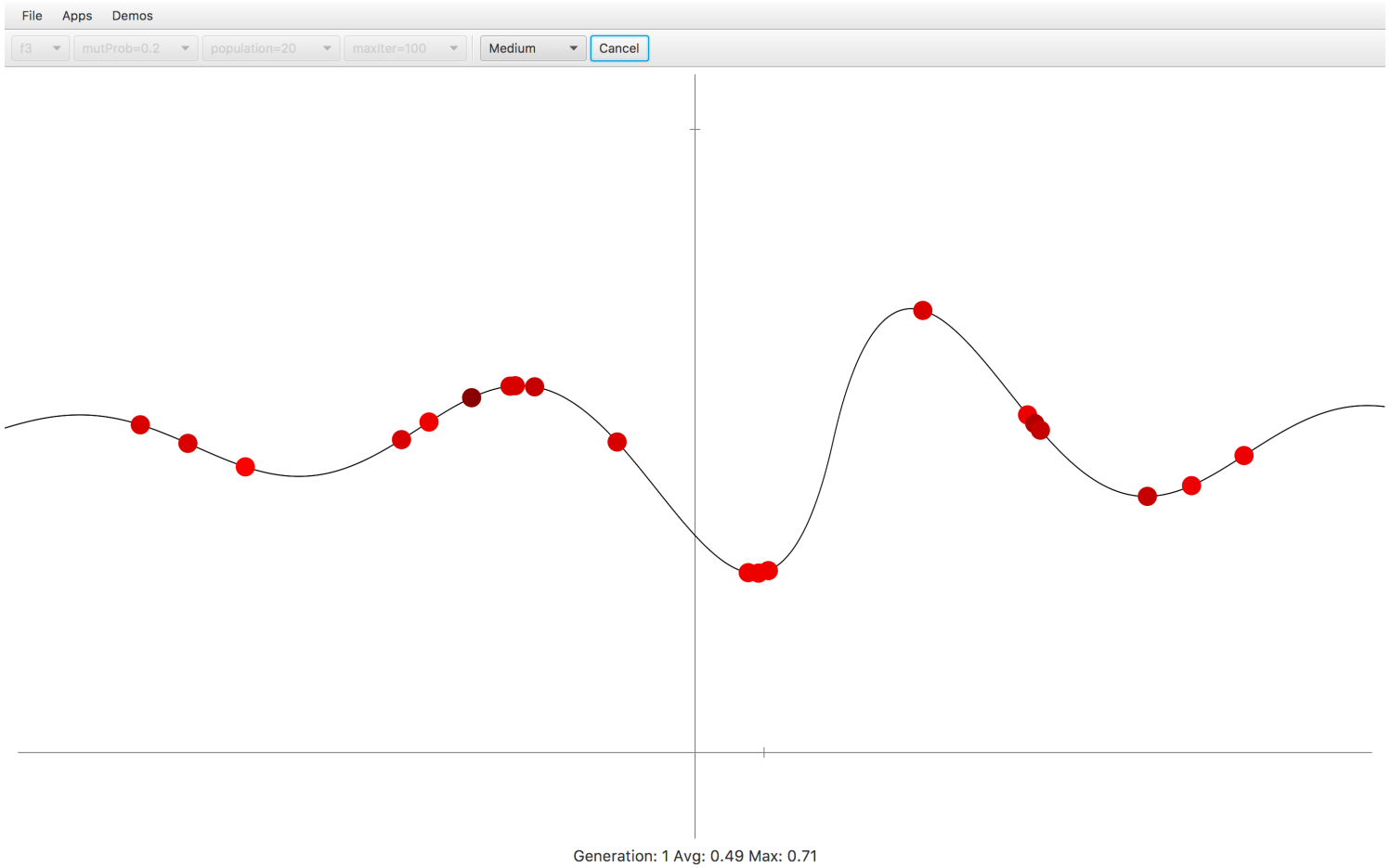


Genetic Algorithm Note

- Evolutionary algorithms (EAs) are a family of population-based search algorithms
- What we introduced is just a simple EA: genetic algorithm
- Other EAs: evolutionary strategy, evolutionary programming, genetic programming as well as their variants.
- Different EAs vary in different search operators (i.e. mutation, crossover and selection)

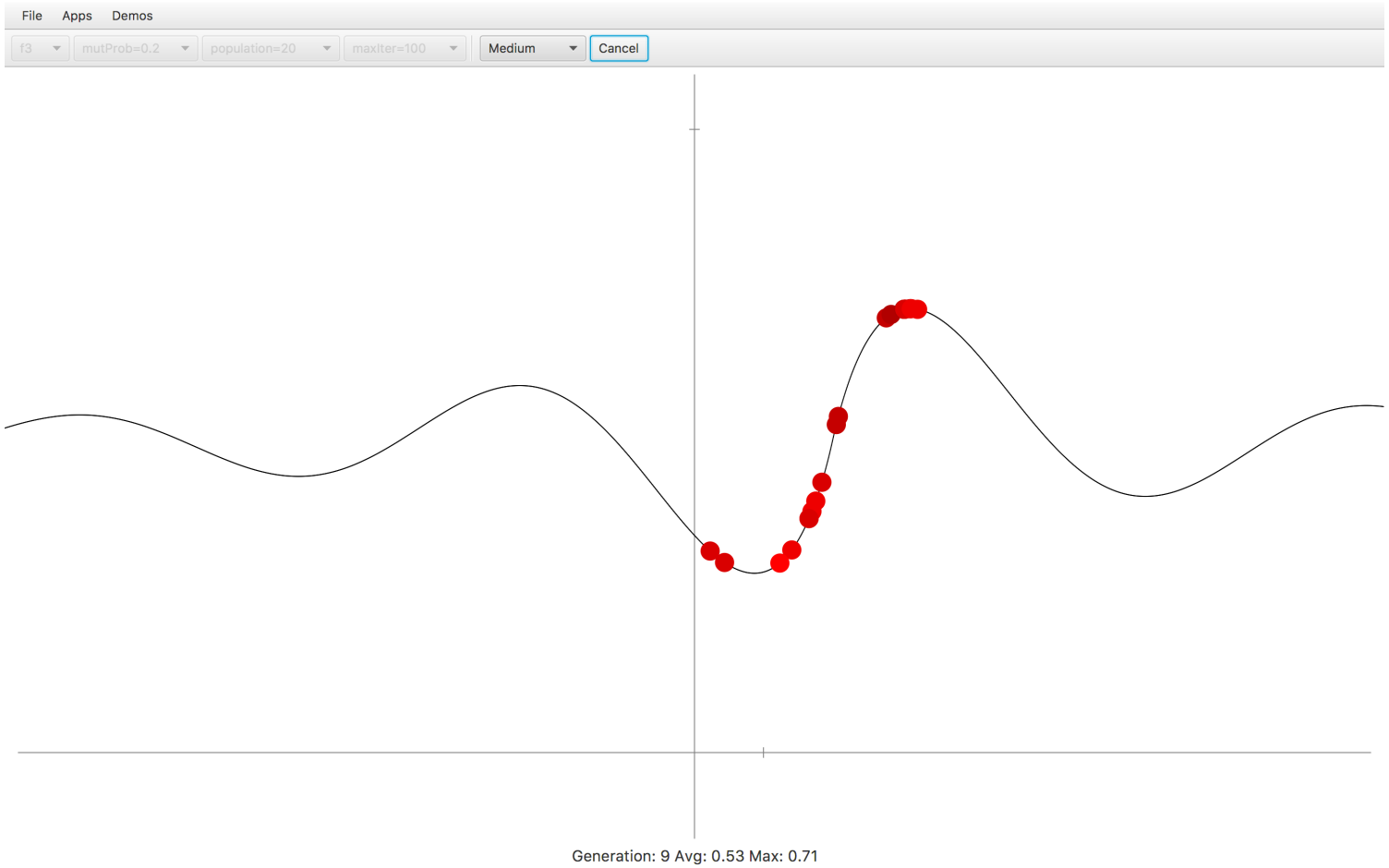
Application of GA for Finding Maximum of Functions

- Beginning



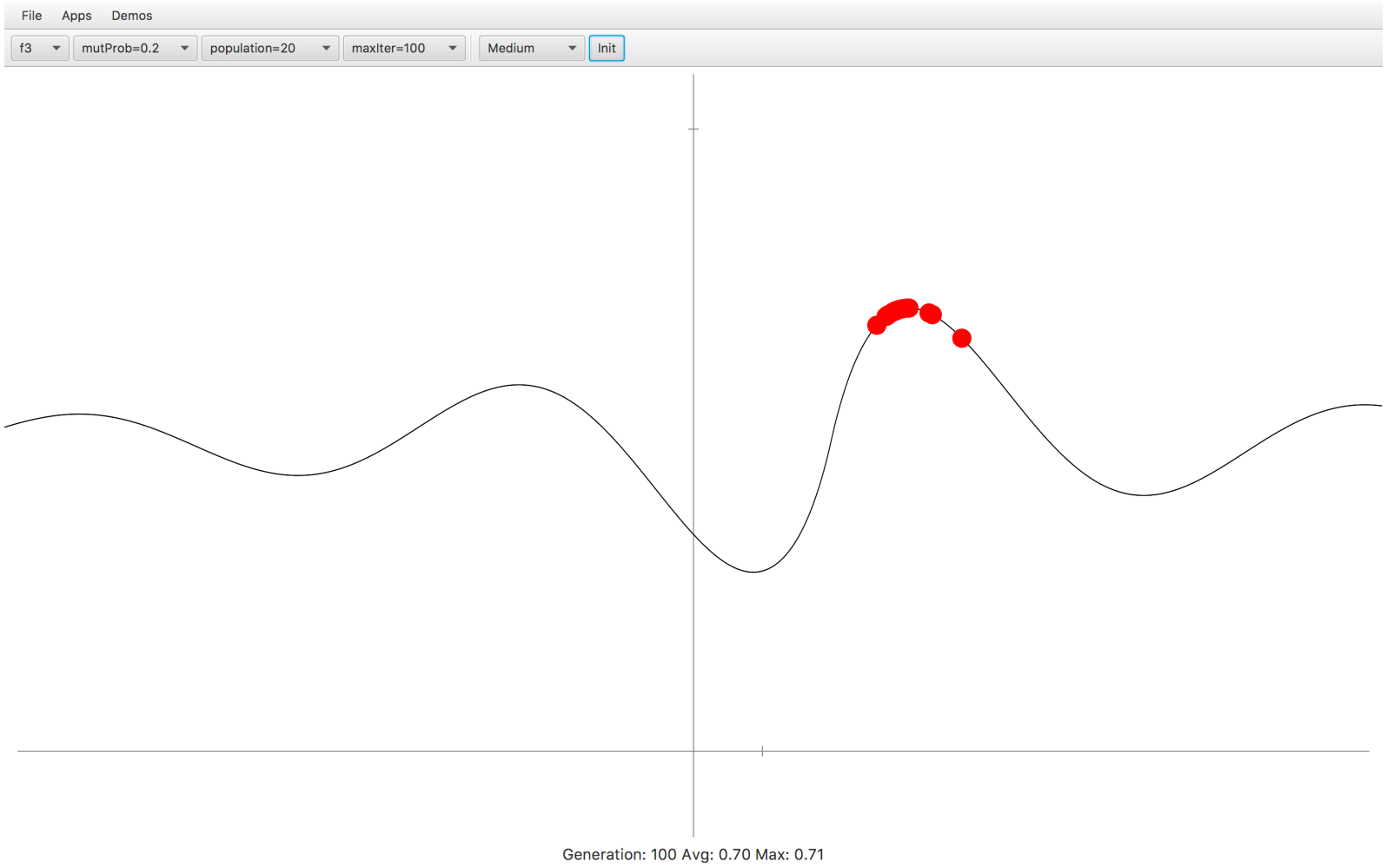
Application of GA for Finding Maximum of Functions

- Midterm



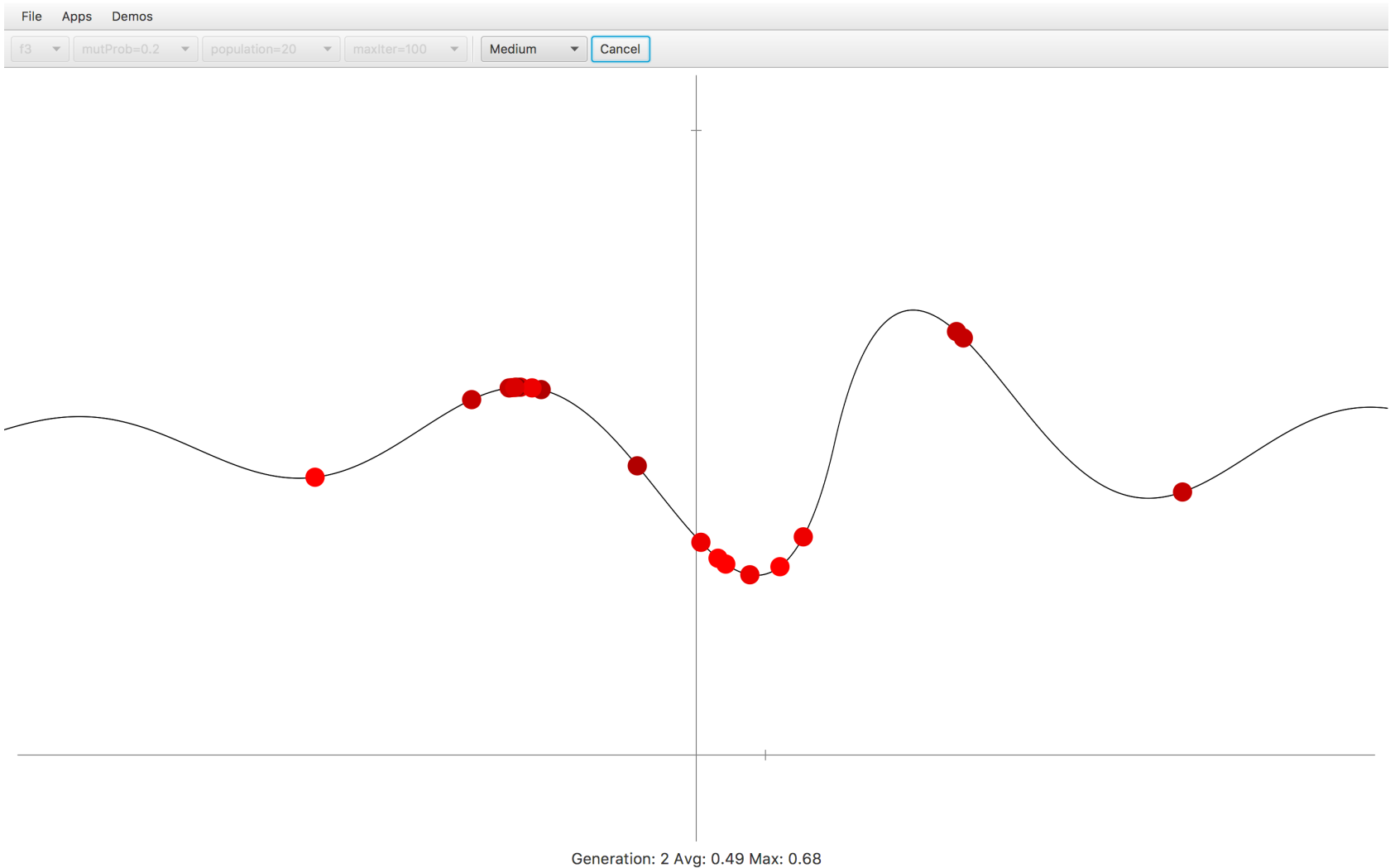
Application of GA for Finding Maximum of Functions

- End



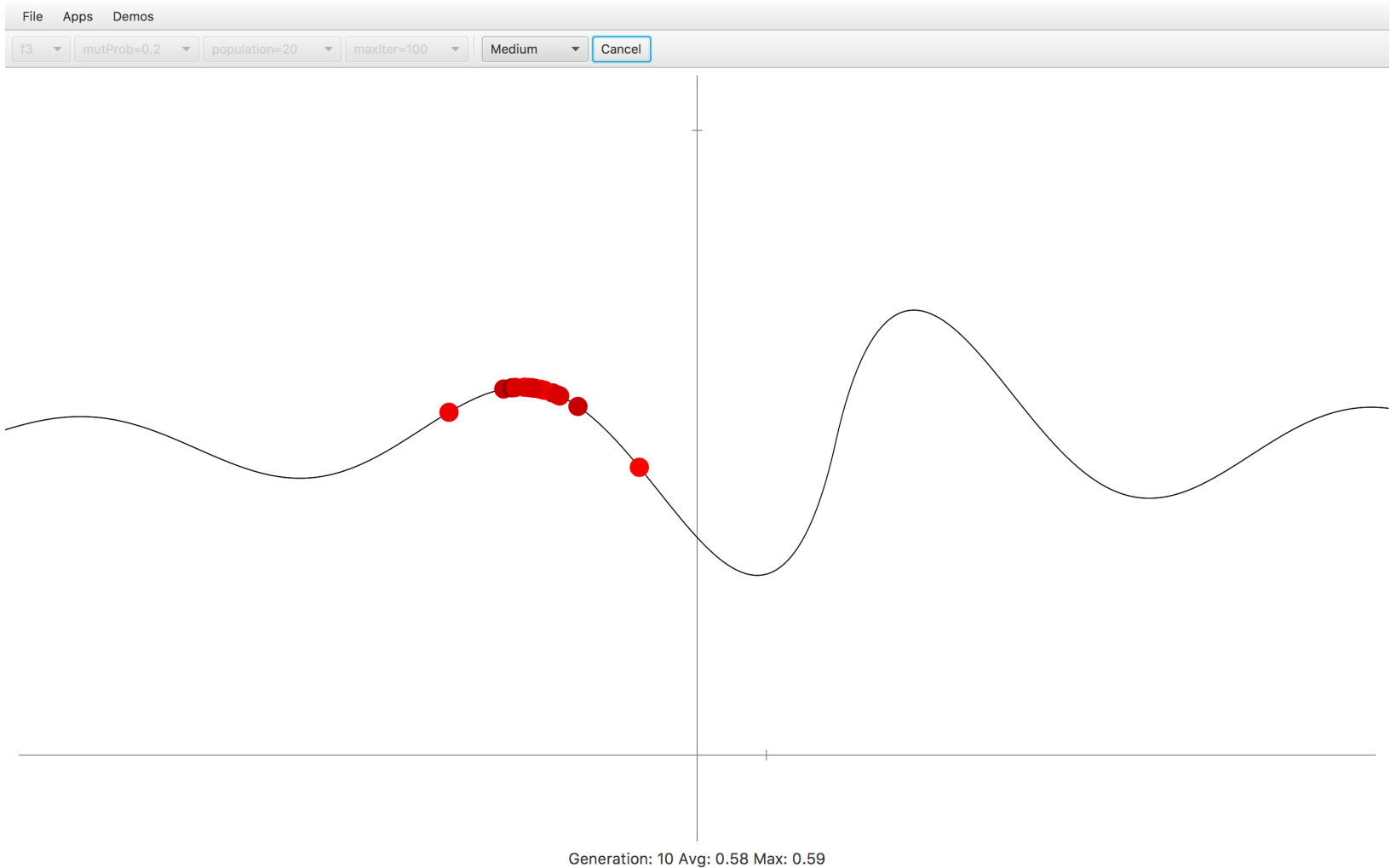
Not Always Get Optimal Solution

- Another run: Beginning



Not Always Get Optimal Solution

- Another run: midterm



Not Always Get Optimal Solution

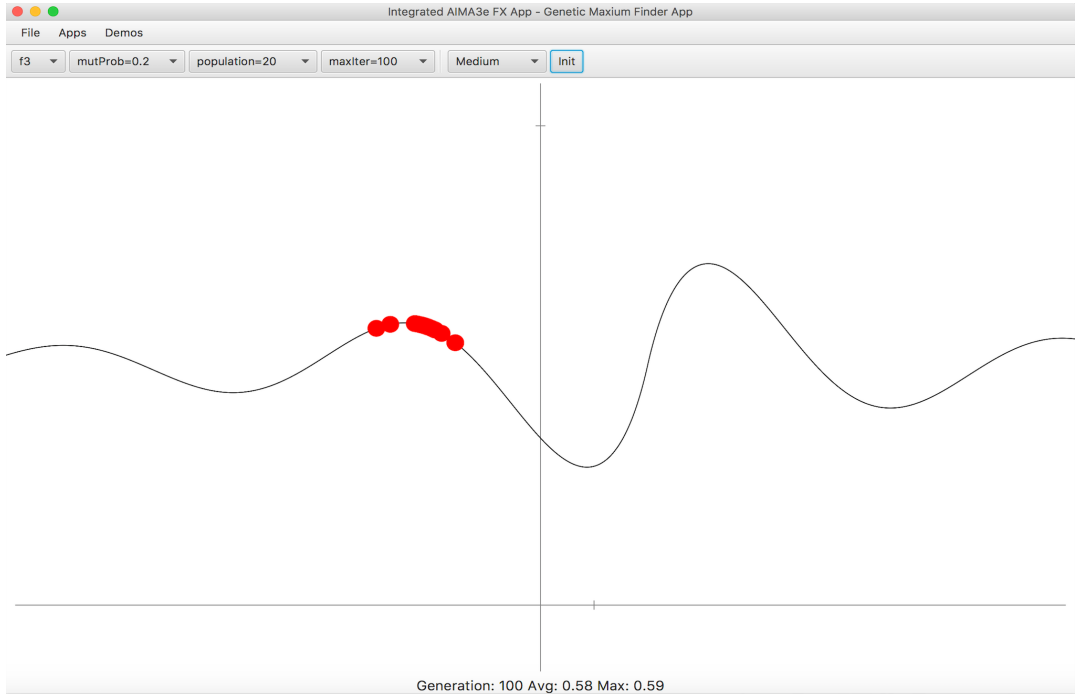
- Another run: end
- For this run, we observe a premature convergence.

Premature convergence occurs when the population has become more difficult to produce offspring better than the parents through crossover and mutation of chromosomes.

Standard crossover simply repeats the current parent.

Further optimization will be completely dependent on the mutation which may be very slow.

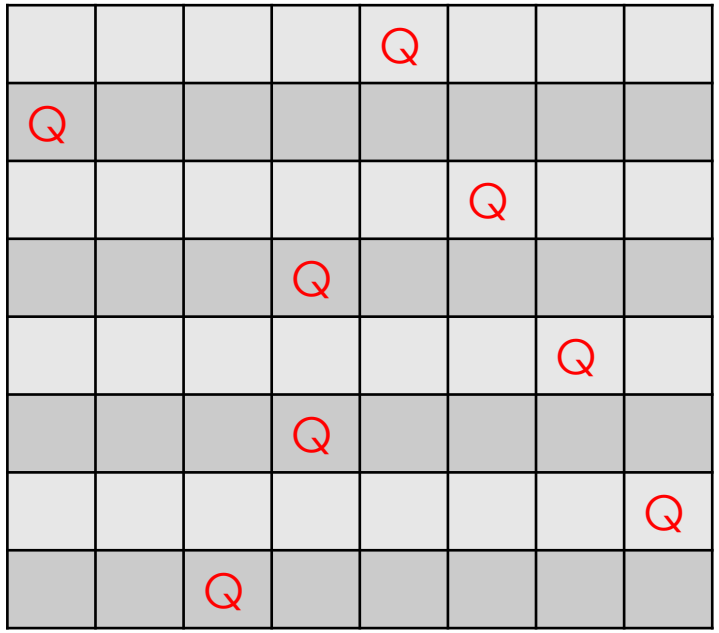
- indicates the randomness of EA

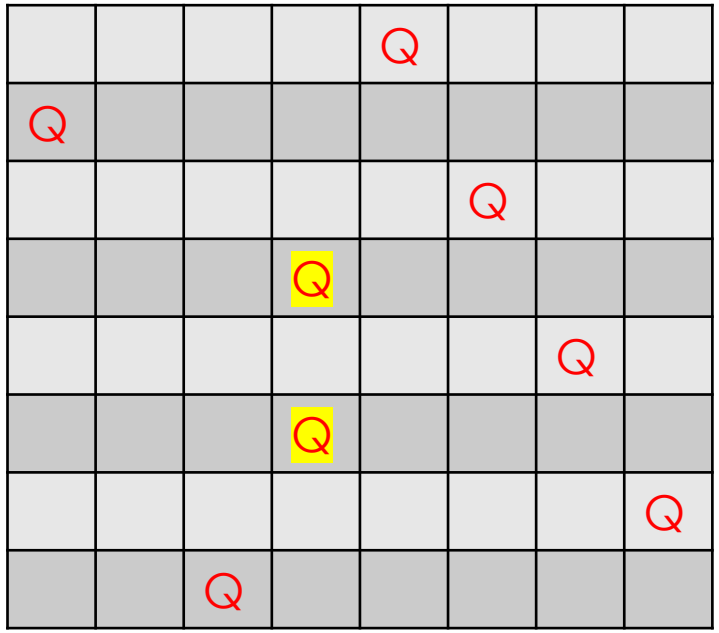


min_conflicts(Local Search for CSP)

- First generate a complete assignment for all variables (this set of assignments may conflict)
- Repeat the following steps until there are no conflicts:
  - Randomly Select a variable that causes conflicts
  - Reassign the value of this variable to another value that with the least constraint conflicts with other variables

min_conflicts：Eight-Queen Problem

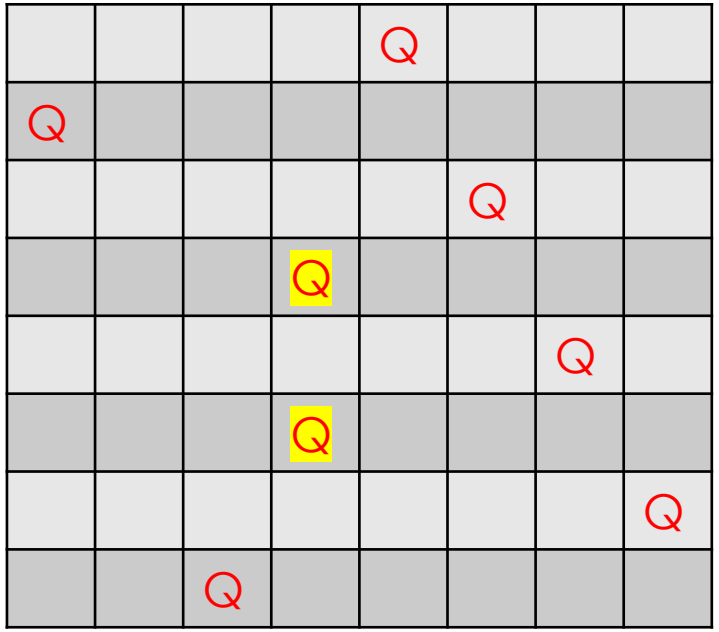


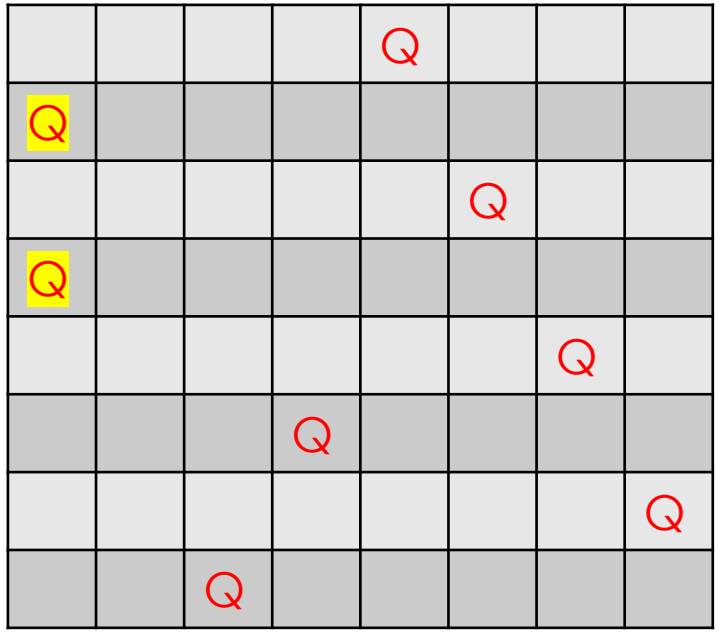


Do an initial complete assignment
(Of course, in most cases this complete assignment will not satisfy all constraints)

Some conflicted variables

min_conflicts： Eight-Queen Problem

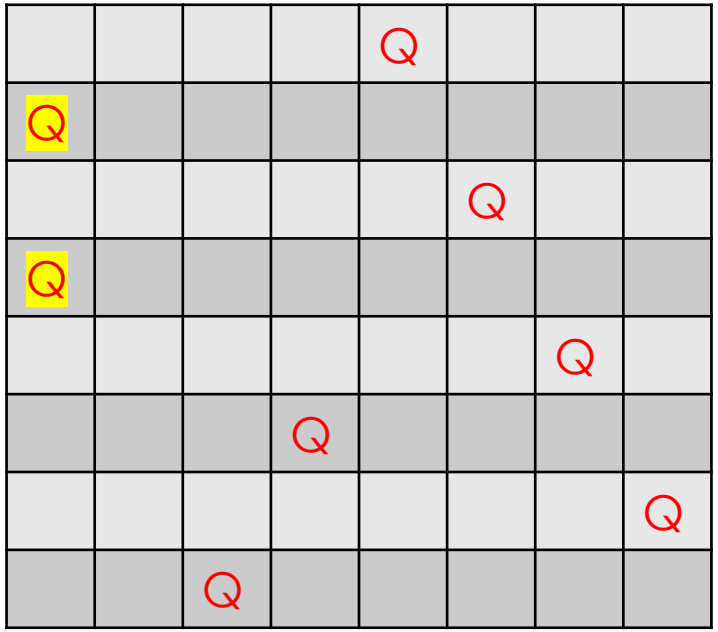


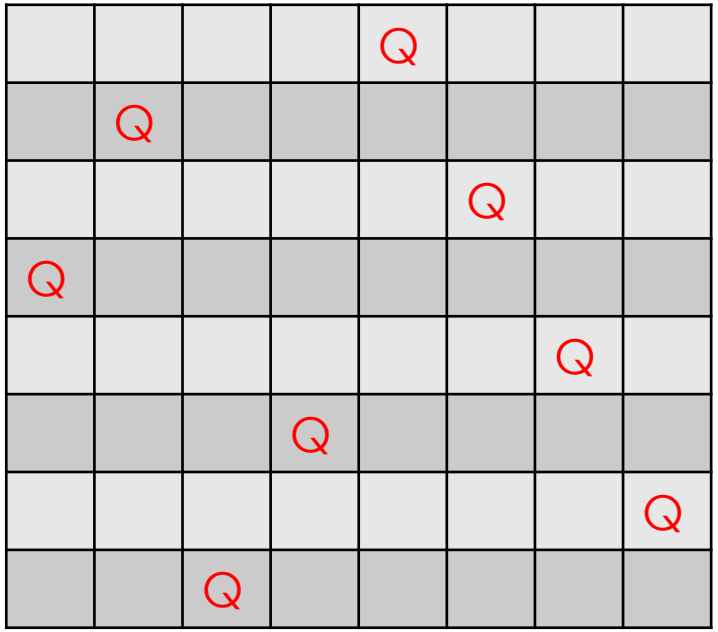


Randomly choose a conflicted variable

Change to the position with the minimum conflict value.

min_conflicts： Eight-Queen Problem





Randomly choose a conflicted variable

Change to the position with the minimum conflict value.

More about Evolutionary Algorithms

- Please refer to “Simple Evolutionary Algorithms” in Chinese or http://www.cs.cmu.edu/~02317/slides/lec_8.pdf in English