07_Metaheuristics

April 28, 2022

1 Metaheuristics

2 Outline

1. Section **??**
2. Section **??**
3. Section **??**
4. Section **??**
5. Section **??**

2.0.1 Setup

[1]:
```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.offsetbox import OffsetImage, AnnotationBbox
import math
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import scipy as sp
import scipy.special as spspecial
```

2.1 Metaheuristics: motivation

- Direct techniques (blind search) are not always possible (they require too much time or memory).
- Weak techniques can be effective if applied correctly on the right kinds of tasks.
  - Typically require domain specific information.

Heuristic: an algorithm which locates 'good enoughì solutions to a problem - do not concerns too much on formal correctness
Trade-off between precision, quality, and accuracy in favor of **feasibility** (computational cost)
Example: greedy search procedure; only takes cost-improving steps.

Metaheuristics are intended to extend the capabilities of heuristics by combining one or more heuristic methods (**procedures**) using a higher-level strategy (hence 'meta'). A procedure in a metaheuristic is considered black-box in that little (if any) prior knowledge is known about it by the metaheuristic, and as such it may be replaced with a different procedure.

2.1.1 Artificial intelligence

AI is a cross-disciplinary field of research that is generally concerned with developing and investigating systems that operate or act intelligently

AI investigates mechanisms underlying intelligent behavior. Traditionally, AI employed a symbolic basis for these mechanisms. A (relatively) newer approach (**scruffy AI**) or soft computing does not necessarily use a symbolic basis, instead patterning these mechanisms after biological or natural processes.

Metaheurstics (MH) is another name for *scruffy AI*

Section **??** define nine properties of MH. 1. Strategies to **guide** the search process.
2. The goal is to efficiently explore the search space in order to find (near-)optimal solutions.
3. They range from simple local search procedures to complex learning processes.
4. MH algorithms are approximate and usually non-deterministic.
5. Usually incorporate mechanisms to avoid getting trapped in confined areas of the search space.
6. The basic concepts of metaheuristics allow for an abstract level description. 7. Not problem-specific. 8. MH may make use of domain-specific knowledge in the form of heuristics that are controlled by the upper level strategy. 9. They have some sort of memory i. e. they **learn** progressively about the search space

C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. ACM Computing Surveys (CSUR), 35(3):268–308, 2003.

2.1.2 Optimization

the search for a combination of parameters (**decision variables**) $x = x_1, x_2, \ldots x_n$ which minimize or maximize some ordinal quantity c (a scalar called **score** or **cost**) assigned by an objective or cost function f that maps $R^n \rightarrow R$ under a set of constraints $g = g_1, g_2, \ldots g_n$

2.1.3 Black box algorithms

Use little, *if any*, information from a problem domain in order to devise a solution. Domain specific knowledge refers to *known relationships between solution representations and the objective cost function.*
Flexibility vs *efficency* tradeoff.
Example: *random search.* Most general and most flexible black box approach and is also the most flexible requiring only the generation of random solutions for a given problem.
Worst case behavior: cost is higher than enumerating the entire search domain.

2.1.4 No free lunch theorem

The Section **??** of search and optimization by Wolpert and Macready proposes that all black box optimization algorithms are the same for searching for the extremum of a cost function when averaged over all possible functions.
Consequence: no **best** general purpose black box optimization algorithm is theoretically possible.

D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation, 1(67):67–82, 1997.

2.1.5 Optimization, 2

Consider the following function:

$$1 + \frac{1}{4000}\sum_{i}^{n} x_i^2 - \prod_{i}^{n} cos(\frac{x_i}{\sqrt{i}})$$

named *Griewank* function.

These are the *first* and *third* order Griewank functions:

How to find the **Global Minimum** of these functions?

How would Gradient Descent perform on these surfaces?

THe Griewank function is one example of rugged function often used as a benchmark in optimization problems.
Other examples include: - *Rastrigin* function: $f(x) = An + \sum_{i=1}^{n} x_i^2 - Acos(2\pi x_i); A = 10, x \in [-5.12, 5.12],\ f(x = 0) = 0$
- *Ackley* function: $f(x, y) = -20e^{-0.2\sqrt{x^2+y^2}} - e^{-0.5(cos(2\pi x)+cos(2\pi y))} + e + 20; f(x = 0) = 0$
- *Rosenbrock* function $f(x, y) = (a - x^2)^2 + b(y - x^2)^2; a = 1, b = 100$ minimum in (a, a^2).

and many more: see https://en.wikipedia.org/wiki/Test_functions_for_optimization

2.1.6 Exercise 1

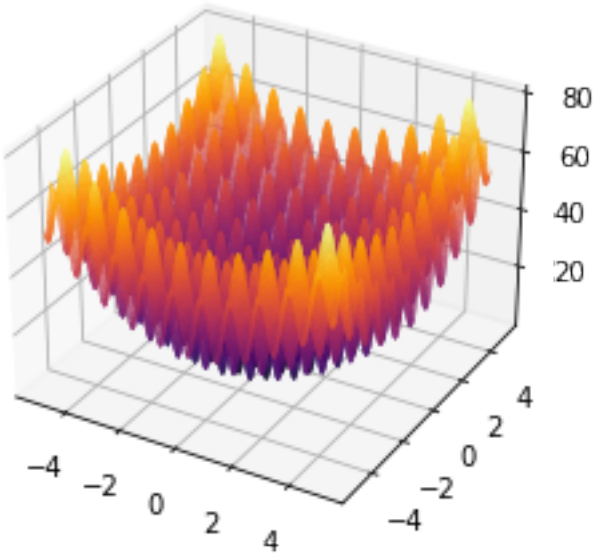
Plot the contours of the 2nd order listed functions.
Hint: *numpy meshgrid*.
Hint: domains for the functions: - Ackley $x_i \in [-5, 5]$ - Rosembrock $x \in [-2, 2], y \in [-1, 3]$

2.1.7 Solution

Rastrigin

[2]:
```
x1 = np.linspace(-5.1, 5.1, 200)
x2 = np.linspace(-5.1, 5.1, 200)
x1, x2 = np.meshgrid(x1, x2)
fx = (x1**2 - 10 * np.cos(2 * np.pi * x1)) + (x2**2 - 10 * np.cos(2 * np.pi *␣
↪x2)) + 20
```

[3]:
```
fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(x1, x2, fx, rstride=1, cstride=1, cmap=mpl.cm.inferno,␣
↪linewidth=0.1)
```

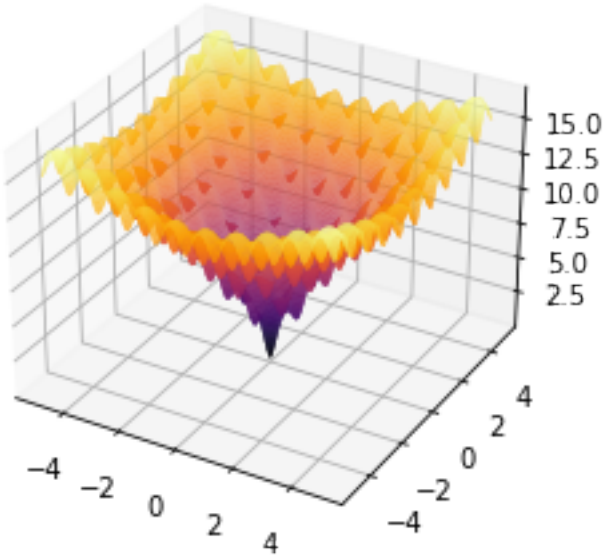


Ackley

[4]:
```
x1 = np.linspace(-5, 5, 200)
x2 = np.linspace(-5, 5, 200)
x1, x2 = np.meshgrid(x1, x2)

fx = -20. * np.exp(-0.2 * np.sqrt(x1**2 + x2**2) ) \
    -np.exp(0.5*(np.cos(2*np.pi*x1) + np.cos(2*np.pi*x2)) ) +\
    np.exp(1) + 20.
```

[5]:
```
fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(x1, x2, fx, rstride=1, cstride=1, cmap=mpl.cm.inferno,␣
 ↪linewidth=0.1)
```

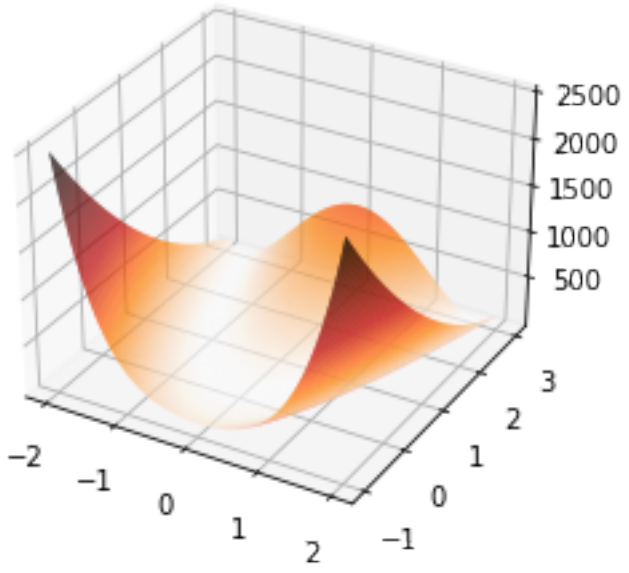


Rosenbrock

[6]:
```
x1 = np.linspace(-2, 2, 200)
x2 = np.linspace(-1, 3, 200)
x1, x2 = np.meshgrid(x1, x2)

fx = (1-x1**2)**2 + 100.*(x2-x1**2)**2
```

[7]:
```
fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(x1, x2, fx, rstride=1, cstride=1, cmap=mpl.cm.
↪gist_heat_r, linewidth=0.1)
```



2.2 Genetic Algorithms: the idea

Genetic Algorithms were first proposed by Section **??** in the '70s and are one of the first proposed metaheuristics.
The idea is to mimick the mechanisms of Dawrinian selection, inheritance and sexual reproduction to explore a search space: starting with a set of candidate solutions these are allowed to change and mix selecting at each step the best ones so far (**survival of the fittest**).

(1) Holland, J. H. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, 1st MIT Press ed.; Complex adaptive systems; MIT Press: Cambridge, Mass, 1992.
(2) Goldberg, D. E. Genetic Algorithms in Search, Optimization, and Machine Learning; Addison-Wesley Pub. Co: Reading, Mass, 1989.

The evolution usually starts from a population of randomly generated individuals and happens in generations.

In each generation, the feasibility (fitness) of every individual in the population is evaluated.
Based on their fitness, parents are selected to reproduce offspring for a new generation; fitter individuals *have more chances to reproduce.*
Offspring *has combination of properties of two parents.*

Reproduction is coupled by *forcing stochastic terms* (mutation) is shaping individuals.

The simulation terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

Terminology **population**, **chromosomes**, **genome**, **genotype**: the current set of candidate solutions

chromosome, **specimen**, **individual**: one specific candidate solution
generation: one step of GA
fitness: the optimization goal e.g. the value of a function

operator a part of the procedure that mimicks one specific mechanism (e. g. mutation)
search space the set of all possible solutions
encoding how the solutions are represented in the procedure **chromosome** blueprint for a **specimen**
trait possible aspect (features) of a specimen **allele** possible settings of trait (value of a feature or coordinate)
locus the position of a gene on the chromosome
genome collection of all chromosomes for an individual

A typical genetic algorithm requires two things to be defined: - a genetic representation of the solution domain, and - a fitness function to evaluate the solution domain.

GA flowchart

GA operators Parent Selection
At each generation, a fraction of the current population breeds new specimens.
Fitter parents have a higher chance to reproduce. Commont parent selection methods include: - roulette wheel seection (the method originally proposed by Holland); each specimen gets a probability of being selected proportional to its fitness - rank selection (aka elitism); the best m specimens are selected as parents - tournament: if m parents are needed, m groups of size t (the *tournament size*) are selected randomly and in each group the best are chosen

Crossover or **Inheritance** or **Recombination** Parents are grouped in pairs and pass their genes to offspring. Each new specimen gets copies half the alleles for each parent. Under a given probability (*crossover probability*) this may involve reordering of bits (in binary representation):

or different interpolation

$$p_0 = [p_{00}, p_{01}], p_1 = [p_{10}, p_{11}]$$

without crossover:

$$c_0 = [0.5(p_{00}+, p_{10}], [0.5(p_{01}+, p_{11}]$$

with crossover:

Mutation
All individuals have a chance (*mutation probability*) that a randomly selected locus will change allele i. e. a coordinate gets a new value sampled from the search space.

Evaluation or **Fit selection**
After children have been added to the population a fraction of the of less fit solutions (children, parents or both) are **selected** i. e. eliminated. This helps keep the diversity of the population large, preventing premature convergence on poor solutions. Popular and well-studied selection methods include roulette wheel selection and tournament selection.

Encoding
Chromosomes may be: - Bit strings (0101 … 1100)

- Real numbers (43.2 -33.1 … 0.0 89.2) - Permutations of elements (E11 E3 E7 … E1 E15) - Lists of rules (R1 R2 R3 … R22 R23) - Any data structure …

```
<img src="chromosome.png" width="500"/>
```

In Holland's original formulation and in many textbooks and papers chromosomes are bit strings.

Limitations of binary encoding:
1. Needs an effort to convert into binary from
2. Accuarcy depends on the binary reprresentation

Advantages:
1. Since operations with binary represntation is faster, it provide a faster implementations of all GA operators and hence the execution of GAs.
2. Any optimization problem has it binary-coded GA implementation

The Schema Theorem Suppose that the set of chromosomes is represented by by binary strings of length 4, the **schema** (pattern) 1*01 represents all those chromosomes beginning with 1 and terminating with 01

For each schema, we can assign two measurements: - **Order**: The number of digits that are fixed (not wildcards) - **Defining length**: The distance between the two furthermost fixed digits

Examples of 4 digit *schemata*:

Schema	Order	Defining Lenght
1101	4	3
001	3	2
10	2	1
1***	1	0
1**1	2	3

Each chromosome *can match* different schemata. The higher the fitness a chromosome, the higher the chances that it will suvive and with it the schemata it matches.
Genetic operators will change the schemata matched by chromosomes and the schemata of low order and short defining length are the ones more likely to survive (e. g. ****).

As a consequence, the frequency of schemata of low order, short defining length, and above-average fitness increases exponentially in successive generations.
I. e. the smaller, simpler sequences correlated to better solutions will become more frequent with passing generations.

Advantages of GA: - No derivatives needed
- Easy to parallelize
- Can escape local minima
- Works on a wide range of problems

Disadvantages:
- Need much more function evaluations than linearized methods
- No guaranteed convergence even to local minimum
- Have to discretize parameter space

2.3 Function optimization

Let's code a simple GA with real encoding a search the GM of the Griewank function. Assume that the population and fitness are real values stored in numpy arrays.
Also, assume the chromosomes are **sorted**.

[8]:
```
def griewank(X, *args, **kwargs):
    """
    Return Griewank function for an array
    """
    squaresSum = np.sum(X**2) / 4000.
    den = np.sqrt(np.arange(1, X.shape[0]+1))
    cosProd = np.prod(np.cos(X/den))
    f = squaresSum - cosProd + 1.0
    return f
```

[9]:
```
def init_population(nspecimen, ndim, sspace):
    """
    Generate nspecimen vectors in ndim dimensions
    from the given search space
    """
    datapoints = list()
    for i in range(nspecimen):
        datapoints.append(np.asarray([np.random.choice(sspace) for i in␣
 ↪range(ndim)]))
    datapoints = np.asarray(datapoints)
    return datapoints
```

[10]:
```
def main_ga_loop(ngeneration, population, sel_press, pCO, pMut, sspace, ffunc,␣
 ↪**kwargs):
    """
    Applies the GA loop to the input population and returns the best
    chromosome, its fitness and the final population
    """
    if not "tsize" in kwargs.keys():
        kwargs["tsize"] = 2
    if not "vstep" in kwargs.keys():
        kwargs["vstep"] = 50
    if not "sel_meth" in kwargs.keys():
        kwargs["sel_meth"] = "rank"
    if not "co_meth" in kwargs.keys():
        kwargs["co_meth"] = "interp"
    if not "alpha" in kwargs.keys():
        kwargs["alpha"] = 0.5
    if not "mut_meth" in kwargs.keys():
        kwargs["mut_meth"] = "const"
    if not "tol" in kwargs.keys():
        kwargs["tol"] = 1e-5
```

```
    if not "ffkwds" in kwargs.keys():
        kwargs["ffkwds"] = dict()
    fitness, population = calc_fitness(min_best, ffunc, population,␣
↪kwargs["ffkwds"])
    best_previous = fitness[0]
    best = list()
    best_f = list()
    for gen in range(ngeneration):
        # select fitter specimens for reproduction
        nmating, parents = ParentSelection(sel_press, kwargs["sel_meth"],␣
↪population, kwargs["tsize"])
        # generate offspring and check cross over probability
        offspring = crossover(pCO, kwargs["alpha"], nmating, parents,␣
↪population, method=kwargs["co_meth"])
        population = np.concatenate((population, offspring))
        # mutate
        mutated, population = mutation(pMut, kwargs["mut_meth"], population,␣
↪sspace)
        # calculate fitness
        fitness, population = calc_fitness(min_best, ffunc, population,␣
↪kwargs["ffkwds"])
        if gen % kwargs["vstep"]== 0:
            print("Generation ",gen, " best specimen ",fitness[0])
        # remove less fit individuals
        population = fit_selection(nmating, population)
        gain = abs(fitness[0]-best_previous)
        if kwargs['tol'] > 0:
            if gen > kwargs["vstep"] and gain < kwargs["tol"]:
                print("Gain/loss less than tolerance: ",gain)
                break
        best.append(population[0])
        best_f.append(fitness[0])
    return gen, best_f, best
```

[11]:
```
def calc_fitness(min_best, ffunc, population, ffkwds):
    fitness = np.apply_along_axis(ffunc, 1, population, **ffkwds)
    order = np.argsort(fitness)
    if not min_best:
        order = order[::-1]
    population = population[order]
    fitness = fitness[order]
    return fitness, population
```

[12]:
```
def ParentSelection(sel_press, sel_meth, pop, tsize):
    nmating = math.ceil(sel_press * pop.shape[0])
    if nmating % 2 !=0:
```

```
        nmating = nmating - 1
    M = list(range(pop.shape[0]))
    if nmating <= 1:
        nmating = 2
    if sel_meth == "tournament":
        parents = TournamentSelection(nmating,  M, pop, tsize)
    elif sel_meth == "rank":
        parents = list(range(nmating))
    return nmating, parents
```

[13]:
```
def crossover(pCO, alpha, nmating,parents, population, method=None):
    if method == None:
        raise ValueError("No Crossover method selected")
    elif method == "interp":
        offspring = interp_crossover(pCO, alpha, nmating, parents, population)
    return offspring
```

[14]:
```
def interp_crossover(pCO, alpha, nmating, parents, population):
    offspring = list()
    for i in range(0,nmating-1,2):
        p0 = population[parents[i]]
        p1 = population[parents[i+1]]
        coin = np.random.rand()
        if coin <= pCO:
            child0 = alpha*p0 + (1.-alpha)*p1
            child1 = (1.-alpha)*p0 + alpha*p1
        else:
            child0 = p0
            child1 = p1
        offspring.append(child0)
        offspring.append(child1)
    return np.asarray(offspring)
```

[15]:
```
def mutation(prob, mut_meth, population, sspace):
    if mut_meth == "const":
        mutated, population = mutation_const(prob, population, sspace)
    elif mut_meth == "prop":
        mutated, population = mutation_prop(prob, population, sspace)
    return mutated, population
```

[16]:
```
def mutation_const(prob, population, sspace):
        mutated = list()
        for s in range(population.shape[0]):
            coin = np.random.rand()
            if coin < prob:
                newallele  = np.random.choice(sspace)
                locus = np.random.choice(population.shape[1])
```

```
                population[s, locus]=newallele
        return mutated, population
```

[17]:
```
def fit_selection(nmating, population):
    return population[:-nmating]
```

Let's see if it find the minimum.

[18]:
```
ndim = 3
domain = (-200.,200.)
sspace = np.arange(domain[0],domain[1],0.001)
min_best = True
```

[19]:
```
mypop = init_population(100, 3, sspace)
mypop.shape
```

[19]:
```
(100, 3)
```

[20]:
```
#def main_ga_loop(ngeneration, population, sel_press, pCO, pMut, sspace, ffunc,␣
 ↪sel_meth="tournament",\
#                    tsize=2,, mut_meth="const", min_best=False, alpha=0.5,␣
 ↪tol=1e-5, vstep)
gen, best_f, best_spec = main_ga_loop(500, mypop, 0.5, 0.5, 0.1, sspace,␣
 ↪griewank,sel_meth="rank",\
                                      min_best=True,vstep=50)
print("Final gen ", gen, "best fitness ", best_f[-1], "best solution ",␣
 ↪best_spec[-1])
```

```
Generation  0  best specimen  0.9025824231571582
Generation  50  best specimen  0.01581648487789511
Generation  100  best specimen  0.01581648487789511
Generation  150  best specimen  0.01581648487789511
Generation  200  best specimen  0.01581648487789511
Generation  250  best specimen  0.01581648487789511
Generation  300  best specimen  0.01581648487789511
Generation  350  best specimen  0.01342884251687304
Generation  400  best specimen  0.01342884251687304
Generation  450  best specimen  0.01342884251687304
Final gen  499 best fitness  0.01342884251687304 best solution  [-3.81249905e-03
4.45200000e+00  5.35375000e+00]
```

2.3.1 Exercise 2

1. Implement the tournament selection method
2. Implement a mutation method such that the number of mutated genes is proportional *to the number of genes in the population*
3. Test on one of the other test functions

2.3.2 Solution

[21]:
```
def TournamentSelection(nmating, M, pop, tsize):
    parents = list()
    for i in range(nmating):
    # selected random parents and sort stsize with best fitness
        competitors = np.random.choice(M, size=tsize)
        added = False
        for i in competitors:
            if i in parents:
                continue
            else:
                parents.append(i)
                added = True
                break
        if added is False:
            parents.append(competitors[0])
    return parents
```

Q 2

[22]:
```
def mutation_prop(prob, population, sspace):
    tot_genes = population.size
    num_mut = math.ceil(prob* tot_genes)
    mut_alleles = np.random.choice(sspace, size=num_mut)
    nspec = num_mut//population.shape[1]
    mut_spec = np.random.choice(sspace, size=num_mut)
    mut_alleles.shape = (nspec, population.shape[1])
    mut_spec = np.random.choice(population.shape[0], size=nspec)
    population[mut_spec] = mut_alleles
    return mut_spec, population
```

Q 3 - Rastrigin function

[23]:
```
def rastrigin(X, *args, **kwargs):
    return (X[0]**2 - 10 * np.cos(2 * np.pi * X[0])) +\
        (X[1]**2 - 10 * np.cos(2 * np.pi * X[1])) + 20
```

[24]:
```
ndim = 2
domain = (-5.1,5.1)
sspace = np.arange(domain[0],domain[1],0.001)
min_best = True
```

[25]:
```
mypop = init_population(100, 2, sspace)
mypop.shape
```

[25]: (100, 2)

Rastrigin function with tournament selection and proportional mutation

[26]:
```
gen, best_f, best_spec = main_ga_loop(500, mypop, 0.5, 0.5, 0.2, sspace,␣
 ↪rastrigin,sel_meth="tournament",\
                                      mut_meth="prop", min_best=True,vstep=50)
print("Final gen ", gen, "best fitness ", best_f[-1], "best solution ",␣
 ↪best_spec[-1])
```

```
Generation  0  best specimen  6.181465304490782
Generation  50  best specimen  0.5807555178405082
Generation  100  best specimen  0.5807555178405082
Generation  150  best specimen  0.5807555178405082
Generation  200  best specimen  0.5807555178405082
Generation  250  best specimen  0.5807555178405082
Generation  300  best specimen  0.5807555178405082
Generation  350  best specimen  0.5807555178405082
Generation  400  best specimen  0.5807555178405082
Generation  450  best specimen  0.5807555178405082
Final gen  499 best fitness  0.33057626857779354 best solution  [ 0.04014063
-0.00796875]
```

Rastrigin function with elitism and constant mutation

[27]:
```
gen, best_f, best_spec = main_ga_loop(500, mypop, 0.5, 0.5, 0.1, sspace,␣
 ↪rastrigin,sel_meth="rank",\
                                      mut_meth="const", min_best=True,vstep=50)
print("Final gen ", gen, "best fitness ", best_f[-1], "best solution ",␣
 ↪best_spec[-1])
```

```
Generation  0  best specimen  5.326555979482933
Generation  50  best specimen  0.24250375954767378
Generation  100  best specimen  0.0022318551054567592
Generation  150  best specimen  0.0017854761939375408
Generation  200  best specimen  0.0007935579604883003
Generation  250  best specimen  0.0007935579604883003
Generation  300  best specimen  0.0007935579604883003
Generation  350  best specimen  0.0007935579604883003
Generation  400  best specimen  0.0007935579604883003
Generation  450  best specimen  0.0007935579604883003
Final gen  499 best fitness  0.0007935579604883003 best solution
[-2.00000000e-03  1.70352621e-12]
```

Q 3 - Ackley function

[28]:
```
def ackley(X, *args, a=20, b=0.2, c=0.5, d=2.*np.pi, **kwargs):
    return -a * np.exp(-b * np.sqrt(X[0]**2 + X[1]**2) ) \
    -np.exp(c*(np.cos(d*X[0]) + np.cos(d*X[1])) ) + np.exp(1) + a
```

[29]:
```
ndim = 2
domain = (-30,30)
sspace = np.arange(domain[0],domain[1],0.001)
min_best = True
```

[30]:
```
mypop = init_population(100, 2, sspace)
mypop.shape
```

[30]:
```
(100, 2)
```

[31]:
```
gen, best_f, best_spec = main_ga_loop(500, mypop, 0.5, 0.5, 0.3, sspace,␣
↪ackley,sel_meth="tournament",\
                                      mut_meth="const", min_best=True,vstep=50)
print("Final gen ", gen, "best fitness ", best_f[-1], "best solution ",␣
↪best_spec[-1])
```

```
Generation  0  best specimen  7.166831089829632
Generation  50  best specimen  0.0494546868877066
Generation  100  best specimen  0.3085260985168965
Generation  150  best specimen  0.31921395584280177
Generation  200  best specimen  0.22475493813603364
Generation  250  best specimen  0.1980604815381639
Generation  300  best specimen  0.18939387830939225
Generation  350  best specimen  0.19438731572871149
Generation  400  best specimen  0.19345082871560848
Generation  450  best specimen  0.19998265444055718
Final gen  499 best fitness  0.2002541507153488 best solution  [-0.03089776
-0.024991  ]
```

Q 3 - Rosenbrock function

[32]:
```
def rosembrock(X, *args, a=1, b=100, **kwargs):
    return (a-X[0]**2)**2 + b*(X[1]-X[0]**2)**2
```

[33]:
```
ndim = 2
domain = (-3,3)
sspace = np.arange(domain[0],domain[1],0.001)
min_best = True
```

[34]:
```
mypop = init_population(100, 2, sspace)
mypop.shape
```

[34]:
```
(100, 2)
```

[35]:
```
gen, best_f, best_spec = main_ga_loop(500, mypop, 0.5, 0.5, 0.1, sspace,␣
↪rosembrock,sel_meth="tournament",\
                                      mut_meth="prop", min_best=True,vstep=50)
```

```
print("Final gen ", gen, "best fitness ", best_f[-1], "best solution ",␣
 ↪best_spec[-1])
```

```
Generation  0  best specimen  0.2836369103366649
Generation  50  best specimen  0.020875216671261455
Generation  100  best specimen  0.012951679380278075
Generation  150  best specimen  0.0012832305870531752
Generation  200  best specimen  0.0012832305870511607
Generation  250  best specimen  0.0012832305870499982
Generation  300  best specimen  0.001283230587048754
Generation  350  best specimen  0.0012832305870480246
Generation  400  best specimen  0.00030741450088156983
Generation  450  best specimen  0.00030741450088156983
Final gen  499 best fitness  0.00030741450088156983 best solution  [1.007 1.013]
```

GAs are not deterministic!

2.4 Traveling Salesman Problem

The Section **??** (TSP) is a famous problem in computer science. It might be summarized as follows: - you are a salesperson who needs to visit some number of cities - you want to minimize costs spent on traveling
hence you want to find out the *most efficient route*, one that will require the least amount of traveling. You are given a coordinate of the cities to visit on a map. How can you find the optimal route?

this is a **combinatorial optimization problem** i. e. our objective function (cost function) is defined a very large but **discrete** domain whose solutions are often found permutating the object of the solutions.
Other examples include: - Bin-Packing: given a set of N objects each with a specified size s, fit them into as few bins (each of size B) as possible - Job-shop Scheduling: given a set of jobs that must be performed, and a limited set of tools with which these jobs can be performed, find a schedule for what jobs should be done when and with what tools that minimizes the total amount of time until all jobs have been completed. *HPC batch systems* are an example.

What about *brute force*? You consider all the different possibilities, calculate the estimated distance for each, and choose the one that is the shortest path. The computational cost would be an **O(n!)**! Sad conclusion: when brute force cannot solve your problem you do not have enough.

2.4.1 Encoding

We use once again a real encoding. We construct a matrix that represent distances between cities i and j such that A_{ij} represents the distance between those two cities. We will represent cities by their indices.

[36]:
```
cities = [0, 1, 2, 3, 4]

adjacency_mat = np.asarray(
    [
        [0.00, 28.02, 17.12, 27.46, 46.07],
```

```
        [28.02, 0.00, 34.00, 25.55, 25.55],
        [17.12, 34.00, 0.00, 18.03, 57.38],
        [27.46, 25.55, 18.03, 0.00, 51.11],
        [46.07, 25.55, 57.38, 51.11, 0.00],
    ]
)
```

In this we assume that each city has a direct route to any other one (a fully connected graph). We can represent a single specimen using a sequence of cities. Each chromosome is a route:

[37]:
```
def init_population(cities, nspecimen):
    return np.asarray([np.random.permutation(cities) for i in range(nspecimen)])
example_pop = init_population(cities,10)
example_pop
```

[37]:
```
array([[0, 1, 4, 3, 2],
       [4, 3, 1, 0, 2],
       [1, 4, 3, 2, 0],
       [1, 2, 0, 3, 4],
       [1, 3, 4, 0, 2],
       [2, 1, 4, 3, 0],
       [0, 3, 2, 4, 1],
       [4, 1, 0, 3, 2],
       [3, 1, 2, 0, 4],
       [4, 3, 1, 0, 2]])
```

The fitness will be the sum of distances along the route:

[38]:
```
def route_fitness(route, amat=adjacency_mat):
    dist = 0.
    for city in range(len(route)-1):
        dist = dist + amat[route[city], route[city+1]]
    return dist
```

[39]:
```
np.apply_along_axis(route_fitness, 1, example_pop, amat=adjacency_mat)
```

[39]:
```
array([122.71, 121.8 , 111.81, 129.69, 139.85, 138.12, 128.42,  99.06,
       122.74, 121.8 ])
```

Mutation can be achieved in different ways as well but: - we cannot just change a random city since each route is complete - a random switch can be highly destructive of the fitness of a given route

so we try to carry out mutation by switching two neighbour cities in a route. In binary encoding this is a **swap mutation**.

[40]:
```
def mutation(prob, mut_meth, population, sspace):
    if mut_meth == "const":
```

```
        mutated, population = mutation_const(prob, population, sspace)
    elif mut_meth == "prop":
        mutated, population = mutation_prop(prob, population, sspace)
    elif mut_meth == "swap":
        mutated, population = mutation_swap(prob, population, sspace)
    return mutated, population
```

[41]:
```
def mutation_swap(prob, population, sspace):
    tot_genes = population.size
    num_swaps = math.ceil(prob * tot_genes)
    nspec = num_swaps//population.shape[1]
    mut_spec = np.random.choice(population.shape[0], size=num_swaps)
    for ms in mut_spec:
        MS = population[ms]
        locus = np.random.choice(len(MS)-1)
        sign = int(np.sign(np.random.rand()-0.5))
        tmp = MS[locus]
        MS[locus] = MS[locus+1*sign]
        MS[locus+1*sign] = tmp
    return mut_spec, population
```

We can now try to optmize the route of a bigger number of cities. For a given number of cities the number of routes is $n!$ which can be approximated with the Stirling formula or calculated with a recursive function.
But we need to calculate the adiacency matrix as well.

2.4.2 Exercise 3

1. Implement the tournament selection method
2. Implement a mutation method such that the number of mutated genes is proportional *to the number of genes in the population*
3. Test on one of the other test functions

2.4.3 Solution

[42]:
```
def calc_amat(cities, min_dist, max_dist):
    n = len(cities)
    amat = np.zeros((n,n))
    for i in range(n-1):
        amat[i, i+1:] = (max_dist-min_dist)*np.random.rand(n-i-1) + min_dist
        amat[i+1:, i] = amat[i, i+1]
    return amat
```

[43]:
```
calc_amat(cities, 10, 50)
```

[43]:
```
array([[ 0.        , 11.93781491, 23.32055195, 19.86130004, 11.45011217],
       [11.93781491,  0.        , 37.3451117 , 21.52771256, 22.22359102],
       [11.93781491, 37.3451117 ,  0.        , 22.45830079, 30.62253689],
```

```
       [11.93781491, 37.3451117 , 22.45830079,  0.        , 21.1760703 ],
       [11.93781491, 37.3451117 , 22.45830079, 21.1760703 ,  0.        ]])
```

2.4.4 Crossover of permutations

For crossover we need to take subroutes from either parent and mix them; we *cannot* just mix cities because *each city must be visited once*

[44]:
```
def crossover(pCO, alpha, nmating,parents, population, method=None):
    if method == None:
        raise ValueError("No Crossover method selected")
    elif method == "interp":
        offspring = interp_crossover(pCO, alpha, nmating, parents, population)
    elif method == "ordered":
        offspring = ordered_crossover(pCO, nmating, parents, population)
    return offspring
```

[45]:
```
def ordered_crossover(pCO, nmating, parents, population):
    offspring = list()
    for i in range(0,nmating-1,2):
        p0 = population[parents[i]]
        p1 = population[parents[i+1]]
        coin = np.random.rand()
        if coin <= pCO:
            pbreak = np.random.choice(len(p0)-1)
            lbreak = np.random.choice(len(p0)-pbreak)
            child0      = -1*np.ones(len(p0), dtype='int')
            child1      = -1*np.ones(len(p1), dtype='int')
            for locus, al in enumerate(p0):
                if locus < pbreak or locus >= pbreak+lbreak:
                    l1 = np.where(p1==al)[0][0]
                    child0[locus] = p1[l1]
                    l1 = np.where(p0==p1[locus])[0][0]
                    child1[locus] = p0[l1]
                else:
                    child0[locus] = p0[locus]
                    child1[locus] = p1[locus]
            if np.any(child0==-1) or np.any(child1==-1) \
            or len(set(child0))!=len(p0) or len(set(child1))!=len(p1):
                raise ValueError
        else:
            child0 = p0
            child1 = p1
        offspring.append(child0)
        offspring.append(child1)
    return np.asarray(offspring)
```

2.4.5 Try to solve the TSP problem

[46]:
```
cities = list(range(20))
adjacency_mat = calc_amat(cities, 10., 50.)
mypop = init_population(cities,100)
fitness = np.apply_along_axis(route_fitness, 1, mypop, amat=adjacency_mat)
np.max(fitness), np.min(fitness), np.mean(fitness), np.std(fitness)
```

[46]:
```
(671.0412978189149, 457.3761230510776, 564.0304759806827, 44.01405522485165)
```

[47]:
```
ffkwds = dict()
ffkwds['amat']=adjacency_mat
gen, best_f[-1], best_spec = main_ga_loop(500, mypop, 0.5, 0.5, 0.3, sspace,␣
 ↪route_fitness,sel_meth="rank",\
                                        ␣
 ↪mut_meth="swap",co_meth="ordered",ffkwds=ffkwds)
```

```
Generation  0  best specimen  469.1876755782273
Generation  50  best specimen  379.858002077454
Generation  100  best specimen  400.34213082893245
Generation  150  best specimen  391.1580925354729
Generation  200  best specimen  394.5843837901358
Generation  250  best specimen  406.21176788093305
Generation  300  best specimen  401.21666810643694
Generation  350  best specimen  393.46460193104457
Generation  400  best specimen  397.0845070987945
Generation  450  best specimen  375.7356017382654
```

[48]:
```
best_spec[-1], len(set(best_spec[-1]))
```

[48]:
```
(array([14, 16,  9, 10, 13, 17,  7,  4, 15,  0, 12, 18,  3,  5,  1,  8,  2,
         6, 11, 19]),
 20)
```

2.5 Constraints: N Queens problem

TSP is an optimization problem with a **single constrain**: each city must be visited exactly once. We could have formulated it in different ways: - imposing that Salesman Joe comes back to the starting city - having some cities being reachable by some other cities only (a **not** fully connected graph).

Imposing constraints often a necessity in practical problems. Let's see how GAs can solve a problem with multiple constraints. One such example is the Section **??**.

The task is to place n chess queens on a n by n board without any two of them threatening each other. In other words, no two queens can share the same row, same column, or same diagonal.

```
<img src="queen_range.png" width="300"/>
```

The problem has multiple solutions for $n \geq 4$. For a traditional 8×8 chessboard there are 92 solutions, or 12 taking into account simmetry

How much big is the search space? It is **any 8 boxes out of 64** i. e. the **binomial coefficent** $\binom{64}{8}$ or 4,426,165,368 total combinations.
We can simplify this gargantuan number a little bit by saying that each row or column must have just one queen; this equals 8! or

[49]:
```
def fact(n):
    if n <= 0:
        return 1
    return n*fact(n-1)
fact(8)
```

[49]:
```
40320
```

2.5.1 Exercise 4

Write a function to calculate the binomial coefficent

Hint: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

[50]:
```
sp.special.binom(64,8)
```

[50]:
```
4426165368.0
```

2.5.2 Problem representation

We have seen how imposing one queen per row or column greatly simplifies the problem so we should take advantage ot this fact.
Hence a possible representation of a candidate solution is any sequence of the first n integers for a $n \times n$ chessboard.

for a normal chessboard:

[51]:
```
a = list(range(8))
np.random.shuffle(a)
print(a)
```

```
[0, 5, 2, 1, 3, 4, 7, 6]
```

is a candidate solution. How we can check if it is valid?

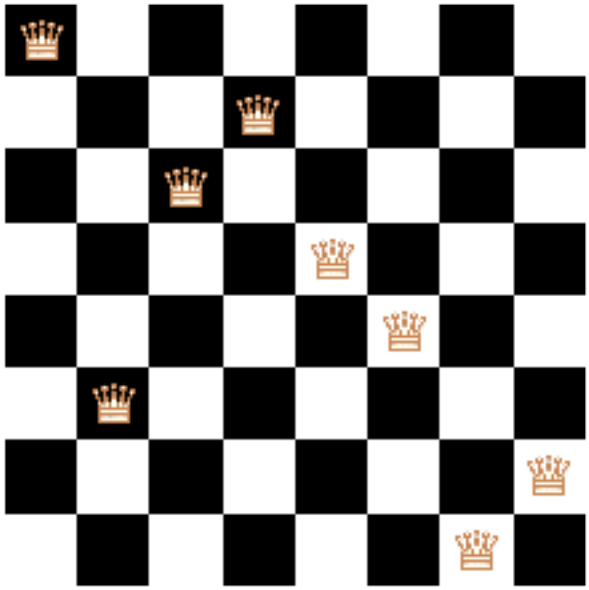
Let's create a function that draws it:

[52]:
```
def draw_chessboard(nrow, ncol, iconcrd):
    chessboard = np.zeros((nrow,ncol))
    chessboard[::2,1::2] = 1
    chessboard[1::2,::2] = 1

    icon = OffsetImage(plt.imread('queen-icon.png'), zoom=.12)
```

```
    fig, ax = plt.subplots()
    plt.imshow(chessboard, cmap='gray')
    for row, col in enumerate(iconcrd):
        ab = AnnotationBbox(icon, (row, col), frameon=False)
        ax.add_artist(ab)
    #plt.figure(figsize=(8,8))
    plt.axis(False)
    plt.show()
```

[53]:
```
draw_chessboard(8,8,a)
```



how many violations do you spot?

From here it is easy to code chromosome (specimen) and population creation

[54]:
```
def queen_chrm(nqueen):
    chromosome = list(range(nqueen))
    np.random.shuffle(chromosome)
    return chromosome

def queen_pop(nchromosomes, nqueen):
    population = list()
    for c in range(nchromosomes):
        population.append(queen_chrm(nqueen))
    population =  np.asarray(population)
    return population
```

What about fitness? Taking inspiration from the plot above we count pair of queens p on the same

diagonal. Any chromosome with $p = 0$ is a valid solution:

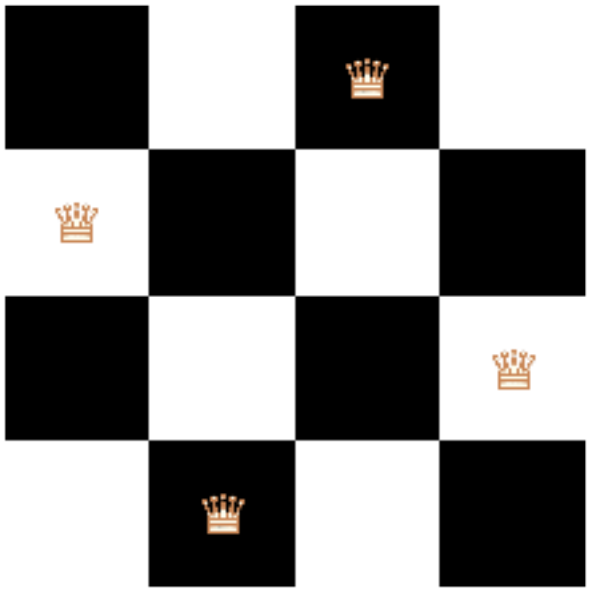
[55]:
```
def violations_fitness(queen_seq):
    L = len(queen_seq)
    violations = 0
    for l0 in range(L-1):
        for l1 in range(l0+1,L):
            d = (l1-l0)
            if queen_seq[l1]==queen_seq[l0]-d or queen_seq[l1]==queen_seq[l0]+d:
                violations += 1
    return violations
```

[56]:
```
a = (1,0,2,3)
b = (1,3,0,2)
```

[57]:
```
violations_fitness(a), violations_fitness(b)
```

[57]: (2, 0)

[58]:
```
draw_chessboard(4,4,b)
```



What about crossover and mutation? The solutions have the same form of the TSP problem, so we could try to use the same operators as well.

2.5.3 8 queens

[59]:
```
mypop = queen_pop(100,8)
print(mypop.shape)
fitness = np.apply_along_axis(violations_fitness, 1, mypop)
np.max(fitness), np.min(fitness), np.mean(fitness), np.std(fitness)
```

```
(100, 8)
```

[59]:
```
(14, 0, 5.38, 2.283768814919759)
```

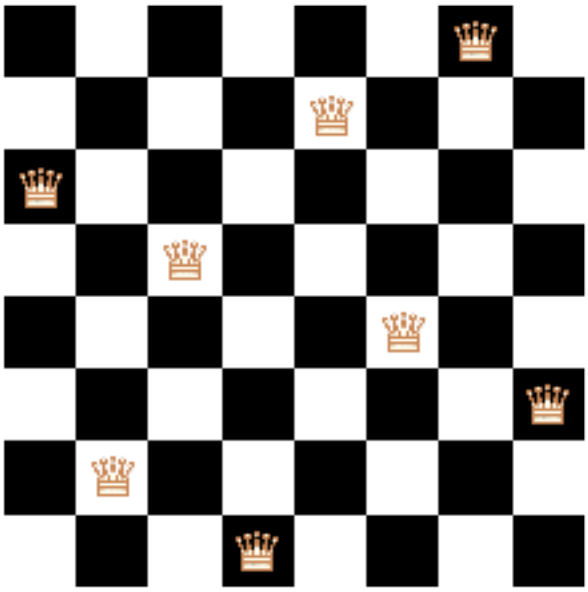
[60]:
```
gen, best_f[-1], best_spec = main_ga_loop(10, mypop, 0.5, 0.5, 0.3, sspace,␣
↪violations_fitness,sel_meth="tournament",\
                                    mut_meth="swap",co_meth="ordered",vstep=1,␣
↪tol=-1)
```

```
Generation  0  best specimen  1
Generation  1  best specimen  0
Generation  2  best specimen  0
Generation  3  best specimen  0
Generation  4  best specimen  0
Generation  5  best specimen  1
Generation  6  best specimen  0
Generation  7  best specimen  0
Generation  8  best specimen  0
Generation  9  best specimen  1
```

[61]:
```
best_spec[-1], violations_fitness(best_spec[-1])
```

[61]:
```
(array([2, 6, 3, 7, 1, 4, 0, 5]), 1)
```

[62]:
```
draw_chessboard(8, 8, best_spec[9])
```



For TSp we swapped just one pair of cities since the route could change too drastically otherwise. For the N queen problem let's try to shuffle a whole chunck.

2.5.4 Exercise 5

Write a mutation method which shuffles a substring of integers of arbitrary length from a random starting point

2.5.5 Solution

[63]:
```
def mutation(prob, mut_meth, population, sspace):
    if mut_meth == "const":
        mutated, population = mutation_const(prob, population, sspace)
    elif mut_meth == "prop":
        mutated, population = mutation_prop(prob, population, sspace)
    elif mut_meth == "swap":
        mutated, population = mutation_swap(prob, population, sspace)
    elif mut_meth == "shuffle":
        mutated, population = mutation_shuffle(prob, population, sspace)
    return mutated, population
```

[64]:
```
def mutation_shuffle(prob, population, sspace):
    tot_genes = population.size
    num_scrambles = math.ceil(prob * tot_genes)
    nspec = num_scrambles//population.shape[1]
    mut_spec = np.random.choice(population.shape[0], size=num_scrambles)
    for ms in mut_spec:
        MS = population[ms]
```

```
        pbreak = np.random.choice(len(MS)-1)
        lbreak = np.random.choice(len(MS)-pbreak)
        np.random.shuffle(MS[pbreak:pbreak+lbreak])
    return mut_spec, population
```

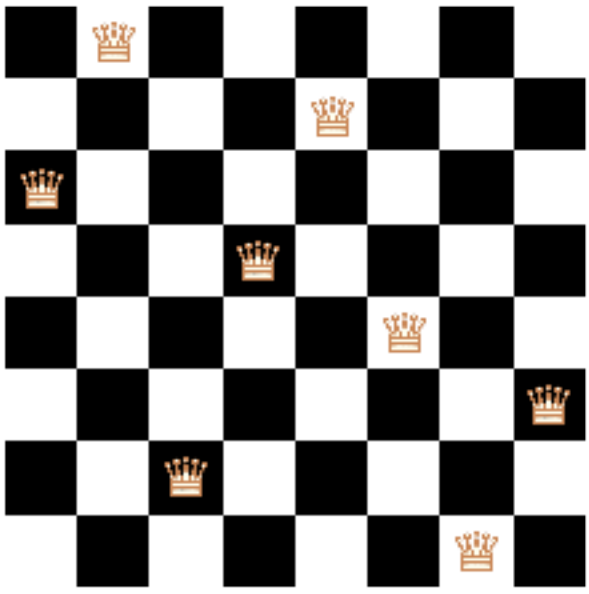
[65]:
```
gen, best_f[-1], best_spec = main_ga_loop(10, mypop, 0.5, 0.6, 0.1, sspace,␣
↪violations_fitness,sel_meth="tournament",\
                                        ␣
↪mut_meth="shuffle",co_meth="ordered",vstep=1, tol=-1)
```

```
Generation  0  best specimen  0
Generation  1  best specimen  0
Generation  2  best specimen  1
Generation  3  best specimen  1
Generation  4  best specimen  1
Generation  5  best specimen  1
Generation  6  best specimen  1
Generation  7  best specimen  1
Generation  8  best specimen  1
Generation  9  best specimen  1
```

[66]:
```
best_spec[-1], violations_fitness(best_spec[-1])
```

[66]:
```
(array([2, 0, 6, 3, 1, 4, 7, 5]), 1)
```

[67]:
```
draw_chessboard(8, 8, best_spec[-1])
```



2.5.6 16 Queens!

This is quite harder. The total number of solutions is

[68]:
```
fact(16)
```

[68]:
```
20922789888000
```

[69]:
```
mypop = queen_pop(300,16)
print(mypop.shape)
fitness = np.apply_along_axis(violations_fitness, 1, mypop)
np.max(fitness), np.min(fitness), np.mean(fitness), np.std(fitness)
```

```
(300, 16)
```

[69]:
```
(20, 4, 10.33, 3.1466013411298226)
```

[73]:
```
gen, best_f, best_spec = main_ga_loop(1000, mypop, 0.6, 0.6, 0.05, sspace,␣
↪violations_fitness,sel_meth="tournament",
                                      ␣
↪mut_meth="shuffle",co_meth="ordered",vstep=100, tol=-1)
```

```
Generation  0  best specimen  4
Generation  100  best specimen  1
Generation  200  best specimen  1
Generation  300  best specimen  1
Generation  400  best specimen  0
Generation  500  best specimen  1
Generation  600  best specimen  1
Generation  700  best specimen  1
Generation  800  best specimen  0
Generation  900  best specimen  1
```

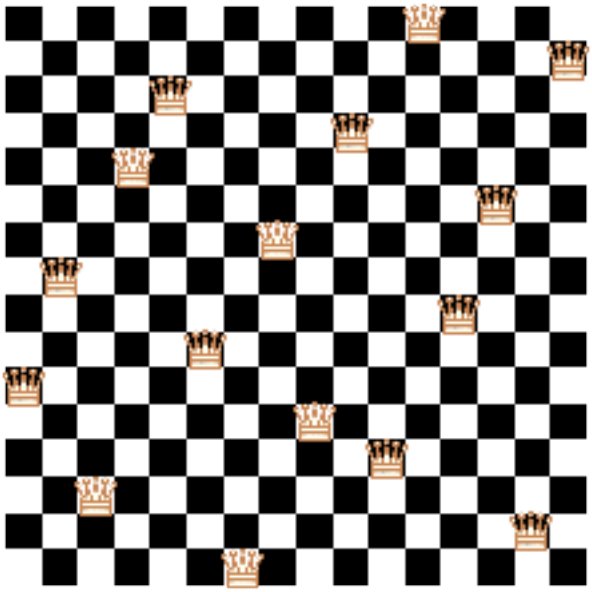
[85]:
```
best_f = np.asarray(best_f)
a = np.where(best_f==0)
a
```

[85]:
```
(array([363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375,
        376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388,
        389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 406,
        425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437,
        438, 439, 440, 441, 442, 443, 444, 445, 775, 776, 777, 778, 779,
        780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792,
        793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805,
        806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818,
        819, 823, 824, 825, 826, 827, 828, 829, 830, 833, 834, 835, 836,
        837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849,
        850, 851, 852, 853, 854, 855, 911]),)
```

[88]:
```
best_spec[363]
```

[88]:
```
array([10,  7, 13,  4,  2,  9, 15,  6, 11,  3, 12,  0,  8,  5, 14,  1])
```

[92]:
```
try:
    draw_chessboard(16, 16, best_spec[a[0][0]])
except:
    pass
```



2.6 The End!