Evolutionary Computation Homework 2

Author: Ben Kovach

N Queens

- Representation: Permutation of the set $\{0..N-1\}$. Each element indicates the row of the queen at index i (i corresponds to the queen's column).
- Parenthood Selection: Tournament selection twice with three combatants.
- Mutation: Swap two elements of an individual's vector
- Crossover: Order crossover
- Survival Selection: Replace the $\frac{N}{3}$ worst individuals with $\frac{N}{3}$ new ones

Code

```
import random
import sys

# order crossover
def crossover(i1, i2):
    n = len(i1)
    r1 = random.randrange(n-1)
    r2 = random.randrange(r1, n)
    child1 = [None] * n
    child2 = [None] * n
    child1[r1:r2] = i1[r1:r2]
    child2[r1:r2] = i2[r1:r2]
    rest1 = filter(lambda e: e not in i1[r1:r2], i2)
    rest2 = filter(lambda e: e not in i2[r1:r2], i1)
    for i in xrange(len(rest1)):
        child1[(r2 + i) % n] = rest1[i]
    for i in xrange(len(rest2)):
        child2[(r2 + i) % n] = rest2[i]
    return child1, child2

# tourament selection
def select(xs, n=3):
    selection = []
    for i in xrange(0, n):
        selection.append(xs[random.randrange(len(xs))])
    fixed = zip(selection, map(lambda path: unfitness(path), selection))
    mini = None
    minind = None
    for ind, fit in fixed:
        if (fit < mini or mini is None):
            mini   = fit
            minind = ind
    return (mini, minind)

# define "unfitness" as the number of collisions between queens
def unfitness(queens):
    colls = 0
    for i, q in enumerate(queens):
        for j, p in enumerate(queens[i+1:]):
            _j  = j + 1
            if (q + _j == p or q - _j == p):
                colls += 1
    return colls

def generate_child(pop, mutate_prob=0.1):
    fit1, p1 = select(pop)
    fit2, p2 = select(pop)
    c1, c2 = crossover(p1, p2)
    # return the child with greater fitness
    if (unfitness(c1) < unfitness(c2)):
        if(c2 in pop):
            return generate_child(pop, mutate_prob)
        if(random.random() < mutate_prob):
            return mutate(c2)
        else:
            return c2
    else:
        if(c1 in pop):
            return generate_child(pop, mutate_prob)
        if(random.random() < mutate_prob):
            return mutate(c1)
        else:
            return c1

# swap two random indices
def mutate(ind):
    return ind
    i1 = random.randrange(len(ind))
    i2 = random.randrange(len(ind))
    ind[i1], ind[i2] = ind[i2], ind[i1]
    return ind

# random permutation of 0..n-1
def individual(n):
    return random.sample(range(n), n)
```

```
def population(n, ind_size):
    return [individual(ind_size) for _ in xrange(n)]

def sort_pop(pop):
    fits = map(unfitness, pop)
    inds = zip(fits, pop)
    return sorted(inds, key=lambda ind: ind[0])

def replace_worst(pop, n=1, mut_prob=0.1):
    keepers = sort_pop(pop)[:-n]
    _pop = map(lambda ind: ind[1], keepers)
    for i in xrange(0, n):
        _pop.append(generate_child(pop, mut_prob))
    return keepers[0][0], _pop

# show a chess board
def show(vs, n):
    vs = zip(xrange(0,len(vs)), vs)
    for x in range(0, n):
        for y in range(0, n):
            if (x, y) in vs:
                sys.stdout.write('Q')
            else:
                sys.stdout.write('-')
        print ""

def nqueens(n):
    POPSIZE = n
    GENERATIONS = n * 1000
    MUTPROB = 1.0/n
    REPLACEMENTS = n/3

    pop = population(POPSIZE, n)

    soln = False

    for g in xrange(GENERATIONS):
        mini, pop = replace_worst(pop, n=REPLACEMENTS, mut_prob=MUTPROB)
        if mini == 0:
            soln = True
            print("Solution for n={0}".format(n))
            show(pop[0], n)
            break
    if(not soln):
        print("Solution for n = {0} not found in {1} generations.".format(n, g))
```

```
for i in xrange(8, 20):
    nqueens(i)

"""
Solution for n=8
--Q-----
----Q---
-Q------
-------Q
-----Q--
---Q----
------Q-
Q-------
Solution for n = 9 not found in 8999 generations.
Solution for n=10
----Q-----
-------Q--
---Q------
------Q---
---------Q
--Q-------
Q---------
-----Q----
--------Q-
-Q--------
Solution for n = 11 not found in 10999 generations.
Solution for n=12
----Q-------
--------Q---
-Q----------
---------Q--
--Q---------
Q-----------
-------Q----
---Q--------
-----------Q
------Q-----
----------Q-
-----Q------
Solution for n = 13 not found in 12999 generations.
Solution for n=14
-----Q--------
--Q-----------
-----------Q--
---Q----------
```

```
--------Q-----
-------------Q
-Q------------
----------Q---
-------Q------
Q-------------
------------Q-
---------Q----
----Q---------
------Q-------
Solution for n=15
----------Q----
-------------Q-
----Q----------
------------Q--
-------Q-------
Q--------------
--Q------------
------Q--------
-Q-------------
--------------Q
--------Q------
-----------Q---
---------Q-----
---Q-----------
-----Q---------
Solution for n = 16 not found in 15999 generations.
Solution for n = 17 not found in 16999 generations.
Solution for n=18
------------Q-----
----------Q-------
-----------------Q
-------Q----------
-Q----------------
---Q--------------
--------Q---------
------Q-----------
---------------Q--
Q-----------------
-----------Q------
----------------Q-
----Q-------------
--Q---------------
--------------Q---
-----Q------------
---------Q--------
```

```
-------------Q----
Solution for n = 19 not found in 18999 generations.
"""
```

Traveling Salesman

- Representation: Permutation of {0..n-1}
- Parenthood Selection: Ternary tournament (3 individuals)
- Mutation: Random swap of two indices
- Crossover: Order crossover
- Survival Selection: Replace single worst entity with new one
- Fitness: Length of the cycle $I_0, I_1, .., I_n, I_0$ for an individual I (minimize)
- Shortest cycle distance found: about 174081.47

Code:

```
from math import *
import random

"""
Some test results:

POPSIZE = 10
GENERATIONS = 10000
MUTPROB = 0.1

Shortest distance = 174081.470967

POPSIZE = 20
GENERATIONS = 5000
MUTPROB = 0.1

Shortest distance = 288226.47026
"""

# want to minimize this
def follow_path(points, indices):
    sm = 0
    for (i, j) in zip(indices, indices[1:] + [indices[0]]):
        p1 = points[i]
        p2 = points[j]
        sm += dist(p1, p2)
    return sm
```

```
# order crossover
def crossover(i1, i2):
    n = len(i1)
    r1 = random.randrange(n-1)
    r2 = random.randrange(r1, n)
    child1 = [None] * n
    child2 = [None] * n
    child1[r1:r2] = i1[r1:r2]
    child2[r1:r2] = i2[r1:r2]
    rest1 = filter(lambda e: e not in i1[r1:r2], i2)
    rest2 = filter(lambda e: e not in i2[r1:r2], i1)
    for i in xrange(len(rest1)):
        child1[(r2 + i) % n] = rest1[i]
    for i in xrange(len(rest2)):
        child2[(r2 + i) % n] = rest2[i]
    return child1, child2

# swap a random one
def mutate(ind):
    return ind
    i1 = random.randrange(len(ind))
    i2 = random.randrange(len(ind))
    ind[i1], ind[i2] = ind[i2], ind[i1]
    return ind

def dist(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return sqrt((x1 - x2)**2 + (y1-y2)**2)

# tourament selection
def select(xs, n=3):
    selection = []
    for i in xrange(0, n):
        selection.append(xs[random.randrange(len(xs))])
    fixed = zip(selection, map(lambda path: unfitness(path), selection))
    mini = None
    minind = None
    for ind, fit in fixed:
        if (fit < mini or mini is None):
            mini   = fit
            minind = ind
    return (mini, minind)

def is_valid(child):
    try:
        c = child.index(None)
        return False
    except ValueError:
        return True

def generate_child(pop, mutate_prob=0.1):
    fit1, p1 = select(pop)
    fit2, p2 = select(pop)
    c1, c2 = crossover(p1, p2)
    while (not is_valid(c1) or not is_valid(c2)) :
        fit1, p1 = select(pop)
        fit2, p2 = select(pop)
        c1, c2 = crossover(p1, p2)
    # return the child with greater fitness
    if (unfitness(c1) < unfitness(c2)):
        if(random.random() < mutate_prob):
            return mutate(c2)
        else:
            return c2
    else:
        if(random.random() < mutate_prob):
            return mutate(c1)
        else:
            return c1

def unfitness(ind):
    return follow_path(points, ind)

def individual(n):
    # random permutation of 0..n-1
    return random.sample(range(n), n)

def population(n, ind_size):
    return [individual(ind_size) for _ in xrange(n)]

def unfitnesses(pop):
    return map(unfitness, pop)

def sort_pop(pop):
    fits = unfitnesses(pop)
    inds = zip(fits, pop)
    return sorted(inds, key=lambda ind: ind[0])

def replace_worst(pop, n=1, mut_prob=0.1):
    keepers = sort_pop(pop)[:-n]
    _pop = map(lambda ind: ind[1], keepers)
    for i in xrange(0, n):
        _pop.append(generate_child(pop, mut_prob))
    return keepers[0][0], _pop

def read_points():
    f = open('in.txt')
    lns = f.readlines()
    return map(tuple, map(lambda ln: map(int, ln.split()), lns))

points = read_points()
ind_size = len(points)

N = len(points)
POPSIZE = 10
GENERATIONS = 50000
MUTPROB = 0.1

pop = population(POPSIZE, N)

for g in xrange(GENERATIONS):
    mini, pop = replace_worst(pop, mut_prob=MUTPROB)
    if(g % 100 == 0):
        print ("Evaluating generation " + str(g))
        print mini
```