

Particle Swarm Optimization

Course: AI and Optimization, ADEO-M2

Author: Alvise de' Faveri Tron

Exercise 1.

a. Add Rosenbrock and Schwefel functions.

```
##
## Rosenbrock function
##
def rosenbrock(sol):
    sum = 0

    for i in range(0, len(sol) - 1):
        sum += 100*((sol[i+1] - (sol[i]**2))**2) + (sol[i] - 1)**2

    return sum
```

```
##
## Schwefel function
##
def schwefel(sol):
    d = len(sol)

    sum = 0

    for i in range(0, d):
        sum += sol[i]*sin(sqrt(abs(sol[i])))

    tot = 418.9829*d - sum

    return tot
```

b. Write a limiting function that manage the problems at the boundaries of the search space when moving. Choose one or more strategies.

We define the `limit` function as follows:

```
## Define INF and SUP as global variables
INF = -600
SUP = 600

# Change INF and SUP for schwefel and rosenbrock
if FUNCTION == "schwefel":
    INF = -500
    SUP = 500
elif FUNCTION == "rosenbrock":
```

```

INF = -5
SUP = 10

# If a coordinate is out of the range, position it on the border
def limit(pos):
    global INF, SUP
    if pos < INF:
        pos = INF
    elif pos > SUP:
        pos = SUP

    return pos

```

We can then use the `limit` function in the `move` function:

```

# Calculate the velocity and move a paticule
def move(particle,dim):
    # ... Calculate velocity as before ...

    # Limit the new position
    for i in range(dim):
        position[i] = limit(particle["pos"][i] + velocity[i])

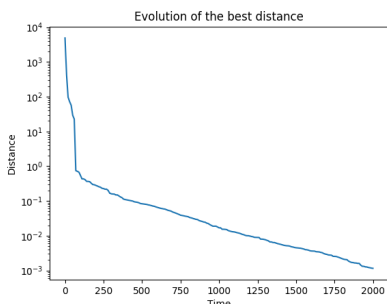
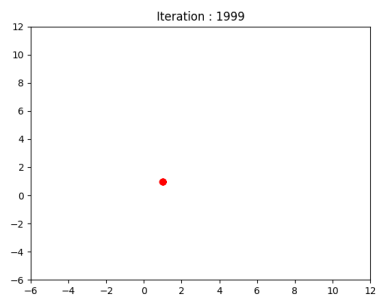
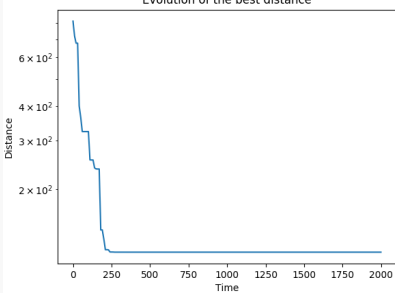
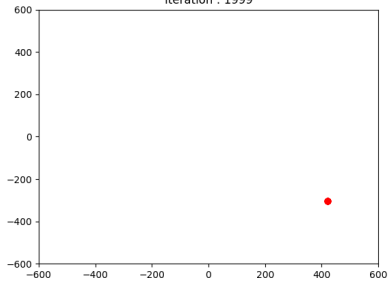
    # ... as before

    return nv

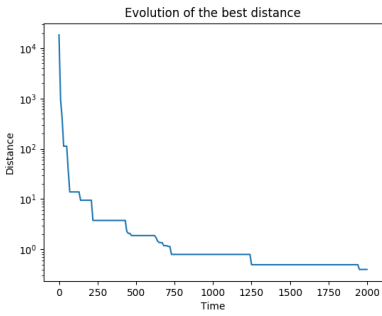
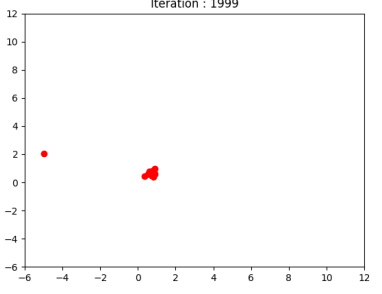
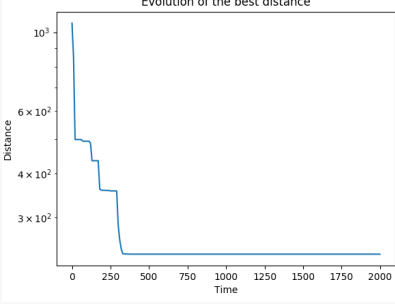
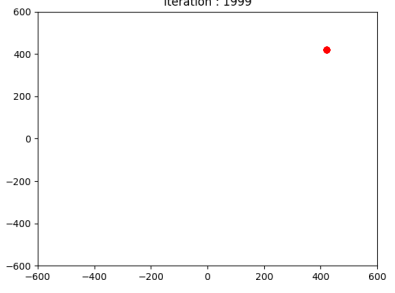
```

c. Test the algorithm with its different parameter values.

The default parameters are $\psi = 0.7$, $c_{max} = 1.47$

Function	Statistics	Output
Rosenbrock		
Schwefel		

The other commented parameters are $\psi = 0.8$, $c_{max} = 1.62$

Function	Statistics	Output
Rosenbrock		
Schwefel		

d. To simplify its parametrization, try to find a suitable relation between the number of particles, the number of cycles and the dimension.

Apparently a good and stable relation can be

- $Nb_cycles = 350 \times DIM$
- $Nb_particle = 20 \times DIM$

For example, for $DIM = 5$ the Schwefel minimum was found at:

```
point = [420.96874671021965, 420.9687459594801, 420.9687461926004,
420.9687469173869, 420.96874737218695]
```

```
eval = 6.363783086271724e-05
```

e. The equation of velocity takes into account the global neighborhood (all particles are informants and the best individual for a particle is the best individual of the swarm). Write the function `localUpdate` that allows to select `nbn` neighbors as informers. To simplify, we will choose informants close to the particle according to their position in the list, it is a social neighborhood.

```
##
## Return the nbn nearest particles in the swarm list
##
def getNeighborhood(particle, swarm, nbn):
    neighbors = []
    tot = (int)(nbn/2)
    initIndex = swarm.index(particle)

    # get successors
    for i in range(initIndex+1, initIndex + tot + 1):
        if i < len(swarm):
```

```

        neighbors.append(swarm[i])
        tot -= 1
    else:
        break

    # get predecessors
    for i in range(initIndex-tot-1, initIndex):
        if i >= 0:
            neighbors.append(swarm[i])
            tot -= 1
        else:
            break

    return neighbors

##
## Update information for the particles
##
def localUpdate(particle, swarm, nbn):
    # get best between neighbors instead of in the whole swarm
    bestParticle = getBest(getNeighborhood(particle, swarm, nbn))

    # update
    nv = dict(particle)
    if(particle["fit"] < particle["bestfit"]):
        nv['bestpos'] = particle["pos"][:]
        nv['bestfit'] = particle["fit"]
    nv['bestvois'] = bestParticle["bestpos"][:]
    return nv

```

We then substitute the update function with `swarm = [localUpdate(e,swarm,nbn) for e in swarm]` in the main loop.

This solution causes the swarm to divide in subgroups which get attracted to different local minimums, but the overall result is consistently better because the particles which find the minimums are less likely to get attracted in other point by the neighbors.

For example, keeping the same settings as in **1.d** and using NBN=10 with 10 dimensions we get

```

point = [420.9687463810648, -302.5249364242619, -500.0, 420.96874678520106,
420.9687468735569, 420.9687469613984, 420.9687458935466, -302.52493600723886,
-302.5249374186065, 420.9687459457745]

eval = 593.7088598600194

```

Exercise 2.

For this exercise, I took the inspiration from the `SATSP34` file of the first lesson, in particular:

- `draw`
- `parse`
- `fluctuationTwo`
- `energyTotale`

a. velocity + velocity = velocity : sum of two speeds is the concatenation of permutations. Write the function add.

Defining a permutation as a couple of indexes (p1, p2), we can say that the velocity is a list of permutations. We have:

```
def add(v1, v2):
    # exclude null speed
    if len(v2) == 0:
        return v1
    if len(v1) == 0:
        return v2

    v = v1.copy()
    v.extend(v2)
    return v
```

b. coefficient * velocity = velocity : We return k% of the permutations of the velocity. If k > 1, we add to the result all the permutations of the initial velocity, we subtract 1 to k and we start again. Write such a function times.

```
def times(v1, coeff):
    if len(v1) == 0:
        return []

    # append v1 to itself as many times as needed
    v = v1.copy()
    n = (int)(coeff*len(v1))

    # return first n values
    while len(v) < n:
        v.extend(v1)

    return v[:n]
```

c. particle - particle = velocity : The distance between two particles is the set of permutations allowing to move from one list to another. Write the function minus which returns a velocity as defined.

```
def minus(p1, p2):
    # define p as p1
    p = p1.copy()
    permlist = []

    # repeat until p has become p2
    while(p1 != p2):
        # check each particle of p and p2
        for i in range(len(p)):
            # when you find two different particles, try to make
            # p similar to p2
            if(p[i] != p2[i]):
                indexToPermute = p.index(p2[i])
                # add the permutation to the list
                permlist.append([i, indexToPermute])
                # perform the permutation
```

```

        permuteTwo(p, i, indexToPermute)

    return permlist

```

d. Deduce that in the program, the equations of velocity and displacement.

```

# Calculate the velocity and move a particule
def move(particle,dim):
    global ksi,c1,c2,psi,cmax

    nv = dict(particle)

    # calculate velocity
    velocity = []

    bp = minus(particle["bestpos"], particle["pos"])
    bv = minus(particle["bestvois"], particle["pos"])

    rbp = times(bp, cmax*random.uniform())
    rbv = times(bv, cmax*random.uniform())
    ps = times(particle["vit"],psi)

    a = add(ps, rbp)
    a = add(a, rbv)

    velocity = a

    # move the particle, i.e. perform all the permutations of the velocity
    position = particle["pos"]
    for t in velocity:
        print(t)
        position = fluctuationTwo(position, t[0], t[1])

    # update the new particle
    nv['vit'] = velocity
    nv['pos'] = position
    nv['fit'] = energyTotale(coords, position)

    return nv

```

e. With the file 14.tsp, test the algorithm with global and local neighborhoods

The only thing left to do is the initialization: we cannot initialize all the particle in the same position and null velocity. In this case, I chose to initialize every particle with a random path:

```

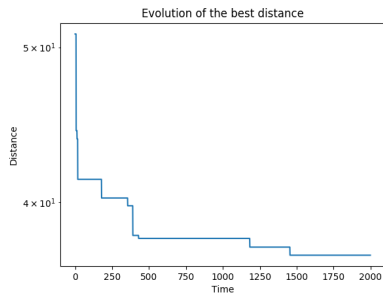
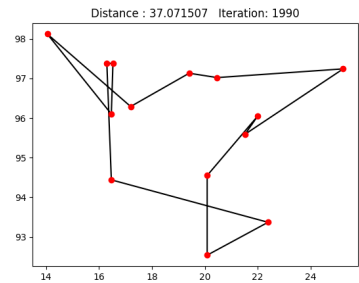
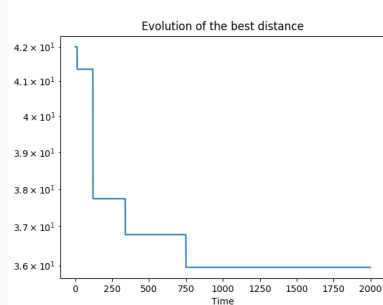
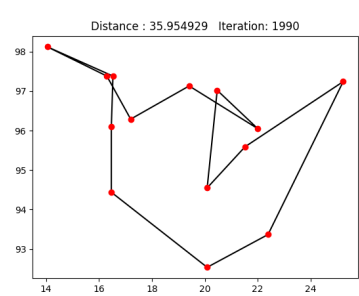
##
## Init a particle
##
def initOne():
    pos = [i for i in range(N)]

    random.shuffle(pos)

    fit = energyTotale(coords,pos)
    return {'vit':[(0,0)], 'pos':pos, 'fit':fit, 'bestpos':pos, 'bestfit':fit,
'bestvois':[]}]

```

The following results were obtained after 2000 iterations with 40 particles, using `psi, cmax = (0.8, 1.62)`.

Update Type	Statistics	Output
Global	 <p>Evolution of the best distance</p>	 <p>Distance : 37.071507 Iteration: 1990</p>
Social Neighborhood NBN=8	 <p>Evolution of the best distance</p>	 <p>Distance : 35.954929 Iteration: 1990</p>