Solving TSP with Simulated Annealing

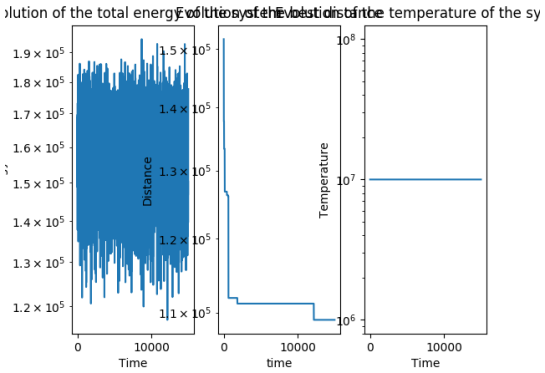
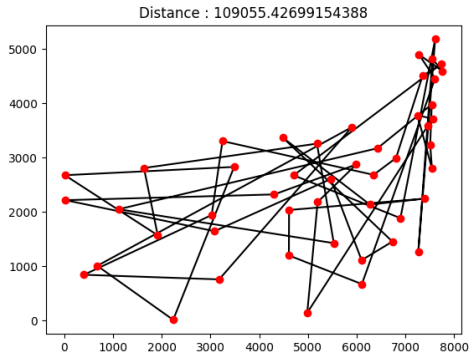
Course: AI and Optimization, ADEO-M2
Author: Alvise de' Faveri Tron

Problem 1

Exercise 1: Understanding the impact of settings

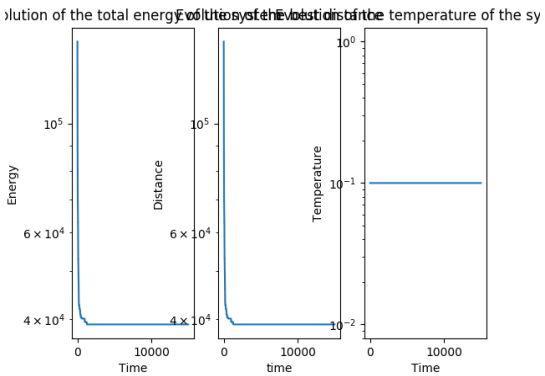
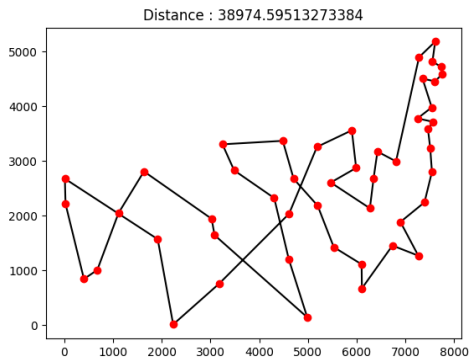
a. Temperature = 10^7

A high temperature means that the algorithm will accept variations to the current solution more easily, even if they are worse in terms of distance. This enables a better exploration of the solution space, but leads to a slower convergence to a solution: in this case, for T= 10^7 the final solution was quite bad.

performance statistics	final solution
	

b. Temperature = 10^{-1}

A low temperature means that the algorithm will reject most of the degrading solutions at each step, converging more rapidly to a local minimum . As we can see, in the case of T=0.1 the energy graph on the left is far smoother than the previous one, and it is strictly descending.

performance statistics	final solution
	

c. Threshold temperature value

The threshold value for the changing of this behavior appears to be near to 1: with T > 1 the algorithm will sometimes accept solutions that are not better than the current one, while with T < 1 the algorithm changes solution only when it finds a better one among its neighbors.

Exercise 2: Performance Measures

a. Display the number of improvements

Results for $T = 10$

```
n_changes: 104760
changes/evaluations: 0.9977142857142857
```

Results for $T = 10^4$

```
n_changes: 1705
changes/evaluations: 0.01623809523809524
```

b. Performance Criteria of the Algorithm

A good performance indicator for the algorithm can be calculated by considering total length of the final solution and the number of steps it took: if an algorithm can produce a shorter solution in the same number of steps or fewer, it is better.

c. Correctness and Stability

The **correctness** can be verified by checking the solution file, which contains the path selected by the algorithm. The path must be a closed Hamiltonian path, i.e. a closed path that touches each point of the graph exactly once.

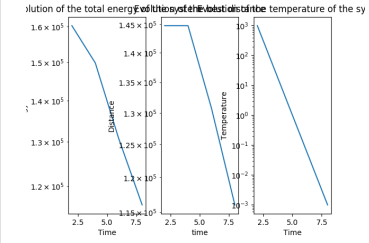
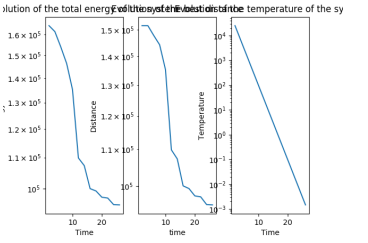
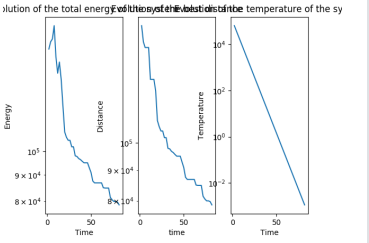
The **stability** of the algorithm can be evaluated by running the algorithm many times with the same graph and looking at how similar the results are at the end.

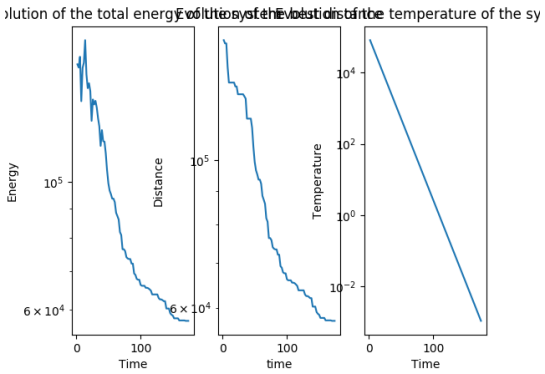
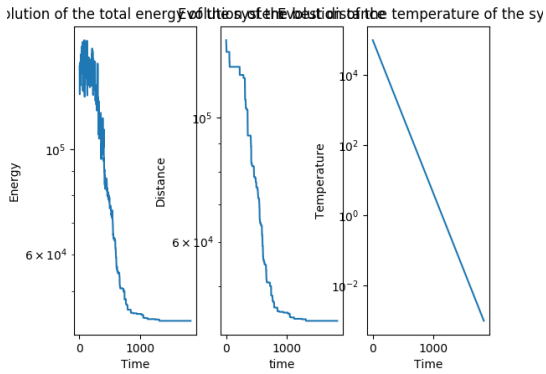
Exercise 3: Temperature

a. and b.: see exercise 1.

c. Set a growth value to 0.1, 0.5, 0.8, 0.9 and 0.99, interpret the behavior.

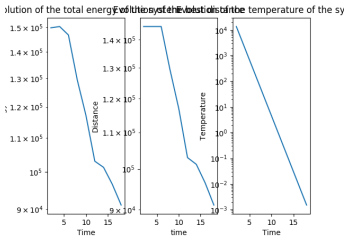
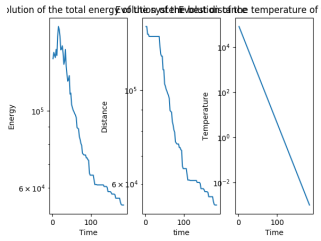
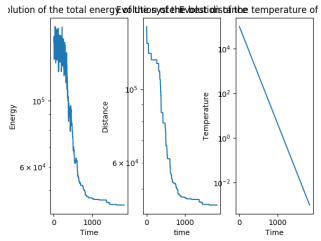
Keeping `10e4` as initial temperature value, we now decrease the temperature of a factor `Alpha` each time. This is the equivalent of *cooling*: the algorithm starts becoming more strict in further iterations and converges more quickly to a solution. The higher is Alpha, the more rapid is the cooling.

0.1	0.5	0.8
		

0.9	0.99
	

d. By varying its value, study the impact of the temperature plateau on the result of the algorithm.

Keeping `10e4` as initial temperature value, we now decrease the temperature of a factor $T_0 * e^{-\frac{t}{\tau}}$ each time. This is another way of simulating the *cooling*: this time the cooling is not linear but exponential.

$\tau = 1$	$\tau = 10$	$\tau = 100$
		

e. Study the difference between the two equations of temperature.

The first equation is linear `T = T*Alpha`.

The second equation is en exponential curve `T = T0*exp(-t/tau)`: is starts descending rapidly but near 0 the temperature starts decreasing less and less rapidly.

Exercise 4: Initialization and Neighborhood

a. Is it wise to initialize the solution as realized? What is the impact of an initialization at random?

Random initialization is not always wise, because it can affect the final result of the algorithm if for example the initialization happens to be near a local minimum but far from a global minimum.

b. The coded disturbance makes a modification of 4 edges. Perform a function that disrupts the solution of only 3 edges.

```
def fluctuationTree(path,i,j,k):
    nv = path[:]
    temp = nv[i]
    nv[i] = nv[j]
    nv[j] = nv[k]
    nv[j] = temp
    return nv
```

c. The 2-opt permutation realizes a disturbance that modifies only 2 edges. Test this permutation.

```
def fluctuationTwo(path,i,j):
    nv = path[:]
    temp = nv[i]
    nv[i] = nv[j]
    nv[j] = temp
    return nv
```

Exercise 5: Convergence

a. Create a convergence criterion that stops (stops) the temperature step when there have been a number of iterations without improvement.

We create a global variable `n_not_improved`. Then in the `metropolis` function we have:

```
if delta <= 0: # if improving
    n_not_improved = 0
    ...
else:
    n_not_improved += 1
    ...
```

Then , in the main loop, we check `n_not_improved` before changing the current temperature.

```
for i in range(IterMax):
# Convergence loop on temperature criterion
    while T> Tmin and iterStep > 0:
    #...
    if n_not_improved < MAX_NOT_IMPROVED:
            T = T0*exp(-t/tau)
```

b. Create a convergence criterion that stops the algorithm when there are a number of iterations without improvement.

Same as before, but in the main loop if `n_not_improved` is too big we stop.

```
for i in range(IterMax):
# Convergence loop on temperature criterion
    while T> Tmin and iterStep > 0:
    #...
    if n_not_improved >= MAX_NOT_IMPROVED:
            break
```

Problem 2.

Exercise 6: Continuous Case

The objective is to adapt the algorithm to solve the problem of minimization on Sphere and Griewank functions on the interval [−600; 600].

a. Write both functions sphere and griewank .

```
def sphere(vett):
    tot = 0
    for i in range(0,len(vett)):
        tot =+ vett[i]**2

    return tot
```

```
def grienwank(vett):
    prod = 1
    tot = 0
    for i in range(0, len(vett)):
        tot += (vett[i]**2)/4000
        prod *= cos(vett[i]/sqrt(i))

    return tot - prod + 1
```

b. Write the function init which returns a point initialized at random in the search space.

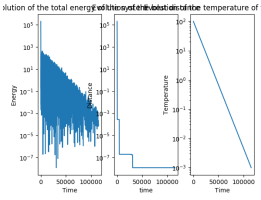
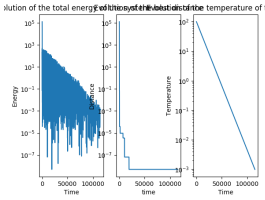
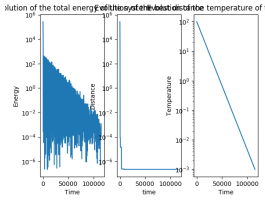
```
def init():
    return uniform(-600, 600)
```

c. Rewrite function fluctuation who returns a neighbor from the current point. We will define a constant STEP , offset.

```
def newFluct(vett, STEP):
    neighborVett = initvett.copy()
    for i in range(0, len(vett))
        neighborVett[i] = initvett[i] + uniform(-STEP, STEP)

    return neighborVett
```

d. Launch the algorithm for each function in dimensions 4, 10 and 50 . Adapt the parameters to the convergence requirements.

	N = 4	N= 10	N = 50
Sphere			
Grienwalk