IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Francesca Sogaro and Peter J. Schmid

February 15, 2017

1 COMBINATORIAL OPTIMIZATION (ANT-COLONY ALGORITHM)

The pure ant-colony optimization (ACO) algorithm applied to the traveling salesman problem (TSP) often leads to a solution that shows path crossings. Even though this solution is not optimal, the algorithm does not seem to "untangle" the path into a path of smaller length.

In this exercise, we want to design and implement an additional feature into the ACO-algorithm to check for non-optimal path crossings and force the "untangling" of these situations. This is referred to as a local-search procedure. In essence, we are looking for configurations where two separated segments of the current path cross each other (see figure 1.1a for an illustration). In this configuration, the path segment $(a-b)$ is crossing the path-segment $(c-d)$. To untangle the path we need to connect node a to node c and node b to node d. When performing this operation, we need to realize that we also have to reverse the original order of the path from b to c, since – after the operation – c is now connected to b, not the other way around. The complete swap is given in the two boxes in figure 1.1. In the literature, this operation is referred to as the 2-opt operation.
In an implementation of this operation, we need to search for a situation where path segments cross. This is best accomplished by computing the lengths $L_{1,\ldots,4}$ as illustrated in figure 1.2. If $L_3 + L_4$ is less than $L_1 + L_2$ we reduce the overall length of the path by untangling the crossing, according to the procedure above.

1.1 EXERCISE 1

Design and implement a `python`-function that searches a given path for crossing path-segments; once a path-crossing has been found, the same function will untangle the path

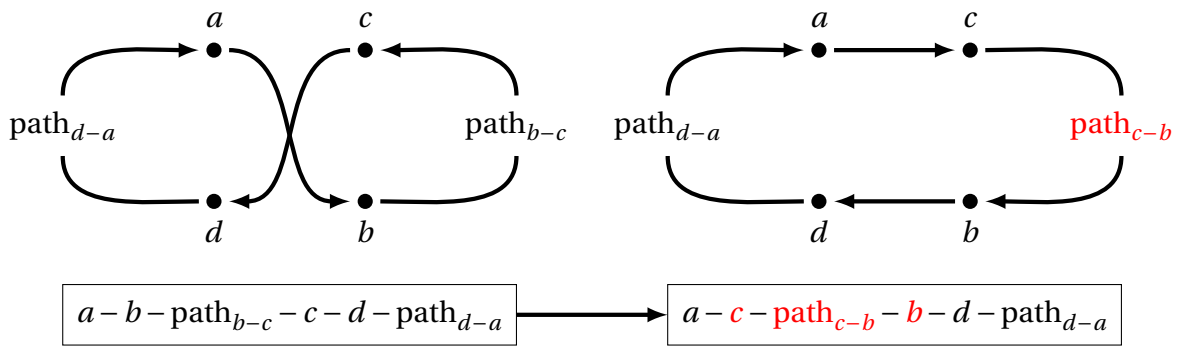

Figure 1.1: Sketch of the 2-opt procedure for "untangling" path-crossings. (a) Original path with two crossing segments $(a-b)$ and $(c-d)$. (b) Corrected path after applying the 2-opt procedure. Note that the path segment $(b-c)$ had to be reversed to $(c-b)$ (in red). The boxes below the paths give the full path-reorganization procedure.

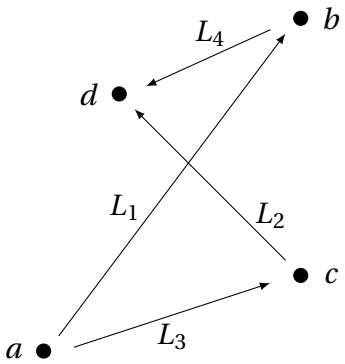

Figure 1.2: Generic path-segment crossing for triggering the 2-opt procedure.

according to the rule above (see boxes in figure 1.1). Apply your function to a random collection of points, using the main code below; also, use the various functions from the tutorial (for calculating the path length or for plotting the final result).

```
1  import numpy as np
2  import scipy as sp
3  import random
4  import math as ma
5  import matplotlib.pyplot as plt
6  import sys
7
8
9  def twoOpt(dist,path):
```

```
10
11      # input:  path  (potentially tangled path)
12      #         dist  (distance matrix)
13      # output: path  (untangled path)
14
15      # add your code here !!
16
17      return path
18
19  if __name__ == '__main__':
20      #-------------------------------------
21      # local search (2-opt) optimization
22      # for traveling salesman problem
23      #-------------------------------------
24
25      nCities = 35   # number of cities
26      maxDist = 100  # maximum distance between cities
27
28      # set city coordinates (randomly)
29      X    = np.random.uniform(0,maxDist,nCities)
30      Y    = np.random.uniform(0,maxDist,nCities)
31      # initial path (0,1,2,3,4,5,...)
32      path = range(nCities)
33      DrawPath(path,X,Y)
34
35      # compute distance between random cities
36      dist = CalcDist(X,Y)
37
38      # main loop (do 100 iterations of 2-opt)
39      for k in range(100):
40          # call your function here
41          path = twoOpt(dist,path)
42      pLen = CalcPathLen(dist,path)
43      print "length of path = ",pLen
44      DrawPath(path,X,Y)
```

1.2 EXERCISE 2

Using the 2-opt technique for the traveling salesman problem does not result in an optimal solution, since it only applies a *local* optimality condition. It does, however, help the convergence of the ACO-algorithm. To show this, take a 2-opt step at the end of each ACO-iteration, before continuing with your ants. Apply this hybrid technique (ACO and 2-opt) to solve the traveling salesman problem for a collection of European capitals encoded in

the function `getCities`. Make the appropriate modifications to the `optAnts`-function to implement the 2-opt-untangling. You should get results similar to figure 1.3 (with 22 cites) or figure 1.4 (for all 45 cities).

```
1  import numpy as np
2  import scipy as sp
3  import math as ma
4  import matplotlib.pyplot as plt
5
6  if __name__ == '__main__':
7
8      nc              = 22 # number of cities considered (max: 45)
9      x,y,D,sc        = getCities(nc) # construct model
10     maxit           = 250  # maximum number of iterations
11     nAnts           = 100  # number of ants (population size)
12     bpath,BestCost = optAnts(nc,x,y,D,maxit,nAnts)
13     for i in bpath:
14         print sc[i],' > ',
15     drawPath(bpath,x,y)
16
17     plt.plot(BestCost)
18     plt.show()
```

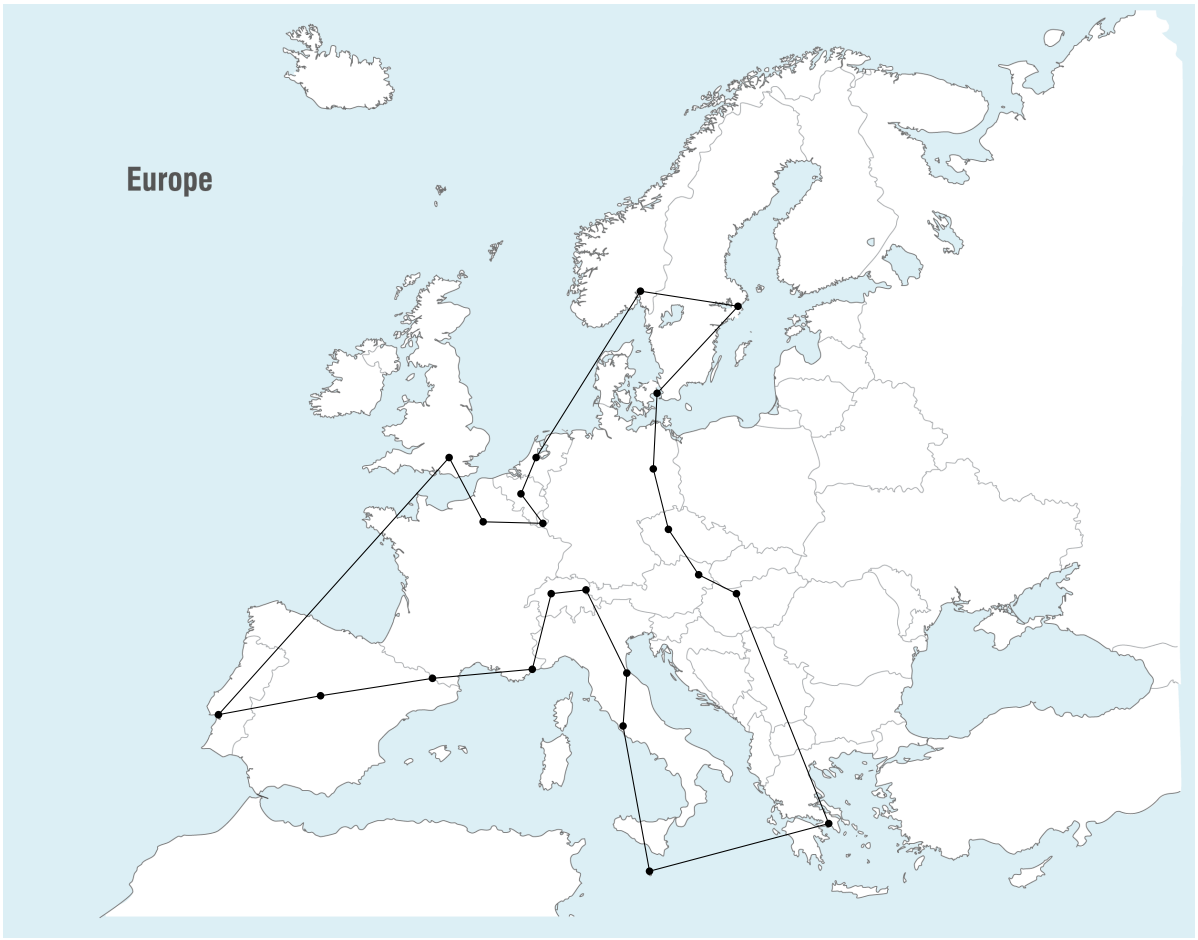

Figure 1.3: “Optimal” path through 22 European capital cities. The path length of this tour is 10863.29 kilometers.

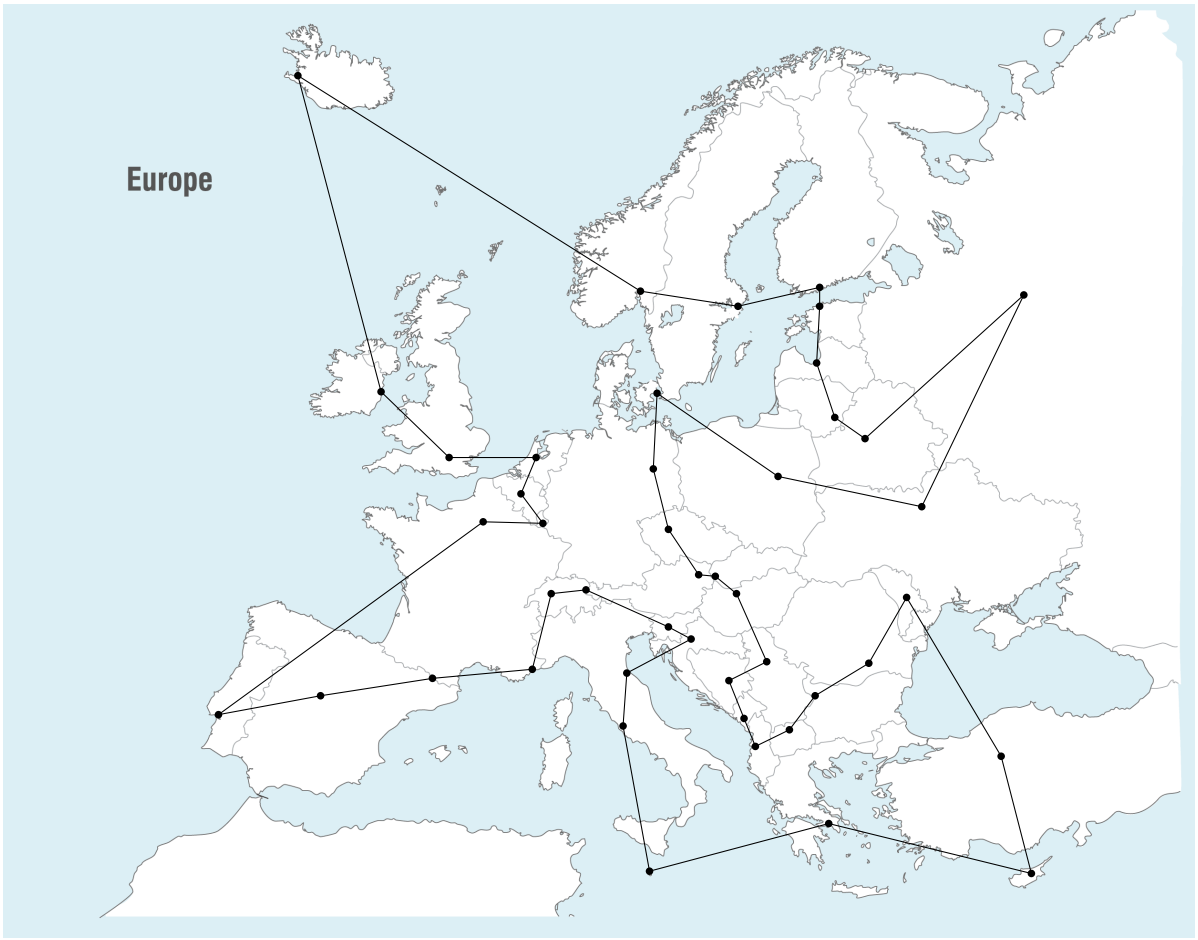

Figure 1.4: “Optimal” path through 45 European capital cities. The path length of this tour is 20402.77 kilometers.