IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

February 20, 2017

1 COMBINATORIAL OPTIMIZATION (ANT-COLONY ALGORITHM)

Combinatorial optimization is concerned with finding an optimal solution from a large but finite set of possibilities. These possibilities represent combinations of subsets drawn from the larger set. Some common problems of combinatorial optimization are the traveling salesman problem (TSP), where a finite set of cities have to be visited only once on a circular path of minimal length, or the minimum spanning tree problem, where all points of a finite set of points are connected by an edge-weighted, undirected graph of minimum total edge weight (and without any cycles). In most cases, an exhaustive search through all combinations is not a viable option. For example, for the traveling salesman problem with 45 cities (as it is solved below) an exhaustive search of all possible roundtrips would require finding the minimum of 45! possibilities, where $45! \approx 1.2\ 10^{56}$. A similar problem with 22 cities (also treated below) still yields $22! \approx 1.1\ 10^{21}$ combinations. It is these staggering numbers why, from an algorithmic point of view, combinatorial optimization problems rank among the most challenging problems in optimization.

In this section we introduce and implement a meta-heuristic algorithm, inspired by nature: the **ant colony algorithm**. It has been introduced in 1992 by Marco Dorigo and develop further by numerous scientists since then. Since its introduction, it has found a great many applications in operations research, artificial intelligence, machine learning and many other fields.

It is designed by observing a group of ants foraging for food sources from their nests, in essence solving a combinatorial path optimization problem. While each individual ant has a limited capability of "solving the food problem", the group, in contrast, is capable of impressive solutions to complex problems. By proper communication between individual

ants, passing on their findings, the path optimization by ants represents a grand example of **swarm intelligence** or intelligent group collaboration. In ant colonies, communication is accomplished by leaving chemicals (**pheromones**) on the chosen trails. These pheromones influence the successive ant's behavior in as much as paths laced with more pheromones are preferred over paths with less pheromones. This preference, however, is *stochastic* in nature: ants will choose a higher-pheromone trail with higher probability than selecting a lower-pheromone path; still, it will choose a lower-pheromones trail from time to time. Spreading and following pheromones thus promotes an **autocatalytic** behavior: as ants more often follow the higher-pheromone trail, this trail becomes increasingly attractive. This mechanism creates a positive feedback loop.

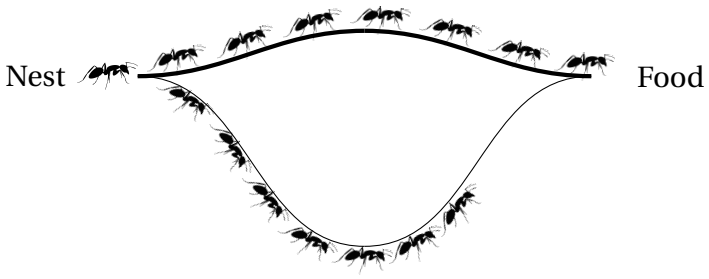

Figure 1.1: Principle of ant colony optimization, explained on a two-branch path.

We explain the ant-colony algorithm by considering the double-path experiment with asymmetric branches. Furthermore, we assume that all ants travel at the same speed and leave the same amount of pheromones on the trail. Ants are foraging for food from the nest. They initially choose paths at random; in our example, ants are as likely to choose the upper as they are to select the lower path. While they move towards the food source, they spread pheromones along the respective trails. The ants on the short, upper path return back to the nest in far shorter time than the ants that chose the longer, lower path. By the time the next ants are exploring the two paths, the upper path will be laced with more pheromones and thus be more likely to be chosen by subsequent ants. Over time, the upper path will dominate over the lower path as the shortest way to the food source.

Casting this behavior into a metaheuristic algorithm, we need to decide on a methodology how to distribute pheromones along trails and determine quantitative measures on how to use the pheromone information when making choices about possible paths.

The probability of choosing a certain path at a crossing point depends on the amount of deposited pheromones. In view of the double-path problem above, we have the following **stochastic model** for the probability of choosing path s from a reference point i over an alternative path l (also from the same reference point i)

$$P_{is}(t) = \frac{(t_s + \phi_{is}(t))^\alpha}{(t_s + \phi_{is}(t))^\alpha + (t_s + \phi_{il}(t))^\alpha} \tag{1.1}$$

with $t_s = l_s / v$ as the travel time along path s, with l_s as the length of path s, and v as the (uniform) velocity at which all ants travel. The quantity ϕ_{is} is the total amount of pheromones

on the branch between reference point i and the end of the path s. The same quantity ϕ_{il} then denotes the amount of pheromones on the alternative route l. The parameter α determines the probability distribution for the decision and is selected experimentally as $\alpha = 2$. Once P_{is} is determined, we then draw a random number from a uniform distribution and decide on the path to take by comparing it to P_{is}; this is akin to the rejection method for making decisions based on general probability distributions. For combinatorial optimization problems that involve a connection between a source (ant heap) and a target (food), it is important to have the ants travel towards the target and back to the source, while depositing pheromones on the forward and backward paths. For our application to the traveling salesman problem (see below), this is not necessary.

While the above outline already constitutes the rudimentary version of the ant-colony optimization algorithm, there are various improvements that can be considered. Loop correction is one of the most common: it proposes to conduct the forward path towards the target without spreading pheromones, after which we check the chosen path for loops; once the loops are eliminated, pheromones are spread *a-posteriori* on the loop-corrected path. Another improvement (which we will also implement below) is concerned with laying down more pheromones on more optimal paths, or considering the current optimality of the path in the probability of choosing a path. For example, we can deposit additional pheromones, indirectly proportional to the length of the trip, on better path combinations. With this trick we need less ants to converge to the optimal solution.

One aspect is also important in ant-colony optimization, which was not included in the original version of the algorithm: **pheromone evaporation**. We assume a constant proportion of pheromones to evaporate and vanish from the path over time. This feature allows the ants to explore less optimal and less-travelled routes; from a mathematical point of view, it helps avoid being trapped in local minima.

The full ant-colony optimization (ACO) algorithm then reads

1. Place a small and uniform pheromone level on all branches of the network.
2. Randomly place ants on the nodes of the network.
3. Move the ants forward through the network, choosing a path by probabilistically selecting the next node based on the relative pheromone levels of surrounding nodes.
4. Eliminate loops in the traced path.
5. Retrace the steps and deposit pheromones.
6. Evaporate the pheromones (globally).
7. Add additional pheromones to the retraced arcs, inversely proportional to the path length.
8. Go back to step 2, until converged.

The main characteristics of the ant-colony optimization algorithm is that it is (i) bio-inspired, (ii) based on stochastic searching of optimal solutions and (iii) relying on a positive feedback mechanism via communication and learning by pheromone levels.

1.1 Application to the traveling salesman problem (TSP)

The traveling salesman problem is the quintessential combinatorial optimization problem and often used to benchmark and showcase combinatorial optimization algorithms. It tries to solve the problem of visiting all points in a plane once on a circular path of minimal length. It is akin to a salesman who has to visit a set of cities once, while minimizing the distance he travels.

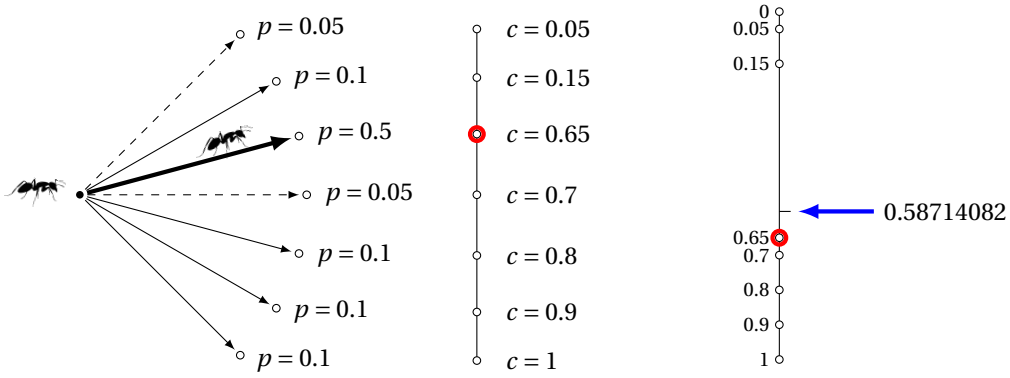

Figure 1.2: Choosing a path. The probabilities p are determined by the amount of pheromones on the trails and the length of the trails. A random number is drawn (see blue arrow) and compared to the cumulative probabilities c, from which a path is selected (indicated by red circle).

Following the ant-colony algorithm outlines above, we initialize all $N(N-1)$ edges with a uniform amount of pheromones. We then choose a starting point for k ants randomly from the N nodes; each ant follows a path, choosing the next node stochastically based on the pheromone level of the various choices. To accomplish a circular path, we introduce a variables `Unv` which records the previously visited nodes. When choosing the next node, this variable guarantees that previously visited sites are excluded. After a circular path is established, its path-length is determined and pheromones are deposited on the path according to the path-length: short paths are "rewarded" more than longer paths. A running record of the best solution will be kept. As a final step in each iteration, the pheromone levels on all paths will be evaporated, which is accomplished by a simple multiplication of the pheromone matrix by a scalar factor less than one. The algorithm is stopped after a fixed number of iterations; other stopping criteria are feasible as well.
The code below is implementing the ant-colony algorithm to solve the traveling salesman problem.

```
import numpy as np
import scipy as sp
import math as ma
import matplotlib.pyplot as plt

def getCities(nc):
    # city data (latitude - longitude)
    yy = np.zeros(45)
    xx = np.zeros(45)
    yy[0] = 48. + 12./60; xx[0] = 16. + 22./60 # Vienna
    .
    .
    .
    yy[44] = 50. + 27./60; xx[44] = 30. + 31.5/60 # Kiev
    .
    .
    .
    # initialize the coordinates of the cities
    x  = np.zeros(nc)
    y  = np.zeros(nc)
    # copy city data and calculate distance matrix
    x  = xx[0:nc]
    y  = yy[0:nc]
    sc = cc[0:nc]
    D  = calcDist(y,x)

    return x,y,D,sc
```

```
def calcLatLonDist(Lat1,Lon1,Lat2,Lon2):
    # calculate the distance between two
    # points given in (Lat,Lon)
    R    = 6373  # average radius of Earth (in km)
    # conversion from degrees
    La1  = Lat1/180*ma.pi
    La2  = Lat2/180*ma.pi
    Lo1  = Lon1/180*ma.pi
    Lo2  = Lon2/180*ma.pi
    # formula from the U.S. Census Bureau website
    dLat = La2-La1
    dLon = Lo2-Lo1
    a    = (ma.sin(dLat/2))**2 + \
           ma.cos(La1)*ma.cos(La2)*(ma.sin(dLon/2))**2
    c    = 2*ma.atan2(ma.sqrt(a),ma.sqrt(1-a))
```

```
        dis  = R*c
        return dis
```

```
def calcDist(La,Lo):
    # calculate the distance matrix between the cities
    n    = len(La)
    dist = np.zeros((n,n))
    for i in range(n):
        for j in range(i+1,n):
            L1 = La[i]
            L2 = La[j]
            B1 = Lo[i]
            B2 = Lo[j]
            dist[i,j] = calcLatLonDist(L1,B1,L2,B2)
            dist[j,i] = dist[i,j]
    return dist
```

```
def drawPath(path,X,Y):
    # graphics output
    nc = len(path)
    XX = list()
    YY = list()
    px = path + [path[0]]
    for i in px:
        XX.append(X[i])
        YY.append(Y[i])

    plt.plot(XX,YY,'o',XX,YY)
    # plt.savefig('PATH.png')
    plt.draw()
    plt.show()
```

```
def pathLength(path,D):
    # step 1: get city-index-pairs
    pairs = zip(path, path[1:] + [path[0]])
    # step 2: sum the city-pair distances
    pL    = sum([D[r,c] for (r,c) in pairs])
    return pL
```

```
def compNextBranch(p):
    r = np.random.random()
    c = np.cumsum(p)
```

```
        j = np.argwhere(c >= r).flatten().min()
        return j
```

```
def costFunction(path,D):
    # could be changed to something else
    C = pathLength(path,D)
    return C
```

```
def genPath(eta,tau):
    # generate a path connecting all cities
    alpha = 2.  # pheromone exponential weight
    beta  = 2.  # heuristic exponential weight
    nc    = len(eta)
    Unv   = np.ones(nc) # keep track of visited cities
    # step 1: randomly choose a city
    curr      = np.random.randint(0,nc-1)
    Unv[curr] = 0
    # step 2: start the path
    path    = [curr]
    # visit all cities from the starting city
    for i in range(nc-1):
        p_tau = np.power(tau[curr,:],alpha)
        p_eta = np.power(eta[curr,:],beta)
        p_et  = (p_tau*p_eta)*Unv
        p     = p_et/sum(p_et) # prob. distrib. to move from current
        curr  = compNextBranch(p)
        Unv[curr] = 0
        path.append(curr)
    return path
```

```
def optAnts(n,x,y,D,maxit,nAnts):
    # optimization parameters
    Q        = 1.
    tau0     = 10.*Q/(n*np.mean(D)) # initial pheromones
    rho      = 0.15;     # evaporation rate
    # initialization
    E        = np.eye(n)
    eta      = 1./(D + E) - E      # heuristic information matrix
    tau      = tau0*np.ones((n,n)) # pheromone matrix
    BestCost = np.zeros(maxit)     # array to hold best cost values
    BestSol  = 1.e30
    AntPath  = np.zeros((nAnts,n), dtype=int)
```

```
13      AntCost  = np.zeros(nAnts)
14
15      for it in range(maxit):
16          # move ants
17          for k in range(nAnts):
18              path         = genPath(eta,tau)
19              AntPath[k,:] = path
20              AntCost[k]   = costFunction(path,D)
21              if (AntCost[k] < BestSol):
22                  BestSol  = AntCost[k]
23                  bpath    = AntPath[k].tolist()
24          pL = pathLength(bpath,D)
25          print it,'pathLength (aco) = ',pL
26
27          #...update pheromones
28          for k in range(nAnts):
29              path  = AntPath[k,:].tolist()
30              pairs = zip(path, path[1:] + [path[0]])
31              # step 2: update the city-pair pheromones
32              for (r,c) in pairs:
33                  tau[r,c] = tau[r,c] + Q/AntCost[k]
34          # evaporation
35          tau = (1.-rho)*tau
36
37          #...store best cost
38          BestCost[it] = BestSol
39
40          #...show iteration information
41          #print 'iteration = ',it,' best cost = ',BestCost[it]
42      return bpath,BestCost
```

```
1   if __name__ == '__main__':
2
3       nc             = 45 # number of cities considered (max: 45)
4       x,y,D,sc       = getCities(nc) # construct model
5       maxit          = 250  # maximum number of iterations
6       nAnts          = 100  # number of ants (population size)
7       bpath,BestCost = optAnts(nc,x,y,D,maxit,nAnts)
8       for i in bpath:
9           print sc[i],' > ',
10      drawPath(bpath,x,y)
11
12      plt.plot(BestCost)
```

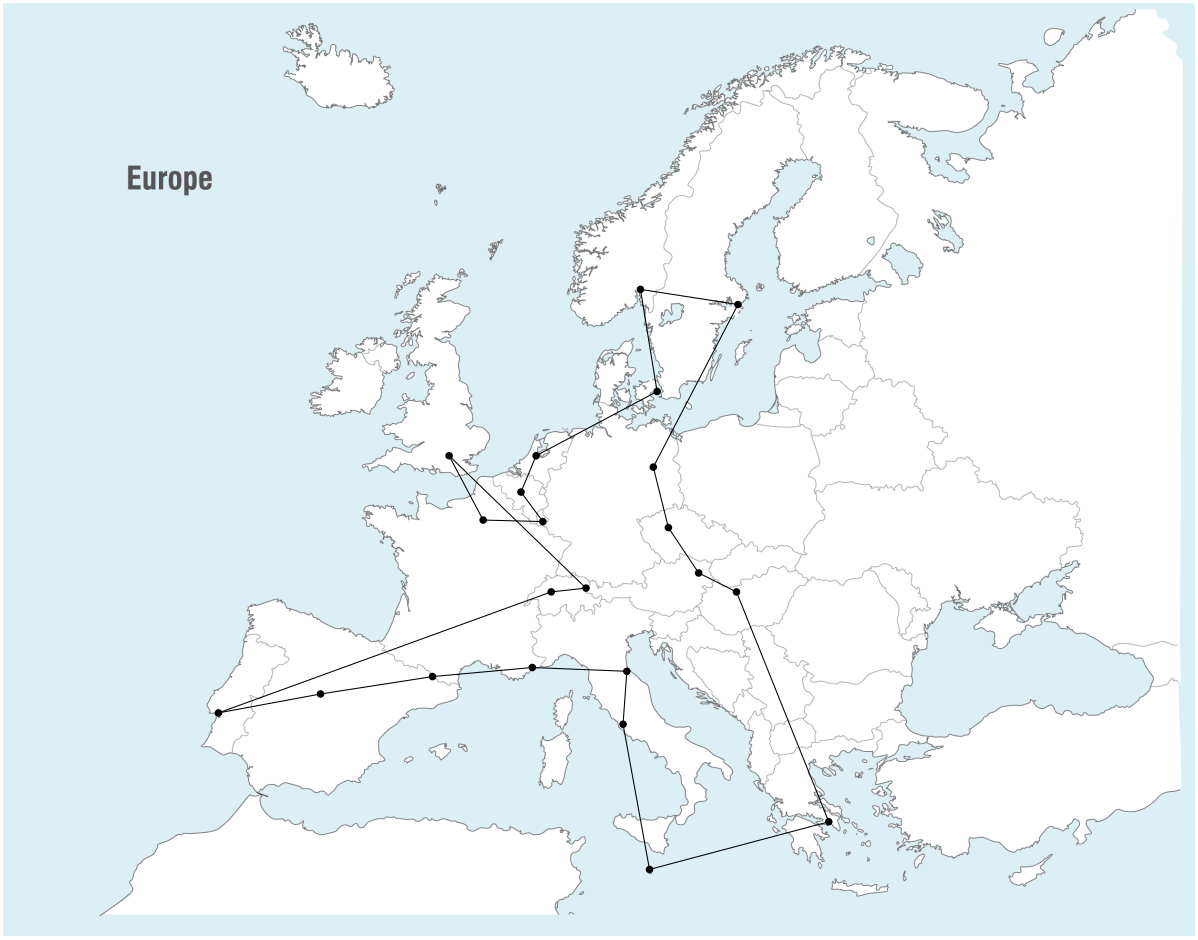

Figure 1.3: Solution to the traveling salesman problem across 22 selected European capital cities. The path length is 11503.74 kilometers. This is the result after 250 iterations, using 100 ants. After these iterations, the path shows crossings, which are clearly suboptimal.

```
13        plt.show()
```

The application of the above code to the traveling salesman problem for a set of European capital sities is given below. Using a colony of 100 ants and running 250 iterations on a 22-city configuration, we observe a rather short, but not optimal circular path. In particular, the path-crossing involving London, Paris, Brussels and Luxembourg adds unnecessary length to the path; by untangling this loop, a shorter and more optimal path seems possible. The algorithm clearly converged to a local minimum. Running the code for more iterations could solve this problem, since due to the pheromone evaporation other path-configurations may be explored. A more direct method for directing the algorithm towards a global minimum (using local corrections) is dealt with in the exercises.
The situation for more cities (now 45 selected European capitals) shows a worsening of the path-crossing problem. Again, we use 100 ants and run the code for 250 iterations (admit-

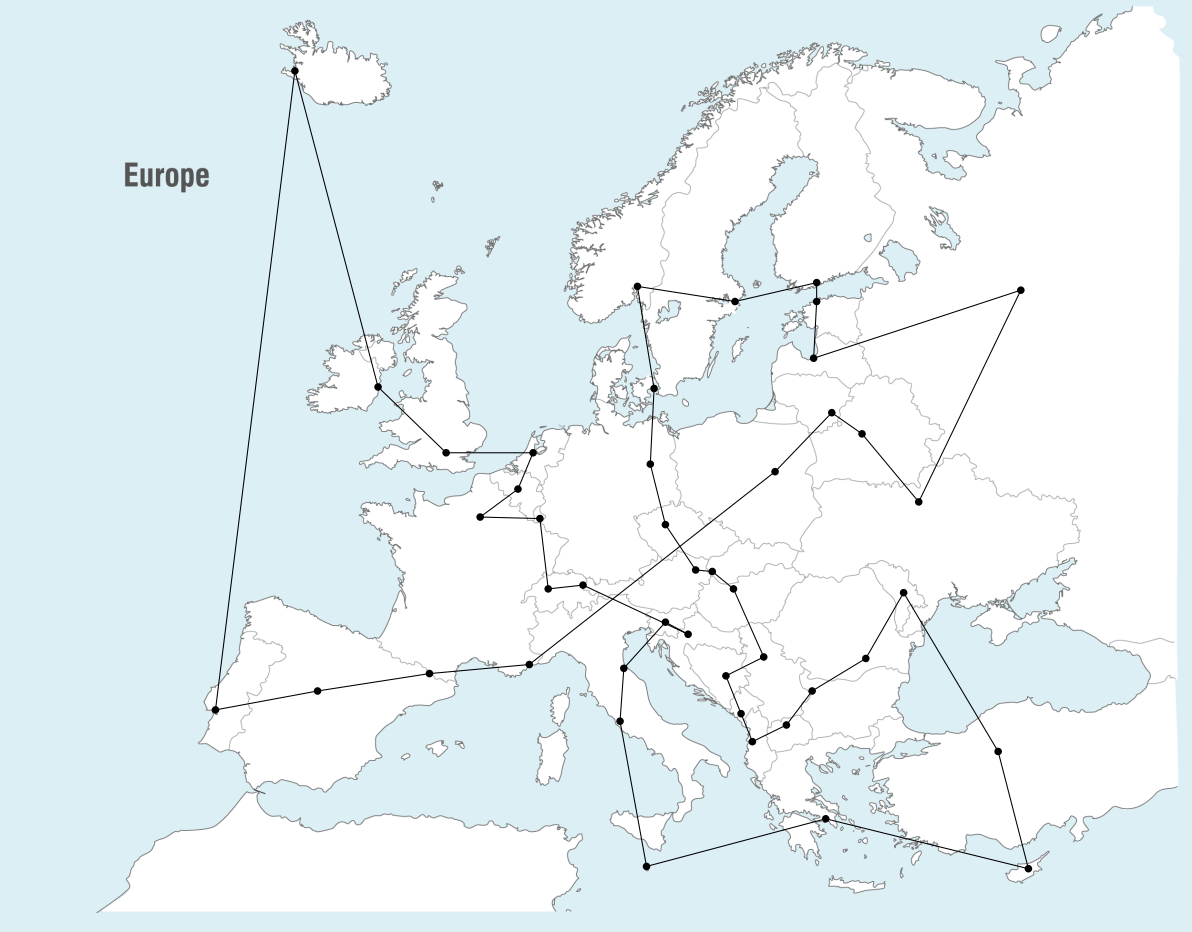

Figure 1.4: Solution to the traveling salesman problem across 45 selected European capital cities. The path length is 21476.80 kilometers. This is the result after 250 iterations, using 100 ants. After these iterations, the path shows crossings, which are clearly suboptimal.

tedly, too low a number of iterations). Even though the path-length of 21476 kilometers is within about a 1000 kilometers of the optimum, the proposed path is clearly not optimal. The trip from Monaco (Southern France) to Warsaw (Poland) will surely not be part of the final, optimal solution. Running the code for more iterations may mitigate the problem, but a simple hybrid modification, combining the global view of the ants with a local path-untangling strategy, will solve the problem a lot more efficiently.