The Traveling Salesman Problem

Didier Maquin

▶ **To cite this version:**

Didier Maquin. The Traveling Salesman Problem. CRAN. 2025. ⟨hal-04909391v3⟩

HAL Id: hal-04909391
https://hal.science/hal-04909391v3

Submitted on 15 Jun 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

The Traveling Salesman Problem

Didier Maquin – didier.maquin@univ-lorraine.fr

June 14, 2025

Contents

1 Statement of the problem **5**

2 History **5**

3 Formal description **5**

4 Complexity **6**

5 Interest **6**

6 Problem solving method **6**

7 The TSP2024 Matlab App **7**

8 Random tour **10**

9 Nearest neighbour **10**
- 9.1 Description . . . 10
- 9.2 Matlab code . . . 11

10 Local optimization – the 2-opt heuristic **13**
- 10.1 Description . . . 13
- 10.2 Matlab code . . . 14

11 Double-ended nearest and loneliest neighbour **15**
- 11.1 Method principle . . . 15
- 11.2 Matlab code . . . 17
- 11.3 Explanations . . . 18

12 Multi-fragment heuristic **19**
- 12.1 Method principle . . . 19
- 12.2 Matlab code . . . 20
- 12.3 Explanations . . . 22

13 Insertion algorithms **22**
13.1 Method principle [24] . . . 22
13.2 Selection of heuristic elements . . . 23
13.3 Insertion from the convex hull method . . . 25
13.3.1 Method principle . . . 25
13.3.2 Matlab code . . . 25
13.3.3 Explanations . . . 27
13.3.4 First alternative approach . . . 27
13.3.5 Matlab code . . . 28
13.3.6 Explanations . . . 29
13.3.7 Second alternative approach . . . 30
13.3.8 Matlab code . . . 31
13.3.9 Explanations . . . 34
13.4 Insertion from a random vertex – cheapest insertion . . . 34
13.4.1 Method principle . . . 34
13.4.2 Matlab code . . . 34
13.5 Insertion based on Delaunay triangulation . . . 36
13.5.1 Method principle . . . 36
13.5.2 Matlab code . . . 37
13.6 Explanations . . . 40

14 Minimum-weight spanning-tree **41**
14.1 Method principle . . . 41
14.2 Matlab code . . . 43
14.3 Explanations . . . 43

15 Bitonic tour based method **44**
15.1 Method principle . . . 44
15.2 Matlab code . . . 48
15.3 Explanations . . . 50

16 Polynomial-time deterministic (PTD) approach **50**
16.1 Method principle . . . 50
16.2 Matlab code . . . 53
16.3 Explanations . . . 56

17 Clarke and Wright heuristic **59**
17.1 Method principle . . . 59
17.2 Matlab code . . . 61
17.3 Explanations . . . 62

18 Divide and conquer approach **63**
18.1 Method principle . . . 63
18.2 Matlab code . . . 65
18.3 Explanation . . . 69

19 Karp's approach **71**
19.1 Method principle 71
19.2 Matlab code 73
19.3 Explanations 75

20 Pair-center algorithm **76**
20.1 Method principle 76
20.2 Matlab code 78
20.3 Explanations 80

21 Space-filling curve **82**
21.1 Method principle 82
21.2 Matlab code 85

22 Metropolis algorithm – Markov Chain Monte Carlo **87**
22.1 Method principle 87
22.2 Matlab code 89

23 Simulated annealing based method **91**
23.1 Method principle 91
23.2 Matlab code 92
23.3 Explanations 95

24 Kohonen map inspired method **95**
24.1 Method principle 95
24.2 Matlab code 98
24.3 Explanations 101

25 Bees algorithm **101**
25.1 Method principle 101
25.2 Matlab code 103

26 Reinforcement learning method **106**
26.1 Method principle 106
26.2 Matlab code 108
26.3 Explanations 109

27 Ant System method **109**
27.1 Method principle 109
27.2 Matlab code 111

28 Genetic algorithm **113**
28.1 Method principle 113
28.2 Matlab code 116
28.3 Explanations 118

29 Black hole algorithm **118**
29.1 Method principle 118
29.2 Matlab code 120
29.3 Explanations 122

30 Firefly algorithm **123**
30.1 Method principle 123
30.2 Matlab code 126
30.3 Explanations 130

31 Tabu search **131**
31.1 Method principle 131
31.2 Matlab code 132
31.3 Explanations 134

References **134**

Licences **138**

Appendix 1 : Loading TSPLIB file **141**

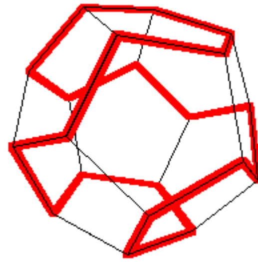
Appendix 2 : Algorithm, efficiency, algorithmic decidability **144**

1 Statement of the problem

A traveling salesman must visit n given cities passing through each city exactly once. It starts with any city and ends by returning to the starting city. The distances between the cities are known. Which path should be chosen in order to minimize the distance travelled?

2 History

19th century — The first mathematical approaches exposed for the traveling salesman problem (TSP) were treated in the 19th century by the mathematicians Sir William Rowan Hamilton and Thomas Penynington Kirkman. Hamilton made a game out of it: *Hamilton's Icosian game*. Players had to complete a tour passing through 20 points using only the predefined connections.



1930s — TSP is covered in more depth by Karl Menger at Harvard. It was then developed at Princeton by the mathematicians Hassler Whitney and Merrill Flood. Particular attention is paid to the connections by Menger and Whitney as well as to the growth of TSP.

1954 — TSP solution for 49 cities by George Dantzig, Delbert Fulkerson and Selmer Johnson by the method of the cutting-plane[1].

1975 — Solution for 67 cities by Camerini, Fratta and Maffioli.

1987 — Solution for 532, then 2 392 cities by Padberg and Rinaldi.

1998 — Solution for the 13 509 cities of the United States.

2001 — Solution for the 15 112 cities in Germany by Applegate, Bixby, Chvàtal and Cook from Rice and Princeton universities[2].

3 Formal description

From a cost matrix $C = (C_{ij})$ where C_{ij} represents the distance between city i and city j $(1 \leq i, j \leq n)$, find a permutation

$$\sigma = (\overset{1}{\sigma_1} \quad \overset{2}{\sigma_2} \quad \dots \quad \overset{n}{\sigma_n})$$

[1] G.B. Dantzig, R. Fulkerson, S.Johnson, *Solution of a large-scale traveling salesman problem*, Operations Research 2: 393-410, 1954.

[2] D. Applegate, R. Bixby, V. Chvàtal and W. Cook, *Implementing the Dantzig-Fulkerson-Johnson algorithm for large traveling salesman problems*, Mathematical Programming Series B, 97:91–153, 2003.

which minimizes the sum of the distances $C_{\sigma(1)\sigma(2)}+C_{\sigma(2)\sigma(3)}+\cdots+C_{\sigma(n-1)\sigma(n)}+C_{\sigma(n)\sigma(1)}$. In other words, one must find a Hamiltonian cycle of minimum length in a valued graph with distance where the vertices are the cities.

4 Complexity

This problem is a representative of the class of NP-complete problems[3]. The existence of a polynomial complexity algorithm remains unknown. A quick calculation of the complexity shows it grows as $O(n!)$ where n is the number of vertices. Table 1 shows the combinatorial explosion of TSP for a complete graph.

# vertices	# edges	# combinations
5	10	12
6	15	60
7	21	360
8	28	2 520
9	36	20 160
10	45	181 440
15	105	43×10^{9}
20	190	6×10^{16}
25	300	31×10^{22}

Table 1: Number of paths depending on the number of cities

If it takes 1 ns to examine a path, with a graph with 100 vertices, it takes 10^{158} ns or 10^{132} times the age of the universe (estimated at 13 billion years)!

5 Interest

The TSP provides an example of a study of a NP-complete problem whose resolution methods can be applied to other discrete math problems. However, it also has direct applications, particularly in transport and logistics. For example, finding the shortest route for school buses or, in industry, finding the shortest distance the mechanical arm of a machine will have to travel to drill the holes in a circuit board (the holes represent cities).

6 Problem solving method

Algorithms for solving TSP can be divided into two classes:

- deterministic algorithms that find the optimal solution; their complexity is exponential, they are very greedy in computing power. The most efficient algorithms are based on the cutting plane method, *cutting-plane* introduced by Ralph Gomory which is based on a formulation in terms of linear optimization.

[3] See appendix 2

- approximation algorithms that provide a sub-optimal solution. They make it possible to find a solution whose cost is close to the cost of the optimal solution. They have the advantage of allowing a solution to be found in a reasonable time.

7 The TSP2024 Matlab App

The TSP2024 Matlab App implements 24 different sub-optimal methods for solving the TSP problem as well as the 2-opt enhancing method. The proposed methods are (Method frame):

- Random tour
- Nearest neighbour
- Double-ended nearest and loneliest neighbour (DENLN)
- Multi-fragment heuristic
- Convex hull insertion
- Cheapest insertion
- Delaunay insertion
- Spanning tree
- Bitonic tour
- PTD approach
- Clarke and Wright heuristic
- Divide and conquer
- Karp's approach
- Pair-center algorithm
- Space-filling curve
- Metropolis algorithm
- Simulated annealing
- Kohonen map
- Bees algorithm
- Reinforcement learning
- Ant system
- Genetic algorithm
- Black hole algorithm
- Tabu search
- 2-opt heuristic (can be applied to any result of one of the methods)

The application was not designed with a performance objective, however, it can work with a **hundred points** (or vertices). Some methods require the choice of certain hyper-parameters. These choices have not been made accessible to the user. The parameters have been adjusted to provide "satisfactory" results on examples with up to a hundred vertices.

The processed data set is either random or read from a file in TSPLIB format. The user can choose this using the buttons on the right of the visualization window below "Choice of data set". At launch, a random set of 20 points is produced and visualized. To change the number of points, simply indicate it in the "Number of points" box and press the "New data set" button. The data set can also come from a file in TSPLIB format. In this case, press the "Load" button. A dialog window listing the files in the current directory will then appear. It enables a user to select the name of a *.tsp file. If the file is valid (for more details, consult appendix 1), the data set is then displayed.

Once again, please note that the application is not designed to handle large sets of points. A message alerts the user when the number of points in the file exceeds 150 (without generating an error). In this case, some methods may not provide results with the selected hyperparameters.

In order to be able to compare the results obtained (essentially through the length of the cycle: Tour length, displayed at the top of the figure), each method can be tested on the same data set: just select a method and press the "Run" button. For a given data set, the length of the best tour obtained (Best length) is stored and displayed at the bottom left of the window as the name of the corresponding method (Method).

The application has been designed to visualize the method in action[4]. For this purpose, the on/off switch at the top right of the window allows the slowing down of the graphic animation to observe each step of the implemented algorithms. The switch below controls how the final result is displayed. If set to "Path", the Hamiltonian cycle and the points are displayed. If set to "Patch", the interior of the polygon defined by the Hamiltonian cycle is displayed. Be careful, this notion of interior can be poorly defined if the cycle has crossings (before using 2-opt optimization). This option is more suitable for large sets of points (or for abstract art lovers!).

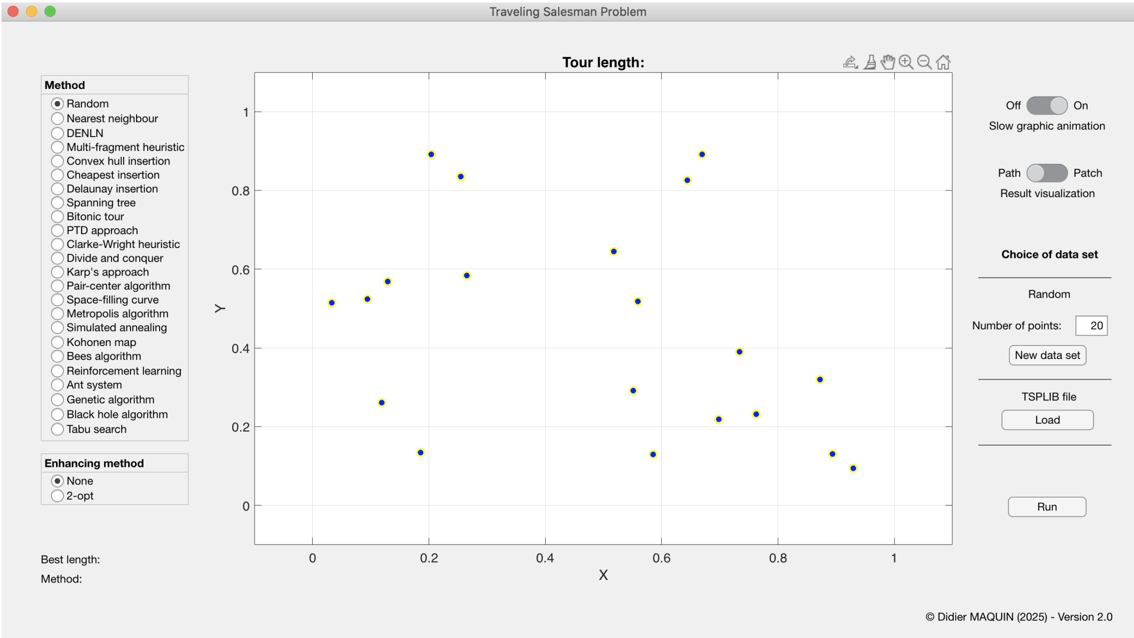


Figure 1: TSP2024 Matlab App interface

Double-ended nearest and loneliest neighbour – Multi-fragment heuristic – Convex hull insertion – Delaunay insertion – Spanning tree – Bitonic tour – PTD approach – Divide and conquer – Karp's approach – Pair-center algorithm – Space-filling curve : these methods are totally deterministic. Launched several times on a data set they will provide the same result.

The other thirteen methods include random choices which can lead to different results with each run.

Random tour starts with a random vertex and builds the Hamiltonian cycle by randomly choosing a new vertex at each iteration.

[4]There was no search for performance in the method implementation.

Nearest neighbour starts with a random vertex.

Cheapest insertion starts with a random vertex and the first Hamiltonian cycle is created with the two closest vertices from that one.

Clarke and Wright heuristic defines the vertex hub as a random vertex.

Metropolis algorithm. In essence, it is a stochastic optimization method where randomness plays a role. The number of learning iterations is set to 10^5.

Simulated annealing method requires setting several parameters. The initial temperature was set to 2000, the cooling factor to 0.98 and the maximum number of iterations to 8000. Visualization of intermediate results is only performed every 100 iterations.

Kohonen map method is implemented with the following parameters : initial learning rate, $\alpha_0 = 0.8$ and decreasing factor, $\beta = 0.9997$. The number of neurons is equal to eight times the numbers of points. Their weights which corresponds to the 2D positions of the neurons are initialized randomly (between 0 and 1 since it is the case of the coordinates of the points to connect). The stopping rule is only indexed on a predetermined number (10 000) of iterations. Visualization of intermediate results is only performed every 200 iterations.

Bees algorithm requires setting several parameters. The number of scout bees is set to 25, the number of elite sites to 4, the number of best sites to 20, the number of elite bees to 300, the number of best bees to 100 and the maximum number of iterations to 400.

Reinforcement learning implements a simple Q-learning approach. The learning rate is set to 0.1, the discount factor to 0.45. To balance exploration and exploitation, actions are selected using an ε-greedy algorithm. ε is initially set to 1 and decreases after each learning episode; it is multiplied by the decay factor 0.9999. The number of episodes is set to 30 000.

Ant system method, as the previous one, has many tuning parameters. The number of ants is set to 50, the pheromone quantity per tour to 1 (arbitrarily), the pheromone exponential weight and heuristic exponential weight to 1, the evaporation rate to 0.04, the number of elitist ants to 8 and the maximum number of iterations to 600.

Genetic algorithm can be implemented in many different ways. Here, the number of generations, corresponding to the maximum number of iterations, is set to 1000, the size of population to 500, the probability of crossover to 0.9 and that of mutation to 0.06, the number of individuals chosen for a tournament in order to select the parents for crossover is set to 20.

Black hole algorithm. Only two parameters have been fixed. The maximum number of iterations is set to 1000 and the size of the population to 400.

Tabu search. The maximum number of iterations of the main loop has been fixed to $100 + n$ where n is the chosen number of cities and the size of the tabu list is half the of the total number of possible movements to explore the neighbourhood of the current solution.

The implemented methods will now be described in detail.

8 Random tour

The method consists of choosing a random vertex and constructing the Hamiltonian cycle by randomly choosing a new vertex at each iteration. Once all the vertices of the graph have been visited, return to the first vertex to close the cycle. Of course, this method gives a very bad solution (see figure 2).

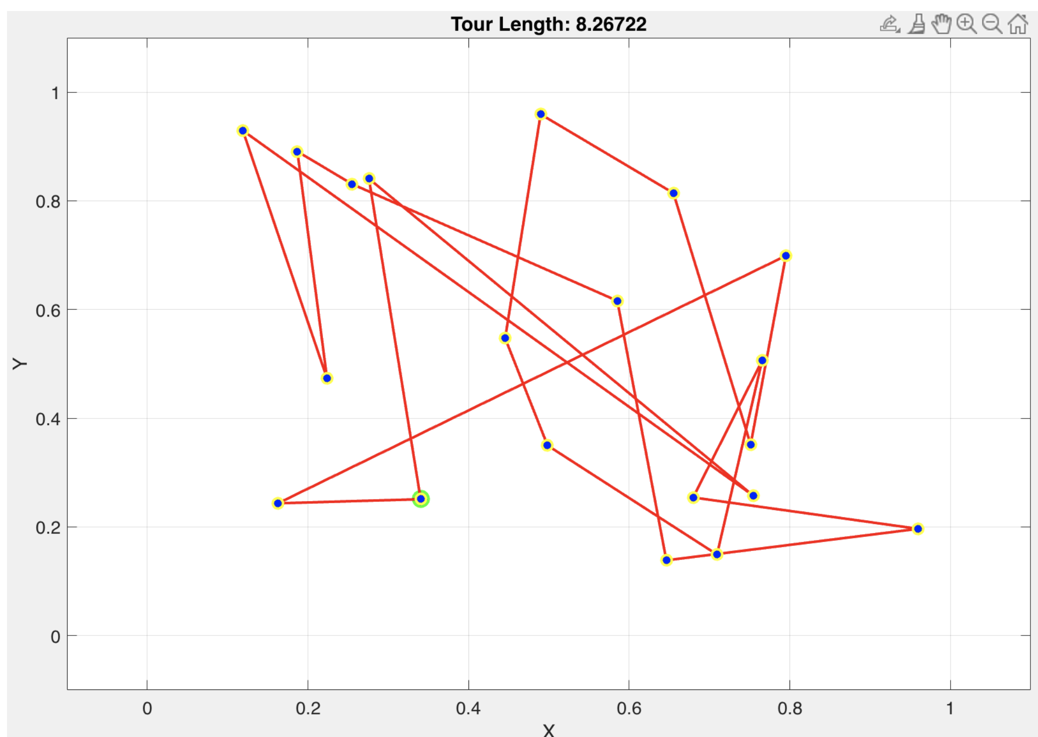


Figure 2: Cycle obtained by a random tour

9 Nearest neighbour

9.1 Description

Greedy algorithms[5] are any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. Nearest neighbour method is one of them. Starting from a vertex, it consists to visit the nearest vertex not yet visited and to repeat the process. Once all the vertices of the graph have been visited, return to the first vertex to close the cycle. This method provides a result in a very short time (complexity in $O(n^2)$). Qualitatively, we see (figure 3) that the result can be improved. It has crossings (which it would be advantageous to eliminate) and when approaching the end, the algorithm must connect scattered vertices which have been neglected before and which are very far apart. The penalty for not taking care of it before is relatively large, as seen in figure 3.

[5] An algorithm is said to be "greedy" when it does the choice of the local optimum at each step.

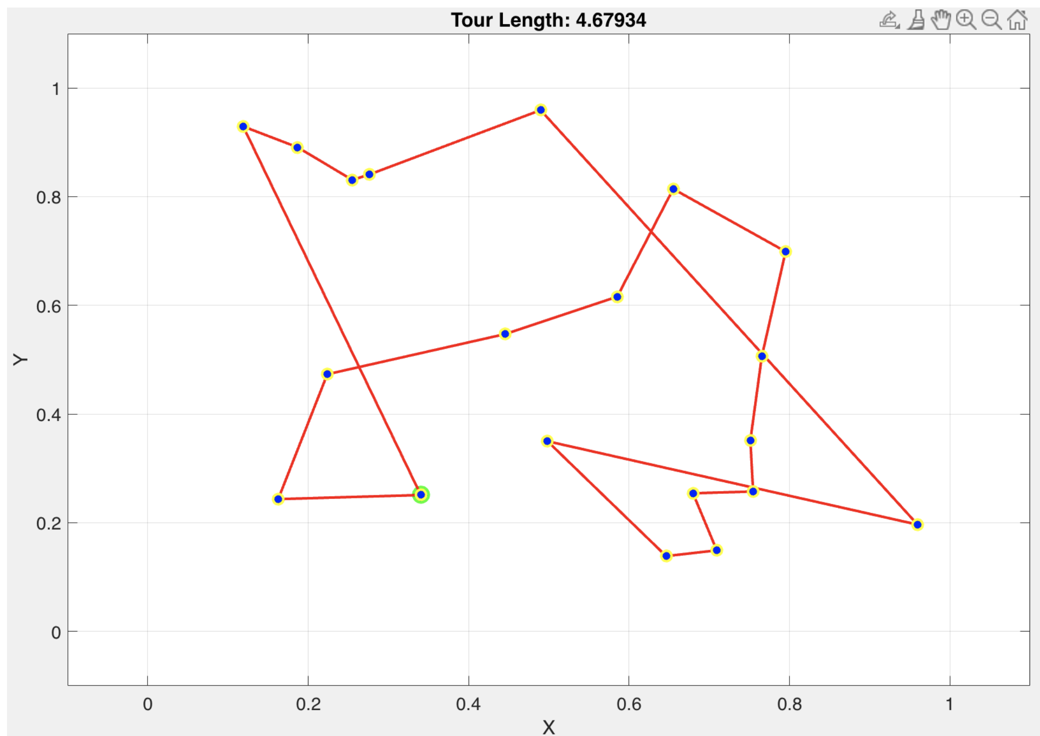


Figure 3: Cycle obtained by the method of the nearest neighbour

9.2 Matlab code

Inspired from Jonas Lundgren (2019).
TSPSEARCH (https://www.mathworks.com/matlabcentral/fileexchange/71226-tspsearch), MATLAB Central File Exchange. Retrieved February 8, 2022.

Main Program

```
%% TSP solving using the nearest neighbour approach followed by
%  the 2-opt optimization
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

%  Part of this software inspired by
%  Jonas Lundgren (2019). TSPSEARCH
%  (https://www.mathworks.com/matlabcentral/fileexchange/71226-tspsearch)
%  MATLAB Central File Exchange.

clear
close all
X = rand(50,2);
[n,dim] = size(X);
% Computation of the distance matrix
D = squareform(pdist(X,'euclidean'));
% Random start point
```

```
start = fix(rand*n)+1;
% Nearest neighbor method
p = greedy(start,D);
% "Dynamic" plot
tspplot1(p,X)
% Enhancement using 2opt
[p,L] = exchange2(p,D,X);
```

Functions

```
function p = greedy(s,D)
% Greedy travel to nearest neighbour
% s starting node
% D distance matrix

n = size(D,1);
p = zeros(1,n);
p(1) = s;

for k = 2:n
    D(s,:) = inf;
    [~,s] = min(D(:,s));
    p(k) = s;
end
end
```

```
function tspplot1(p,X)
%TSPPLOT Plot 2D tour
%   TSPPLOT(p,X), p is the tour and X is the coordinate matrix

%   Author: Jonas Lundgren <splinefit@gmail.com> 2012
%   Modified by Didier Maquin 2022

x = X(p,1); x = [x;x(1)];
y = X(p,2); y = [y;y(1)];

% Plot
plot(x,y,'kx',x(1),y(1),'ob','MarkerSize',12, 'LineWidth',2)
hold on
pause(0.2)
for i=2:length(p)+1
    plot([x(i-1) x(i)],[y(i-1) y(i)],'r','LineWidth',2)
    pause(0.2)
end

% Add title
L = sqrt(diff(x).^2 + diff(y).^2);
str = sprintf('Tour Length: %g',sum(L));
title(str)
end
```

10 Local optimization – the 2-opt heuristic

10.1 Description

The solutions provided by the preceding methods are unsatisfactory because they include crossings or very long edges. In order to improve the solutions obtained, one can implement a local optimization. The principle of the 2-opt heuristic is to examine each pair of non-consecutive edges. Let a be the edge linking vertices A and B and b be the edge linking vertices C and D in the initial cycle. If $d(A, B) + d(C, D) > d(A, D) + d(B, C)$, where $d(X, Y)$ is the distance between vertices X and Y, we create a new cycle by removing a and b and inserting two new edges, edge c between A and D and edge d between C and B.

Consider for example the path on the left of figure 4 (representing part of a Hamiltonian cycle) consisting of the succession of vertices 1–2–3–4–5–6–7–8–9. If we carry out the "permutation" of the edges 2–3 and 7–8 (decreasing of the edges in this case), we obtain the route represented on the right 1–2–7–6–5–4–3–8–9.

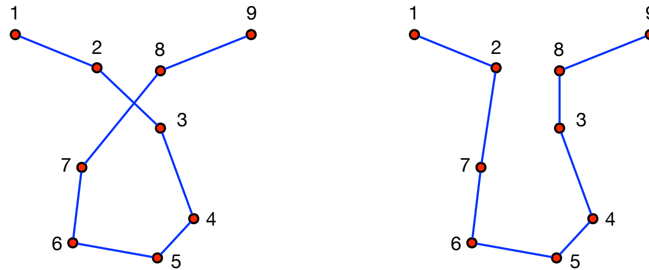


Figure 4: Illustration of the 2-opt heuristic

It will be noted that the second path is obtained easily. Here, among the four vertices involved (2, 3, 7 and 8) vertex 2 is the first in the sequence describing the initial Hamiltonian cycle and vertex 8 the last. The new path is easily deduced from the previous one by reversing the order of the vertices located between this first and this last vertex (1–2–7–6–5–4–3–8–9).

This local optimization can be applied to any Hamiltonian cycle obtained by a suboptimal method in order to refine it. This is the reason why, at the Matlab app interface, this option appears separately from the proposed methods.

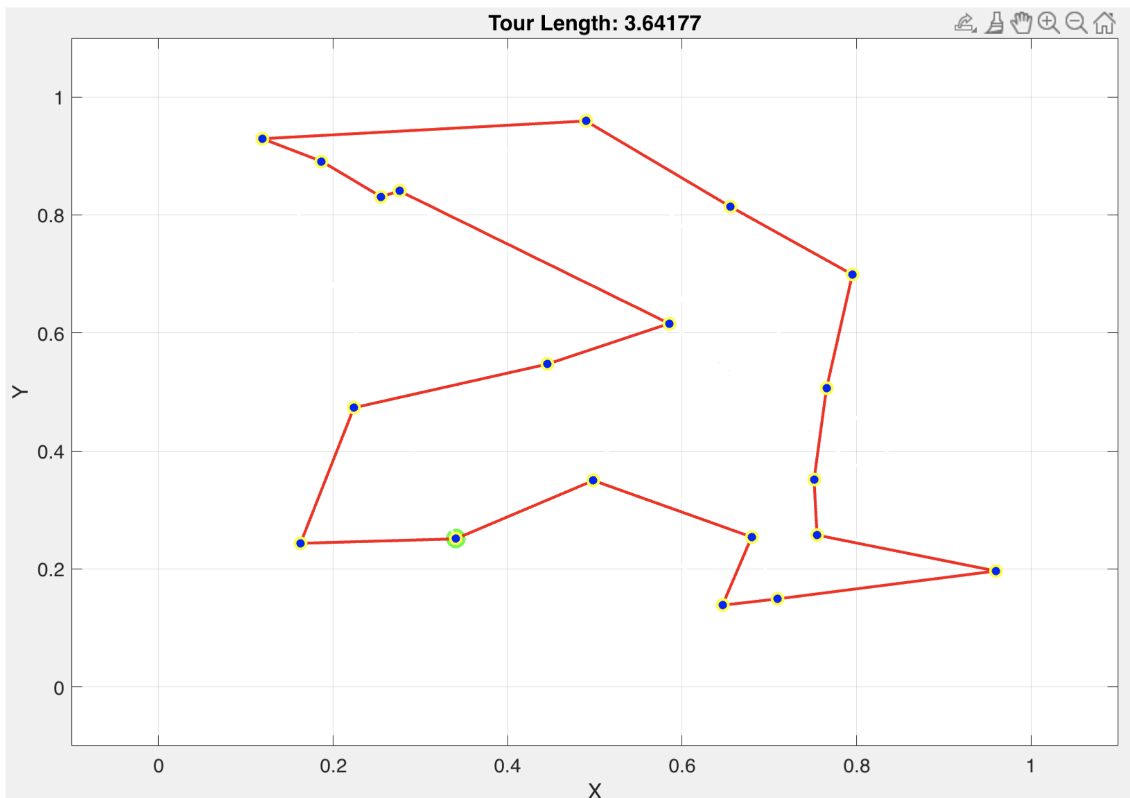


Figure 5: Cycle after applying the 2-opt heuristic

10.2 Matlab code

```
function [p,L] = exchange2(p,D,X)
%EXCHANGE2 Improve tour p by 2-opt heuristics (pairwise exchange of edges).
%   The basic operation is to exchange the edge pair (ab,cd) with the pair
%   (ac,bd). The algorithm examines all possible edge pairs in the tour and
%   applies the best exchange. This procedure continues as long as the
%   tour length decreases. The resulting tour is called 2-optimal.

n = numel(p);

% Tour length
L = D(p(n),p(1));
for j = 2:n
    L = L + D(p(j-1),p(j));
end
zmin = -L;

% Iterate until the tour is 2-optimal
while zmin/L < -1e-6
    zmin = 0;
    i = 0;
    b = p(n);

    % Loop over all edge pairs (ab,cd)
```

```
    while i < n-2
        a = b;
        i = i+1;
        b = p(i);
        Dab = D(a,b);
        j = i+1;
        d = p(j);
        while j < n
            c = d;
            j = j+1;
            d = p(j);
            % Tour length diff z
            % Note: a == d will occur and give z = 0
            z = (D(a,c) - D(c,d)) + D(b,d) - Dab;
            % Keep best exchange
            if z < zmin
                zmin = z;
                imin = i;
                jmin = j;
            end
        end
    end

    % Apply exchange
    if zmin < 0
        p(imin:jmin-1) = p(jmin-1:-1:imin);
        L = L + zmin;
        iminm1=imin-1 ; jminm1=jmin-1;
        % Closed polygon: point number "0" becomes point "n"
        if imin==1, iminm1=n; end
        if jmin==1, jminm1=n; end
        x = X(p,1);
        y = X(p,2);
        % Highlight the edges
        plot([x(iminm1) x(jminm1)],[y(iminm1) y(jminm1)],'b','LineWidth',2)
        plot([x(imin) x(jmin)],[y(imin) y(jmin)],'b','LineWidth',2)
        pause
        % Erasure of them
        plot([x(iminm1) x(jminm1)],[y(iminm1) y(jminm1)],'w','LineWidth',2)
        plot([x(imin) x(jmin)],[y(imin) y(jmin)],'w','LineWidth',2)
        % Plot of new edges
        plot([x(iminm1) x(imin)],[y(iminm1) y(imin)],'r','LineWidth',2)
        plot([x(jminm1) x(jmin)],[y(jminm1) y(jmin)],'r','LineWidth',2)
        pause
        str = sprintf('Tour Length: %g',L);
        title(str)
    end
end
```

11 Double-ended nearest and loneliest neighbour

11.1 Method principle

The nearest neighbor method gives rather poor results. However, it can be improved in several ways. As soon as the path (which will be closed to become a cycle) under construc-

tion is made up of two vertices, it has two end vertices. A first variation of the heuristic is to consider the new vertices closer to each of the route's ends and add to the tour the one that is closer to the route's respective endpoint. This way, the route grows with successive augmentations to both of its ends. This heuristic is known as the double-ended nearest neighbour.

Another lesser known variation of the nearest neighbour heuristic consists of determining and selecting the shortest edge of the distance matrix as the tour starting edge and then proceed exactly as in the nearest neighbour to include the following vertices. This favourably solves the problem of not knowing which vertex to start with and, statistically, it leads to better results than the nearest neighbour method.

The nearest neighbour heuristic and its variations suffer from a major problem. Their greedy nature of systematically and solely trying to reach the next closest vertex leads to the postponement of the connection of more distant vertices to the route. As a consequence, later in the tour construction, several vertices still remain that are quite apart from each other, forcing the method to include them at a higher cost.

This observation suggests that, among nearby vertices, a heuristic that gives some priority to those standing in more distant locations, would probably be more successful than the nearest neighbour heuristic.

In the paper "Double-ended nearest and loneliest neighbour – a nearest neighbour heuristic variation for the travelling salesman problem" [26], Fernando Pimentel proposes to take into account such priorities by modifying the distance matrix. To make it possible, he introduces the concept of loneliness of a vertex, computed from the average distance of that vertex to all others. Together with the distance to the closest neighbours, the loneliness of the closer neighbours will be also criteria for selecting the next vertex to be added to the route. Lonelier neighbours will be preferred over the others.

The proposed modified distance matrix is computed as follows

- Calculate the average of the distances of each city to all others
- Calculate the minimum, maximum and average (between both) of the average distances of each city to all others
- Calculate the symmetric of the distance of each city to all others with respect to the average calculated in the previous step (this will guarantee that a higher loneliness is rewarded with a shorter cost in the distance matrix).
- Calculate the new distance matrix where each new entry is the average between its old entry (the initial cost) and the respective entry of the matrix obtained in the previous step.

Based on this new metric, the algorithm proceeds as a double-ended nearest neighbour heuristic that starts with including the shortest edge (of the original distance matrix) among all.

11.2 Matlab code

Main Program

```
%% TSP solving using Doubled-ended nearest and loneliest neighbour approach
%  Included in TSP20024 app
%  Didier Maquin (2025). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

% Must be followed by a 2-opt optimization for avoiding crossings

clear
close all
tempo = 0.1;
rng(1)
X = rand(200,2);
n = length(X);
% Plot the points
plot(X(:,1),X(:,2),'kx','MarkerSize',12,'LineWidth',2)
hold on
D = squareform(pdist(X,'euclidean'));
% Compute the modified distance matrix (distance matrix pre-processing)
distset = sum(D);
min_d = min(distset);
max_d = max(distset);
average_d = (min_d+max_d)/2;
gt = distset > average_d;
lt = distset < average_d;
distset(gt) = average_d - (distset(gt) - average_d);
distset(lt) = average_d + (average_d - distset(lt));
newD = zeros(size(D));
for j=1:n
    newD(:,j) = (n*D(:,j)+distset(j))/2;
end
%
% Cancelation of the diagonal of D (0 --> inf)
D = D+diag(inf*ones(1,length(D)));
mini = min(D,[],'all');
% Search the first edge (closest vertices) in the initial distance matrix
[lig,col] = find(D==mini);
end1 = lig(1)
end2 = col(1)

% Cancelation of the diagonal of newD (0 --> inf)
newD = newD+diag(inf*ones(1,length(newD)));
% Include the first edge in the route
list = [end2 end1];
% Draw this edge in green
plot([X(end1,1) X(end2,1)],[X(end1,2) X(end2,2)],'g','LineWidth',2)

for k = 3:n
    sauvl1 = newD(list(end-1),:);
    sauvc1 = newD(:,list(end-1));
    newD(list(end-1),:) = inf;
    newD(:,list(end-1)) = inf;
    temp = newD(list(end),:);
    temp(list(1)) = inf;
```

```
    [mini1,s1] = min(temp);
    newD(list(end-1),:) = sauvl1;
    newD(:,list(end-1)) = sauvc1;
    %
    sauvl2 = newD(list(2),:);
    sauvc2 = newD(:,list(2));
    newD(list(2),:) = inf;
    newD(:,list(2)) = inf;
    temp = newD(list(1),:);
    temp(list(end)) =inf;
    [mini2,s2] = min(temp);
    newD(list(2),:) = sauvl2;
    newD(:,list(2)) = sauvc2;
    %
    if mini1 < mini2
        list = [list s1];
        newD(list(end-1),:) = inf;
        newD(:,list(end-1)) = inf;
        plot([X(list(end-1),1) X(list(end),1)],[X(list(end-1),2) ...
            X(list(end),2)],'r','LineWidth',2)
        pause(tempo)
    else
        list = [s2 list];
        newD(list(2),:) = inf;
        newD(:,list(2)) = inf;
        plot([X(list(1),1) X(list(2),1)],[X(list(1),2) ...
            X(list(2),2)],'b','LineWidth',2)
        pause(tempo)
    end
end
% Closing of the polygon for length calculus and drawing
p = [list list(1)];
plot(X(p,1),X(p,2),'r','LineWidth',2)
% Add title
L = sum(sqrt(diff(X(p,1)).^2 + diff(X(p,2)).^2));
str = sprintf('Tour Length: %g',L);
title(str)
```

11.3 Explanations

The calculus of the modified distance matrix follows the description given in the cited paper (C code). The two closest vertices are determined and constitute the initial path *list*. We then search for the vertices *s1* and *s2* closest to each of the ends of the path being constructed and we retain the closest one by comparing the distances *mini1* and *mini2*. Depending on which "side" (right or left) the nearest vertex is added, the vector *list* is updated and the corresponding edge is drawn (in red on the right and in blue on the left). At the end, the Hamiltonian cycle is closed by adding an edge between the first and the last vertices of *list*.

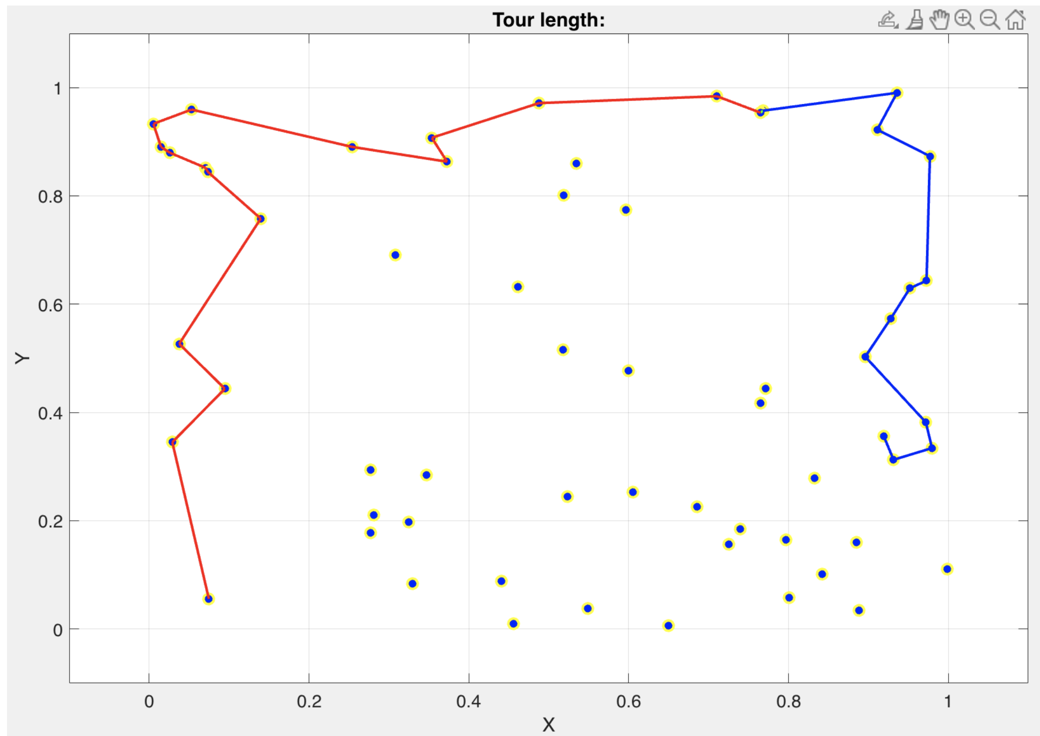


Figure 6: Tour construction in progress

12 Multi-fragment heuristic

12.1 Method principle

This method, also called “greedy algorithm” is very simple. The idea is to select edges accordingly to their respective length. All the edges of the complete graph with n vertices are sorted by length, shortest to longest. Then the shortest edge that will neither create a vertex with more than 2 edges (degree 2), nor a cycle with less than the total number of vertices is added to the tour. This is repeated until a cycle containing all of the vertices is obtained.

This approach can be summarized by the following algorithm:

Algorithm 1 Multi-fragment heuristic

Require: X : matrix of the vertex coordinates of dimension $n \times 2$

1: compute the euclidean distance matrix d
2: sort the set E of edges in increasing length
3: **for** each $e \in E$ **do**
4:   **if** (e is closing *tour* and size(*tour*) $< n$) or (one of end vertices of e is of degree 2) **then**
5:     go to the next edge
6:   **end if**
7:   **if** (e is closing *tour* and size(*tour*) $= n$) **then**
8:     add e to *tour*
9:     **return** *tour*
10:   **end if**
11:   add e to *tour*
12: **end for**

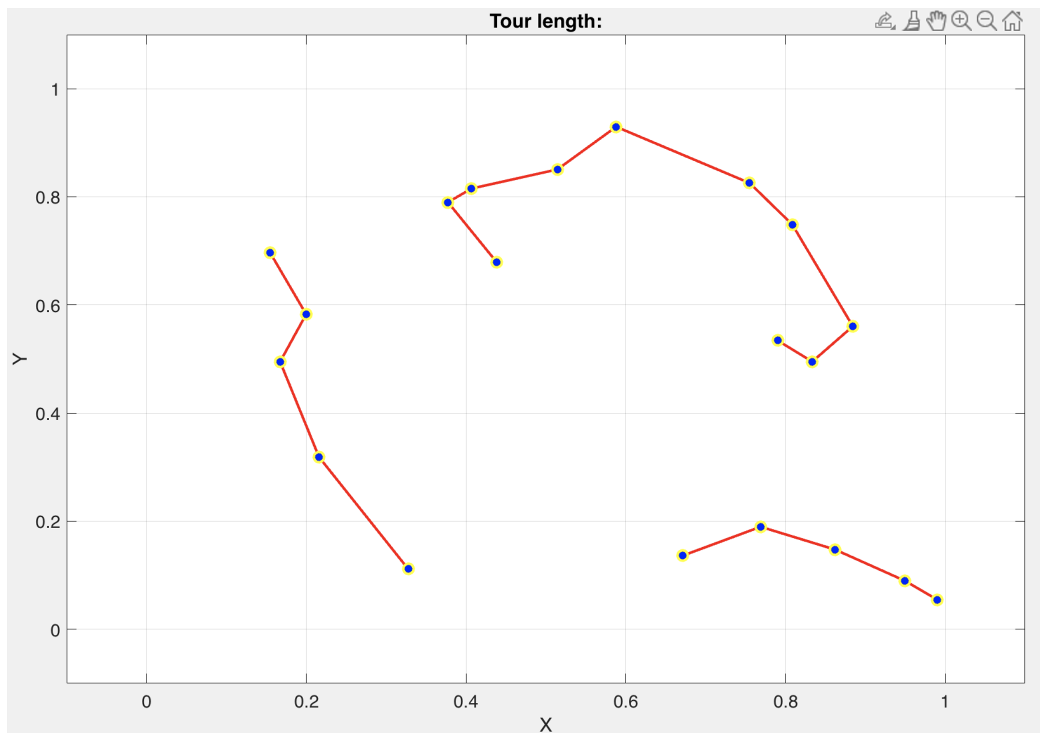


Figure 7: Multi-fragment heuristic in action

12.2 Matlab code

Main Program

```
%% TSP solving using multi-fragment heuristic (also called greedy algorithm)
%  Included in TSP20024 app
```

```
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

clear
close all
X = rand(40,2);
n = length(X);
D = squareform(pdist(X,'euclidean'));
% Changing the distance (i,i) from zero to infinity
D = D+diag(inf*ones(1,n));
plot(X(:,1),X(:,2),'kx','MarkerSize',8,'LineWidth',2)
hold on
% Graph creation with nodes only
G = graph;
G = addnode(G,n);
% Ordered distances (matrix with 3 columns : i,j,dist)
k = 0;
for i=1:n
    for j=i+1:n
        k = k+1;
        s(k,:) = [ i j D(i,j) ];
    end
end
[~,order] = sort(s(:,3),'ascend');
s = s(order,:);

listnode = [];         % List of nodes included in the tour
degrees = zeros(1,n); % Vertex degrees
minParent = 1:n;       % At initialization, each vertex belongs to its path

edge_idx = 1;          % Number of the edge to be examined

while sum(degrees) < 2*n  % Try until all the vertices have a degree equal 2
    % Edge to be inserted
    i = s(edge_idx,1);
    j = s(edge_idx,2);
    % If the edge is closing the tour and size(tour) < n or
    % the edge has a city already connected to two others go to the next
    % edge
    if (minParent(i)==minParent(j) && length(listnode) < n) || ...
        degrees(i)==2 || degrees(j)==2
        edge_idx = edge_idx+1;
        continue
    % If the edge is closing the tour and size(tour) = n
    elseif (minParent(i)==minParent(j) && length(listnode) == n)
        % If all nodes are in the same path add the last edge
        if length(unique(minParent))==1
            plot([X(i,1) X(j,1)],[X(i,2) X(j,2)],'r','LineWidth',2)
            G = addedge(G,i,j);
            degrees(i) = degrees(i)+1;
            degrees(j) = degrees(j)+1;
            edge_idx = edge_idx+1;
        % Else go to the next edge
        else
            edge_idx = edge_idx+1;
            continue
        end
```

```
    % For the other cases add the edge to the tour
    else
        plot([X(i,1) X(j,1)],[X(i,2) X(j,2)],'r','LineWidth',2)
        G = addedge(G,i,j);
        degrees(i) = degrees(i)+1;
        degrees(j) = degrees(j)+1;
        pause(0.4)
        if minParent(i)<minParent(j)
            minParent(minParent==minParent(j))=minParent(i);
        else
            minParent(minParent==minParent(i))=minParent(j);
        end
        if ~ismember(i,listnode)
            listnode = [listnode i];
        end
        if ~ismember(j,listnode)
            listnode = [listnode j];
        end
        edge_idx = edge_idx+1;
    end
end
tour=dfsearch(G,1);

% Closing of the polygon for length calculus
p = [tour ; tour(1)];
% Add title
L = sum(sqrt(diff(X(p,1)).^2 + diff(X(p,2)).^2));
str = sprintf('Tour Length: %g',L);
title(str)
```

12.3 Explanations

The *listnode* vector contains the list of the nodes included in the tour. Each time an edge is included in the tour, its end vertices are added to *listnode* provided they are not already there. The *minParent* vector is regularly updated to know if two vertices are part of the same path. At initialization, $minParent(i) = i$; each vertex is part of its own path. Then in order to know if two vertices belong to the same path, $minParent(i)$ is updated by the value of the smallest index of the vertices in the path involving vertex i.

If both end vertices of an edge belong to the same path, the edge is not added to the tour unless it is the last edge closing the Hamiltonian cycle, that is to say if all the vertices are already in the tour. As the final Hamiltonian cycle is only constructed by successive addition of edges, the sequence of vertices is obtained by performing a depth-first search of the graph thus constructed.

13 Insertion algorithms

13.1 Method principle [24]

Many straightforward heuristics have been suggested for the TSP. Among these are several that are classified as insertion algorithms. Due to their simplicity, these algorithms produce reasonable solutions to the TSP very quickly. The better ones often produce good solutions

(i.e. within 5% of optimality). This kind of method is also part of the greedy algorithms. An insertion algorithm constructs a feasible tour by successively adding one vertex to an existing subtour. Let $G(N, A)$ the graph composed of the set of vertices N and the set of edges E with a cost (distance) $d(i, j)$ associated with the edge $(i, j) \in E$. The general form of such algorithms is as follows:

1. Obtain a tour for a subset of the vertices $N' \subset N$.
2. Identify a vertex $r \in N - N'$ which is to be added to the existing subtour (**selection criterion**) .
3. Identify an edge (i, j) connecting vertices i and j in the subtour on N'. Insert vertex r between vertices i and j, and add r to N' (**insertion criterion**).
4. Stop if $N' = N$. Otherwise return to Step 2.

Steps 2 and 3 may be interchanged to read:

2. For each vertex $r \in N - N'$, identify an edge (i, j) connecting vertices i and j in the subtour on N' where r may best be inserted (**insertion criterion**).
3. Identify a vertex $r \in N - N'$ which is to be added to the existing subtour (**selection criterion**), and add it where it may best be inserted according to Step 2.

The heuristic elements in the above format include the choice of an initial subtour in Step 1, and the choices of selection and insertion criteria.

13.2 Selection of heuristic elements

Initial subtour. A single vertex selected at random can be chosen as the initial subtour. To produce better solutions, the chosen algorithm can be repeated using different vertices and the best solution over several trials is then used as the final solution.

When the TSP is Euclidean, better alternatives exist for the initial subtour. In particular, one can use the convex hull of the set of vertices N. Indeed, it has been shown that the optimal tour visits vertices on the boundary of the convex hull in the same order as if the boundary itself were followed.

Selection criterion. Many criteria have been suggested for the selection of the vertex to be next inserted in the current subtour. Among them, the most used are:

- Choosing the vertex r closest to any vertex of the current subtour (nearest insertion). Choose vertex $r \in N - N'$ and $i \in N'$ such that $d(i, r)$ is minimal.

- Choosing the vertex r farthest to any vertex of the current subtour (farthest insertion). Choose vertex $r \in N - N'$ and $i \in N'$ such that $d(i, r)$ is maximal.

- Choosing the vertex r that may be inserted at minimal increased cost (cheapest insertion). Choose vertex $r \in N - N'$ and $i, j \in N'$ such that $d(i, r) + d(r, j) - d(i, j)$ is minimal.

- Choosing the vertex r such that the proportional increase in cost is minimal (ratio insertion). Choose vertex $r \in N - N'$ and $i, j \in N'$ such that $(d(i, r) + d(r, j))/d(i, j)$ is minimal.

- Choosing the vertex r such that the product of ratio and distance is minimized. Choose vertex $r \in N - N'$ and $i, j \in N'$ such that $((d(i,r)+d(r,j))/d(i,j))(d(i,r)+d(r,j)-d(i,j))$ is minimal.

- Choosing the vertex $r \in N - N'$ and $i, j \in N'$ such that the angle formed by the two edges (i, r) and (r, j) is maximal (greatest angle).

We can also add to this list an approach based on Delaunay triangulation (DT):

- Choosing the vertex $r \in N - N'$ which has the lowest degree in the DT of vertices in N, breaking ties regarding the decreasing order of the sum of the lengths of all edges in DT that have that vertex as an endpoint.

and the Less Flexibility First (LFF) approach first introduced in solving the block placement and rectangle packing problems in automated VLSI design. This latter, based on the cheapest insertion, will be described in section 13.3.7.

These are just a few of many alternatives that might be employed as selection criteria.

Insertion criterion. Two criteria are mainly used:

- Cheapest insertion. Insert the vertex $r \in N - N'$ between those two connected vertices, $i, j \in N'$ that minimize the quantity $d(i,r) + d(r,j) - d(i,j)$.

- Insertion and selection criteria identical. Insert the vertex r that has been selected between those two vertices $i, j \in N'$ that caused this selection.

The cheapest insertion criterion can be used for selection as well as insertion criterion.

There are therefore many different algorithms depending on the combination (initial subtour, selection criterion and insertion criterion) chosen. They all offer fairly similar performances. Different approaches have been implemented of which only three are included in the TSP2024 Matlab App:

- (convex hull, cheapest insertion, ratio selection), named "Convex hull insertion" in the TSP2024 Matlab App (file name: TSPinsert_conv_hull.m)

- (single vertex at random, nearest selection, cheapest insertion), named "Cheapest insertion" in the TSP2024 Matlab App (file name: TSPinsert_cheapest.m)

- (convex hull, selection criteria: lowest degree in the DT of vertices in N, breaking ties regarding the decreasing order of the sum of the lengths of all edges in DT that have that vertex as an endpoint, insertion criteria: cheapest), named "Delaunay insertion" in the TSP2024 Matlab App (file name: TSPinsert_delaunay.m)

Two other insertion methods are provided in the downloadable file from Matlab File Exchange, but have not been included in the TSP2024 Matlab App:

- (convex hull, nearest insertion, nearest selection) (file name: TSPinsert_conv_hull_2.m)

- (convex hull, LFF insertion, LFF selection) (file name: TSPinsert_LFF.m)

13.3 Insertion from the convex hull method

13.3.1 Method principle

If H is the convex hull of the vertices in two-dimensional space, then the order in which the vertices on the boundary of H appear in the optimal tour will follow the order in which they appear in H. This observation serves as impetus for the convex hull heuristic procedure:

1. Form the convex hull of the set of vertices. The hull gives an initial subtour.
2. For each vertex r not yet contained in the subtour decide between which two vertices i and j on the subtour to insert vertex r. That is, for each such r, find (i, j) such that $d(i, r) + d(r, j) - d(i, j)$ is minimal (insertion criterion).
3. From all (i, r, j) found in Step 2, determine the (i^*, r^*, j^*) such that $(d(i^*, r^*) + d(r^*, j^*))/d(i^*, j^*)$ is minimal (selection criterion).
4. Insert vertex r^* in subtour between vertices i^* and j^*.
5. Repeat Steps 2 through 4 until a Hamiltonian cycle is obtained.

13.3.2 Matlab code

Main Program

```
%% TSP solving using the convex hull insertion method
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

% Must be followed by a 2-opt optimization for avoiding crossings

% Method class: insertion
% Initial subtour: convex hull
% Insertion criterion: cheapest
% Selection criterion: ratio

clear
close all
X=rand(60,2);
n = length(X);
x = X(:,1);
y = X(:,2);
% Computing and animated drawing of the convex hull
k = convhull(x,y);
plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
hold on
for i=2:length(k)
  plot([x(k(i-1)) x(k(i))],[y(k(i-1)) y(k(i))],'r','LineWidth',2)
  pause(0.2)
end

D = squareform(pdist(X,'euclidean'));
% List of the vertices not belonging to the tour: kbar
```

```
k(end) = []; % Remove the last point as it's a cycle
kbar = 1:n; kbar(k) = [];

% While all the vertices are not included in the cycle
while ~isempty(kbar)
    insert = zeros(length(kbar));
    % For all vertices in kbar
    for r=1:length(kbar)
        best_sol = inf;
        % For each vertex r not yet contained in the subtour decide between
        % which two nodes i and j on the subtour to insert node r.
        % That is, for each such r, find (i,j) such that
        % d(i,r) + d(r,j) - d(i,j) is minimal
        % Rather than working with the vertex numbers we will work with
        % indices of vectors k (current tour) and kbar (vertices to insert)
        % In that case, we have j=i+1
        l_k = length(k);
        % Increment in the length tour for all vertices pairs of indices i
        % and i+1
        inc = zeros(l_k,1);
        for indk=1:l_k-1
            inc(indk) = ...
               D(k(indk),kbar(r))+D(kbar(r),k(indk+1))-D(k(indk),k(indk+1));
        end
        % Increment when inserting r between the last and the first vertices
        inc(l_k) = D(k(l_k),kbar(r))+D(kbar(r),k(1))-D(k(l_k),k(1));
        % Search for minimal increment; i is the index in k
        [mininc,i] = min(inc);
        insert(r) = i;
    end
    ratio = inf;
    % Choice of the vertex r to insert
    % From all (i,r,j) found previously, determine the (i*,r*,j*) such
    % that (d(i*,r*)+ d(r*,j*))/d(i*,j*) is minimal.
    for r=1:length(kbar)
        i = insert(r);
        j = i+1;
        if i==length(k)
            j=1;
        end
        newratio = (D(k(i),kbar(r))+D(kbar(r),k(j))) / D(k(i),k(j));
        if newratio < ratio
            ratio = newratio;
            ind = r;
        end
    end

    i = insert(ind);
    ip1=i+1;
    % If it's the last point, the next one is the first
    if i==length(k)
       ip1=1;
    end

    % Erasure of the edge i ; i+1 and point vizualisation
    plot([x(k(i)) x(k(ip1))],[y(k(i)) y(k(ip1))],'w','LineWidth',2)
    plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
    % Insertion of the new vertex
```

```
    k = [k(1:i); kbar(ind) ; k(i+1:end)];
    % Update of the list of vertices to be treated
    kbar(ind) = [];
    pause(0.2)
    % Closing of the polygon for length calculus and plot
    p = [k ; k(1)];
    plot(x(p),y(p),'r','LineWidth',2)
    % Add title
    L = sum(sqrt(diff(x(p)).^2 + diff(y(p)).^2));
    str = sprintf('Tour Length: %g',L);
    title(str)
end
```

13.3.3 Explanations

The algorithm begins by finding the convex polygon such that all the vertices are inside the polygon. There is a Matlab function performing this processing: `convhull`. The vertices are originally numbered from 1 to n and the polygon is described by a vector indicating their order. The vertices are separated into vertices of the polygon (vector of index k ordered according to the route) and vertices not belonging to the polygon (vector of index $\overline{k}$).

Rather than working with the vertex numbers, we will work with indices of vectors k (current tour) and $\overline{k}$ (vertices to insert). In that case, for all $\overline{k}(r)$, we search $k(i)$ such that the increase in tour length $d(k(i), \overline{k}(r)) + d(\overline{k}(r), k(i+1)) - d(k(i), k(i+1))$ is minimal. Indeed, the insertion must be done between two consecutive vertices in k. This information is stored in the vector `insert`. Then for all $\overline{k}(r)$, the proportional increase in tour length $(d(k(i), \overline{k}(r)) + d(\overline{k}(r), k(i+1)))/d(k(i), k(i+1))$ is computed and the vertex $\overline{k}(r)$ with the minimal increase is inserted between $k(i)$ and $k(i+1)$.

For the graphic animation, the edge $(k(i), k(i+1))$ is erased and edges $(k(i), \overline{k}(r))$ and $(\overline{k}(r), k(i+1))$ are drawn.

Let us denote $\ell = \text{card}(k)$. A difficulty arises in the management of the indices when it is necessary to insert the vertex $\overline{k}(r)$ after the vertex $k(\ell)$ in the sequence. Since the polygon is closed, this means that the vertex $\overline{k}(r)$ must be inserted between the vertex with index ℓ and the vertex with index 1 in k. Consequently, if $i = \ell$, the index $i+1$ must be replaced by 1.

13.3.4 First alternative approach

A similar procedure can be implemented based on nearest selection and insertion criteria:

1. Form the convex hull of the set of vertices. The hull gives an initial subtour.

2. Chose the vertex j not yet contained in the subtour which is nearest of a vertex of the subtour ; let i that vertex (selection criterion).

3. If $d(i-1, j) + d(j, i) < d(i, j) + d(j, i+1)$ insert vertex r between $i-1$ and i else insert r between i and $i+1$ (insertion criterion).

4. Repeat Steps 2 through 3 until a Hamiltonian cycle is obtained.

13.3.5 Matlab code

Main program

```
%% TSP solving using the convex hull insertion method
%  Not included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

% Must be followed by a 2-opt optimization for avoiding crossings

% Method class: insertion
% Initial subtour: convex hull
% Insertion criterion: nearest
% Selection criterion: nearest

clear
close all
X=rand(80,2);
n = length(X);

x = X(:,1);
y = X(:,2);
% Computing and animated drawing of the convex hull
k = convhull(x,y);

plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
hold on
for i=2:length(k)
  plot([x(k(i-1)) x(k(i))],[y(k(i-1)) y(k(i))],'r','LineWidth',2)
  pause(0.2)
end

% List of the points not belonging to the polygon: kbarre
k(end) = []; % Remove the last point as it's a cycle
kbarre = 1:n; kbarre(k) = [];

% For all points in kbarre
for kb=1:length(kbarre)
   % Search the point of kbarre nearest a point of the polygon
   Xkbarre = X(kbarre,:); Xk = X(k,:);
   D = pdist2(Xkbarre,Xk,'euclidean');
   minimum = min(min(D));
   % j : index of the point in kbarre
   % i : index of the point in k
   [j,i]=find(D==minimum,1);
   num_p_close = kbarre(j);
   im1=i-1; ip1=i+1;
   % If it's the first point, the previous one is the last
   if i==1
       im1=length(k);
   end
   % If it's the last point, the next one is the first
   if i==length(k)
       ip1=1;
   end
```

```
   im1_i = pdist([x(k(im1))      y(k(im1))      ; x(k(i))        y(k(i))]);
   im1_j = pdist([x(k(im1))      y(k(im1))      ; x(num_p_close) ...
       y(num_p_close)]);
   i_ip1 = pdist([x(k(i))        y(k(i))        ; x(k(ip1))      y(k(ip1))]);
   j_ip1 = pdist([x(num_p_close) y(num_p_close) ; x(k(ip1))      y(k(ip1))]);
   i_j   = pdist([x(k(i))        y(k(i))        ; x(num_p_close) ...
       y(num_p_close)]);

   l1 = im1_i + i_j + j_ip1; % segment i-1 ; i ; j ; i+1
   l2 = im1_j + i_j + i_ip1; % segment i-1 ; j ; i ; i+1
   if l1 > l2
          % Erasure of the edge i-1; i and point vizualisation
          plot([x(k(im1)) x(k(i))],[y(k(im1)) y(k(i))],'w','LineWidth',2)
          plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
          k = [k(1:i-1); num_p_close ; k(i:end)];
   else
          % Erasure of the edge i ; i+1 and point vizualisation
          plot([x(k(i)) x(k(ip1))],[y(k(i)) y(k(ip1))],'w','LineWidth',2)
          plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
          k = [k(1:i); num_p_close ; k(i+1:end)];
   end

   % Update of the list of points to be treated
   kbarre(j) = [];
   % Closing of the polygon for length calculus and plot
   p = [k ; k(1)];
   plot(x(p),y(p),'r','LineWidth',2)
   % Add title
   L = sum(sqrt(diff(x(p)).^2 + diff(y(p)).^2));
   str = sprintf('Tour Length: %g',L);
   title(str)
   pause(0.2)
end
```

13.3.6 Explanations

As previously, the vertices are separated into vertices of the polygon (vector of index k ordered according to the route) and vertices not belonging to the polygon (vector of index $\overline{k}$). Rather than working with the vertex numbers, we will work with indices of vectors k (current tour) and $\overline{k}$ (vertices to insert). So the vertex $j \in \overline{k}$ is the closest of the polygon vertex $k(i)$.

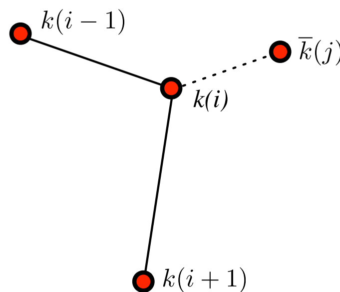


Figure 8: Initial route

Two situations are then to be considered according to the sum of the lengths of the edges connecting $\overline{k}(j)$ to the vertices of index $i-1$ and $i+1$ in k. If $d(k(i-1), \overline{k}(j)) + d(\overline{k}(j), k(i)) + d(k(i), k(i+1)) < d(k(i-1), k(i)) + d(k(i), \overline{k}(j)) + d(\overline{k}(j), k(i+1))$ then we retain the first solution (figure 9, left) otherwise we retain the second (figure 9, right).

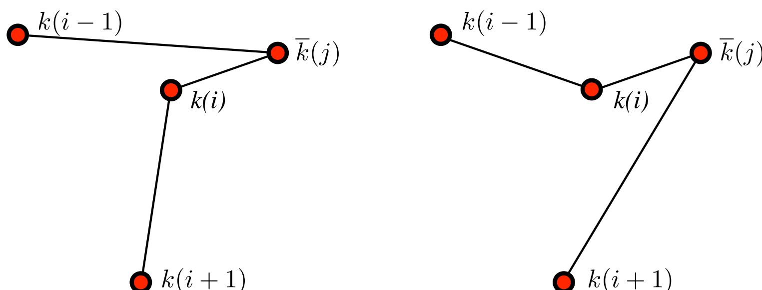


Figure 9: Two possible insertions

In the first case, the sequence of vertices of the polygon is modified by inserting the vertex $\overline{k}(j)$ between the vertices of index $i-1$ and i. For the graphic animation, we delete the edge $(k(i-1), k(i))$ and we add the edges $(k(i-1), \overline{k}(j))$ and $(\overline{k}(j), k(i))$. In the second case, the sequence of vertices of the polygon is modified by inserting the vertex j between the vertices of index i and $i+1$ and for the graphic animation, we delete the edge $(k(i), k(i+1))$ and we add the edges $(k(i), \overline{k}(j))$ and $(\overline{k}(j), k(i+1))$.

13.3.7 Second alternative approach

The solution proposed by Sheqin Dong *et al.* [10] is inspired and generalize the so-called "Less Flexibility First" (LFF) principle first introduced in solving the block placement and rectangle packing problems in automated VLSI design. As mentioned by the authors, this heuristic is closely related to Cheapest Insertion from Convex Hull and provide a tour which cannot be worse than this method. This improvement, however, comes at the expense of calculation time because it explores a much larger number of solutions.

The authors define what they called the "flexibility" of a vertex which corresponds to the quantity used for the cheapest insertion criterion. The flexibility of vertex r is the smallest increase in the length of the tour caused by the insertion of this vertex between two consecutive vertices of the current tour T.

The proposed approach is summarized by the two following algorithms:

LFF algorithm: Less Flexibility First construction heuristic

Input : a TSP instance

Output : a Hamiltonian tour

1. Form the convex hull of the set of vertices. The hull gives an initial subtour T.
2. For each vertex r not yet contained in the subtour, calculate $FFV(r, T)$.

3. Insert the vertex with least FFV and update the current partial tour T
4. Go to Step 2 until no vertices left.

FFV algorithm: Fitness cost Function Value

Input : a vertex r and a partial tour T

Output : fitness value of r

1. Perform the cheapest insertion of r in T
2. For each vertex s not yet contained in the subtour T add them to the tour using the cheapest insertion approach.
3. Repeat Step 2 until a Hamiltonian cycle is obtained.
4. Return the length of the obtained Hamiltonian tour (fitness value of vertex r).

The selection criterion for the vertex to be inserted is here less "local" than the first insertion method. The vertex to be inserted is that which will lead to the smallest length of the tour, once all the vertices will be inserted. In order to be consistent, the insertion criterion remains the cheapest insertion criterion.

13.3.8 Matlab code

Main program

```
%% TSP solving using LFF principle
%  Not included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

% Inspired by:
% Sheqin Dong, Fan Guo, Jun Yuan, Rensheng Wang, Xianlong Hong
% A novel tour construction heuristic for Traveling
% Salesman Problem using LFF principle
% Proceedings of 9th Joint International Conference on Information Sciences,
% JCIS-06, p. 433-436, Atlantis Press. doi:10.2991/jcis.2006.214

% Method class: insertion
% Initial subtour: convex hull
% Insertion criterion: Less Flexibility First
% Selection criterion: Less Flexibility First
close all
X=rand(80,2);
n = length(X);

% Step 1: Generate a partial tour
x = X(:,1); y = X(:,2);
% Computing and animated drawing of the convex hull
k = convhull(x,y);

plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
hold on
```

```
for i=2:length(k)
  plot([x(k(i-1)) x(k(i))],[y(k(i-1)) y(k(i))],'r','LineWidth',2)
end
pause(0.2)
D = squareform(pdist(X,'euclidean'));

% List of the vertices not belonging to the tour: kbar
k(end) = []; % Remove the last point as it's a cycle
kbar = 1:n; kbar(k) = [];

% While all the vertices are not included in the cycle
while ~isempty(kbar)
    % Step 2: For each vertex c not included in the tour,
    % calculate FFV(k,c,X,D);
    tab_FFV = zeros(length(kbar),2);
    for i=1:length(kbar)
        tab_FFV(i,:) = FFV(k,kbar(i),X,D);
    end

    % Step 3: Insert the vertex with least FFV and update the tour
    [~,r] = min(tab_FFV(:,1));
    i = tab_FFV(r,2);
    % vertex to insert: kbar(r)
    % where to insert: between k(i) et k(i+1)
    ip1=i+1;
    % If it's the last point, the next one is the first
    if i==length(k)
        ip1=1;
    end

    % Erasure of the edge i ; i+1 and point vizualisation
    plot([X(k(i),1) X(k(ip1),1)],[X(k(i),2) X(k(ip1),2)],'w','LineWidth',2)
    plot(X(:,1),X(:,2),'kx','MarkerSize',12,'LineWidth',2)
    % Insertion of the new vertex
    k = [k(1:i); kbar(r) ; k(i+1:end)];
    % Update of the list of vertices to be treated
    kbar(r) = [];
    % Closing of the polygon for drawing
    p = [k ; k(1)];
    plot(X(p,1),X(p,2),'r','LineWidth',2)
    % Add title
    L = tour_length(X,k);
    str = sprintf('Tour Length: %g',L);
    title(str)
    pause(0.2)
end
```

Functions

```
function outFFV = FFV(tour,r,X,D)
n = length(X);
l_tour = length(tour);
% Increment in the length tour for all vertices pairs of indices i
% and i+1
inc = zeros(l_tour,1);
for i=1:l_tour-1
    inc(i) = D(tour(i),r)+D(r,tour(i+1))-D(tour(i),tour(i+1));
end
```

```
% Increment when inserting between the last and the first vertices
inc(l_tour) = D(tour(l_tour),r)+D(r,tour(1))-D(tour(l_tour),tour(1));
% Search for minimal increment; i is the index in k~
[~,insert] = min(inc);
tour = [tour(1:insert); r ; tour(insert+1:end)];

% Second step: Perform an LFFM inserting a city not included
% in the current partial tour
% Third step: Goto step 2 until no cities left
k = tour;
% List of the vertices not belonging to the tour: kbar
kbar = 1:n; kbar(k) = [];

% While all the vertices are not included in the cycle
while ~isempty(kbar)
    % Best_sol initialization [index in kbarre  length_increment index in k ...
        for insertion]
    best_sol = [0 inf 0];
    % For all vertices in kbarre
    for r=1:length(kbar)
        % For each vertex r not yet contained in the subtour decide between
        % which two nodes i and j on the subtour to insert node r.
        % That is, for each such r, find (i,j) such that
        % d(i,r) + d(r,j) - d(i,j) is minimal
        % Rather than working with the vertex numbers we will work with
        % indices of vectors k (current tour) and kbar (vertices to insert)
        % In that case, we have j=i+1
        l_k = length(k);
        % Increment in the length tour for all vertices pairs of indices i
        % and i+1
        inc = zeros(l_k,1);
        for indk=1:l_k-1
            inc(indk) = ...
                D(k(indk),kbar(r))+D(kbar(r),k(indk+1))-D(k(indk),k(indk+1));
        end
        % Increment when inserting between the last and the first vertices
        inc(l_k) = D(k(l_k),kbar(r))+D(kbar(r),k(1))-D(k(l_k),k(1));
        % Search for minimal increment; i is the index in k
        [mininc,i] = min(inc);
        if mininc < best_sol(2)
            best_sol = [r mininc i];
        end
    end
    r = best_sol(1);
    i = best_sol(3);
    k = [k(1:i); kbar(r) ; k(i+1:end)];
    % Update of the list of points to be treated
    kbar(r) = [];
end
%Fourth step: Return the fitness cost function value
L = tour_length(X,tour);
outFFV = [L insert];
end
```

```
function tlength = tour_length(X,tour)
    tour = [tour ; tour(1)];
    tlength = sum(sqrt(diff(X(tour,1)).^2+diff(X(tour,2)).^2));
end
```

13.3.9 Explanations

The function `FFV` computes the length of a Hamiltonian tour and the index `insert` indicating the insertion position of the vertex r (between the vertices of index `insert` and `insert+1` in k) on the basis of a current tour `tour` and a vertex r to be inserted first. These two values are combined in the vector `outFFV`.

The main program implements the LFF algorithm. As long as not all vertices are part of the cycle, the fitness cost value is computed for these vertices. The vertex with the smaller fitness cost value is then inserted in the polygon.

13.4 Insertion from a random vertex – cheapest insertion

13.4.1 Method principle

The algorithm begins with the random choice of a vertex and the initial polygon is constructed with the closest vertex. Selection and insertion criteria are cheapest.

13.4.2 Matlab code

Main program

```
%% TSP solving using the cheapest insertion method
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

% Must be followed by a 2-opt optimization for avoiding crossings

% Method class: insertion
% Initial subtour: random vertex
% Insertion criterion: cheapest
% Selection criterion: cheapest

clear
close all
X=rand(60,2);
n = length(X);
% Computation of the distance matrix
D = squareform(pdist(X,'euclidean'));
x = X(:,1); y = X(:,2);
plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
hold on
% Starting point and first polygon
rnd_vertex = randi(n);
k = rnd_vertex;
D(rnd_vertex,:) = inf;
```

```
[~,s] = min(D(:,rnd_vertex));
k(2) = s;

% Polygon plot
plot(x(k),y(k),'r','LineWidth',2)
pause(0.4)

D = squareform(pdist(X,'euclidean'));
% List of the vertices not belonging to the tour: kbar
kbar = 1:n; kbar(k) = [];

% While all the vertices are not included in the cycle
while ~isempty(kbar)
    % Best_sol initialization
    % [index in kbarre    length_increment   index in k for insertion]
    best_sol = [0 inf 0];
    % For all vertices in kbarre
    for r=1:length(kbar)
        % For each vertex r not yet contained in the subtour decide between
        % which two nodes i and j on the subtour to insert node r.
        % That is, for each such r, find (i,j) such that
        % d(i,r) + d(r,j) - d(i,j) is minimal
        % Rather than working with the vertex numbers we will work with
        % indices of vectors k (current tour) and kbar (vertices to insert)
        % In that case, we have j=i+1
        l_k = length(k);
        % Increment in the length tour for all vertices pairs of indices i
        % and i+1
        inc = zeros(l_k,1);
        for indk=1:l_k-1
            inc(indk) = ...
                D(k(indk),kbar(r))+D(kbar(r),k(indk+1))-D(k(indk),k(indk+1));
        end
        % Increment when inserting between the last and the first vertices
        inc(l_k) = D(k(l_k),kbar(r))+D(kbar(r),k(1))-D(k(l_k),k(1));
        % Search for minimal increment; i is the index in k
        [mininc,i] = min(inc);
        if mininc < best_sol(2)
            best_sol = [r mininc i];
        end
    end
    r = best_sol(1);
    i = best_sol(3);
    ip1=i+1;
    % If it's the last point, the next one is the first
    if i==length(k)
       ip1=1;
    end
    % Erasure of the edge i ; i+1 and point vizualisation
    plot([x(k(i)) x(k(ip1))],[y(k(i)) y(k(ip1))],'w','LineWidth',2)
    plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
    k = [k(1:i); kbar(r) ; k(i+1:end)];
    % Update of the list of points to be treated
    kbar(r) = [];
    % Closing of the polygon for length calculus and plot
    p = [k ; k(1)];
    plot(x(p),y(p),'r','LineWidth',2)
    % Add title
```

```
    L = sum(sqrt(diff(x(p)).^2 + diff(y(p)).^2));
    str = sprintf('Tour Length: %g',L);
    title(str)
    pause(0.4)
end
```

13.5 Insertion based on Delaunay triangulation

13.5.1 Method principle

As mentioned previously, the initial subtour is the convex hull of the vertices; selection criteria is the lowest degree in the Delaunay graph of all vertices, breaking ties regarding the decreasing order of the sum of the lengths of all edges in Delaunay graph that have that vertex as an endpoint; insertion criteria is cheapest.

Let us begin by a definition. A Delaunay triangulation of a set of points $\{p_1, \dots p_n\}$ in the plane subdivides their convex hull into triangles whose circumcircles do not contain any of the points. As a consequence, this maximizes the size of the smallest angle in any of the triangles (see figure 10).

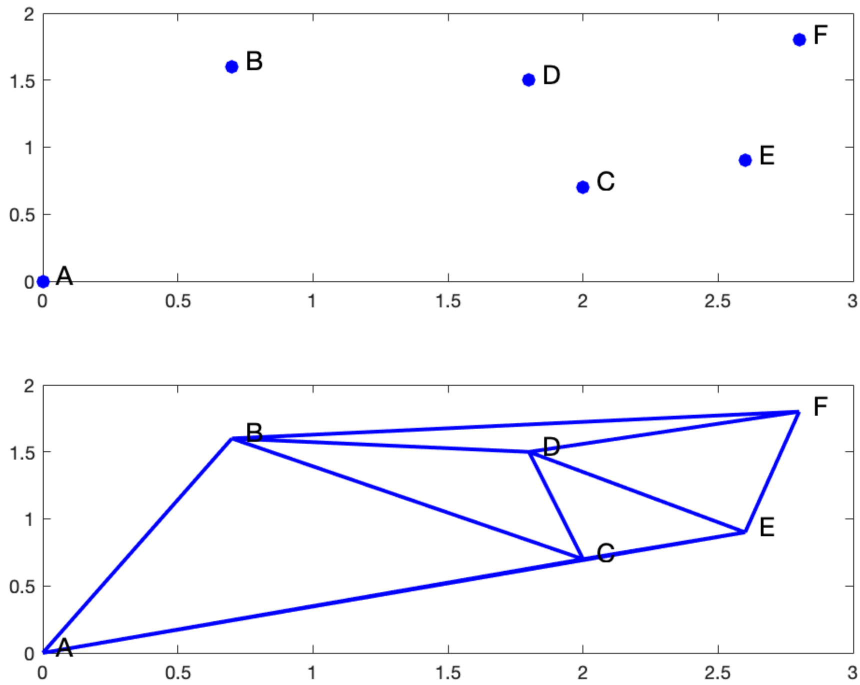


Figure 10: Delaunay triangulation of 6 points in $\mathbb{R}^2$

The property of this triangulation can be verified on the example. Considering the triangle A, B, C, the corresponding circumcircle with center $O1$ is drawn on figure 11-left: it contains no other points. Figure 11-right shows the same behavior for the triangle C, D, E and its circumcircle centered in $O2$.

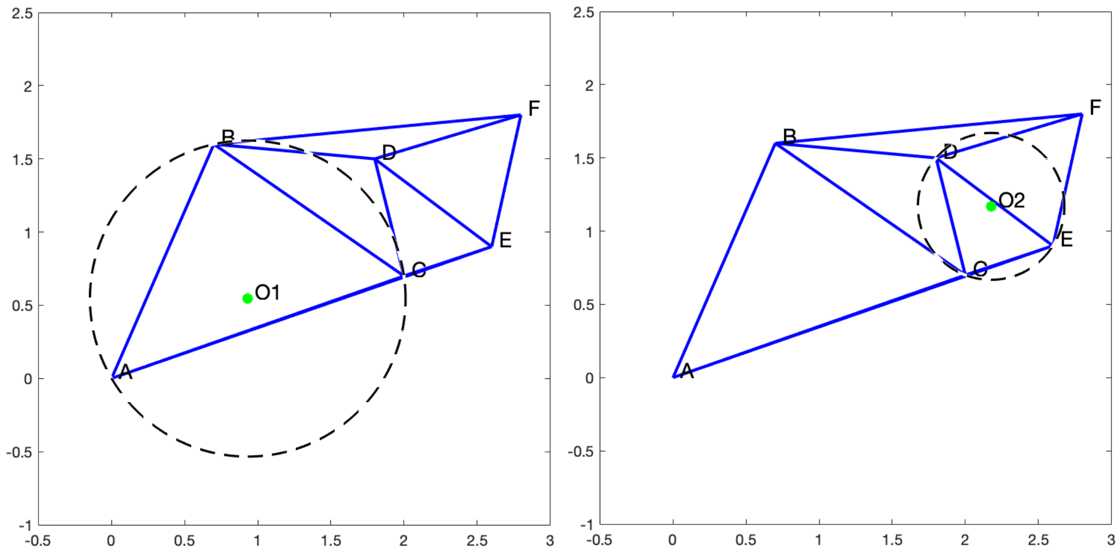


Figure 11: Circumcircles for two triangles

It has been shown [9] that the optimal tour for the Euclidean symmetric TSP is not necessary a subgraph of the Delaunay graph. However, the percentage of edges which belong to the optimal tour and are not present in the Delaunay graph is very low. This indicates that the Delaunay graph has some good information about the candidate set of possible edges to be considered as being part of the optimal tour.

The authors of the method [19] propose to select and insert first the vertices which have low degrees in the Delaunay graph. It corresponds to a farthest insertion criterion. Of course, there are many vertices with the same number of neighbours in the Delaunay graph so another criterion to derandomize the selection is needed. For each vertex, the sum of the length of all edges in Delaunay graph that have that vertex as an endpoint is computed. The authors have tested two different criteria, both of them based on the use of this sum. The insertion is done either in the descending order of that sum or in ascending order. In conclusion of their work, they recommend using the insertion criterion based on descending order (even if the results differ relatively little when choosing ascending order).

13.5.2 Matlab code

Main program

```
%% TSP solving using the one city insertion from Delaunay triangulation method
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

% Must be followed by a 2-opt optimization for avoiding crossings

% Inspired by:
% Natalio Krasnogor, Pablo Moscato, Michael G. Norman
% A New Hybrid Heuristic For Large Geometric Traveling Salesman Problems
% Based On The Delaunay Triangulation
```

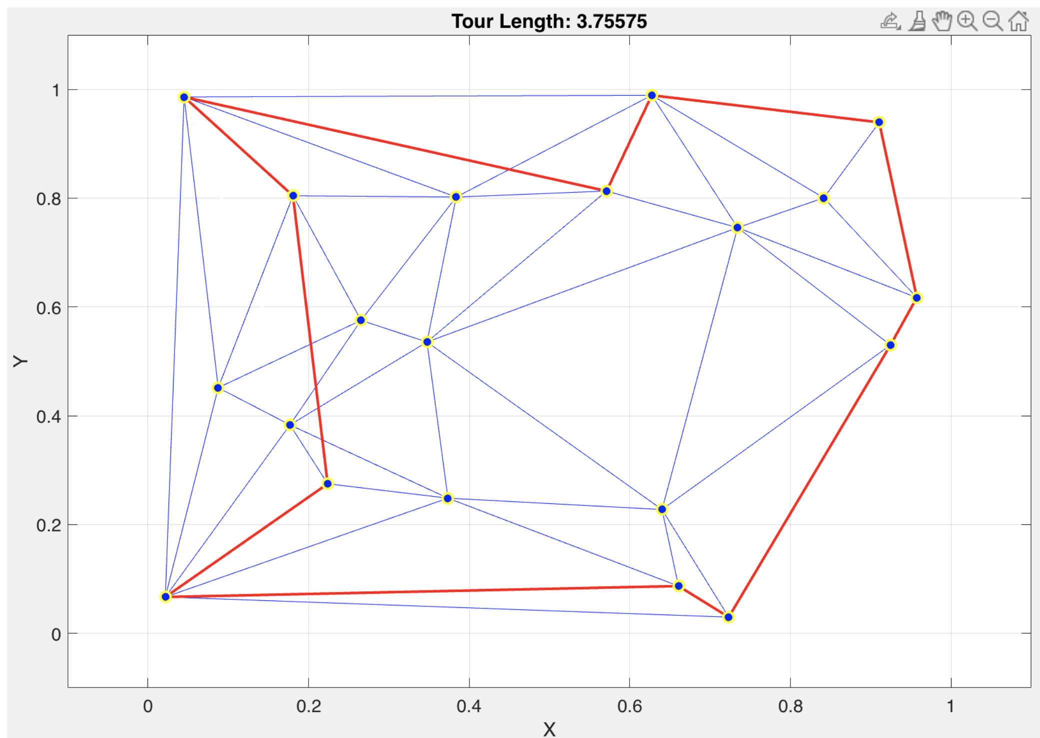


Figure 12: Delaunay insertion in progress

```
% Anales del XXVII Simposio Brasileiro de Pesquisa Operacional, November 1995

% Method class: insertion
% Initial subtour: convex hull
% Selection criterion: min degree in the Delaunay triangulation and max
% sum of lengths of all edges in DT that have that vertex as an endpoint
% Insertion criterion: cheapest

clear
close all
%rng(2)
X=rand(80,2);
D = squareform(pdist(X,'euclidean'));
[n,~] = size(X);

x = X(:,1);
y = X(:,2);
% Computing and animated drawing of the convex hull
k = convhull(x,y);

plot(x,y,'kx','MarkerSize',8,'LineWidth',2)
hold on
plot(x(k),y(k),'r','LineWidth',2)
pause(0.5)

k(end) = []; % Remove the last point as it's a cycle

% Delaunay triangulation and initialization
```

```
DT = delaunayTriangulation(X);
sum_edge_length = zeros(n,1);
vertex_degree = zeros(n,1);
for n_vertex = 1:n
    % Return the indices in the Connectivity list of triangles attached
    % to vertex n_vertex
    V = vertexAttachments(DT,n_vertex);
    ind = cell2mat(V);
    % Select the corresponding rows in the Connectivity list
    DT1 = DT.ConnectivityList(ind,:);
    listedge = [];
    [nl,~] = size(DT1);
    % Transform the triangles (Connectivity list) in a list of edges
    for i=1:nl
        if DT1(i,1) == n_vertex
            listedge = [listedge ; DT1(i,[1 2])];
            listedge = [listedge ; DT1(i,[1 3])];
        elseif DT1(i,2) == n_vertex
            listedge = [listedge ; DT1(i,[2 1])];
            listedge = [listedge ; DT1(i,[2 3])];
        elseif DT1(i,3) == n_vertex
            listedge = [listedge ; DT1(i,[3 1])];
            listedge = [listedge ; DT1(i,[3 2])];
        end
    end
    % Eliminate duplicates
    listedge = unique(listedge,'rows');

    % Compute the degree of each vertex
    vertex_degree(n_vertex) = length(listedge);

    % Compute the sum of lengths of edges connected to vertex n_vertex
    for i=1:vertex_degree(n_vertex)
        sum_edge_length(n_vertex) = ...
            sum_edge_length(n_vertex)+D(listedge(i,1),listedge(i,2));
    end
end
% Decision table with
% the first column as vertex_degree and
% the second as sum of length of edges connected to the vertex
Tab_decision = [vertex_degree -sum_edge_length];
% Minus sign on sum_edge_length for choosing the largest sum (H1 criterion)
[~,ordered_insert]=sortrows(Tab_decision,[1 2],'ascend');
% Eliminate the vertices of the convhull
for i=1:length(k)
    ordered_insert(find(ordered_insert==k(i)))=[];
end
% For all points not belonging to the convex hull in the order of the selection
for v=1:length(ordered_insert)
    r = ordered_insert(v);
    l_k = length(k);
    % Increment in the length tour for all vertices pairs of indices i
    % and i+1
    inc = zeros(l_k,1);
    for indk=1:l_k-1
        inc(indk) = D(k(indk),r)+D(r,k(indk+1))-D(k(indk),k(indk+1));
    end
    % Increment when inserting between the last and the first vertices
```

```
    inc(l_k) = D(k(l_k),r)+D(r,k(1))-D(k(l_k),k(1));
    % Search for minimal increment; i is the index in k
    [mininc,i] = min(inc);
    ip1=i+1;
    % If it's the last point, the next one is the first
    if i==length(k)
       ip1=1;
    end
    % Erasure of the edge i ; i+1 and point vizualisation
    plot([x(k(i)) x(k(ip1))],[y(k(i)) y(k(ip1))],'w','LineWidth',2)
    plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
    k = [k(1:i); r ; k(i+1:end)];
    % Closing of the polygon for length calculus and plot
    p = [k ; k(1)];
    plot(x(p),y(p),'r','LineWidth',2)
    % Add title
    L = sum(sqrt(diff(x(p)).^2 + diff(y(p)).^2));
    str = sprintf('Tour Length: %g',L);
    title(str)
    pause(0.4)
end
```

13.6 Explanations

Matlab is able to compute a Delaunay triangulation (function `delaunayTriangulation`). The output of the function comprises the so-called `ConnectivityList`. For a 2D triangulation, each row specifies a triangle (list of three vertices). The Matlab function `vertexAttachments` gives the indices in the `ConnectivityList` of the triangles attached to a particular vertex. Once these triangles identified, an exhaustive list of the edges constituting these triangles is established from the `ConnectivityList`. Of course, this list comprises duplicate edges that are eliminated using the Matlab function `unique`. For each vertices, we therefore have the list of its incident edges. It is then easy to determine the degree of each vertex as well as the sum of the lengths of its incident edges.

The matrix `Tab_decision` brings together these two pieces of information (in columns). The rows of this matrix are then sorted according to the ascending order of their degree and the descending order of the sum of the lengths of the edges. The rows corresponding to vertices belonging to the convex hull are then canceled. The vector `ordered_insert` then contains the ordered list of vertices to insert.

These vertices are then inserted, one by one, into the Hamiltonian cycle respecting the cheapest insertion criterion (choice of two consecutive vertices in the cycle such that the insertion of the vertex considered between these two vertices causes the smallest increase in the length of the cycle).

14 Minimum-weight spanning-tree

14.1 Method principle

If the lengths of the edges respect the triangle inequality, a minimum-weight spanning-tree can be built thanks to the Prim's algorithm. Depending on its implementation, the complexity of the algorithm varies. The best implementation (using binary heaps) has a complexity of $O(n^2 \log n)$. The cycle is obtained by performing a depth-first search of the tree as in the figure 13.

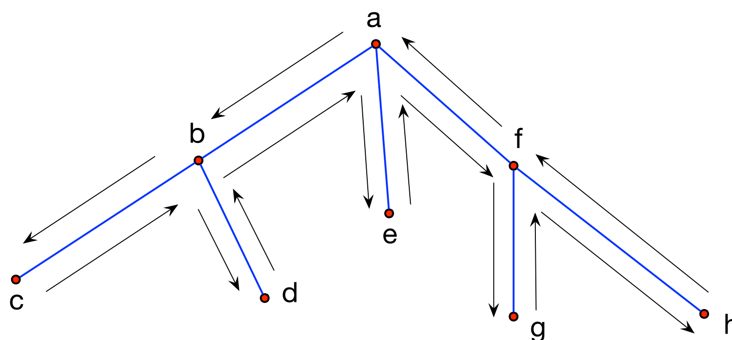


Figure 13: Depth-first search

If vertex marking is not used, a vertex with k children will appear $k+1$ times in the path. For the example considered, the path is abcbdbaeafgfhfa. Each edge is thus traversed twice: once in each "direction". Thus, the length of the path is exactly twice the weight of the spanning tree. Now, let's delete the vertices appearing more than once, keeping only their first occurrence in the path, then deleting the following ones. The path given as an example becomes: abcdefgh. If we add an occurrence of the first vertex at the end, we obtain a Hamiltonian cycle. By doing so, the cycle length can only be less than twice the weight of the spanning tree. Indeed, the triangle inequality guarantees that the length of the path does not increase when we remove the recurrent vertices.

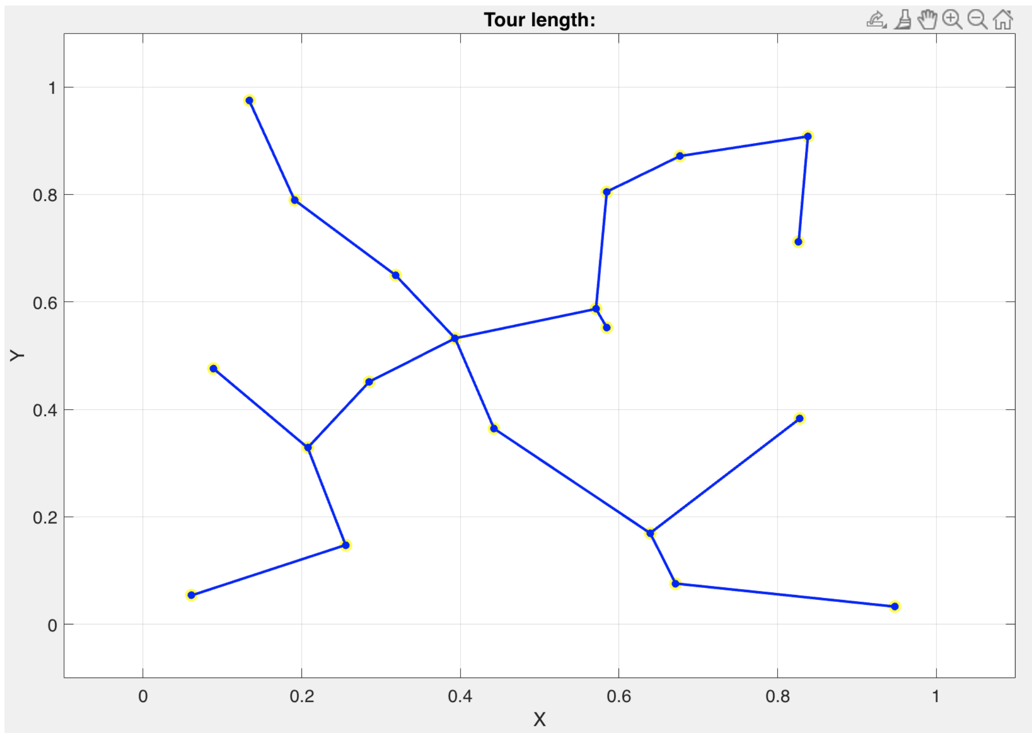


Figure 14: Minimum-weight spanning-tree

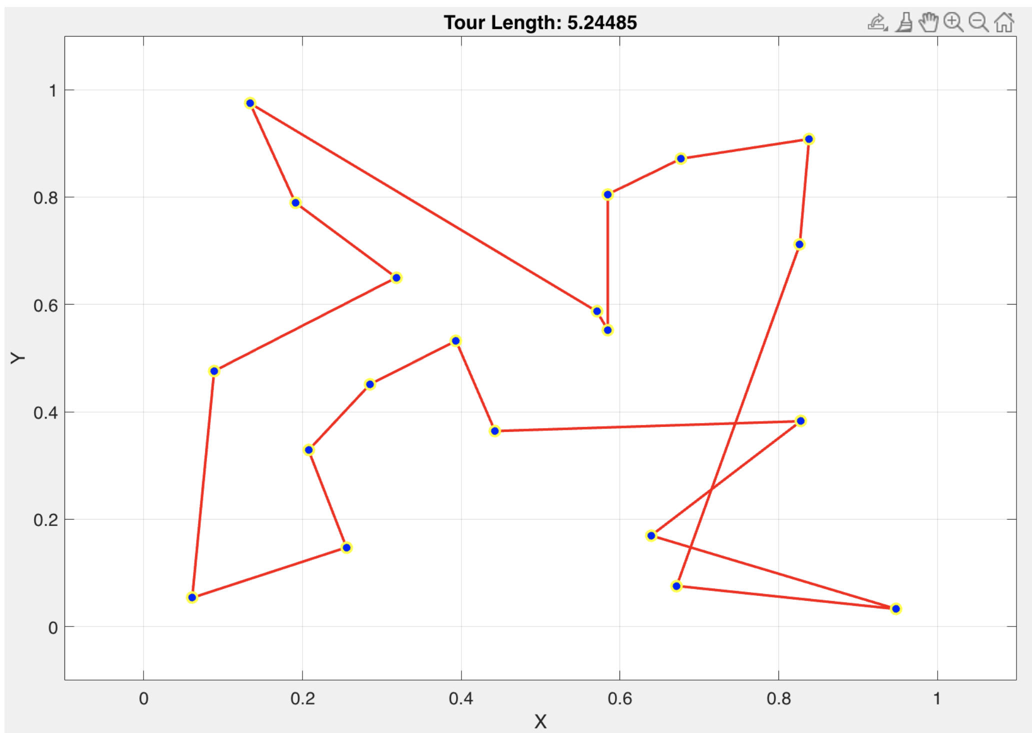


Figure 15: Cycle based on the previous tree

14.2 Matlab code

Main program

```
%% TSP solving using depth first search of the minimum weight spanning tree
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

% Must be followed by a 2-opt optimization for avoiding crossings

clear
close all
X = rand(50,2);
n = length(X);
x = X(:,1); y = X(:,2);
% Creating the source and target node lists
s=[];t=[];
for i=1:n-1
    s=[s i*ones(1,n-i)];
    t=[t i+1:n];
end
dist = pdist(X);
% Graph creation and minimum spanning tree search
G = graph(s,t,dist);
tree = minspantree(G);
edges = tree.Edges.EndNodes;
% Graph route using depth first search (from node 1)
p = dfsearch(tree,1);
k = [p ; 1];
plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
hold on
% Tree plot
for i=1:length(edges)
    plot([x(edges(i,1)) x(edges(i,2))],[y(edges(i,1)) y(edges(i,2))],'b ...
        ','LineWidth',2)
end
pause(0.5)
cla
plot(x,y,'kx','MarkerSize',12,'LineWidth',2)
% Polygon plot
plot(x(k),y(k),'r','LineWidth',2)

% Add title
L = sqrt(diff(x(k)).^2 + diff(y(k)).^2);
str = sprintf('Tour Length: %g',sum(L));
title(str)
```

14.3 Explanations

Matlab has functions managing graphs. In particular, for a given graph described by two vectors containing the numbers of the source vertices and destination vertices, the `minspantree` function creates the spanning tree of minimum weight (where here the weights correspond to the distances between vertices). Depth-first search of the tree can then be done with the `dfsearch` function (which eliminates duplicates).

15 Bitonic tour based method

15.1 Method principle

The objective is to construct an Hamiltonian cycle as a monotone polygon corresponding to the so called bitonic cycle or bitonic tour. Such polygon can be partitioned into exactly two monotone chains[6]. For the considered TSP problem, the vertices are first classified according to the increasing values of their abscissa (or respectively according to their ordinates). The bitonic cycle can be described as this: starting from the left-most vertex, strictly passing some of the vertices from the left to the right and finally reaching the right-most vertex, after which coming back passing the remaining vertices, from the right to the left. The path from the first vertex (left-most vertex) to the nth (right-most vertex) is called the forward path and the path from the nth (right-most vertex) to the first vertex (left-most vertex), the backward path.

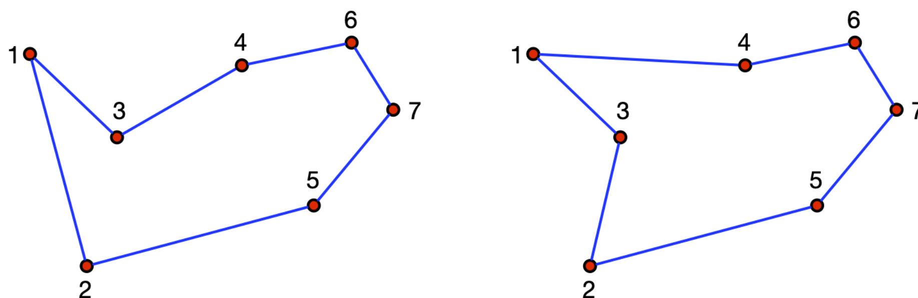


Figure 16: Bitonic and non bitonic cycles (with regard an horizontal line)

Figure 16 illustrates bitonic and non bitonic cycles. In the left-hand side figure, which corresponds to a bitonic cycle, the forward path from vertices 1 to 7 is such that the vertices $1 - 3 - 4 - 6 - 7$ are ordered according to their increasing abscissa. Similarly, for the backward path from 7 to 1, the vertices are ordered according to their decreasing abscissa.

For the right-hand side figure, the backward path is $7 - 5 - 2 - 3 - 1$, but the abscissa of vertex 3, which comes after 2 in the path, is bigger than those of vertex 2. This cycle is not a bitonic cycle[7].

The desired bitonic cycle, solution of the TSP problem, is such that the sum of the lengths of the two paths (forward and backward) is as small as possible.

Before giving an algorithm able to solve that problem [21], let us make the following remark: In a bitonic cycle including n vertices, there are two vertices connected to the right-most vertex n. One is always the vertex $n - 1$ while the other can be any one from 1 to $n - 2$.

[6] A polygonal chain is called monotone if there is a straight line L such that every line perpendicular to L intersects the chain at most once.

[7] It's easy to draw a vertical line at an abscissa comprised between that of vertices 2 and 3 and to observe that the polygon intersects this line 4 times (rather than only 2 for a monotone polygon).

Indeed, two situations can occur:

- edge $(n-1, n)$ is part of the forward path from vertex 1 to vertex n ;
- edge $(n-1, n)$ is not part of the forward path and, necessarily, as all the vertices must be included in one of the two forward or backward paths, edge $(n, n-1)$ is part of this backward path.

The vertex that directly connects the right-most vertex, except $n-1$, is called the key vertex of the cycle.

Dynamic programming algorithm can find a solution of the previous problem in a recursive manner. Consider a $(\ell+1)$-vertex problem constructed on the basis of a ℓ-vertex problem and for which the additional vertex can only be added to the right of the rightmost vertex of the ℓ-vertex problem.

Let us denote $M(\ell)$ the length of the shortest bitonic cycle for the ℓ-vertex problem and $d(i, j)$, the euclidean distance between vertex i and vertex j.

The following lemma can then be stated:
Lemma: For a n-vertex problem, $n > 2$, when k is the key vertex of the cycle, $1 \leq k \leq n-2$, the length of the optimal bitonic cycle is:

$$M(n) = d(k, n) + \sum_{i=k+1}^{n-1} d(i, i+1) + M(k+1) - D(k, k+1)$$

Proof: Consider the bitonic cycle depicted in figure 17.

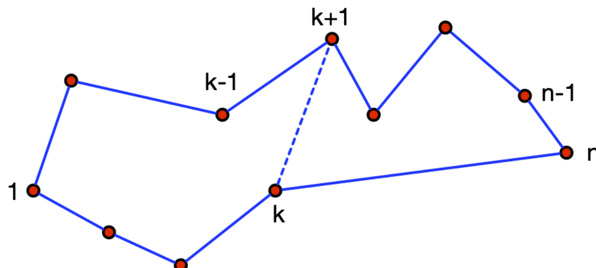


Figure 17: Bitonic cycle

The cycle can be divided into two parts, namely a right-hand chain $k+1, \ldots, n-1, n, k$ (in green on figure 18) and a left-hand chain $k, \ldots, 1, \ldots, k-1, k+1$ (in red).

The key vertex k is directly connected to n. Edge (k, n) belongs to the cycle and its length is $d(k, n)$. Assume the edge (k, n) belongs to the backward path (the proof for the forward path is the same), so that all the vertices from $k+1$ to n belong to the forward path. Assume vertex ℓ, for $k < \ell < n$ is in the backward path, then according to the bitonic features, vertex ℓ being at the right of vertex k, the order of these vertices in the backward path is (n, ℓ, k): this is in contradiction with the initial hypothesis, the vertex k is directly connected to n; so the assumption is false.

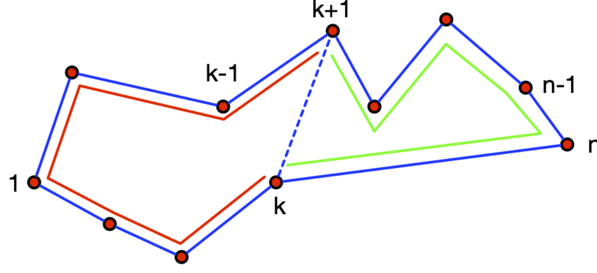


Figure 18: Bitonic cycle splited into two chains

Then the length of the green right-hand chain can be computed as:

$$\sum_{i=k+1}^{n-1} d(i,i+1) + d(k,n)$$

When the key vertex is determined, the length of the right-hand chain is fixed and is the shortest one. Now, the left-hand chain must fulfill the bitonic features and be of minimum length. This chain can be determined from the bitonic cycle of minimal length constructed for the $(k+1)$-vertex problem from which the edge $(k,k+1)$ is removed (red chain in figure 18). Its length is equal to $M(k+1) - d(k,k+1)$.

In the lemma, the key vertex can be any vertex ranging from 1 to $n-2$. To find the optimal solution, it is therefore necessary to examine all cases. The following theorem can then be stated.

Theorem: The length of the optimal bitonic cycle for the n-vertex problem (in the Euclidean $\mathbb{R}^2$ plane) is defined by:

- M(1)=0
- M(2)=2d(1,2) (length of the cycle 1–2–1)
- for $n \geq 3$, $M(n) = \min_{1 \leq k \leq n-2} \left(d(k,n) + \sum_{i=k+1}^{n-1} d(i,i+1) + M(k+1) - d(k,k+1) \right)$

The following algorithm implements the search for a bitonic cycle of smallest length.

Algorithm 2 Bitonic cycle

Require: X : matrix of the vertex coordinates of dimension $n \times 2$

```
1:  compute the euclidean distance matrix d
2:  M(1) ← 0
3:  M(2) ← 2d(1,2)
4:  K(2) ← 1
5:  for i = 3 to n do
6:      M(i) ← ∞
7:      link ← 0
8:      for k = i − 2 downto 1 do
9:          link ← link + d(k+1, k+2)
10:         local_length ← link + d(k,i) + M(k+1) − D(k,k+1)
11:         if local_length < M(i) then
12:             M(i) ← local_length
13:             K(i) ← k
14:         end if
15:     end for
16: end for
17: return K, M
```

In the previous algorithm, *link* is the length of the right-hand part of the forward chain (i.e. chain from $k+1$ to n) and *local_length* is the length of the local optimal cycle. The ith component of vector K is used to store the key vertex of the optimal cycle of the i-vertex problem.

The final cycle is obtained recursively by progressively selecting the edges that will be part of it. In order to reconstruct the final cycle, we can use the memorization of the key vertices of the different solutions for the successive i-vertex problems, $2 < i < n$.

The following algorithm helps to plot the final cycle.

Algorithm 3 Bitonic cycle plot

Require: X: matrix of the vertex coordinates of dimension $n \times 2$

Require: K: vector of the key vertices of the solutions for i-vertex problems

```
1:  line(n − 1,n)
2:  cont ← true
3:  while cont do
4:      k ← K(n)
5:      line(k,n)
6:      for i = k + 1 to n − 2 do
7:          line(i,i + 1)
8:      end for
9:      n ← k + 1
10:     cont ← k ≠ 1
11: end while
12: line(1,2)
```

15.2 Matlab code

Main program

```
%% TSP solving using bitonic cycle method
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

clear
close all

X = rand(25,2);
n = length(X);

% Bitonic tour along the abcissae
% Sort according the first component
[~,orderx] = sort(X(:,1));
Xx = X(orderx,:);
D = squareform(pdist(Xx,'euclidean'));
Kx = dyn_prog(D);
Gx = create_cycle(X,Kx);
px = dfsearch(Gx,1);
% Tour length
Lx = D(px(n),px(1));
for j=2:n
    Lx = Lx + D(px(j-1),px(j));
end
L = Lx;
id_best_tour = 'x';

% Bitonic tour along the ordinates
% Exchange of the two coordinates
Xy=fliplr(X);
% Sort according the first component
[~,ordery] = sort(Xy(:,1));
Xy = Xy(ordery,:);
D = squareform(pdist(Xy,'euclidean'));
Ky = dyn_prog(D);
Gy = create_cycle(Xy,Ky);
py = dfsearch(Gy,1);
% Tour length
Ly = D(py(n),py(1));
for j=2:n
    Ly = Ly + D(py(j-1),py(j));
end
% Best bitonic tour (vs abscissae or ordinates)
if Ly < L
    L = Ly;
    id_best_tour = 'y';
end
% Polygon plot
if id_best_tour == 'x'
    plot(Xx(:,1),Xx(:,2),'kx','Markersize',12,'LineWidth',2)
    hold on
    plot_cycle(Xx,Kx);
    xlabel('Bitonic tour along the abscissae')
```

```
else
    plot(Xy(:,2),Xy(:,1),'kx','Markersize',12,'LineWidth',2)
    hold on
    plot_cycle(fliplr(Xy),Ky);
    xlabel('Bitonic tour along the ordinates')
end
str = sprintf('Tour length: %g',L);
title(str)
```

Functions

```
function K = dyn_prog(D)
n=length(D);
K=zeros(n,1);
M=zeros(n,1);
M(2)=2*D(1,2);
K(2)=1;
for i=3:n
    M(i)=inf;
    link=0;
    for k=i-2:-1:1
        link=link+D(k+1,k+2);
        locallen=link+D(k,i)+M(k+1)-D(k,k+1);
        if locallen < M(i)
            M(i)=locallen;
            K(i)=k;
        end
    end
end
end
```

```
function G = create_cycle(X,K)

n = length(X);
source=[]; target=[];
% edge(n-1,n)
source = [source n-1]; target = [target n];

cont = true;
while cont
    k = K(n);
    % edge(k,n)
    source = [source k]; target = [target n];
    for i=k+1:n-2
        % edge(i,i+1)
        source = [source i]; target = [target i+1];
    end
    n = k+1;
    cont = k~=1;
end
% edge(1,2)
source = [source 1]; target = [target 2];
G = graph(source,target);
end
```

```
function plot_cycle(X,K)
n = length(X);

% line(n-1,n)
plot([X(n-1,1);X(n,1)],[X(n-1,2);X(n,2)],'Color','red','LineWidth',2)
pause(0.5)
cont = true;
while cont
    k = K(n);
    % line(k,n)
    plot([X(k,1);X(n,1)],[X(k,2);X(n,2)],'Color','red','LineWidth',2)
    for i=k+1:n-2
        %line(i,i+1)
        plot([X(i,1);X(i+1,1)],[X(i,2);X(i+1,2)],'Color','red','LineWidth',2)
    end
    n = k+1;
    pause(0.5)
    cont = k~=1;
end
% line(1,2)
plot([X(1,1);X(2,1)],[X(1,2);X(2,2)],'Color','red','LineWidth',2)
end
```

15.3 Explanations

As explained in the method principle section, a monotone polygon is monotone with respect to a straight line. The initial data being in two dimensions, we arbitrarily chose to determine the bitonic cycles with regard to the abscissa axis and the ordinate axis. Then, only the best of the two cycles (the shortest one) is displayed.

The function *dyn_prog* implements algorithm 2. The function *create_cycle* creates the graph by gradually adding the edges of the forward and backward paths according algorithm 3. When adding the edges, the end vertices are stored in the two *source* and *target* vectors. The resulting graph can thus be constructed (Matlab `graph` function). A depth first search exploration of the graph (Matlab `dfsearch` function) then makes it possible to generate the sequence of vertices constituting the desired cycle.

The function *plot_cycle* is identical to the previous one but plots the corresponding polygon (from the right-most vertex to the left-most one).

16 Polynomial-time deterministic (PTD) approach

16.1 Method principle

This approach has been published by Ali Jazayeri and Hiroki Sayama in 2020 [17]. In summary, the proposed algorithm ranks vertices based on their priorities calculated using a power function of means and standard deviations of their distances from other vertices and then connects the vertices to their neighbors in the order of their priorities. When connecting a vertex, a neighbor is selected based on their neighbors' priorities calculated as another power function that additionally includes their distance from the focal vertex

to be connected. This repeats until all the vertices are connected into a single loop.

Let us denote μ_i and σ_i the mean and standard deviation of distances between vertex i and the other vertices and $d_{i,j}$ the distance between vertex i and vertex j. The first power function to determine the priority of vertex i is defined by :

$$p_i = \mu_i^{\alpha} \sigma_i^{\beta} \tag{a}$$

The second power function is defined by:

$$c_j = \frac{\mu_i^{\delta} \sigma_i^{\varepsilon}}{d_{i,j}^{\gamma}} \tag{b}$$

The proposed algorithm proceeds as follows (also see Algorithm 4):

- In the first main step, vertices are processed in the order of the ranking calculated using Equation (a). Each vertex is connected to a neighbor vertex with the highest priority given by (b), under the conditions that the degree of the focal vertex before connection should be zero and that the degree of the neighbor vertex should be less than two. This process is repeated for all vertices. Once this first main step is complete, it is guaranteed that all the vertices have at least one neighbor. Some may have two, as a result of being selected as the neighbor for other vertices.

- In the second main step, Equation (a) is calculated again by using vertices' mean and standard deviation of distances from all other vertices. Then the vertices that have just one neighbor are connected to another neighbor with the highest priority given by Equation (b), under the conditions that the connection of the neighbor vertex to the focal vertex will not create a cycle (unless this connection is the very last one to form a tour) and that the degree of the neighbor vertex is less than two before connection. This process is repeated for all the vertices. Once this second main step is complete, a single looped tour is generated.

Algorithm 4 Polynomial-time deterministic (PTD) approach

Require: X: matrix of the vertices coordinates of dimension $n \times 2$

Require: Value sets for exponents of power functions used for ranking vertices

1: Shortest $= \infty$
2: Compute the distance matrix d between all pairs of vertices
3: MeanList = list of means of distances from each vertex to all other vertices
4: STDList = list of standard deviations of distances from each vertex to all other vertices
5: **for** different combinations of exponent values **do**
6: $E = 0$
7: $OtherEndList = [1 \dots n]$
8: **for** $Step$ from 1 to 2 **do**
9: $ListOne = [\,]$;
10: **for** $Vertex$ from 1 to n **do**
11: **if** $degree(Vertex) < 2$ **then**
12: Calculate Equation (a) using $MeanList(Vertex)$, $STDList(Vertex)$ and the current exponent values
13: Append($Vertex$, Equation(a) result) to $ListOne$
14: **end if**
15: **end for**
16: **while** $ListOne \neq [\,]$ **do**
17: $Vertex$ = the vertex with largest 2nd entry value in $ListOne$
18: Remove $Vertex$ from $ListOne$
19: **if** $degree(Vertex) < Step$ **then**
20: $ListTwo = [\,]$
21: **for** $Neighbor$ from 1 to n except for $Vertex$ **do**
22: **if** $degree(Neighbor) < 2$ **then**
23: **if** $OtherEndList(Vertex) \neq Neighbor$ **or** $E = n - 1$ **then**
24: Calculate Equation(b) using $MeanList(Neighbor)$,
25: $STDList(Neighbor)$ and the current exponent values
26: Append($Neighbor$, Equation(b) result) to $ListTwo$
27: **end if**
28: **end if**
29: **end for**
30: $Neighbor$ = the vertex with the largest 2nd entry value in $ListTwo$
31: $OtherEndList(OtherEndList(Vertex)), OtherEndList(OtherEndList(Neighbor)) = OtherEndList(Neighbor), OtherEndList(Vertex)$
32: Connect $Vertex$ and $Neighbor$
33: $E = E + 1$
34: **end if**
35: **end while**
36: **end for**
37: $FinalTourLength$ = length of the current tour
38: **if** $FinalTourLength < Shortest$ **then**
39: $Shortest = FinalTourLength$
40: Remember used exponent values and connections among vertices
41: **end if**
42: **end for**

16.2 Matlab code

Main program

```
%% TSP solving using a polynomial-time deterministic approach (PTD)
%  Included in TSP20024 app
%  Didier Maquin (2025). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

% Must be followed by a 2-opt optimization for avoiding crossings

% From the article Ali Jazayeri & Hiroki Sayama (2020) A polynomial-time
% deterministic approach to the travelling salesperson problem,
% International Journal of Parallel, Emergent and Distributed Systems,
% 35:4, 454-460, DOI: 10.1080/17445760.2020.1776867

clear
close all
X=rand(40,2);
n = length(X);
% Length of the shortest path
Shortest = inf;
% Distance matrix
D = squareform(pdist(X(:,1:2),'euclidean'));
% List of means of distances from each vertex to all other vertices;
MeanList = sum(D)/(n-1);
% List of standard deviations of distances from each vertex to all other ...
    vertices;
for i=1:n
    V = D(:,i);
    V(i) = [];
    STDList(i) = std(V);
end

% For different combinations of exponent value
% (with regard the cited article, we limit the number of combinations by
% choosing delta=alpha, epsilon=beta and gamma=1 ; as in the article the
% parameters alpha and beta are selected from an exponent valueset {0,0.5,1})
%
for alph=[0 0.5 1]
    for bet=[0 0.5 1]
        delt = alph; epsilon = bet ; gam = 1;
        % Graph creation with nodes only
        G = graph;
        G = addnode(G,n);
        E = 0;
        OtherEndList = [1:n];
        for Step=1:2
            ListOne = [];
            for Vertex=1:n
                if degree(G,Vertex) < 2
                    Eq1 = (MeanList(Vertex)^alph)*(STDList(Vertex)^bet);
                    ListOne = [ListOne ; [Vertex Eq1]];
                end
            end
            while ~isempty(ListOne)
                % Vertex = the vertex with largest 2nd entry value in ListOne
```

```
                [~,i] = max(ListOne(:,2));
                Vertex = ListOne(i,1);
                % Remove Vertex from ListOne
                ListOne(i,:) = [];
                if degree(G,Vertex) < Step
                    ListTwo = [];
                    for Neighbour=1:n
                        if Neighbour == Vertex
                            continue
                        end
                        if degree(G,Neighbour) < 2
                            if (ne(OtherEndList(Vertex),Neighbour))||(E==(n-1))
                                Eq2 = ((MeanList(Neighbour)^delt)* ...
                                (STDList(Neighbour)^epsilon))/ ...
                                (D(Vertex,Neighbour)^gam);
                                ListTwo = [ListTwo ; [Neighbour Eq2]];
                            end
                        end
                    end
                    % Neighbour = the vertex with the largest 2nd entry ...
                        value in ListTwo;
                    [~,i] = max(ListTwo(:,2));
                    Neighbour = ListTwo(i,1);
                    NewOtherEndList = OtherEndList;
                    NewOtherEndList(OtherEndList(Vertex)) = ...
                        OtherEndList(Neighbour);
                    NewOtherEndList(OtherEndList(Neighbour)) = ...
                        OtherEndList(Vertex);
                    OtherEndList = NewOtherEndList;
                    % Connect Vertex and Neighbour;
                    G = addedge(G,Vertex,Neighbour);
                    E = E+1;
                end
            end
        end
        tour = dfsearch(G,1);
        tour = tour';
        L = tour_length(X,tour);
        if L < Shortest
            Shortest = L;
            memalph = alph ; membet = bet;
        end
    end
end
% The best exponents that give the shortest tour have been found
% Now we redo the treatment with these parameters in order to show
% the progressive construction of the tour
alph = memalph ; bet = membet;
plot(X(:,1),X(:,2),'kx','MarkerSize',8,'LineWidth',2)
hold on
% Graph creation with nodes only
G = graph;
G = addnode(G,n);
E = 0;
figtitle = {'First step','Second step'};
colordraw = {'r','b'};
OtherEndList = [1:n];
```

```
for Step=1:2
    title(figtitle{Step})
    ListOne = [];
    for Vertex=1:n
        if degree(G,Vertex) < 2
            Eq1 = (MeanList(Vertex)^alph)*(STDList(Vertex)^bet);
            ListOne = [ListOne ; [Vertex Eq1]];
        end
    end
    while ~isempty(ListOne)
        % Vertex = the vertex with largest 2nd entry value in ListOne
        [~,i] = max(ListOne(:,2));
        Vertex = ListOne(i,1);
        % Remove Vertex from ListOne
        ListOne(i,:) = [];
        if degree(G,Vertex) < Step
            ListTwo = [];
            for Neighbour=1:n
                if Neighbour == Vertex
                    continue
                end
                if degree(G,Neighbour) < 2
                    if (ne(OtherEndList(Vertex),Neighbour)) || (E==(n-1))
                        Eq2 = ((MeanList(Neighbour)^delt)* ...
                        (STDList(Neighbour)^epsilon))/(D(Vertex,Neighbour)^gam);
                        ListTwo = [ListTwo ; [Neighbour Eq2]];
                    end
                end
            end
            % Neighbour = the vertex with the largest 2nd entry value in ...
                ListTwo;
            [~,i] = max(ListTwo(:,2));
            Neighbour = ListTwo(i,1);
            % Update of path extremity
            OC = OtherEndList(City);
            ON = OtherEndList(Neighbour);
            OtherEndList(OC) = ON;
            OtherEndList(ON) = OC;
            % Connect Vertex and Neighbour;
            G = addedge(G,Vertex,Neighbour);
            plot([X(Vertex,1) X(Neighbour,1)],[X(Vertex,2) ...
                X(Neighbour,2)],colordraw{Step},'LineWidth',2)
            E = E+1;
            pause(0.4)
        end
    end
    pause(2)
end
tour = dfsearch(G,1);
tour = tour';
cla
tourdisp = [tour tour(1)];
plot(X(tourdisp,1),X(tourdisp,2),'r','LineWidth',2);
plot(X(:,1),X(:,2),'kx','MarkerSize',8,'LineWidth',2)
% Add title
str = sprintf('Tour Length: %g',Shortest);
title(str)
```

Functions

```
function L = tour_length(X,tour)
tour = [tour tour(1)];
L = sum(sqrt(diff(X(tour,1)).^2+diff(X(tour,2)).^2));
end
```

16.3 Explanations

The algorithm proposed in the article is very clear and directly follows the given explanations. However, notice that the priorities defined by Equation (a) and stored in the *ListOne* matrix must be computed for vertices of degree 0 in the first step and vertices of degree 1 in the second step. To have only one processing within the two-step loop, these priorities are calculated for all vertices of degree strictly less than 2 (0 or 1) (line 11) and the insertion of a possible neighbor in *ListTwo* is conditioned by the fact that the degree of the vertex considered $Vertex$ is less than *Step* (the step number variable) (line 19). Some priorities are therefore calculated unnecessarily for the first step.

The only subtlety of the algorithm is to ensure that connecting a vertex to a neighbor will not close a cycle unless it involves connecting the last two vertices together (all the others already constituting a chain). To do this, we maintain the list *OtherEndList*, each element i of which contains the number of the vertex located at the other end of the chain to which it belongs. Thus, when examining the different candidate vertices to be neighbors of a particular vertex i, if $OtherEndList(i) = j$, then vertex j cannot be considered as a neighbor because the connection $i - j$ would create a cycle except in the situation where adding the edge $i - j$ closes the Hamiltonian cycle, hence the test in line 23 of the algorithm:

$$\textbf{if } OtherEndList(Vertex) \neq Neighbor \textbf{ or } E = n - 1 \textbf{ then},$$

E being the number of edges already added in the constitution of the Hamiltonian cycle (a Hamiltonian cycle passing through n vertices contains $n - 1$ edges).

Please note that updating the two elements of the *OtherEndList* list must be simultaneous, hence the syntax of instruction 31 of the algorithm. For the Matlab implementation, two temporary scalars OC and ON have been used.

In the original article, five parameters were introduced in the power functions: α and β in Equation (a) and δ, ε and γ in Equation (b). The authors of the article propose to select these parameters from the exponent value set $\{0, 0.5, 1\}$ and to retain the best solution obtained (shortest Hamiltonian cycle). This involves computing $3^5 = 243$ solutions for a given problem. We decided to limit the exploration of the solution domain here by fixing $\gamma \equiv 1$, $\delta = \alpha$ and $\varepsilon = \beta$ (only $3^2 = 9$ solutions are examined and compared).

Once the best parameters have been determined (those which lead to the shortest Hamiltonian cycle) and in order to demonstrate in an animated way how the two-step algorithm works, the calculation is carried out again with the chosen set of parameters.

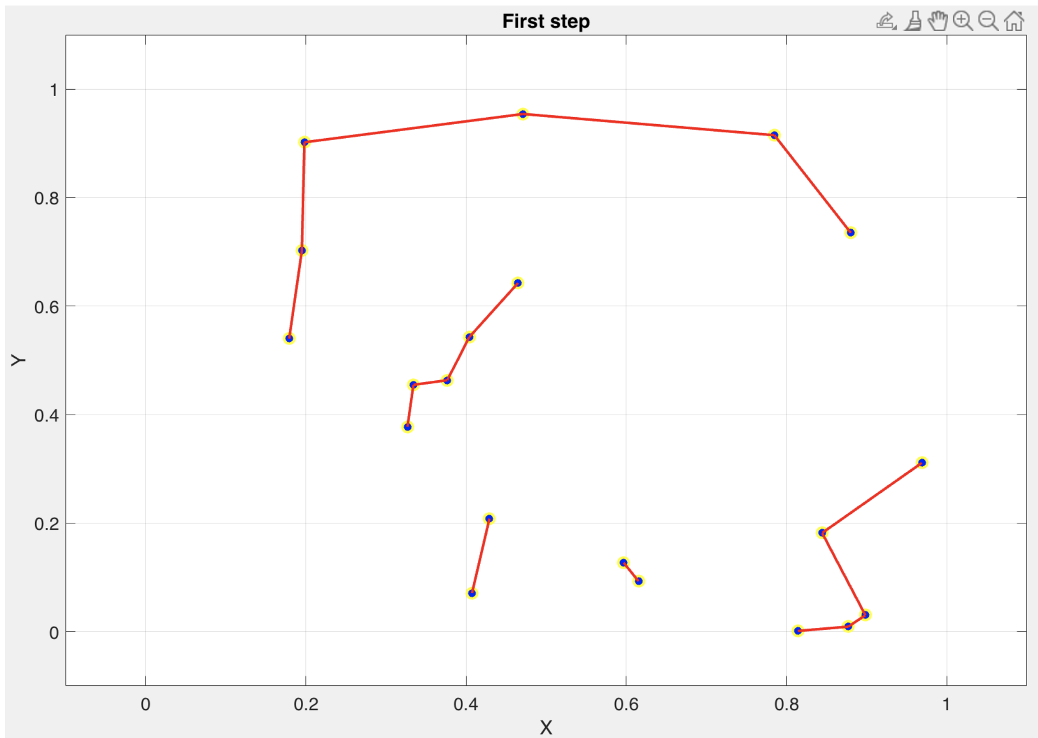


Figure 19: First step of the PTD approach

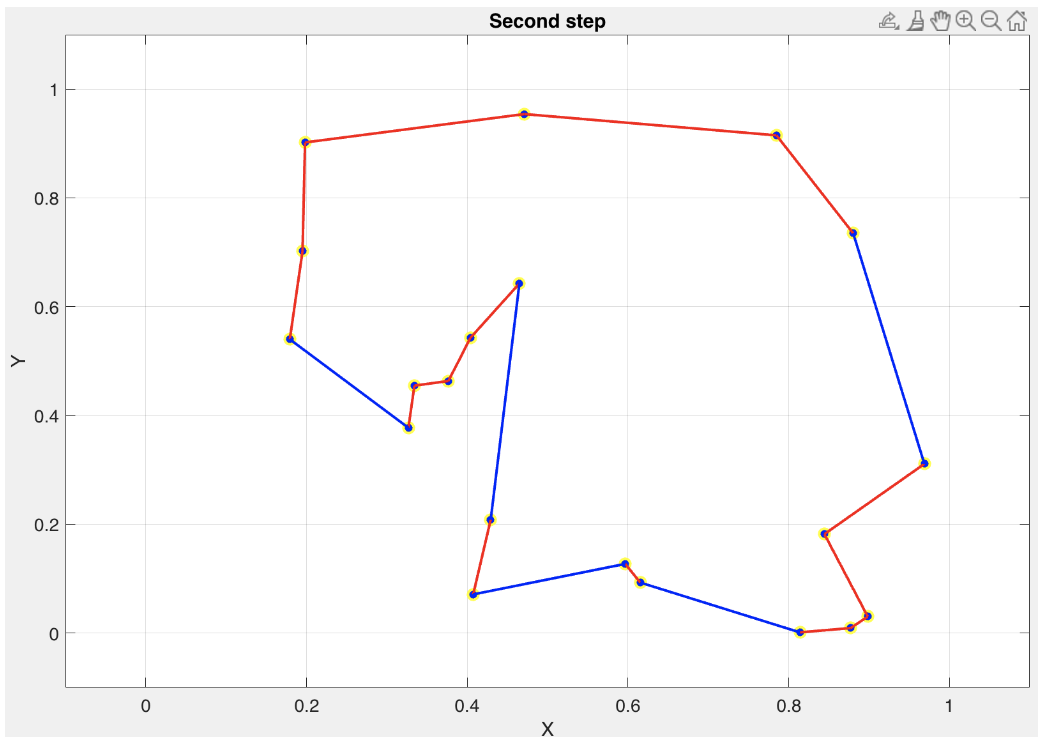


Figure 20: Second step of the PTD approach

As said in the introduction, the Patch visualization mode is more suitable for large data set. Below, the final result for the "very big file" (for TSP2024!) rat575 comprising 575 vertices.

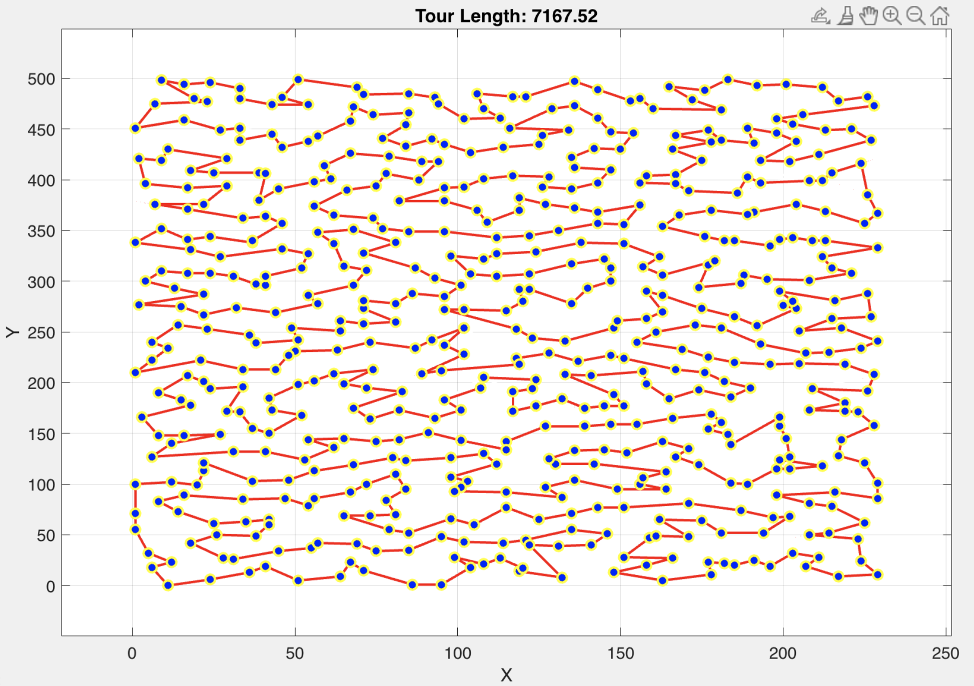


Figure 21: Final result with the Path visualization option

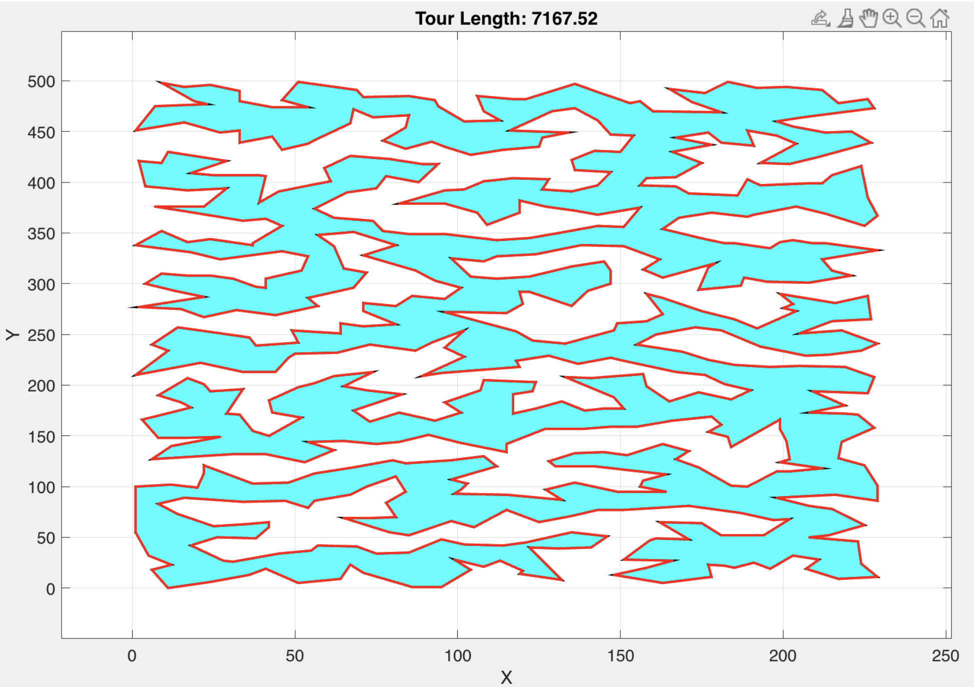


Figure 22: Final result with the Patch visualization option

17 Clarke and Wright heuristic

17.1 Method principle

The Clarke-Wright saving's algorithm is an algorithm originally designed to solve the vehicle routing problem (VRP), a problem very similar to the TSP.

To start off, a vertex h is randomly picked among the n vertices to be linked. This vertex is considered as a hub. The idea is then to create $n-1$ pseudo-tours (cycles) in the form of (h, i, h), where $i, i = 1, \ldots, n, i \neq h$ are indices of other vertices.

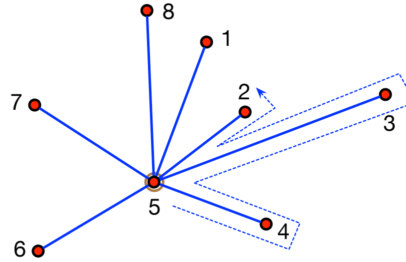


Figure 23: Initial cycle

Then these pseudo-tours are iteratively combined in such a way that an Hamiltonian cycle with the shortest possible distance is created. For each pair of two vertices, other than the hub, we calculate how much distance we would save if we connect the two pseudo-tours those vertices participate in.

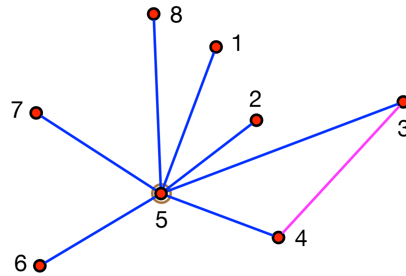


Figure 24: Shortcut from vertex 4 to vertex 3

The length of the original cycle for visiting vertex 3 and 4 from the hub vertex 5 is equal to $d(5,3) + d(3,5) + d(5,4) + d(4,5) = 2(d(5,3) + d(5,4))$ where $d(i,j)$ is the euclidean distance between vertices i and j. With the shortcut of figure 24, the length is equal to $d(5,4) + d(4,3) + d(3,5)$. The saving is then equal to $d(5,3) + d(5,4) - d(4,3)$ which is necessary positive due to the triangle inequality.

More generally, the strategy consists to replace pseudo-tours (h, i, h) and (h, j, h) with (h, i, j, h). This is also extended to pseudo-tours with more than one vertex, for instance, if the saving by connecting vertices i and j is equal to $s(i,j)$, these vertices can be connected if they belong to two distinct pseudo-tours, one of which starts or ends with i, and one of which starts or ends with j. For example, we combine pseudo-tours $(h, i, ..., h)$ with $(h, ..., j, h)$ into $(h, ..., i, j, ..., h)$.

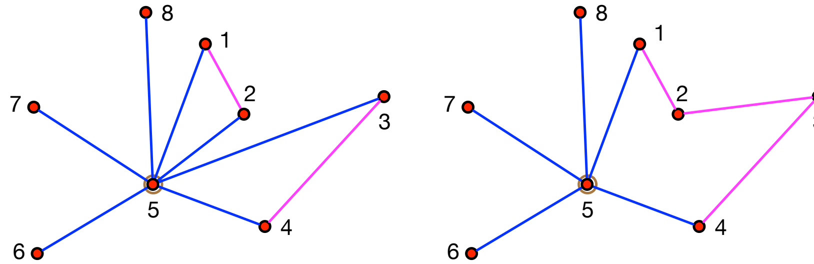


Figure 25: Merging of two pseudo-tours by linking vertices 2 and 3

With these ideas, the simple pseudo-code of the Clarke-Wright algorithm for the TSP can be formulated [28]:

Algorithm 5 Clarke and Wright heuristic

Require: X: matrix of the vertices coordinates of dimension $n \times 2$

1: compute the distance matrix d between all pairs of vertices
2: select any vertex as a hub which we denote as vertex 0
3: create a set T of pseudo-tours $(0, i, 0)$ where $i = 1, 2, ...n - 1$
4: compute savings $s(i, j) = d(0, i) + d(j, 0) - d(i, j)$ for all pairs of vertices
5: sort the savings by decreasing values
6: starting at the top of the list of savings $s(i, j)$, try to merge the two pseudo-tours in which i and j intervene, provided that:
   (a) the two vertices are not already on the same pseudo-tour
   (b) neither vertex is interior to its pseudo-tour, meaning that both vertices are still directly connected to the hub vertex on their respective pseudo-tour
7: repeat step (6) until no additional savings can be achieved.
8: **return** the last constructed tour

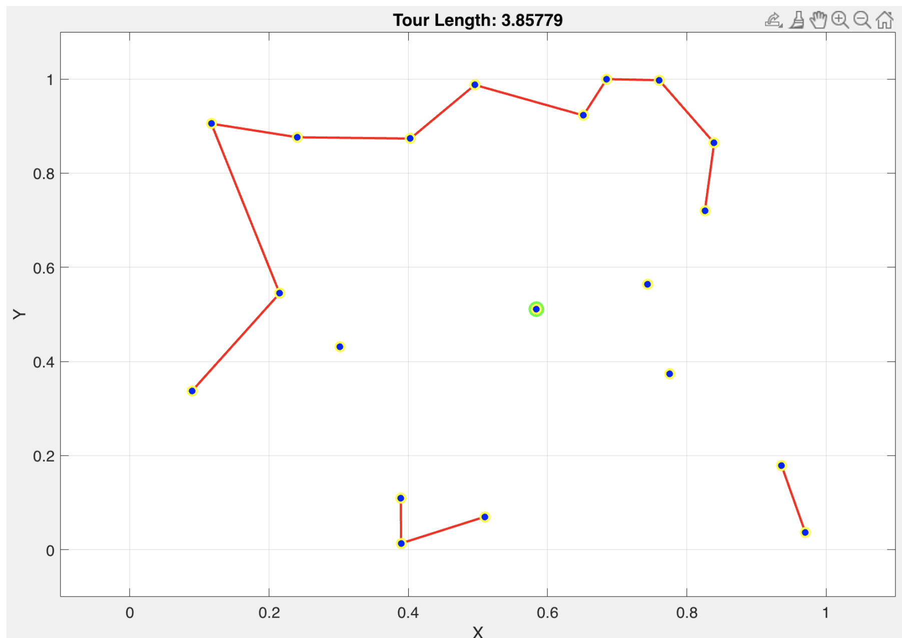


Figure 26: Clarke-Wright heuristic in action

17.2 Matlab code

Main program

```
%% TSP solving using the Clark and Wright heuristic
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

clear
close all
X = rand(10,2);
n = length(X);
D = squareform(pdist(X));
plot(X(:,1),X(:,2),'kx','MarkerSize',8,'LineWidth',2)
% Add node numbers
% for k = 1:n
%   str = sprintf(' %d',k);
%   text(X(k,1),X(k,2),str)
% end
hold on

% With Statistics and Machine Learning Toolbox
% center = mean(X,1);               % mean of data
% hub = knnsearch(X,center,'k',1);  % nearest record to center

hub = randi(n);
plot(X(hub,1),X(hub,2),'og','MarkerSize',12,'MarkerFaceColor','b','LineWidth',2)
% Graph creation with nodes only
G = graph;
G = addnode(G,n);

% Computing of the savings (matrix with 3 columns : i,j,sav)
k = 0;
for i=1:n
    for j=i+1:n
        if i ~= hub && j ~= hub
            k = k+1;
            s(k,:) = [ i j D(hub,i)+D(hub,j)-D(i,j)];
        end
    end
end

% Sort of the savings
[~,order] = sort(s(:,3),'descend');
s = s(order,:);

minParent = 1:n;

Vh = zeros(1,n);      % Vh(i)=1 if the vertex is included in the final tour
Vh(hub) = 1;
VhCount = n-1;        % Number of vertices to examine
degrees = zeros(1,n); % Vertex degrees

edge_idx = 1;         % Number of the saving to be examined
```

```
while VhCount>2       % Try until only two vertices remain
    % Edge to be inserted
    i = s(edge_idx,1);
    j = s(edge_idx,2);

    if Vh(i)==0 && Vh(j)==0 && (minParent(i)~=minParent(j))

        degrees(i) = degrees(i)+1;
        degrees(j) = degrees(j)+1;
        plot([X(i,1) X(j,1)],[X(i,2) X(j,2)],'r','LineWidth',2)
        G=addedge(G,i,j);
        pause(0.2)

        if minParent(i)<minParent(j)
            minParent(minParent==minParent(j))=minParent(i);
        else
            minParent(minParent==minParent(i))=minParent(j);
        end

        if degrees(i)==2
            Vh(i) = 1;
            VhCount = VhCount-1;
        end

        if degrees(j)==2
            Vh(j) = 1;
            VhCount = VhCount-1;
        end
    end
    edge_idx = edge_idx+1;

end

remain = find(Vh==0); % last remaining nodes

% Add of the two last edges from the vertice hub to the 2 unvisited
% vertices
G = addedge(G,hub,remain(1));
G = addedge(G,hub,remain(2));
plot([X(hub,1) X(remain(1),1)],[X(hub,2) X(remain(1),2)],'r','LineWidth',2)
plot([X(hub,1) X(remain(2),1)],[X(hub,2) X(remain(2),2)],'r','LineWidth',2)

tour = dfsearch(G,hub);

% Closing of the polygon for length calculus
p = [tour ; tour(1)];
% Add title
L = sqrt(diff(X(p,1)).^2 + diff(X(p,2)).^2);
str = sprintf('Tour Length: %g',sum(L));
title(str)
```

17.3 Explanations

In the above implementation, the pseudo-tours are not explicitly built. Only the shortcut edges, which will constitute the final Hamiltonian cycle, are created progressively. The elements $Vh(i), i = 1, \ldots, n$ are such that $Vh(i) = 1$ if the vertex i is included in the final tour and $Vh(i) = 0$ otherwise.

The $minParent$ vector is regularly updated to know if two vertices are part of the same pseudo-tour. At initialization, $minParent(i) = i$; each vertex is part of its own pseudo-tour. Then in order to know if two vertices belong to the same pseudo-tour (if the vertices have been connected by an edge), $minParent(i)$ is updated by the value of the smallest index of the vertices in the pseudo-tour involving vertex i.

Managing the degrees of the vertices then allows to know if a vertex is interior to its pseudo-tour. The condition for adding the edge (i, j) to the final cycle then concerns the fact that the vertices i and j have not already been definitively included in the final cycle (degree equal to 2) and that they are not part of the same pseudo-tour (different $minParent$).

Processing stops when there are only two unintegrated vertices left in the final cycle. The edges connecting these vertices to the hub vertex are then added. As the final Hamiltonian cycle is only constructed by successive addition of edges, the sequence of vertices is obtained by performing a depth-first search of the graph thus constructed.

18 Divide and conquer approach

18.1 Method principle

The divide and conquer approach is not very efficient, but it is very elegant to implement using a recursive procedure. The basic idea is to split the vertices by a cut in the plane, solve the TSP on the two halves, and then somehow merge the solutions [5]. The cut can be orthogonal to the coordinate axes. In particular, we can find in which of the two dimensions the coordinates of the vertices have a larger spread, find the mean point along that dimension and cut the plane in two halves by a straight line passing through this mean point. To merge the solutions, the two Hamiltonian cycles can be joined through and "edge pairing".

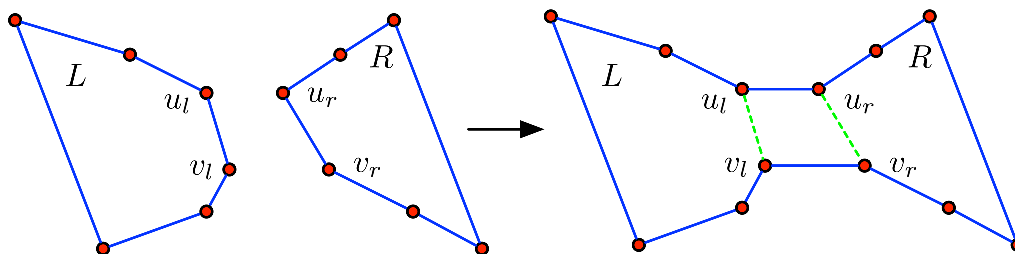


Figure 27: Merging two cycles

The edges $e_l = (u_l, v_l)$ from the left tour and $e_r = (u_r, v_r)$ from the right tour are chosen to minimize the increase in the length of the resulting Hamiltonian cycle:

$$\min_{e_l, e_r} \quad c(e_l, e_r) = d(u_l, u_r) + d(v_r, v_l) - d(u_l, v_l) - d(u_r, v_r)$$

where $d(u, v)$ is the Euclidean distance between vertex u and vertex v.

The Matlab function computing the Hamiltonian cycle which is defined recursively is particularly simple and elegant.

```
function tour = DivCon(list_X,X,D)
n = length(list_X);
% For sets with one or two vertices, the Hamiltonian cycle is the list
if n==1 || n==2
    tour = list_X;
else
    % Divide the list of vertices in two parts
    [list_X1,list_X2] = part_list_X(list_X,X);
    % Search an Hamiltonian cyle in each part
    tour1 = DivCon(list_X1,X,D);
    tour2 = DivCon(list_X2,X,D);
    % Merge the two obtained Hamiltonian cycles
    tour = merge_tour(tour1,tour2,X,D);
end
end
```

The `part_list_X` function splits a list of vertices in two parts and the `merge_tour` function merges two Hamiltonian cycles into one at the lowest cost (minimizing the increase in its length).

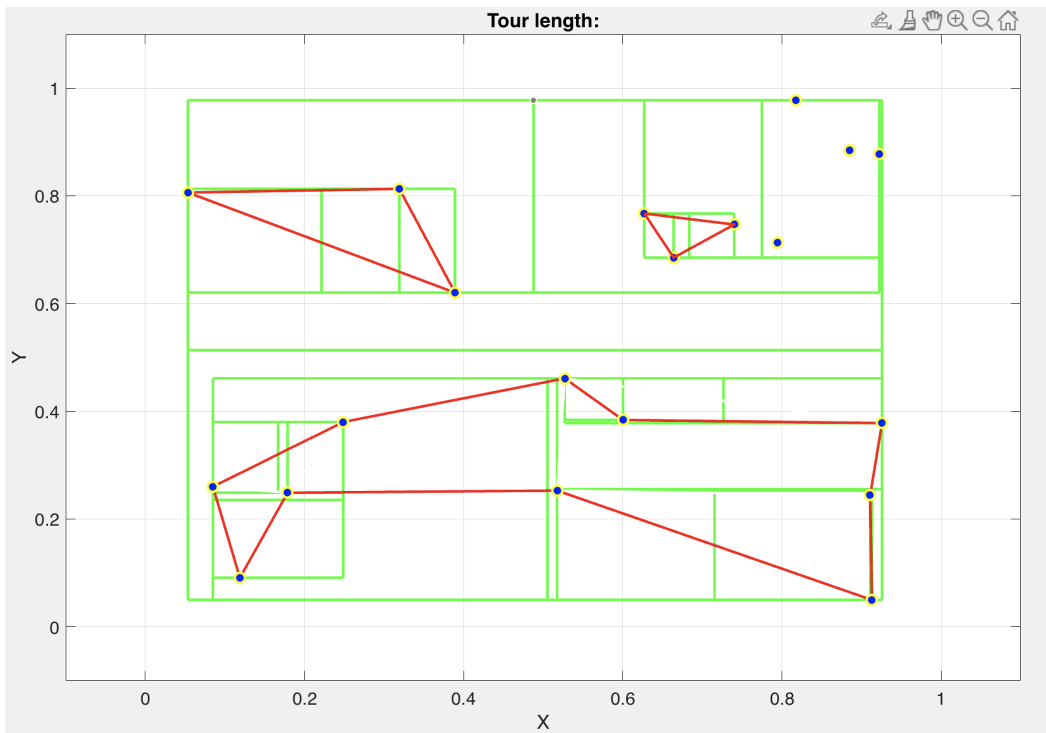


Figure 28: Divide and conquer in action

18.2 Matlab code

Main program

```
%% TSP solving using Divide & Conquer approach
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

clear
close all
X=rand(100,2);
D = squareform(pdist(X,'euclidean'));
list_X = 1:length(X);
% Call the Divide & Conquer function
tour = DivCon(list_X,X,D);
pause(1)
cla
plot(X(:,1),X(:,2),'kx','MarkerSize',12,'LineWidth',2)
hold on
L = tour_length(X,tour);
% Plot
tour = [tour tour(1)];
plot(X(tour,1),X(tour,2),'r','LineWidth',2);
% Add title
str = sprintf('Tour Length: %g',L);
title(str)
```

Functions

```
function tour = DivCon(list_X,X,D)
n = length(list_X);
% For sets with one or two vertices, the Hamiltonian cycle is the list
if n==1 || n==2
    tour = list_X;
    % Comment the following lines to mask the partition
    x = X(list_X,1); y = X(list_X,2);
    plot([min(x) max(x)],[min(y) min(y)],'g','LineWidth',2)
    plot([min(x) max(x)],[max(y) max(y)],'g','LineWidth',2)
    plot([min(x) min(x)],[min(y) max(y)],'g','LineWidth',2)
    plot([max(x) max(x)],[min(y) max(y)],'g','LineWidth',2)
else
    % Divide the list of vertices in two parts
    [list_X1,list_X2] = part_list_X(list_X,X);
    % Search an Hamiltonian cyle in each part
    tour1 = DivCon(list_X1,X,D);
    tour2 = DivCon(list_X2,X,D);
    % Merge the two obtained Hamiltonian cycles
    tour = merge_tour(tour1,tour2,X,D);
    % Comment the following line if you don't want the merged tour
    % to be optimized at each merge
    [tour,~] = exchange2(tour,D,X,0);
    % Polygon plot
    plot(X(:,1),X(:,2),'kx','MarkerSize',12,'LineWidth',2)
    hold on
```

```
    tourdisp = [tour tour(1)];
    plot(X(tourdisp,1),X(tourdisp,2),'r','LineWidth',2);
    pause(0.2)
end
end
```

```
function tour = merge_tour(tour1,tour2,X,D)
% The not so simple function merging to Hamiltonian cycles
n1 = length(tour1);
n2 = length(tour2);
% Closing the cycles
tour1=[tour1 tour1(1)];
tour2=[tour2 tour2(1)];
% The number of vertices of tour1 is less or equal than those of tour2
if n2<n1
    temp = tour1;  ntemp = n1;
    tour1 = tour2; n1 = n2;
    tour2 = temp;  n2 = ntemp;
end
% The merge of two cycles with one vertex each
if n1==1 & n2==1
    tour = [tour1(1) tour2(1)];
end
% The merge of a single vertex with a cycle
if n1==1 & n2~=1
    minl = inf;
    for j=2:n2+1
        toura = [tour2(1:j-1) tour1(1) tour2(j:n2)];
        if tour_length(X,toura) < minl
            tour = toura;
            minl = tour_length(X,toura);
            memj = j-1;
        end
    end
    % Plot: erasing of original edge
    plot([X(tour2(memj),1) X(tour2(memj+1),1)],[[X(tour2(memj),2) ...
        X(tour2(memj+1),2)]],'w','LineWidth',2)
% The merge of two cyles of dimension 2
elseif n1==2 & n2==2
    toura = [tour1(1) tour2(1) tour2(2) tour1(2)];
    tour = toura;
    minl = tour_length(X,tour);
    tourb = [tour1(1) tour2(2) tour2(1) tour1(2)];
    if tour_length(X,tourb) < minl
        tour = tourb;
    end
% The merge of a cycle of dimension 2 with any cycle comprising more than 2 ...
    vertices
elseif n1==2 & n2>2
    minl = inf;
    for j=2:n2+1
        addl = D(tour1(1),tour2(j-1))+D(tour1(2),tour2(j))...
               -D(tour2(j-1),tour2(j));
        if addl < minl
            minl = addl;
            edge2 = [j-1 j];
        end
```

```
        addl = D(tour1(1),tour2(j))+D(tour1(2),tour2(j-1))...
                -D(tour2(j-1),tour2(j));
        if addl < minl
            minl = addl;
            edge2 = [j j-1];
        end
    end
    % Restoration of original cycles
    tour1=tour1(1:end-1);
    tour2=tour2(1:end-1);

    % Fliping the second tour  and updating indexes
    if edge2(2) < edge2(1)
        tour2 = flip(tour2);
        edge2 = [n2-edge2(1)+1 n2-edge2(2)+1];
    end
    % The edge where the graft is carried out is placed in first position ...
       in the cycle
    tour2 = circshift(tour2,1-edge2(1));

    % The two possible merging are examined
    tour = [tour1(1) tour2(1) tour2(n2:-1:2) tour1(2)];
    minl = tour_length(X,tour);
    toura = [tour1(1) tour2(1:n2) tour1(2)];
    if tour_length(X,toura) < minl
        tour = toura;
    end
    % Plot: erasing of the original edge
    plot([X(tour2(1),1) X(tour2(2),1)],[[X(tour2(1),2) ...
        X(tour2(2),2)]],'w','LineWidth',2)
%The merge of two any cycles
else
    minl = inf;
    for i=2:n1+1
        for j=2:n2+1
            addl = D(tour1(i-1),tour2(j-1))+D(tour1(i),tour2(j))...
                -D(tour1(i-1),tour1(i))-D(tour2(j-1),tour2(j));
            if addl < minl
                minl = addl;
                edge1 = [i-1 i];
                edge2 = [j-1 j];
            end
            addl = D(tour1(i-1),tour2(j))+D(tour1(i),tour2(j-1))...
                -D(tour1(i-1),tour1(i))-D(tour2(j-1),tour2(j));
            if addl < minl
                minl = addl;
                edge1 = [i-1 i];
                edge2 = [j j-1];
            end
        end
    end
    % Restoration of original cycles
    tour1 = tour1(1:end-1);
    tour2 = tour2(1:end-1);

    % Fliping the second tour and updating indexes
    if edge2(2) < edge2(1)
        tour2 = flip(tour2);
```

```
        edge2 = [n2-edge2(1)+1 n2-edge2(2)+1];
    end
    % The edges where the graft is carried out are placed
    % in first position in their cycles
    tour1 = circshift(tour1,1-edge1(1));
    tour2 = circshift(tour2,1-edge2(1));

    % The two possible merging are examined
    tour = [tour1(1) tour2(1) tour2(n2:-1:2) tour1(2:n1)];
    minl = tour_length(X,tour);
    toura = [tour1(1) tour2(1:n2) tour1(2:n1)];
    if tour_length(X,toura) < minl
        tour = toura;
    end
    % Plot: erasing of the edges in the two tours
    plot([X(tour1(1),1) X(tour1(2),1)],[X(tour1(1),2) ...
        X(tour1(2),2)],'w','LineWidth',2)
    plot([X(tour2(1),1) X(tour2(2),1)],[X(tour2(1),2) ...
        X(tour2(2),2)],'w','LineWidth',2)
end
end
```

```
function [list_X1,list_X2] = part_list_X(list_X,X)
x = X(list_X,1); y = X(list_X,2);
rangex = max(x)-min(x);
rangey = max(y)-min(y);
if rangey>rangex
    ymid = min(y)+rangey/2;
    indX1 = find(y<=ymid);
    % Comment the following lines to mask the partition
    plot([min(x) max(x)],[ymid ymid],'g','LineWidth',2)
    plot([min(x) max(x)],[min(y) min(y)],'g','LineWidth',2)
    plot([min(x) max(x)],[max(y) max(y)],'g','LineWidth',2)
    plot([min(x) min(x)],[min(y) max(y)],'g','LineWidth',2)
    plot([max(x) max(x)],[min(y) max(y)],'g','LineWidth',2)
else
    xmid = min(x)+rangex/2;
    indX1 = find(x<=xmid);
    % Comment the following lines to mask the partition
    plot([xmid xmid],[min(y) max(y)],'g','LineWidth',2)
    plot([min(x) min(x)],[min(y) max(y)],'g','LineWidth',2)
    plot([max(x) max(x)],[min(y) max(y)],'g','LineWidth',2)
    plot([min(x) max(x)],[min(y) min(y)],'g','LineWidth',2)
    plot([min(x) max(x)],[max(y) max(y)],'g','LineWidth',2)
end
list_X1 = list_X(indX1);
list_X2 = list_X;
list_X2(indX1)=[];
end
```

```
% Alternative function for space partitioning
% Needs the Statistics and Machine Learning Toolbox
% The vertices are clustered in two class using kmeans
% Suppress or rename the previous function and
% rename the function below as part_list_X
```

```
function [list_X1,list_X2] = part_list_X_alt(list_X,X)
[idx,~] = kmeans(X(list_X,:),2);
list_X1 = list_X(idx==1);
list_X2 = list_X(idx==2);
end
```

18.3 Explanation

The purpose of the `merge_tour` function is easy to understand. However, it is more complex to write than it seems. Let us begin by the more general case of merging two cycles with more than 3 nodes each. The first step consists of determining the two edges allowing the graft to be carried out according to the criterion stated previously. For a given pair of edges, each in its cycle, the two connection possibilities ((u_l, u_r) and (v_l, v_r) or (u_l, v_r) and (v_l, u_r)) must be tested.

Two difficulties appear to generate the cycle resulting from the merging. Let us first recall that a cycle is described by an ordered vector of the vertex numbers to be traversed. Thus, a cycle comprising n vertices can be represented by $2n$ different vectors. For example, the cycle described by the vector (3, 5, 6, 2, 8, 4, 1, 7) is the same by circular permutation as that described by (2, 8, 4, 1, 7, 3, 5, 6) and the same using reverse order as that described by (7, 1, 4, 8, 2, 6, 5, 3).

The first difficulty appears when one of the two selected edges (or both) is the edge connecting the last vertex of the vector to the first one. In order to take this difficulty into account, the selected edges will be systematically brought to the first position of the vectors using a circular permutation (Matlab function `circshift`).

The second difficulty comes from the "direction of rotation" (in the plane, trigonometric or anti-trigonometric direction) of the description of the cycles which influences how to create the merged cycle.

When searching for edges, they are identified by two consecutive indices in the vectors describing the cycles. Consider the example of figure 29.

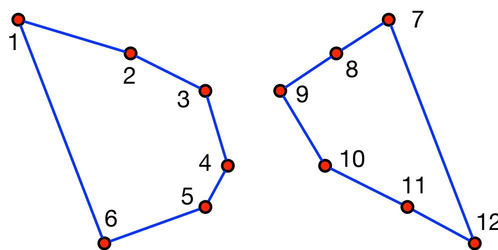


Figure 29: Two cycles to merge

If the first cycle is described by the vector (2, 3, 4, 5, 6, 1) and the second by (12, 7, 8, 9, 10, 11). The indices in the first vector describing the edge (3, 4) are (2, 3) and those in the second vector describing the edge (9, 10) are (4, 5). But if the second cycle is described by

the vector (11, 10, 9, 8, 7, 12) – reverse order – the corresponding indices are now (3, 2). The second index being smaller than the first, this means that the direction of rotation of the second cycle is the opposite of the first. In that case, the cycle is flipped (reversed) and the indices are updated accordingly.

With these two preprocessings, the second cycle is inserted between the first and the second vertex of the first cycle, according to the two possible directions of rotation. The shortest cycle is then retained. For the graphic animation, the two edges allowing the graft (one in each tour) are erased.

Some special cases must also be considered:

- When the cycles to merge are composed each of only one vertex, say i and j, the merged cycle is simply (i, j).
- When one of the two cycles is composed of only one vertex, merging the cycles consists to insert this isolated vertex j between two vertices of the other cycle of indices k and $k+1$ such that the length of the obtained cycle is minimum. For the graphic animation, the edge between the vertices of index k and $k+1$ is erased.

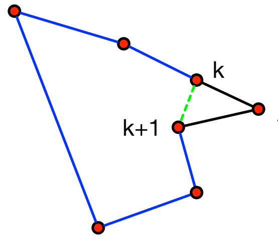


Figure 30: Insertion of a vertex in a cycle

- When the two cycles are both composed of two vertices, the two connection possibilities are tested and that producing the minimal length cycle is retained. Let say that the first cycle is made up of the two vertices u_1 and v_1 and the second is made up of u_2 and v_2. The lengths of the two cycles (u_1, v_1, v_2, u_2) and (u_1, v_1, u_2, v_2) are compared to determine the shortest one. For the graphic animation, there is nothing to erase.
- When one of the two cycles is composed of two vertices, the other comprising more than two vertices, the treatment is almost identical to the general case except that one of the two edges where the grafting of the two cycles takes place is fixed *a priori*. For the graphic animation, the grafting edge in the cycle comprising more than two vertices is erased.

The "basic" `part_list_X` function splits a list of vertices into two parts depending only on geometric considerations. The function divides the initial space into two half-spaces by cutting it orthogonally to the axis where the coordinate variation interval is the greatest.

Many variations can be considered. We can cut the space comprising n vertices so that the two subspaces include an identical number (within one) of vertices (see the Karp's

approach below). Another approach is to take into account the proximity of the vertices. The two subspaces can be obtained by a 2-class classification algorithm. Among the simplest, we can use the k-means algorithm (see the Matlab code).

Merging two Hamiltonian cycles can generate intersecting edges. After each merge, a 2-opt local optimization can be used.

19 Karp's approach

19.1 Method principle

Richard Karp published, in 1977, an article describing partitioning algorithms [18]. As, the previous method, these algorithms subdivide the set of vertices into small groups, construct an optimum tour through each group and then patch the subtours together to form a tour through all the vertices. So this approach is very similar to the previous one. What differs is the technique of cutting the plane and the way of merging two Hamiltonian cycles (which is linked to this cutting).

Karp's approach to divide and conquer employs a recursive bisection routine which repeatedly subdivides the original problem into smaller and smaller rectangles in the Euclidean plane, stopping when the subproblem sizes become less than some predetermined threshold. Let rectangle R contain m cities. Let y be the y-coordinate of the $[m/2]$th closest vertex to the top edge of R. A horizontal cut through y subdivides R into two rectangles, an upper rectangle and a lower rectangle. The situation is illustrated in figure 31 (left). The effect is to place half the vertices either side of the bisecting line with at least one city on the bisector. In a similar fashion, a vertical cut could be applied to bisect the cities through x, which is the x-coordinate of the $[m/2]$th closest vertex to the left edge of R (figure 31, right). In Karp's algorithm the direction of the cut is always parallel to the shorter side of the rectangle.

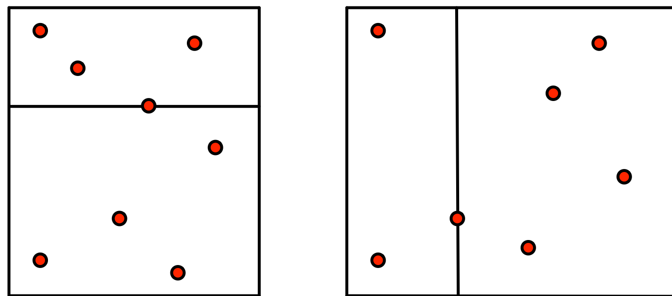


Figure 31: Karp's bissection

Once this partition is carried out, Hamiltonian cycles for all the vertices of the subrectangles can be searched (figure 32, left).

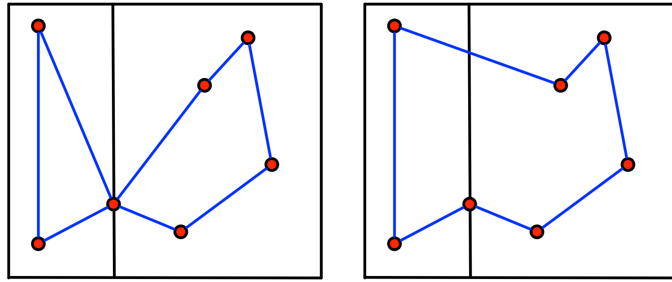


Figure 32: Patching subtours

The next step is to patch or to merge these two cycles. Because these two cycles share a common vertex, Karp introduces what he calls a “spanning walk” and two reduction operations to obtain this merge[8]. We propose here a simplified version of this merge [32]. As shown in figure 32, left, the four incident edges to the shared city must be reduced to two. This is achieved by the removal of two of the incident edges, one from each cycle, and the creation of a new edge between the two “stranded” cities (figure 32, right). As there are only four possible ways this patching can be done, they are all tried and one that results in the shortest patched tour is selected.

The structure of the algorithm is the same as the previous one. The `karp_tour` function computes the Hamiltonian cycle recursively. As in the previous method, a 2-opt local optimization is used after the merge of two cycles.

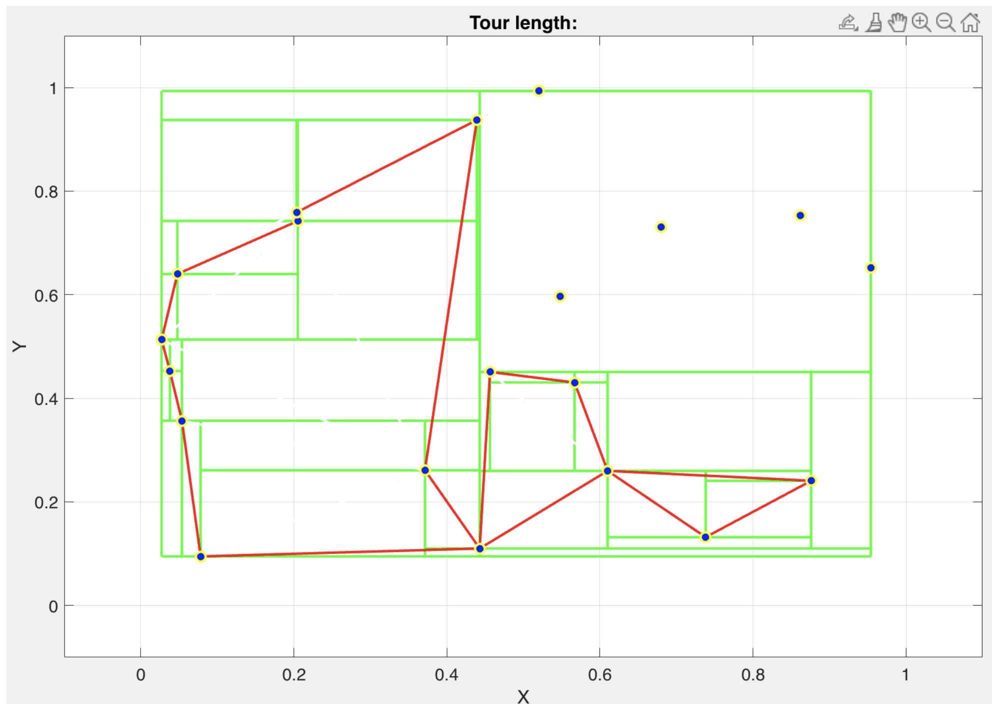


Figure 33: Karp’s partitioning in action

[8]Given a multigraph (possibly with loops and multiple edges), a tour (Hamilton cycle) is a connected graph including all the vertices of the graph and every vertex has degree 2 and a spanning walk is a connected graph including all the vertices of the graph and all vertices are of even degree.

19.2 Matlab code

Main program

```
%% TSP solving using Karp's partition
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

clear
close all
X=rand(60,2);
n = length(X);
D = squareform(pdist(X,'euclidean'));
list_X = 1:length(X);

% Call the Karp function
tour = karp_tour(list_X,X,D);
L = tour_length(X,tour);
hold off; cla
plot(X(:,1),X(:,2),'kx','MarkerSize',12,'LineWidth',2)
hold on
% Plot
tourdisp = [tour tour(1)];
plot([X(tourdisp,1),X(tourdisp,2),'r','LineWidth',2);
% Add title
str = sprintf('Tour Length: %g',L);
title(str)
```

Functions

```
function tour = karp_tour(list_X,X,D);
n = length(list_X);
if n==2
    tour = list_X;
    % Uncomment the following lines to see the partition
    % x = X(list_X,1); y = X(list_X,2);
    % plot([min(x) max(x)],[min(y) min(y)],'g','LineWidth',2)
    % plot([min(x) max(x)],[max(y) max(y)],'g','LineWidth',2)
    % plot([min(x) min(x)],[min(y) max(y)],'g','LineWidth',2)
    % plot([max(x) max(x)],[min(y) max(y)],'g','LineWidth',2)
else
    % Divide the list of vertices in two parts
    [list_X1,list_X2] = bisect(list_X,X);
    % Search an Hamiltonian cyle in each part
    tour1 = karp_tour(list_X1,X,D);
    tour2 = karp_tour(list_X2,X,D);
    % Patching the two obtained Hamiltonian cycles
    tour = patch_tour(tour1,tour2,X,D);
    % Comment the following line if you don't want the merged tour
    % to be optimized at each merge
    [tour,L] = exchange2(tour,D,X,0);
    % Polygon plot
    plot(X(:,1),X(:,2),'kx','MarkerSize',12,'LineWidth',2)
    hold on
    tourdisp = [tour tour(1)];
```

```
    plot(X(tourdisp,1),X(tourdisp,2),'r','LineWidth',2);
    pause(0.2)
end
end
```

```
function tour = patch_tour(tour1,tour2,X,D);
% Common vertex of the two tour
common = intersect(tour1,tour2);
pos1 = find(tour1==common);
% Place the edge with this vertex in first position of the tour 1
tour1 = circshift(tour1,1-pos1);
% Place the edge with this vertex in first position of the tour 1
pos2 = find(tour2==common);
tour2 = circshift(tour2,1-pos2);
% Merge the tours (4 possibilities) and memorize the edges to erase
min_length = inf;
toura = [tour1(1:end) tour2(2:end)];
tlengtha = tour_length(X,toura);
if tlengtha < min_length
    tour = toura;
    min_length = tlengtha;
    erase1 = [tour1(1) tour1(end)];
    erase2 = [tour2(1:2)];
end
tourb = [tour1(1:end) tour2(end:-1:2)];
tlengthb = tour_length(X,tourb);
if tlengthb < min_length
    tour = tourb;
    min_length = tlengthb;
    erase1 = [tour1(1) tour1(end)];
    erase2 = [tour2(1) tour2(end)];
end
tourc = [tour1(1) tour1(end:-1:2) tour2(2:end)];
tlengthc = tour_length(X,tourc);
if tlengthc < min_length
    tour = tourc;
    min_length = tlengthc;
    erase1 = [tour1(1:2)];
    erase2 = [tour2(1:2)];
end
tourd = [tour2(1:end) tour1(2:end)];
tlengthd = tour_length(X,tourd);
if tlengthd < min_length
    tour = tourd;
    min_length = tlengthd;
    erase1 = [tour1(1:2)];
    erase2 = [tour2(1) tour2(end)];
end
% Plot: erasing of original edges
    plot([X(erase1(1),1) X(erase1(2),1)],[X(erase1(1),2) ...
        X(erase1(2),2)],'w','LineWidth',2)
    plot([X(erase2(1),1) X(erase2(2),1)],[X(erase2(1),2) ...
        X(erase2(2),2)],'w','LineWidth',2)
end
```

```
function [list_X1,list_X2] = bisect(list_X,X)
x = X(list_X,1); y = X(list_X,2);
rangex = max(x)-min(x);
rangey = max(y)-min(y);
if rangey>rangex
    ymed = median(y);
    [~,ind] = min(abs(y-ymed));
    indX1 = find(y<=y(ind));
    indX2 = find(y>=y(ind));
    % Uncomment the following lines to see the partition
    %     plot([min(x) max(x)],[y(ind) y(ind)],'g','LineWidth',2)
    %     plot([min(x) max(x)],[min(y) min(y)],'g','LineWidth',2)
    %     plot([min(x) max(x)],[max(y) max(y)],'g','LineWidth',2)
    %     plot([min(x) min(x)],[min(y) max(y)],'g','LineWidth',2)
    %     plot([max(x) max(x)],[min(y) max(y)],'g','LineWidth',2)
else
    xmed = median(x);
    [~,ind] = min(abs(x-xmed));
    indX1 = find(x<=x(ind));
    indX2 = find(x>=x(ind));
    % Uncomment the following lines to see the partition
    %     plot([x(ind) x(ind)],[min(y) max(y)],'g','LineWidth',2)
    %     plot([min(x) min(x)],[min(y) max(y)],'g','LineWidth',2)
    %     plot([max(x) max(x)],[min(y) max(y)],'g','LineWidth',2)
    %     plot([min(x) max(x)],[min(y) min(y)],'g','LineWidth',2)
    %     plot([min(x) max(x)],[max(y) max(y)],'g','LineWidth',2)
end
list_X1 = list_X(indX1);
list_X2 = list_X(indX2);
end
```

19.3 Explanations

The `patch_tour` function merges two Hamiltonian cycles having a common vertex. First, this common vertex is searched and brought to the first position of the vectors describing the cycles using a circular permutation (Matlab function `circshift`). Then, the four possibilities of merging are examined. Let us consider the following two tours: (2, 4, 9, 10, 7, 3) and (2, 8, 1, 5, 6). The four edges incident to the common vertex are (2, 4), (2, 3), (2, 8) and (2, 6). Removing one edge in each tour leads to the four following situations:

- If edges (2, 3) and (2, 8) are removed, it leads to the merged tour (2, 4, 9, 10, 7, 3 | 8, 1, 5, 6).
- If edges (2, 3) and (2, 6) are removed, it leads to the merged tour (2, 4, 9, 10, 7, 3 | 6, 5, 1, 8).
- If edges (2, 4) and (2, 8) are removed, it leads to the merged tour (2 | 3, 7, 10, 9, 4 | 8, 1, 5, 6).
- If edges (2, 4) and (2, 6) are removed, it leads to the merged tour (2, 8, 1, 5, 6 | 4, 9, 10, 7, 3).

where the | sign separates the elements of the first tour from that of the second. All these tours can be easily constructed. Their length are then compared and the shortest-length cycle is retained.

20 Pair-center algorithm

20.1 Method principle

The pair-center algorithm was proposed by Arno Formella in the article "Quasi-linear time heuristic to solve the Euclidean traveling salesman problem with low gap, Journal of Computational Science 82:102424 (2024)" [12]. The following description summarizes the terms of this article.

Given a set of vertices (points) in the Euclidean plane, the pair-center algorithm runs in two phases: a clustering phase and a construction phase. In the first phase, the clustering phase, a binary tree over the points is built by successively substituting the currently closest pair of points by its center point. Eventually, only one center point remains as the root of the tree. The leaves of the tree contain the original vertices. The inner nodes of the tree contain the constructed center points. This binary tree can be viewed as a hierarchical clustering of the point set where each subtree of the binary tree represents a cluster of the points at its leaves.

In the second phase, the construction phase, a traversal of the binary tree takes place to construct the complete tour. The traversal of the tree starts at the root of the binary tree that has been built in the first phase. A point of an inner node belonging to the current tour is replaced by its two generating points. When all inner nodes will have been replaced, the tour passes through all original vertices.

In the proposed article, the author uses particular data structures for the implementation of the algorithm in order to make it efficient in computation time. In particular, he uses dynamic tree, binary tree and heap. Here we only want to show how the algorithm works and we are not looking for performance. So we implemented the method using simple lists (this approach is not very efficient but it is more than enough to handle problems with 100 vertices).

Let P be a given set of n points in the Euclidean plane. Store all the points of P as the leaves of a binary tree B (still without inner nodes). Build a dynamic tree T of all points in P. The two phases of the pair-center algorithms are:

1. **Complete the binary tree B in the following way.**
   While there is still a closest pair in T:
   a) Take the closest pair.
   b) Delete the two corresponding points, say p_i and p_j, from the dynamic tree T.
   c) Calculate the center point c_{ij} of the two points p_i and p_j, i.e. $c_{ij} = 0.5(p_i + p_j)$.
   d) Insert c_{ij} into the dynamic tree T .
   e) Insert c_{ij} as an inner node into the binary tree B with p_i and p_j as child nodes.

At the end, the dynamic tree T is reduced to just one point and the binary tree B contains $2n-1$ nodes, namely, $n-1$ inner nodes as the pair-centers and n leaves as points of P.

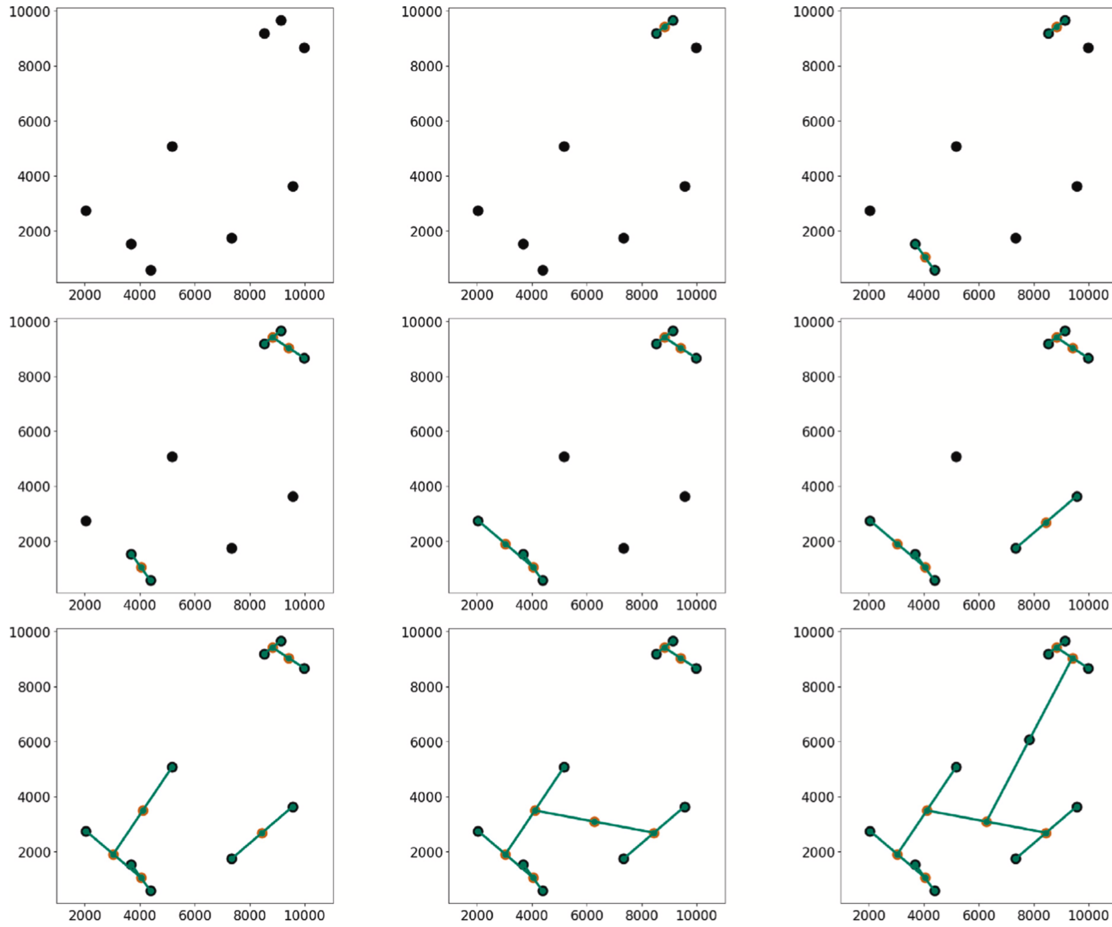


Figure 34: Visualization of the first phase – the clustering phase – of the pair-center algorithm (from A. Formella [12])

2. Traverse the binary tree and construct the tour by substituting points in the current tour by their generating pairs of center points. The order of substitution takes place according to the distances of the corresponding tree nodes. The order of substitution is according to the length of the segment joining the generating pairs, the points at the center of the longest segments being replaced first.

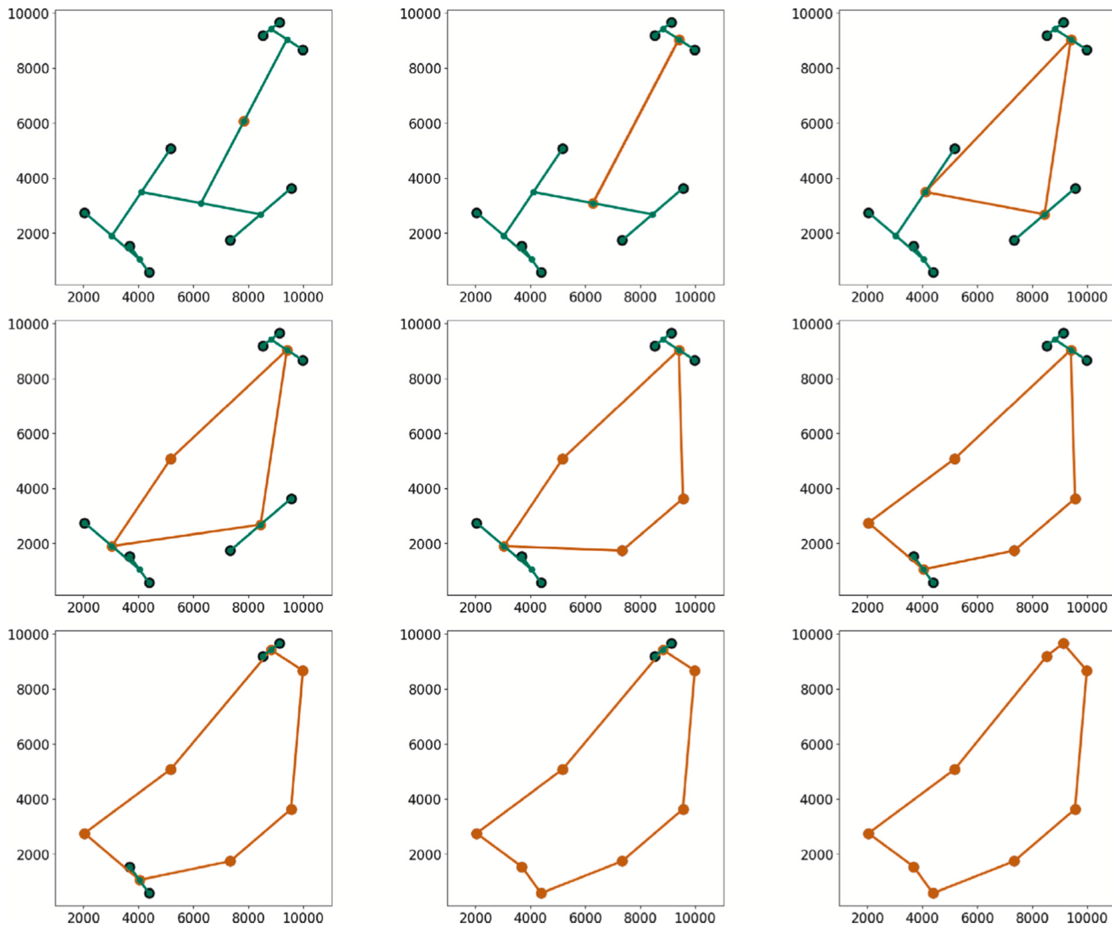


Figure 35: Visualization of the second phase – the construction phase – of the pair-center algorithm (from A. Formella [12])

20.2 Matlab code

Main program

```
%% TSP solving using the pair-center algorithm
%  Included in TSP20024 app
%  Didier Maquin (2025). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

% Must be followed by a 2-opt optimization for avoiding crossings

clear
close all
X = rand(60,2);
n = length(X);
plot(X(:,1),X(:,2),'rx','Linewidth',2)
% First phase: binary tree building
title('First phase: clustering')
```

```
%
Xc = [X (1:n)'];
center = [];
for k = 1:n-1
    % Computation of the distance matrix
    D = squareform(pdist(Xc(:,1:2),'euclidean'));
    % Cancelation of the diagonal (0 --> inf)
    D = D+diag(inf*ones(1,length(D)));
    % Search of the minimum distance between points:
    % original vertices and center points
    mini = min(D,[],'all');
    % Determination of point numbers
    [lig,col] = find(D==mini);
    ilig = lig(1);
    icol = col(1);
    hold on
    % Plotting the corresponding segment
    plot([Xc(ilig,1) Xc(icol,1)],[Xc(ilig,2) Xc(icol,2)],'b','Linewidth',1)
    pause(0.6)
    Xc(end+1,:) = [(Xc(ilig,1:2)+Xc(icol,1:2))/2 n+k];
    center = [center ; Xc(end,1:2) Xc(ilig,3) Xc(icol,3) ...
        pdist([Xc(ilig,1:2);Xc(icol,1:2)])];
    Xc(ilig,:) = [];
    Xc(icol,:) = [];
end
% Matrix of all the points: original vertices and center points
Xe = [X;center(:,1:2)];
tour = center(end,3:4);
title('Second phase: polygon construction')
% Erase the segment joining the center points
plot([Xe(center(end,3),1) Xe(center(end,4),1)],[Xe(center(end,3),2) ...
    Xe(center(end,4),2)],'w','LineWidth',2)
% Closing of the polygon for ploting
p = [tour  tour(1)];
h = plot(Xe(p,1),Xe(p,2),'g','LineWidth',2);
% For all the remaining center points
for k=1:n-2
    ltour = length(tour);
    maxi = 0;
    % For all vertices of the intermediate polygon
    for i=1:ltour
        % If a point is an original vertex do nothing
        if tour(i)<=n
            continue
        end
        % If it is a center point search the largest "tangent" segment
        newmaxi = center(tour(i)-n,5);
        if newmaxi > maxi
            maxi = newmaxi;
            expandpoint = tour(i);
            ind = i;
        end
    end
    % Replace the central point with the two points from which it originates
    tour1 = [tour(1:ind-1) center(expandpoint-n,[3 4]) tour(ind+1:end)];
    tour2 = [tour(1:ind-1) center(expandpoint-n,[4 3]) tour(ind+1:end)];
    L1 = tour_length(Xe,tour1);
    L2 = tour_length(Xe,tour2);
```

```
    % Check which order gives the lowest tour increase
    tour = tour1;
    if L2 < L1
        tour = tour2;
    end
    % Closing of the polygon for length calculus and plot
    p = [tour  tour(1)];
    % Erase the previous intermediate polygon
    delete(h)
    % Erase the segment joining the center points
    plot([Xe(center(expandpoint-n,3),1) ...
        Xe(center(expandpoint-n,4),1)],[Xe(center(expandpoint-n,3),2) ...
        Xe(center(expandpoint-n,4),2)],'w','LineWidth',2)
    % Plot the new polygon
    h=plot(Xe(p,1),Xe(p,2),'g','LineWidth',2);
    plot(X(:,1),X(:,2),'rx','Linewidth',2)
    pause(0.6)
end
% Plot the final polygon
clf(1)
plot(Xe(p,1),Xe(p,2),'g','LineWidth',2)
hold on
plot(X(:,1),X(:,2),'rx','Linewidth',2)
% Add title
L = tour_length(X,p);
str = sprintf('Tour Length: %g',L);
title(str)
```

Functions

```
function L = tour_length(X,tour)
tour = [tour tour(1)];
L = sum(sqrt(diff(X(tour,1)).^2+diff(X(tour,2)).^2));
end
```

20.3 Explanations

For the first phase, rather than implementing a binary tree, we used simple lists (matrices). Here, a matrix Xc is created. The two first columns contain the coordinates of the original vertices and the third their order numbers. Within an iteration loop, we determine the closest points, we calculate their center which we insert into Xc by assigning it an increasing order number from $n+1$ and we delete these closest points from Xc. In parallel, we create the matrix *center* which contains in its first two columns, the coordinates of the calculated centers, in the third and fourth columns, the numbers of the vertices relative to the center considered and in the fifth and last column, the distance between the two vertices. The binary tree is also progressively drawn within this loop. The matrix *center* together with the matrix X of the original vertices provide all the necessary information relating to the described binary tree.

Before constructing the polygon representing the Hamiltonian path, we gather in the same two-column matrix Xe the coordinates of the original vertices and those of the different centers. This matrix has $2n-1$ rows.

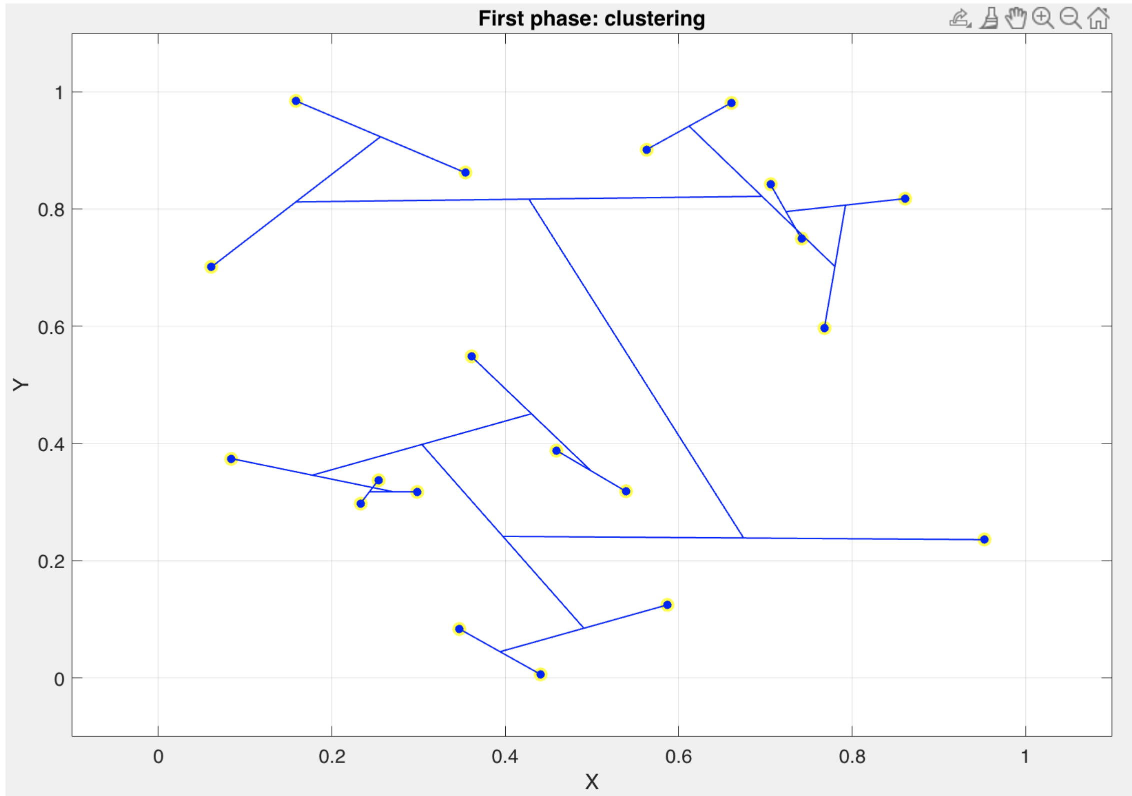


Figure 36: Binary tree at the end of the first phase

The construction phase is based on the analysis of *center* and Xe. First the tour is initialized as the list of the two points defining the last element of *center* ; it corresponds to the biggest distance between center points. Of course, as usual, the polygon is closed by adding at the end of the list its first element. For graphic animation, the corresponding segment is erased and the polygon is drawn. Then, within a loop iteration the different centers are progressively replaced by their generating pairs.

Inside this loop, we examine each of the vertices of the current polygon. If a vertex of the polygon is an initial vertex (leaf of the binary tree), we do nothing. It corresponds to vertices with a number less or equal n. For all polygon vertices that are center points, we look for the one whose corresponding segment is the largest and we replace it in the list by its generating pair ; this center is named *expandpoint*. This information is available in the matrix *center*. Let say that the actual vertex sequence defining the polygon is $q_p - q - q_n$ where q is the center point between q_i and q_j. Then we check which option gives the lowest tour increase when replacing the tour segment $q_p - q - q_n$ either with $q_p - q_i - q_j - q_n$ or with $q_p - q_j - q_i - q_n$. As previously, for graphic animation, the segment having as center q is erased as well as the current polygon and the newly defined polygon is drawn.

In the end, all the center points have been substituted and the polygon contains all the original vertices.

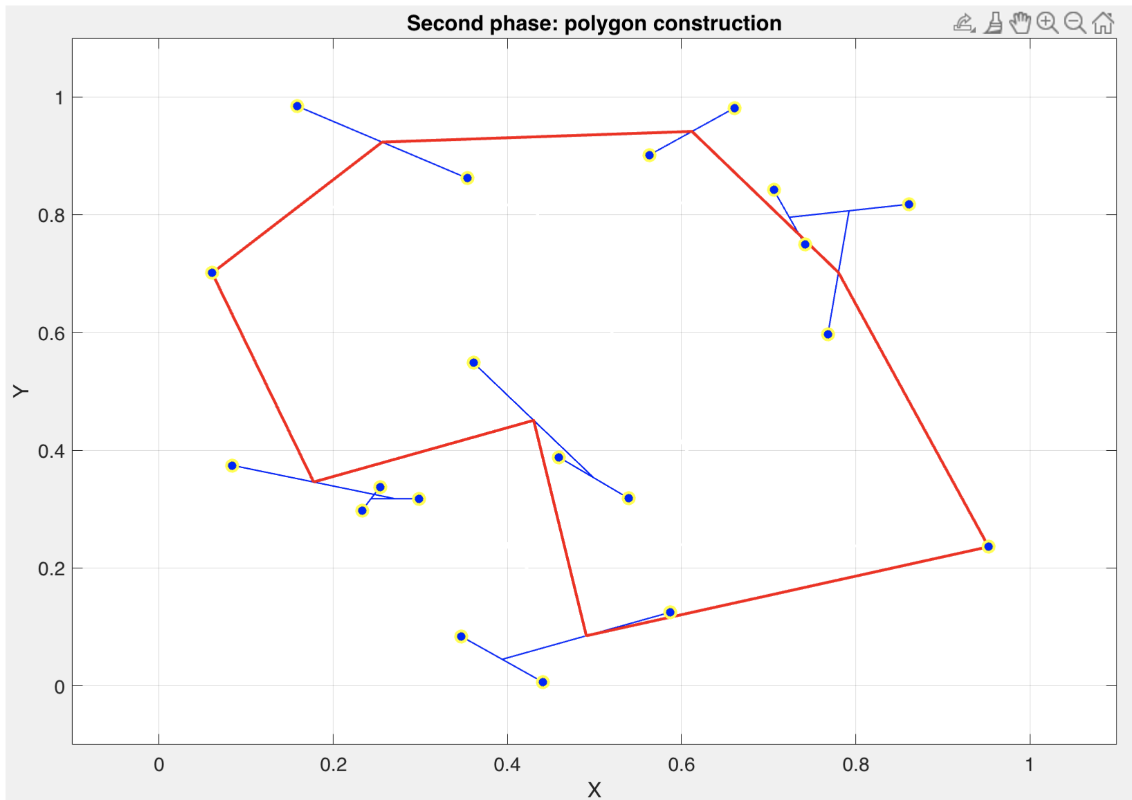


Figure 37: Construction phase in progress

21 Space-filling curve

21.1 Method principle

A space-filling curve is a curve whose range reaches every point in a higher dimensional region, typically the unit square (or more generally an n-dimensional unit hypercube). In 1890, Giuseppe Peano discovered a continuous curve, now called the Peano curve, that passes through every point of the unit square. His purpose was to construct a continuous mapping from the unit interval onto the unit square.

A useful property of a space-filling curve is that it tends to visit all the points in a region once it has entered that region. Thus points that are close together in the plane will tend to be close together in appearance along the curve. This forms the basis of the heuristic, invented by John Bartholdi III and Loren Platzman [3], to produce a reasonably short Travelling Salesman's Tour of n given locations: Simply visit them in the same sequence as does the space-filling curve. For example, a short tour of the points marked in black is indicated by the red lines, which connect the points in the same sequence as their appearance on the space-filling curve (figure 38 bottom-right).

The Hilbert curve was first described by the German mathematician David Hilbert in 1891, as a variant of the space-filling Peano curves. The Hilbert curve is constructed as a limit of piecewise linear curves. The length of the nth curve is $2^n - \frac{1}{2^n}$, i.e., the length

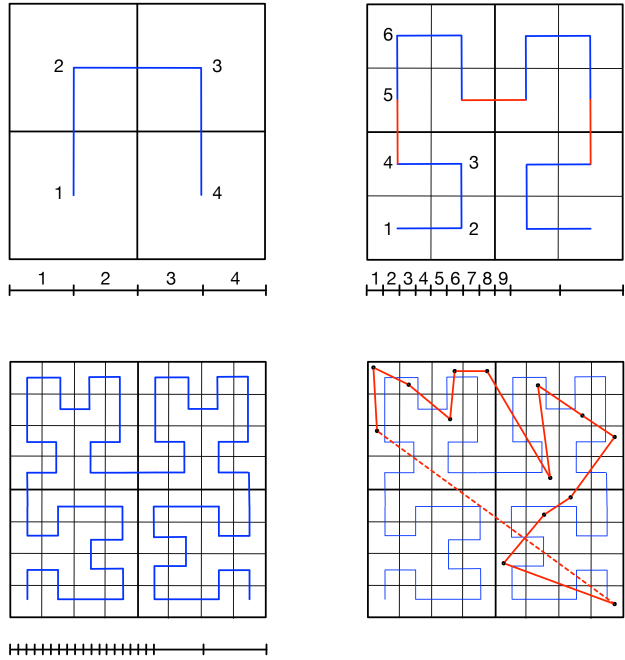


Figure 38: Hilbert curve

grows exponentially with n, even though each curve is contained in a square with area 1.

Consider the following construction: Divide the unit interval into four interval and the unit square into four squares and assign each interval to one of the square. Connect the centers of the four squares as shown in figure 38 top-left. Divide each square again in four identical squares and connect their centers respecting one of the production rules presented in figure 39.

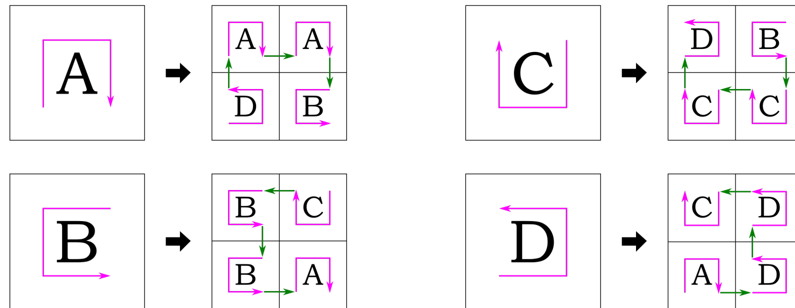


Figure 39: Production rules

This process can be continued recursively and furthermore it can be done in such a way that neighboring intervals are assigned to neighboring squares. The first three steps of this construction are shown in figure 38. In the limit this yields a surjective, continuous map from the unit interval to the unit square.

For our purposes it suffices to repeat the subdivision process until we can read off the

order of the points along the curve. Since the Hilbert curve starts in the lower left corner and ends in the lower right corner, the last edge of the round-trip (shown in figure 38 bottom-right as dashed line) is likely to be long. Thus, the Hilbert curve is only suited for applications where a short path through the points is needed. If a closed tour is required as in the case of the traveling salesman problem then a closed space-filling curve is more suited, i.e., a curve starting and ending in the same point. The Moore and the Sierpinski curves shown in figure 40 are examples for closed curves.

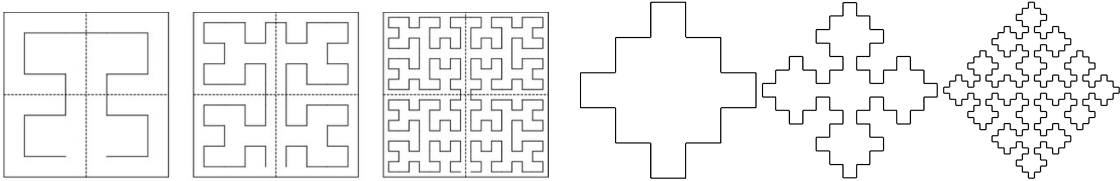


Figure 40: Moore and Sierpinski curves

The space-filling curve heuristic for the traveling salesman problem yields a tour which is at most a logarithmic factor longer than the shortest tour. In the case of uniformly distributed points it is with high probability only longer by a constant factor.

The search algorithm is particularly simple:

Algorithm 6 General space-filling curve heuristic with space-filling curve ψ

Require: X: matrix of the vertex coordinates (x, y) of dimension $n \times 2$

1: **for** i from 1 to n **do**
2: compute $\theta_i \in [0, 1]$ such $\psi(\theta_i) = (x_i, y_i)$
3: sort $(x_1, y_1), \ldots, (x_n, y_n)$ by the order $(x_i, y_i) \prec (x_j, y_j) \Leftrightarrow \theta_i < \theta_j$, for $1 \leq i, j \leq n$
4: **end for**

For space-filling curves like the Hilbert curve which can be constructed by recursive subdivision, preimages of points in $[0, 1]^2$ can be computed based on the subdivision. Indeed, the specific form of ψ is not essential but requires some properties including the ability to easily calculate an inverse image. In their original article John Bartholdi III and Loren Platzman [3] propose a space-filling curve $\psi(\theta)$ constructed recursively by dividing the square into four identical subsquares and filling each with a space-filling curve rotated so that they link to form a circuit (figure 41). They also provide a program in BASIC for computing the ϕ function such that $\psi(\phi(x, y)) = (x, y)$.

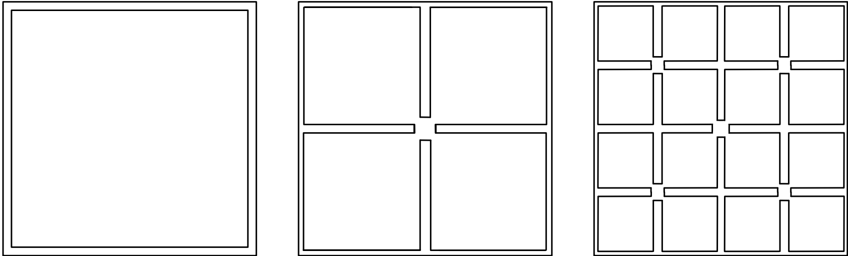


Figure 41: Space-filling curve proposed by Platzman

In their paper published in 1989 [27], they re-visited their idea: the curve is constructed by successively partitioning the domain $S = [0,1]^2$ using triangles. The kth partition of S consists of $2k$ identical triangles, each labeled with the binary representation of an integer in the range $0, \ldots, 2k-1$.

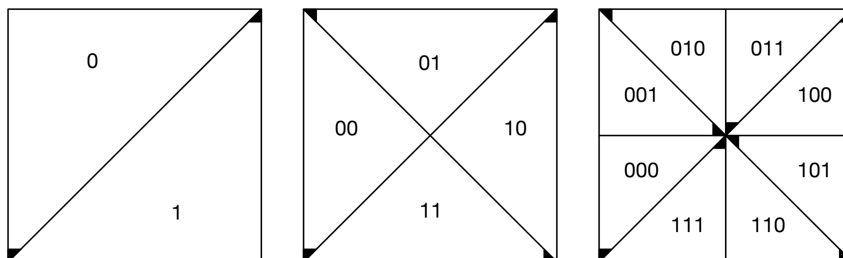


Figure 42: Partition in 2^k identical triangles

One vertex of each triangle is marked to distinguish it from the others. The marked vertices are therefore visited in a sequence determined by their labels as shown in figure 41 ($k = 2$, left, $k = 4$, middle and $k = 8$ right).

This partition is clever because the effort required to invert ψ is linear in the size of the input data (see the proof and the algorithm to compute positions along the space-filling curve given in the paper).

21.2 Matlab code

Main program

```
%% TSP solving using space-filling curves
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

%  Inspired from
%  John Bartholdi III, Loren Platzman
%  An O(N log N) planar travelling salesman heuristic based on spacefilling
% curves. Operations Research Letters, 1(4):121:125, September 1982

%  Loren Platzman, John Bartholdi III
%  Spacefilling curves and the planar Travelling Salesman Problem
%  Journal of the Association for Computing Machinery, 36(4):719-737, ...
    October 1989

clear
close all
X=rand(40,2);
n = length(X);
% Normalized coordinates
x = X(:,1); y = X(:,2);
xrange = max(x)-min(x); yrange = max(y)-min(y);
if xrange > yrange
    xnorm = (x-min(x))/xrange;
    ynorm = (y-min(y))/xrange;
```

```
else
    xnorm = (x-min(x))/yrange;
    ynorm = (y-min(y))/yrange;
end
theta = zeros(n,1);
for i=1:n
    theta(i)=compth(app,xnorm(i),ynorm(i));
end
plot(X(:,1),X(:,2),'kx','MarkerSize',10,'Linewidth',2);
hold on
[~,tour]=sort(theta);
tourdisp = [tour ; tour(1)];
plot(X(tourdisp,1),X(tourdisp,2),'r','Linewidth',2);
% Add title
L = sum(sqrt(diff(X(tourdisp,1)).^2 + diff(X(tourdisp,2)).^2));
str = sprintf('Tour Length: %g',L);
title(str)
```

Functions

```
function theta=compth(x,y)
% Compute the inverse mapping of psi
% psi : [0, 1] --> [0, 1]^2
%    : theta --> (x,y)
% i.e. find theta such that (x,y) = psi(theta)
M = 1;
kmax = 10;
theta = 0;
k = 1;
if x > y
    theta = 1;
    x = M - x;
    y = M - y;
end
while k < kmax
    theta = theta + theta;
    k = k + 1;
    if x+y > M
        theta = theta + 1;
        z = M - y;
        y = x;
        x = z;
    end
    if k < kmax
        theta = theta + theta;
        k = k + 1;
        x = x + x;
        y = y + y;
        if y > M
            theta = theta + 1;
            z = y - M;
            y = M - x;
            x = z;
        end
    end
end
end
```

22 Metropolis algorithm – Markov Chain Monte Carlo

22.1 Method principle

A discrete-time Markov chain is a sequence of random variables $X_1, X_2, X_3, \ldots$, with the Markov property, namely that the probability of moving to the next state depends only on the present state and not on the previous states:

$$\Pr(X_{n+1} = x \mid X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n) = \Pr(X_{n+1} = x \mid X_n = x_n)$$

If the state space is finite, the transition probability distribution or transition kernel can be represented by a matrix, called the transition matrix, with the (i, j)th element of P equal to:

$$p_{ij} = \Pr(X_{n+1} = j \mid X_n = i)$$

Each row of P sums to one and all elements are non-negative. A stationary distribution π is a vector, whose entries are non-negative and sum to 1, that is unchanged by the operation of transition matrix P on it:

$$\pi P = \pi$$

Markov chain Monte Carlo (MCMC) is a class of algorithms used to draw samples from a probability distribution. Given a probability distribution, one can construct a Markov chain whose elements' distribution approximates it; that is, the Markov chain's stationary distribution matches the target distribution.

The Metropolis algorithm is a Markov chain Monte Carlo (MCMC) method. It can draw samples from any probability distribution with probability density $P(x)$, provided a function $f(x)$ proportional to the density P is known and the values of $f(x)$ can be calculated. The requirement that $f(x)$ must only be proportional to the density, rather than exactly equal to it, makes the Metropolis algorithm particularly useful.

To accomplish this, the algorithm uses a Markov process, which asymptotically reaches a unique stationary distribution $\pi(x)$ such that $\pi(x) = P(x)$. A Markov process is uniquely defined by its transition probabilities $P(x' \mid x)$, the probability of transitioning from any given state x to any other given state x'. It has a unique stationary distribution $\pi(x)$ when the following two conditions are met: a) Existence of stationary distribution; b) Uniqueness of stationary distribution. The Metropolis algorithm involves designing a Markov process (by constructing transition probabilities) that fulfills the two above conditions, such that its stationary distribution $\pi(x)$ is chosen to be $P(x)$.

The method used to propose new candidates is characterized by the probability distribution $g(x \mid y)$ of a new proposed sample x given the previous sample y. This is called the proposal density, proposal function, or jumping distribution. A common choice for $g(x \mid y)$ is a Gaussian distribution centered at y, so that points closer to y are more likely to be visited next, making the sequence of samples into a Gaussian random walk.

Metropolis algorithm

Let $f(x)$ be a function that is proportional to the desired probability density function $P(x)$.

1. Initialization: Choose an arbitrary point x_0 to be the first observation in the sample and choose a proposal function $g(x \mid y)$. In this section, g is assumed to be symmetric; in other words, it must satisfy $g(x \mid y) = g(y \mid x)$.
2. For each iteration t:
   - Propose a candidate x' for the next sample by picking from the distribution $g(x' \mid x_t)$.
   - Calculate the acceptance ratio $\alpha = f(x')/f(x_t)$, which will be used to decide whether to accept or reject the candidate. Because f is proportional to the density P, we have that $\alpha = f(x')/f(x_t) = P(x')/P(x_t)$.
   - Accept or reject:
     - Generate a uniform random number $u \in [0, 1]$.
     - If $u \leq \alpha$, then accept the candidate by setting $x_{t+1} = x'$,
     - If $u > \alpha$, then reject the candidate and set $x_{t+1} = x_t$ instead.

Now how to use this algorithm to perform stochastic optimization? We are interested in the problem of minimizing a function J on a finite set $\mathcal{X}$ and with values in $\mathbb{R}$. A stochastic optimization method will most often seek to construct a sequence of random variables x_1, x_2 ... which converges in a probabilistic sense towards a minimizer of J. This function J is called energy function. Let us denote $\underline{J} = \min_{x \in \mathcal{X}} J(x)$.

We wish to construct a stochastic algorithm which converges to a minimum of J more precisely a point of $\operatorname{argmin} J = \{x \in \mathcal{X}, J(x) = \underline{J}\}$ (there may be several items in this set). A first idea is to consider a probability law μ_β (β a fixed parameter) which focuses on points of $\mathcal{X}$ where J is close to $\underline{J}$ and use the Metropolis algorithm to construct a Markov chain on $\mathcal{X}$ with invariant law μ_β. The simulation of this Markov chain is an algorithm which goes through $\mathcal{X}$ by focusing more on points of $\mathcal{X}$ where J is close to $\underline{J}$.

A family of probability laws μ_β that focuses on $\operatorname{argmin} J$ is the Gibbs measure (coming from statistical physics). It is a measure which gives the probability of the system being in state x (equivalently, of the random variable X having value x) as:

$$P(X = x) = \mu_\beta(x) = \frac{1}{Z(\beta)} e^{-\beta J(x)}, \quad \text{where} \quad Z(\beta) = \sum_{y \in \mathcal{X}} e^{-\beta J(y)}$$

The normalizing constant $Z(\beta)$ is the partition function. When $\beta > 0$, the probability $\mu_\beta(x)$ is greater than the energy $J(x)$ is small (compared to other values of J), the probability measure $\mu_\beta(x)$ therefore favors low energy configurations, all the more so as β is large. When $\beta = 0$ we obtain the uniform probability measure on $\mathcal{X}$.

We now have all the tools to solve the traveling salesman problem. For a problem including n vertices, the states correspond to the $(n-1)!/2$ permutation vectors of the first n integers.

The transition kernel, which allows, from a current cycle x to generate a candidate x' for the next cycle, can be chosen in different ways. Note that the probability distribution $g(x \mid y)$ needs not to be explicit in the Metropolis algorithm ; it just has to check $g(x \mid y) = g(y \mid x)$. The method needs only to know how to generate the sample x' from x. We can choose a simple swap operator (permutation of two indices i and j) in a vector describing a cycle, or more efficiently, use the 2-opt operator by reversing the sequence between two indices i and j. These transformations can be assimilated to the previous mentioned "random walk". The criterion to be optimized, the energy function, corresponds to the length of the Hamiltonian cycle ; for a state x, let us denote its length by $\ell(x)$. The acceptance ratio $\alpha = f(x')/f(x_t)$ of the Metropolis algorithm becomes:

$$\alpha = \frac{\mu_\beta(x')}{\mu_\beta(x_t)} = \frac{e^{-\beta l(x')}}{e^{-\beta l(x_t)}} = e^{-\beta(l(x')-l(x_t)}$$

Then, the metropolis algorithm for solving he TSP is as follows:

Algorithm 7 Metropolis algorithm for TSP

Require: X: matrix of the vertex coordinates of dimension $n \times 2$
Require: $maxiter$: number of calculus iterations
1: $\beta \leftarrow 50$ // *inverse temperature coefficient*
2: $tour \leftarrow$ random permutation of the integers from 1 to n
3: $tour_opt \leftarrow tour$
4: $min_length \leftarrow tour_length(X, tour)$
5: **for** $iter$ from 1 to $maxiter$ **do**
6: $new_tour \leftarrow reverse(tour)$
7: $\Delta \leftarrow tour_length(X, new_tour) - tour_length(X, tour)$
8: $r = \exp(-\beta\Delta)$
9: $u \leftarrow$ random number uniformly distributed between 0 to 1
10: **if** $u < r$ **then**
11: $tour \leftarrow new_tour$
12: **end if**
13: **if** $tour_length(X, tour) < min_length$ **then**
14: $tour_opt \leftarrow tour$
15: $min_length \leftarrow tour_length(X, tour)$
16: **end if**
17: **end for**
18: **return** $tour_opt$

where the $tour_length()$ function calculates the tour length and the $reverse()$ function realized a 2-opt operation as described in section 10 between two arbitrary chosen indices i and j independently of the decrease in cycle length.

22.2 Matlab code

Main program

```
%% TSP solving using Metropolis algorithm
%  Included in TSP20024 app
```

```
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

%  Inspired from
%  https://courses.cs.washington.edu/courses/cse312/...
%  22wi/files/student_drive/9.6.pdf
%  and
%  https://souleymanebagayogo.wordpress.com/2016/04/22/...
%. maltabla-methode-mcmc-pour-resoudre-le-probleme-du-voyageur-de-commerce/

clear
close all
X = rand(80,2);
D = squareform(pdist(X,'euclidean'));
maxiter = 10^5;
[tour_opt,min_length,TL] = metropolis(X,maxiter);
tour_opt = [tour_opt tour_opt(1,:)];
figure(1)
plot(X(:,1), X(:,2),'kx','MarkerSize',8,'LineWidth',2);
hold on
plot(X(tour_opt,1), X(tour_opt,2), 'r','LineWidth',2);
% Add title
str = sprintf('Tour length: %g',min_length);
title(str)
figure(2)
plot(TL)
title('Decreasing of the minimum length of the Hamiltonian cycle')
str = sprintf('%i updates / %1.1g iterations',length(TL),maxiter);
xlabel(str)
```

Functions

```
function [tour_opt,min_length,TL] = metropolis(X,itermax)
% Algorithm initialisation
tour = 1:length(X);
tour_opt = tour;
min_length = tour_length(X,tour_opt);
TL=min_length;
% Main loop
for i=1:itermax
    new_tour = reverse(tour);
    u = rand(1);
    r = exp(-50*(tour_length(X,new_tour)-tour_length(X,tour)));
    tour = new_tour*(u<r) + tour*(u>=r);
    if tour_length(X,tour) < min_length
        min_length = tour_length(X,tour);
        tour_opt = tour;
        TL=[TL min_length];
    end
end
end
```

```
function reverse_tour = reverse(new_tour)
    reverse_tour = new_tour;
    n = length(new_tour);
    ind = randperm(n,2);
```

```
    i1=ind(1); i2=ind(2);
    if i1 < i2
        reverse_tour(i1:i2) = new_tour(i2:-1:i1);
    else
        reverse_tour(i1:-1:i2) = new_tour(i2:i1);
    end
end
```

```
function L = tour_length(X,tour)
    tour = [tour tour(1)];
    L = sum(sqrt(diff(X(tour,1)).^2+diff(X(tour,2)).^2));
end
```

23 Simulated annealing based method

23.1 Method principle

The simulated annealing method is inspired by metallurgy where it is sometimes necessary to anneal metal parts to change their physical properties. Indeed, when a piece of metal is deformed, defects in the crystal lattice are created which have the effect of making the part more brittle. To restore the plasticity of the metal part, it is brought to a high temperature to increase the mobility of the atoms of the crystal then it is cooled slowly by leaving the atoms to resume their position in the crystal lattice. The heating of metal provides the atoms with the energy necessary to extract themselves from a local minimum of energy to end up in a global minimum (perfect crystal lattice).

To determine the shortest Hamiltonian cycle in the traveling salesman problem, one can start with any cycle and then make random modifications, keeping only those which induce a reduction in the length of the cycle. However, by proceeding in this way, we very quickly find ourselves in a local minimum, that is to say a cycle whose length can no longer be reduced by a single small change. Faced with this situation, it is better to accept a change that increases the length of the cycle and hope that the following changes will reduce it further.

The search algorithm, which is an enhancement of the previous Metropolis algorithm, then proceeds as follows:

- Generate a random cycle passing through the set of vertices
- As long as the temperature is above a threshold
  - Randomly swap two cycle vertices
  - If the cycle length has decreased, keep the obtained cycle
  - Otherwise keep the cycle obtained with a certain probability linked to the temperature
  - Cool down temperature

More precisely, the implementation is described by algorithm 8. In this algorithm, the *tour_length*() function calculates the length of the cycle defined by the sequence of vertices *tour*. Three parameters must be chosen by the user: the initial temperature (*initial_temp*),

the final temperature ($temp_threshold$) beyond which the algorithm will stop as well as the cooling factor ($cool_factor$).

Algorithm 8 Simulated annealing

Require: X: matrix of the vertex coordinates of dimension $n \times 2$
Require: $initial_temp$: initial temperature
Require: $cool_factor$: cooling factor
Require: $temp_threshold$: equilibrium temperature
1: $temp = initial_temp$
2: $tour \leftarrow$ random permutation of the integers from 1 to n
3: $previous_tour_length \leftarrow tour_length(X, tour)$
4: **while** $temp \geq temp_threshold$ **do**
5: $i \leftarrow$ random integer uniformly distributed between 1 to n
6: $j \leftarrow$ random integer uniformly distributed between 1 to n
7: $new_tour \leftarrow$ swap i and j in $tour$
8: $actual_tour_length \leftarrow tour_length(X, new_tour)$
9: $\Delta \leftarrow abs(actual_tour_length - previous_tour_length)$
10: **if** $actual_tour_length < previous_tour_length$ **then**
11: $previous_tour_length \leftarrow actual_tour_length$
12: $tour \leftarrow new_tour$
13: **else**
14: $r \leftarrow$ random number uniformly distributed between 0 to 1
15: **if** $r < e^{-\Delta/temp}$ **then**
16: $previous_tour_length \leftarrow actual_tour_length$
17: $tour \leftarrow new_tour$
18: **end if**
19: **end if**
20: $temp \leftarrow temp \times cool_factor$
21: **end while**
22: **return** $tour$

23.2 Matlab code

Inspired from Aravind Seshadri (2006).
Traveling Salesman Problem (TSP) using Simulated Annealing
https://fr.mathworks.com/matlabcentral/fileexchange/9612-traveling-salesman-problem-tsp-using-simulated-annealing. MATLAB Central File Exchange. Retrieved January 27, 2023.

Main program

```
%% TSP solving using simulated annealing
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

%  Part of this software inspired by Aravind Seshadri (2006).
%  Traveling Salesman Problem (TSP) using Simulated Annealing
%  https://fr.mathworks.com/matlabcentral/fileexchange/...
%. 9612-traveling-salesman-problem-tsp- using-simulated-annealing.
```

```
%  MATLAB Central File Exchange.

clear
close all
X=rand(50,2)
n = length(X);
p = randperm(n);
plot(X(:,1),X(:,2),'kx','MarkerSize',12,'LineWidth',2);
hold on

[p,L] = simulatedannealing(X,2000,0.980,8000,p);
```

Functions

```
function [p,L] = simulatedannealing(X,initial_temp,cooling_rate,threshold,p)
% SIMULATEDANNEALING

% %The input arguments are
% X            - The coordinates for n cities are represented by a matrix
%                with n rows and 2 columns
% initial_temp - The initial temperature to start the
%                simulatedannealing process.
% cooling_rate - Cooling rate for the simulatedannealing process.
%                Cooling rate should always be less than one.
% threshold    - Threshold is the stopping criteria in terms of
%                number of iterations
%
% Set the current temperature to initial temperature.
temp = initial_temp;
% Initialize the iteration number.
num_it = 1;
% This is a flag used to cool the current temperature after 10 iterations
% irrespective of wether or not the function is minimized. This is my
% recipe and done based on my experience. This is not part of the
% original algorithm.
complete_temperature_iterations = 0;

% Tour length
previous_dist = tour_length(X,p);

while num_it < threshold
    %  2 points are randomly swapped
    %p_temp = swapcities(p);
    p_temp = reverse(p); % Best choice
    current_dist = tour_length(X,p_temp);
    delta = abs(current_dist - previous_dist);
    if current_dist < previous_dist
        p = p_temp;
        if rem(num_it,100) == 0
            cla
            plot(x,y,'kx','MarkerSize',12,'LineWidth',2);
            % Polygon plot
            tourdisp =[p p(1)];
            plot(X(tourdisp(:,1),X(tourdisp(:,2),'r','LineWidth',2)
            pause(0.3)
            % Add title
            str = sprintf('Tour Length: %g',current_dist);
            title(str)
```

```
        end
        if complete_temperature_iterations >= 10
            temp = cooling_rate*temp;
            complete_temperature_iterations = 0;
        end
        previous_dist = current_dist;
        num_it = num_it + 1;
        complete_temperature_iterations = complete_temperature_iterations + 1;
    else
        if rand(1) < exp(-delta/(temp))
            p = p_temp;
            if rem(num_it,100) == 0
                cla
                plot(x,y,'kx','MarkerSize',12,'LineWidth',2);
                % Polygon plot
                tourdisp =[p p(1)];
                plot(X(tourdisp(:,1),X(tourdisp(:,2),'r','LineWidth',2)
                pause(0.3)
                % Add title
                str = sprintf('Tour Length: %g',current_dist);
                title(str)
            end
            previous_dist = current_dist;
            complete_temperature_iterations = ...
                complete_temperature_iterations + 1;
            num_it = num_it + 1;
        end
    end
end
L = current_dist;
cla
plot(x,y,'kx','MarkerSize',12,'LineWidth',2);

% Polygon plot
tourdisp =[p p(1)];
plot(X(tourdisp(:,1),X(tourdisp(:,2),'r','LineWidth',2)
% Add title
str = sprintf('Tour Length: %g',current_dist);
title(str)
end
```

```
function s = swapcities(p)
% SWAPCITIES
% s=SWAPCITIES(p) returns a set of cities where two cities are randomly swaped.
s = p;
city_1 = round(length(p)*rand(1));
if city_1 < 1
    city_1 = 1;
end
city_2 = round(length(p)*rand(1));
if city_2 < 1
    city_2 = 1;
end
temp = s(city_1);
s(city_1) = s(city_2);
s(city_2) = temp;
end
```

```
function L = tour_length(X,tour)
    tour = [tour tour(1)];
    L = sum(sqrt(diff(X(tour,1)).^2+diff(X(tour,2)).^2));
end
```

```
function reverse_tour = reverse(new_tour)
    reverse_tour = new_tour;
    n = length(new_tour);
    ind = randperm(n,2);
    i1=ind(1); i2=ind(2);
    if i1 < i2
        reverse_tour(i1:i2) = new_tour(i2:-1:i1);
    else
        reverse_tour(i1:-1:i2) = new_tour(i2:i1);
    end
end
```

23.3 Explanations

The implemented functions differ slightly from the previously described generic algorithm. The stopping rule does not rely on reaching an equilibrium temperature, but only on an arbitrarily fixed number of iterations. The temperature decreases after 10 iterations whether the length of the cycle has decreased or not. This method has many tuning parameters that are not easy to adjust. Here, all these parameters have been fixed (which does not always lead to very good results). The initial temperature was set to 2000, the cooling factor to 0.98 and the maximum number of iterations to 8000.

24 Kohonen map inspired method

24.1 Method principle

The traveling salesman problem can be solved by a method inspired by the development of a Kohonen map also called a Self-Organized Map – SOM. Self-organized maps or topological Kohonen maps are a particular class of neural networks called "competitive neural networks" where each neuron competes with its neighbours (topological aspect) for updating its synaptic weights. In general, self-learning (unsupervised learning) involves frequently changing the synaptic weights of the network in response to a set of input data. Changes in weights are made according to a set of learning rules. The purpose of a Kohonen map is to capture the topology and probability distribution of the input data. Originally, the networks used were based on a mono or bi-dimensional architecture of neurons. A competitive algorithm is used to train the network. In this training mode, a "winning" neuron is allowed to learn, but some neurons in its vicinity are also allowed to learn but with a decreasing learning rate as the distance (to be defined) to the winning neuron increases. The synaptic weights of the neurons are gradually modified in order to preserve the topological information of the input data when presented to the network.

To apply this approach to solving the traveling salesman problem, we build a two-layer network that consists of a two-dimensional input layer and an output layer with m neurons.

The evolution of the network can be imagined geometrically as the stretching of a rubber band towards the coordinates of the vertices to be traversed.

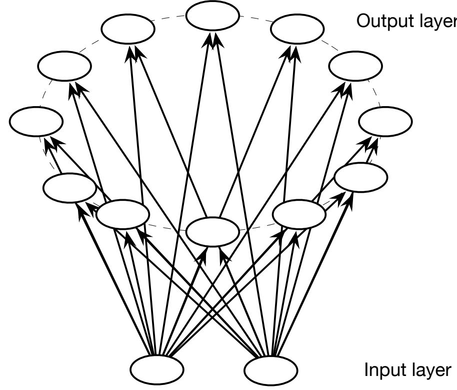


Figure 43: Topological structure of the network

Input data are coordinates of vertices $e_i \in \mathbb{R}^2, i = 1, \ldots, n,\ e_i = \begin{pmatrix} e_{x,i} \\ e_{y,i} \end{pmatrix}$. Each neuron $j = 1, \ldots, m$ of the network is also characterized by spatial coordinates in $\mathbb{R}^2$ which are its synaptic weights $s_j = \begin{pmatrix} s_{x,j} \\ s_{y,j} \end{pmatrix}$.

As with training any neural network, the weights can be initialized randomly. Here, as these weights have a geometric meaning (the 2D position of the neurons), other choices can be made. One can choose to arrange the neurons on a regular grid covering the area where the vertices are located. We can also arrange the neurons on a circle. Whatever the choice made, the neurons are numbered and constitute the sequence describing the previously mentioned “rubber band”.

The input data (coordinates of the vertices) are presented to the network in a random order and a competition based on the Euclidean distance between the presented vertex e_i and the set of neurons is then performed. The winning neuron J is the one closest to the vertex presented:

$$J = \text{Argmin}_j ||e_i - s_j||_2 \tag{1}$$

where $||\cdot||_2$ represent the Euclidean distance.

We then move the winning neuron but also its neighbours towards the vertex e_i. Different functions defining the neighbourhood can be defined. The Gauss function (exponential decrease according to a distance) is the most used:

$$f(\sigma, d) = e^{-d^2/\sigma^2} \tag{2}$$

The standard deviation σ conventionally defines the range of attraction. The chosen distance d is counted in number of edges separating two neurons in the cycle that they constitute by taking them in the order of their numbering.

The updating of the weights (of the position of the neurons) is then carried out according to:

$$s_j = s_j + \alpha f(\sigma, d)\,(e_i - s_j) \tag{3}$$

where α is a learning rate to set.

Consider the cycle of neurons

$$1 - 2 - 3 - 4 - 5 - 6 - 7 - 8$$

Suppose the winning neuron is number 6, the distances to consider are then:

$$3 - 4 - 3 - 2 - 1 - 0 - 1 - 2$$

indeed, neuron 6 is at a zero distance from itself, neuron 8 is indeed at a distance equal to 2 (number of edges of the chain $6 - 7 - 8$) and neuron 1 at a distance equal to 3 (number of edges in the chain $6 - 7 - 8 - 1$ because it is a cycle). Mathematically, this distance is defined by:

$$d = \min(|J - j|, m - |J - j|)$$

Indeed, if we take the previous example, we have:

$$\begin{array}{rl} \text{initial cycle} & 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 \\ |J - j| & 5 - 4 - 3 - 2 - 1 - 0 - 1 - 2 \\ m - |J - j| & 3 - 4 - 5 - 6 - 7 - 8 - 7 - 6 \\ d & 3 - 4 - 3 - 2 - 1 - 0 - 1 - 2 \end{array}$$

To ensure the convergence of the process, the two tuning parameters: learning rate α and standard deviation defining the neighbourhood σ can be adapted during the iterations of the calculation. The literature proposes various adaptation laws. For example, starting from initial values α_0 and σ_0, Kohonen proposes to adapt as follows:

$$\alpha_k = \alpha_0 e^{-k/\tau_1}, \qquad \sigma_k = \sigma_0 e^{-k/\tau_2}$$

where k is the iteration number and τ_1 and τ_2 time constants to choose. Other authors propose, on the basis of a heuristic, to choose:

$$\alpha_k = \frac{1}{\sqrt[4]{k}}, \qquad \sigma_k = \sigma_{k-1}(1 - 0.01k) \text{ with } \sigma_0 = 10$$

We can also, more simply, implement a very simple adaptation law such as:

$$\alpha_k = \beta \alpha_{k-1}, \qquad \sigma_k = \beta \sigma_{k-1} \tag{4}$$

which requires defining the initial values α_0 and σ_0 as well as the decay factor β.

The stopping rules of this iterative calculation are multiple. We can stop when the number of iterations reaches a predefined value (number of examples presented to the network), when the learning rate is below a given threshold (this rate intervenes in a multiplicative way in the updating of weight and s_j no longer changes when this rate is too low) or when the standard deviation which defines the zone of attraction is less than a given threshold

(in this case the function $f(\sigma, d)$ tends to 0 and again s_j no longer evolves).

Once out of the iterative calculation, it is necessary to create the permutation vector describing the sought cycle (order in which it is necessary to visit the various vertices). It suffices for this, to associate with each vertex, an index corresponding to the number of the nearest neuron, then to sort these indices (remember that each neuron was initially assigned a sequence number defining the initial sequence which describes the shape of the rubber band).

The algorithm succinctly summarizing the sequence of processing is as follows:

Algorithm 9 Kohonen map

Require: X: matrix of the vertex coordinates of dimension $n \times 2$
Require: $iter$: max number of iterations
Require: α: initial learning_rate
Require: β: decreasing factor
1: $number_of_neurons \leftarrow 8n$
2: randomize weights (positions) of all neurons
3: **for** $it = 1$ to $iter$ **do**
4: choose one random vertex from X
5: find the closest neuron (the winning one) (eq. 1)
6: compute the neighbourhood of the winning neuron (eq. 2)
7: update the synaptic weights of the neurons (eq. 3)
8: decay the learning rate and neighbourhood function (eq. 4)
9: **end for**
10: **for** $i = 1$ to n **do**
11: $index(i) \leftarrow$ queueing number of the closest neuron to the ith vertex
12: **end for**
13: sort $index$ and compute the permutation vector $tour$
14: **return** $tour$

24.2 Matlab code

Inspired from Diego Vicente (2018).
som-tsp: Python codes uploaded on https://github.com/diego-vicente/som-tsp/.

Main program

```
%% TSP solving using Kohonen or Self Organizing Map (SOM)
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

clear
close all
X=rand(50,2);
```

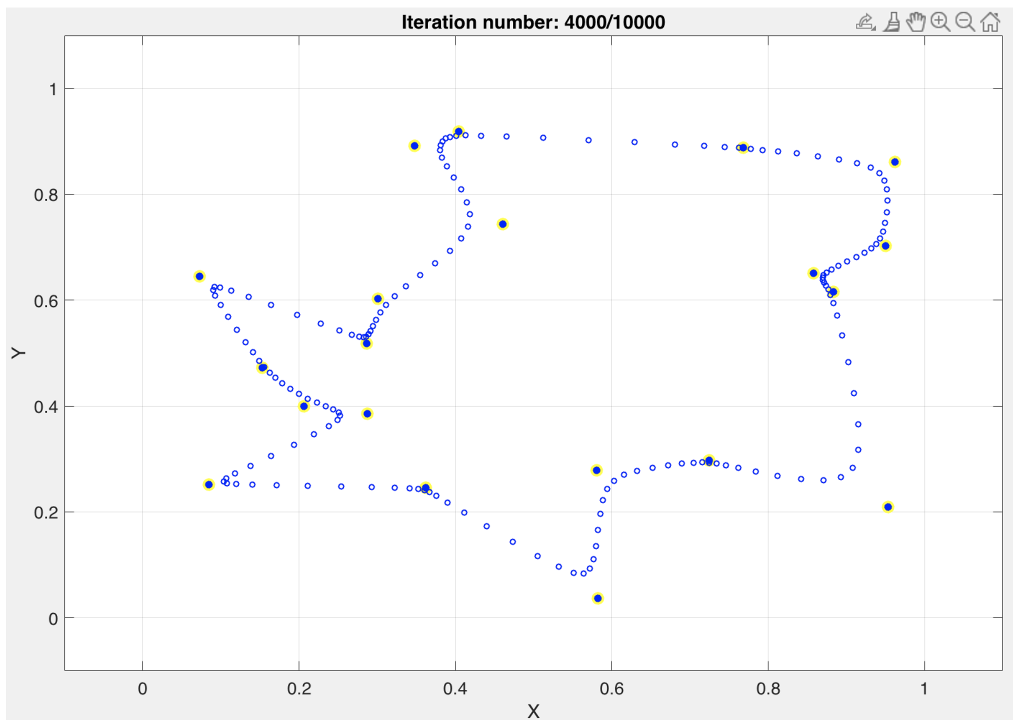


Figure 44: Intermediate stretching of the rubber band

```
lx=length(X);

iter=10000;
learning_rate=0.8;

% Normalization
Xn=normalize(X);
% The population of neurons is 8 times the number of cities
n=length(X)*8;
% Generate the network of neurons
network=rand(n,2);
range=n

for i=1:iter
    % Choose a random point
    rand_point=Xn(randperm(lx,1),:);
    winner_idx=select_closest(network,rand_point);
    % Generate a filter that applies changes to the winner's Gaussian
    gaussian=get_neighborhood(winner_idx,fix(range/10),n);
    % Update the network's weights (closer to the point)
    network=network+gaussian.*(learning_rate*(rand_point-network));
    % Decay the variables
    learning_rate=learning_rate*0.9997;
    range=range*0.9997;
    % Check if any parameter has completely decayed
    if range<1
        disp('Range')
        break
    end
```

```
    if learning_rate < 0.001
        disp('Learning_rate')
        break
    end
    % Show the result each 200 iterations
    if rem(i,200)==0
       cla
       plot(Xn(:,1),Xn(:,2),'kx','Markersize',12,'LineWidth',2)
       hold on
       x = network(:,1); x = [x;x(1)];
       y = network(:,2); y = [y;y(1)];
       plot(x,y,'LineWidth',1)
       plot(x,y,'ok')
       str=['Iteration : ' num2str(i)];
       title(str)
       pause(0.3)
    end
end
p = get_route(Xn,network);

cla
plot(X(:,1),X(:,2),'kx','Markersize',12,'LineWidth',2)
hold('on')
tourdisp = [p p(1)];
plot(X(tourdisp,1),X(tourdisp,2),'r','LineWidth',2)
```

Functions

```
function closest = select_closest(candidates,origin)
% return the index of the closest candidate to the origin point
[~,closest] = min(pdist2(candidates,origin));
end
```

```
function gaussian = get_neighborhood(center,radix,domain)
if radix < 1
    radix=1;
end
deltas = abs(center-[1:domain]');
distances = min(deltas,domain-deltas);
gaussian = exp(-(distances.*distances)/(2*radix*radix));
end
```

```
function norm_points = normalize(points)
% return the normalized version of a given vector points
% Remove the initial offset and normalize the points in a proportional
% interval [0, 1] on y maintaining the original ratio on x
ratio = ...
    [(max(points(:,1))-min(points(:,1)))/(max(points(:,2))-min(points(:,2))) ...
    1];
ratio = ratio/max(ratio);
norm_points(:,1) = ...
    (points(:,1)-min(points(:,1)))./(max(points(:,1))-min(points(:,1)));
norm_points(:,2) = ...
    (points(:,2)-min(points(:,2)))./(max(points(:,2))-min(points(:,2)));
norm_points = norm_points.*ratio;
end
```

```
function ordre = get_route(cities,network)
% return the route computed by a network
for i=1:length(cities)
    winner(i) = select_closest(network,cities(i,:));
end
[~,ordre]=sort(winner);
end
```

24.3 Explanations

For the application implemented, the last adaptation law mentioned was deployed for the learning rate with $\alpha_0 = 0.8$. The standard deviation is adapted according to a value *range* initially fixed at the number m of neurons and which decreases according to the factor β at each iteration. The standard deviation is then calculated as the integer part of $range/10$ with a minimum value equal to 1. The normalization function is not necessarily useful when the data (vertex coordinates) are generated in the square $0 \leq x \leq 1$ and $0 \leq y \leq 1$.

25 Bees algorithm

25.1 Method principle

The bees algorithm is a population-based search algorithm which was developed by Pham, Ghanbarzadeh *et al.* in 2005 [25]. It mimics the food foraging behaviour of honey bee colonies. In its basic version the algorithm performs a neighbourhood search combined with a global search. A simplified version of this algorithm adapted to the TSP problem is presented here. The interested reader will be able to discover how the behavior of the bees influenced the development of the algorithm, however, applied to the TSP, it is not necessary to resort to this explanation to understand how the research is carried out.

The algorithm maintains an ordered list of the best Hamiltonian cycles with their respective lengths. It starts by generating n_{scout} random Hamiltonian cycles (each route corresponds to a scout bee). Then, iteratively (with a number of iterations fixed in advance) the cycles are classified according to their lengths (from the smallest - the best one - to the largest). The n_{best} first cycles are denoted “best cycles”. Among them, the n_{elite} first cycles, are called “elite cycles”. All the best cycles are then the subject of “local” improvement attempts by carrying out random operations of swap, reverse or insertion of vertices (disruption of the cycle with the aim of trying to find a shorter cycle). These improvement efforts differ depending on whether they are elite cycles or the remaining best cycles : n_{elite_bee} improvement attempts will be done for elite cycles and only n_{best_bee} ($n_{best_bee} < n_{elite_bee}$) for the remaining best cycles. The last $n_{scout} - n_{best}$ cycles are replaced by new cycles drawn randomly. The list of Hamiltonian cycles thus obtained is again sorted according to their length and the iterative process continues.

During the local search, three different operators are frequently used. In the swap operation, the order of two randomly selected vertices is replaced with each other.

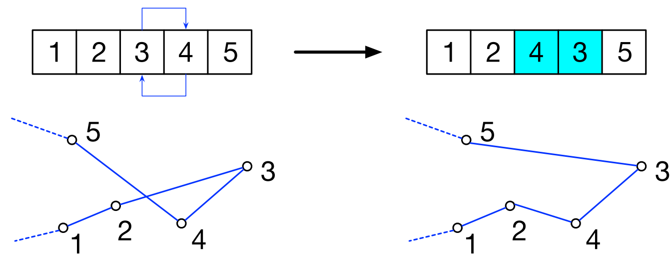


Figure 45: Swap operator

As shown in the figures, this operation, like the two following ones, pursues the same objective as the 2-opt procedure presented in section 10. As a consequence of the triangular inequality in a Euclidean space, the elimination of all the "crossings" of edges constituting the cycle makes it possible to reduce the length of the cycle.

In the insertion operation, the ranking of a randomly selected vertex is inserted to a different order, which is also randomly selected.

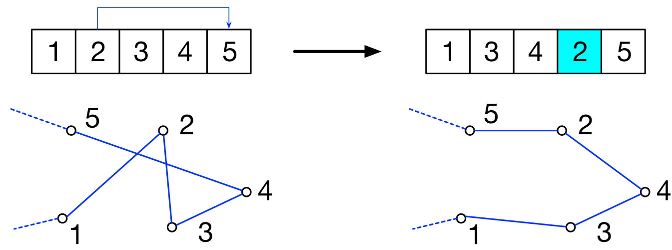


Figure 46: Insertion operator

In reversion, the order between two randomly determined vertices is reversed.

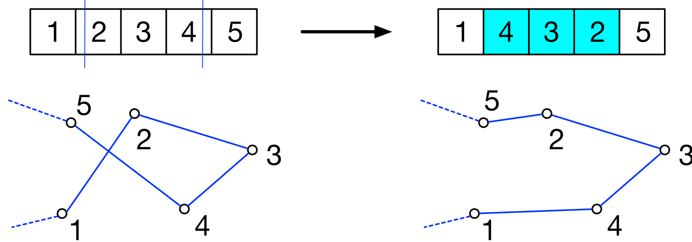


Figure 47: Reverse operator

Algorithm 5 describes the bees algorithm.

Algorithm 10 Bees algorithm

Require: X: matrix of the vertex coordinates of dimension $n \times 2$
1: *maxiter*: max number of iterations
2: *n_scout*: Number of scout bees
3: *n_elite*: Number of elite sites
4: *n_best*: Number of best sites
5: *n_elite_bee*: Number of recruited bees for elite sites
6: *n_best_bee*: Number of recruited bees for best sites
7: initialize the *n_scout* tours with random Hamiltonian cycles
8: sort the tours by ascending length
9: **for** $iter = 1$ to *maxiter* **do**
10:    **for** $i = 1$ to *n_elite* **do**
11:       *// Local optimisation for elite tours*
12:       **for** $j = 1$ to *n_elite_bee* **do**
13:          try to enhance $tour(i)$ applying randomly swap, inverse or insertion ops
14:       **end for**
15:    **end for**
16:    *// Local optimisation for the remaining best tours*
17:    **for** $i = n_elite + 1$ to *n_best* **do**
18:       **for** $j = 1$ to *n_best_bee* **do**
19:          try to enhance $tour(i)$ applying randomly swap, inverse or insertion ops
20:       **end for**
21:    **end for**
22:    *// Global optimisation - exploration of the domain*
23:    **for** $i = n_best + 1$ to *n_scout* **do**
24:       assign a random Hamilonian cycle to $tour(i)$
25:    **end for**
26:    sort the tours by ascending length
27: **end for**
28: **return** $tour(1)$

25.2 Matlab code

Main program

```
%% TSP solving using bees algorithm
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

clear
close all
X=rand(30,2);
n=length(X);
```

```
plot(X(:,1),X(:,2),'kx','MarkerSize',12,'LineWidth',2)
hold on

maxiter = 400;
n_scout = 25;       % Number of scout bees
n_elite = 4;        % Number of elite sites
n_best = 20;        % Number of best sites
n_elite_bee = 300;  % Number of recruited bees for elite sites
n_best_bee = 100;   % Number of recruited bees for best sites

% Initialization
tour = zeros(n_scout,n);
tlength = zeros(n_scout,1);

% Random tour for all scout bees
for i = 1:n_scout
    tour(i,:) = randperm(n);
    tlength(i) = tour_length(X,tour(i,:));
end

% Sorting of the solutions
[tlength,order] = sort(tlength);
tour = tour(order,:);

% Main iteration
for iter = 1:maxiter
    % Local optimization around elite site
    [tour,tlength] = local_search(1,n_elite,n_elite_bee,X,tour,tlength);
    % Local optimization around best site
    [tour,tlength] = local_search(n_elite+1,n_best,n_best_bee,X,tour,tlength);
    % Global optimisation
    for i = n_best+1:n_scout
        tour(i,:) = randperm(n);
        tlength(i) = tour_length(X,tour(i,:));
    end
    % Sorting of the solutions
    [tlength,order] = sort(tlength);
    tour = tour(order,:);

    % Polygon plot
    if rem(iter,20)==0
        cla
        best_tour = tour(1,:);
        best_length = tour_length(X,best_tour);
        tourdisp = [best_tour best_tour(1)];
        plot(X(tourdisp,1),X(tourdisp,2),'r','LineWidth',2);
        plot(X(:,1),X(:,2),'kx','MarkerSize',8,'LineWidth',2)
        pause(0.1)
    title(['Iteration number : ' num2str(iter)])
     end
end
% Add title
str = sprintf('Tour length: %g',tlength(1));
title(str)
```

Functions

```
function [tour,tlength] = ...
    local_search(n_first_site,n_last_site,n_tour,X,tour,tlength)
for i = n_first_site:n_last_site
    best_length = inf;
    for j = 1:n_tour
        new_tour = tour(i,:);
        p = randperm(3,1);
        if p==1
            new_tour = swap(new_tour);
            new_length = tour_length(X,new_tour);
            if new_length < best_length
                best_tour = new_tour;
                best_length = new_length;
            end
        elseif p==2
            new_tour = reverse(new_tour);
            new_length = tour_length(X,new_tour);
            if new_length < best_length
                best_tour = new_tour;
                best_length = new_length;
            end
        else
            new_tour = insert(new_tour);
            new_length = tour_length(X,new_tour);
            if new_length < best_length
                best_tour = new_tour;
                best_length = new_length;
            end
        end
    end
    if best_length < tour_length(X,tour(i,:))
        tour(i,:) = best_tour;
        tlength(i) = best_length;
    end
end
end
```

```
function swap_tour = swap(new_tour)
    swap_tour = new_tour;
    n = length(new_tour);
    ind = randperm(n,2);
    i1=ind(1); i2=ind(2);
    swap_tour(i1) = new_tour(i2);
    swap_tour(i2) = new_tour(i1);
end
```

```
function new_tour = reverse(new_tour)
    reverse_tour = new_tour;
    n = length(new_tour);
    ind = randperm(n,2);
    i1=ind(1); i2=ind(2);
    if i1 < i2
        reverse_tour(i1:i2) = new_tour(i2:-1:i1);
```

```
    else
        reverse_tour(i1:-1:i2) = new_tour(i2:i1);
    end
end
```

```
function insert_tour = insert(new_tour)
    insert_tour = new_tour;
    n=length(new_tour);
    ind = randperm(n,2);
    i1=ind(1); i2=ind(2);
    if i1 < i2
        insert_tour = new_tour([1:i1-1 i1+1:i2 i1 i2+1:end]);
    else
        insert_tour = new_tour([1:i2 i1 i2+1:i1-1 i1+1:end]);
    end
end
```

```
function L = tour_length(X,tour)
    tour = [tour tour(1)];
    L = sum(sqrt(diff(X(tour,1)).^2+diff(X(tour,2)).^2));
end
```

26 Reinforcement learning method

26.1 Method principle

Reinforcement learning[9] involves an agent, a set of states $\mathcal{S}$, and a set $\mathcal{A}$ of actions per state. By performing an action $a \in \mathcal{A}$, the agent transitions from state to state. Executing an action in a specific state provides the agent with a reward (a numerical score).

The goal of the agent is to maximize its total reward. It does this by adding the maximum reward attainable from future states to the reward for achieving its current state, effectively influencing the current action by the potential future reward. This potential reward is a weighted sum of expected values of the rewards of all future steps starting from the current state.

The algorithm, therefore, has a function that calculates the quality of a state–action combination:

$$Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$$

In this Q matrix, each element $Q(S_t, A_t)$ corresponds to a state S and an action A taken at time step t. Before learning begins, Q is initialized to a possibly arbitrary fixed value (chosen by the programmer). Then, at each time t the agent selects an action A_t, observes a reward R_{t+1}, enters a new state S_{t+1} (that may depend on both the previous state S_t and the selected action), and Q is updated. The core of the algorithm is a Bellman equation

[9] https://en.wikipedia.org/wiki/Q-learning

as a simple value iteration update, using the weighted average of the current value and the new information:

$$Q^{new}(S_t, A_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(S_t, A_t)}_{\text{current value}} + \alpha \cdot \Bigg(\underbrace{R_{t+1}}_{\text{reward}} + \gamma \cdot \underbrace{\max_a Q(S_{t+1}, a)}_{\text{estimate of optimal future value}} \Bigg)$$

where R_{t+1} is the reward received when moving from the state S_t to the state S_{t+1}, α is the learning rate ($0 < \alpha \leq 1$) and γ is the discount factor ($0 < \gamma \leq 1$).

The learning factor α determines to what extent the new calculated information will outperform the old one. If $\alpha = 0$, the agent learns nothing. Conversely, if $\alpha = 1$, the agent still ignores everything it has learned so far and will only consider the latest information. The discount factor γ determines the magnitude of future rewards. A factor $\gamma = 0$ would make the agent myopic by only considering current rewards, while a factor γ close to 1 would also involve more distant rewards. If the discount factor γ is close to or equal to 1, the value of Q may diverge.

Note that $Q^{new}(S_t, A_t)$ is the sum of three factors:

- $(1-\alpha)Q(S_t, A_t)$: the current value (weighted by one minus the learning rate)
- $\alpha \, R_{t+1}$: the reward R_{t+1} to obtain if action A_t is taken when in state S_t (weighted by learning rate)
- $\alpha\gamma \max_a Q(S_{t+1}, a)$: the maximum reward that can be obtained from state S_{t+1} (weighted by learning rate and discount factor)

During the process of reinforcement learning, it is important to balance between exploitation and exploration. This means that the algorithm cannot rely solely on the values in the Q matrix and must continue to explore new actions, rather than solely relying on experience. The ε-greedy strategy is used to determine when to exploit the Q matrix and when to explore randomly. The ε-greedy strategy selects the action with the highest Q value with a probability of $1 - \varepsilon$, and a random action with a probability of ε. The value of ε is initially set to a high value to encourage exploration, but it is gradually decreased as the learning progresses to encourage exploitation of the Q matrix.

When applying reinforcement learning to TSP problems, it is necessary to define a model that includes states, actions, and rewards. Here, the model is defined as follows:

- State: The state of the agent is defined as the current vertex the agent is visiting. In other words, the state of the agent is the vertex that has been visited by the agent so far.
- Action: The action of the agent is defined as the next vertex to visit. In other words, the agent chooses an action by selecting the next vertex to visit from the set of unvisited vertices.
- Reward: In TSP, the reward is related to the distance traveled between vertices. The goal of the agent is to minimize the total distance traveled. So different reward functions can be used: $R_{ij} = -d_{ij}$, $R_{ij} = -(d_{ij})^2$ or $R_{ij} = 1/d_{ij}$, where d_{ij} is the distance between the current vertex and the next vertex to visit.

As the number of learning cycles increases, the value of ε gradually decreases. Initially, route selection is completely random, but as reinforcement learning progresses, the value of ε decreases, and the selection becomes less random and more biased towards selecting the maximum value. Finally, reinforcement learning relies entirely on the maximum value to choose the next route.

26.2 Matlab code

Main program

```
%% TSP solving using Reinforcement learning by Q-learning
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange

% Inspired from
% Jiaying Wang, Chenglong Xiao, Shanshan Wang, Yaqi Ruan
% Reinforcement learning for the traveling salesman problem: Performance ...
    comparison of three algorithms
% The Journal of Engineering, e12303:2023.
% https://doi.org/10.1049/tje2.12303

clear
close all

X = rand(40,2);
n = length(X);
% Computation of the distance matrix
D = squareform(pdist(X,'euclidean'));
% Reward matrix
R = 1./D; % Or R = -D or R = -D.^2
% Q matrix
Q = zeros(size(R));
% Q-learning parameters
maxiter = 30000;    % Maximum number of iterations
epsilon = 1;        % Epsilon greedy strategy
alpha = 0.1;        % Learning rate
gamma = 0.45;       % Discount coefficient
best_tour = randperm(n);
best_length = tour_length(X,best_tour);

for iter=1:maxiter
    state = randi(n); % Initial vertex
    tour = state;
    for l=2:n
        tourb = 1:n; % Complementary list of unexplored vertices
        tourb(tour) = [];
        state = tour(end);
        % Epsilon-greedy strategy
        u = rand;
        if u < epsilon
            new_state = tourb(randi(length(tourb)));
        else
            current_Q = Q(state,:);
            current_Q(tour) = -inf;
```

```
            [maxQ,new_state] = max(current_Q);
            % Update of the Q matrix
            Q(state,new_state) =...
            (1-alpha)*Q(state,new_state)+alpha*(R(state,new_state)+gamma*maxQ);
        end
        tour = [tour new_state];
    end

    % Decrease of epsilon
    epsilon = epsilon*0.9999;
    if tour_length(X,tour) < best_length
        best_length = tour_length(X,tour);
        best_tour = tour;
    end
end
% Polygon plot
tourdisp = [best_tour best_tour(1)];
plot(X(:,1), X(:,2),'kx','MarkerSize',8,'LineWidth',2);
hold on
plot(X(tourdisp,1), X(tourdisp,2), 'r','LineWidth',2);
% Add title
str = sprintf('Tour length: %g',best_length);
title(str)
```

Functions

```
function L = tour_length(X,tour)
    tour = [tour tour(1)];
    L = sum(sqrt(diff(X(tour,1)).^2+diff(X(tour,2)).^2));
end
```

26.3 Explanations

For the proposed implementation, the learning rate α is set to 0.1, the discount factor γ to 0.45. The reward function is the inverse distance between the vertices. The Q matrix is initially set to 0. The number of learning episodes is set to 30 000. After each learning episode, ε is multiplied by the decay factor 0.9999.

27 Ant System method

27.1 Method principle

Artificial ants stand for multi-agent methods inspired by the behavior of real ants. The pheromone-based communication of biological ants is often the predominant paradigm used. The Ant System method was first presented by Dorigo [11]. We consider a colony of n_{ants} ants. Each ant is placed arbitrarily on a vertex of the considered graph.

Each ant constructs a Hamiltonian cycle within the graph. To select an edge in its tour, an ant considers the length of each edge available from its current position, as well as the corresponding pheromone level. Thus, each ant k computes a set $N_k(x)$ of vertices adjacent to vertex x that have not been visited yet by the kth ant and moves to one of these

in probability. For ant k, the probability p_{xy}^k of moving from vertex x to vertex y depends on the combination of two values, the attractiveness η_{xy} of the edge, as computed by some heuristic indicating the *a priori* (local) desirability of that move and the pheromone level τ_{xy} of the edge, indicating how proficient it has been in the past to use that edge. The pheromone level represents *a posteriori* (global) indication of the desirability of that edge.

The kth ant moves from vertex x to vertex y with probability

$$p_{xy}^k = \begin{cases} \frac{(\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})}{\sum_{z \in N_k(x)} (\tau_{xz}^{\alpha})(\eta_{xz}^{\beta})} & \text{if } j \in N_k(x) \\ 0 & \text{if } j \notin N_k(x) \end{cases}$$

where τ_{xy} is the amount of pheromone deposited for transition from vertex x to vertex y, $0 \leq \alpha$ is a parameter to control the influence of τ_{xy}, η_{xy} is the desirability of vertex transition xy (a priori knowledge, typically $1/d_{xy}$, where d_{xy} is the length of the edge xy) and $\beta \geq 1$ is a parameter to control the influence of η_{xy}. τ_{xz} and η_{xz} represent the pheromone levels and attractiveness for the other possible vertices transitions.

Once all ants have completed their tour, the levels of pheromone of all edges are updated according to the rule

$$\tau_{xy} \leftarrow (1 - \rho) \left(\tau_{xy} + \sum_{k=1}^{n_{ants}} \Delta\tau_{xy}^k \right)$$

where τ_{xy} is the amount of pheromone deposited for a vertex transition xy, ρ is the pheromone evaporation coefficient and $\Delta\tau_{xy}^k$ is the amount of pheromone deposited by k^{th} ant given by

$$\Delta\tau_{xy}^k = \begin{cases} Q/L_k & \text{if ant } k \text{ uses edge } xy \text{ in its tour} \\ 0 & \text{otherwise} \end{cases}$$

where L_k is the length of the kth ant's tour and Q is a constant.

Another rule for updating the edge level of pheromone called Elitist Ant System, consists to reinforce, at each iteration, the deposit of pheromone along the edges of the best known Hamiltonian cycle leading to:

$$\tau_{xy} \leftarrow (1 - \rho) \left(\tau_{xy} + \sum_{k=1}^{n_{ants}} \Delta\tau_{xy}^k + e\Delta\tau_{xy}^{bks} \right)$$

with

$$\Delta\tau_{xy}^{bks} = \begin{cases} Q/L_{bks} & \text{if the best tour uses edge } xy \\ 0 & \text{otherwise} \end{cases}$$

where L_{bks} is the length of the best ant's tour and e a constant coefficient corresponding to the chosen number of elitist ants. The idea of the elitist strategy is to give extra emphasis to the best tour found so far after every iteration.

27.2 Matlab code

Main program

```
%% TSP solving using ant colony optimization
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

clear;
close all;

X = rand(50,2)
n = length(X)
% Computation of the distance matrix
D = squareform(pdist(X,'euclidean'));

%% ACO Parameters

maxiter = 600;    % Maximum number of iterations
n_ants = 50;      % Number of ants
Q = 1;            % Pheromone quantity per tour
tau0 = 10*Q/(n_ants*mean(D(:)));    % Initial pheromone
alpha = 1;        % Pheromone exponential weight
beta = 1;         % Heuristic exponential weight
rho = 0.04;       % Evaporation rate
e = 8;            % Number of elitist ants

%% Initialization

eta = 1./D;               % Heuristic information matrix
tau = tau0*ones(n,n);     % Pheromone matrix
tour = zeros(n_ants,n);
tlength = zeros(n_ants,1);
best_length = inf;

%% Ant System Main Loop
for iter=1:maxiter
    % Move Ants
    for k=1:n_ants
        new_tour = randi(n);
        for l=2:n
            i = new_tour(end);
            P = tau(i,:).^alpha.*eta(i,:).^beta;
            P(new_tour) = 0;
            P = P/sum(P);
            j = roulette(P);
            new_tour = [new_tour j];
        end
        tour(k,:) = new_tour;
        tlength(k) = tour_length(X,tour(k,:));
        if tlength(k) < best_length
            best_tour = tour(k,:);
            best_length = tlength(k);
        end
    end
```

```
    % Update pheromone
    for k=1:n_ants
        ptour = tour(k,:);
        ptour = [ptour tour(k,1)];
        for l=1:n
            i = ptour(l);
            j = ptour(l+1);
            tau(i,j) = tau(i,j)+Q/tlength(k);
        end
    end

    % Reinforcement of pheromone for elitist ants
    ptour = best_tour;
    ptour = [ptour best_tour(1)];
    for l=1:n
        i = ptour(l);
        j = ptour(l+1);
        tau(i,j) = tau(i,j)+e*Q/best_length;
    end

    % Evaporation
    tau=(1-rho)*tau;

    % Polygon drawing
    cla
    tourdisp = [best_tour best_tour(1)];
    plot(X(tourdisp,1),X(tourdisp,2),'r','LineWidth',2);
    plot(X(:,1),X(:,2),'kx','MarkerSize',12,'LineWidth',2)
    pause(0)
    title(['Iteration number: ' num2str(iter)])
end
% Add title
str = sprintf('Tour length: %g',best_length);
title(str)
```

Functions

```
function node_j=roulette(app,P)
    r = rand;
    C = cumsum(P);
    node_j=find(r<=C,1);
end
```

```
function L = tour_length(X,tour)
    tour = [tour tour(1)];
    L = sum(sqrt(diff(X(tour,1)).^2+diff(X(tour,2)).^2));
end
```

28 Genetic algorithm

28.1 Method principle

A genetic algorithm (GA) is a meta-heuristic inspired by the process of natural selection. A population of candidate solutions to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties (its chromosomes or genotype) which can be mutated and altered.

The evolution is an iterative process and the population in each iteration is called a generation. This evolution usually starts from a population of randomly generated individuals. In each generation, the fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. The more fit individuals are stochastically selected from the current population, and each individual's genome is modified (recombined and possibly randomly mutated) to form a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

During each successive generation, a portion of the existing population is selected to reproduce for a new generation. This new generation of solutions is created from those selected, through a combination of genetic operators: crossover and mutation. For each new solution to be produced, a pair of "parent" solutions is selected for breeding from the pool selected previously. By producing a "child" solution using the methods of crossover and mutation, a new solution is created which typically shares many of the characteristics of its "parents". New parents are selected for each new child, and the process continues until a new population of solutions of appropriate size is generated.

Different parent selection methods can be used, the two most used being roulette wheel (or fitness proportionate) and tournament.

In roulette wheel selection, the fitness function assigns a fitness to possible solutions or chromosomes. This fitness level is used to associate a probability of selection with each individual chromosome. If f_i is the fitness of individual i in the population, its probability of being selected is

$$p_i = \frac{f_i}{\Sigma_{j=1}^{n} f_j}$$

where n is the number of individuals in the population.

Tournament selection involves running several "tournaments" among a few individuals (or "chromosomes") chosen at random from the population. The winner of each tournament (the one with the best fitness) is selected for crossover. Selection pressure, a probabilistic measure of a chromosome's likelihood of participation in the tournament based on the participant selection pool size, is easily adjusted by changing the tournament size. The reason is that if the tournament size is larger, weak individuals have a smaller chance to be selected, because, if a weak individual is selected to be in a tournament, there is a higher probability that a stronger individual is also in that tournament.

Some tuning parameters such as the mutation probability, crossover probability, population size and tournament size must be chosen to find reasonable settings for the problem class being worked on.

For solving the TSP using Genetic Algorithm, an Hamiltonian cycle is represented as a chromosome. Let us recall that a cycle is coded as an ordered sequence of the first n integers. Starting with a population of valid chromosomes, ordinary crossover and mutation operators cause problems. Indeed offspring generated by means of the ordinary operators have a high chance of being invalid with some cities missing, and others repeated. Many specialized crossover operators have been proposed in the literature to create only valid child chromosomes (i.e. constituted as a permutation of the first n integers). Let us cite the Partially-Mapped Crossover (PMX), the Order-Crossover (OX1) or Order Based Crossover (OX2) to name only these.

Let us explain PMX crossover operator. For the TSP, as an Hamiltonian cycle (a closed chain) is coded by a chromosome only one crossover point needs to be defined (corresponding to two gene segments). Given two parents s and t, PMX randomly picks a crossover point cp (an integer between 1 and the size of the chromosome). A child is then constructed in the following way. The first cp elements of s are copied in the first cp elements of the child chromosome. To keep the string a valid chromosome the cities in these positions are not just overwritten. To set position p to city c, the city in position p and city c swap positions. Below is an example of this coding and special crossover technique for two sample chromosomes: 5, 7, 1, 3, 6, 4, 2 and 4, 6, 2, 7, 3, 1, 5 and with $cp = 3$.

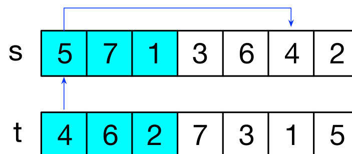


Figure 48: First step

The first element of t (4) is copied in the first position of s. The 5 (first element of s) and 4 (6th element of s) are swapped.

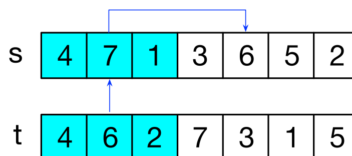


Figure 49: Second step

The second element of t (6) is copied in the second position of s. The 7 (second element of s) and 6 (5th element of s) are swapped.

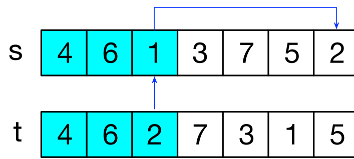


Figure 50: Third step

The third element of t (2) is copied in the third position of s. The 1 (third element of s) and 2 (7th element of s) are swapped.

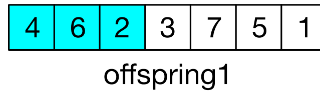


Figure 51: Chid chromosome

The first child chromosome is then obtained (figure 51). A second child can be obtained by exchanging the role of the parents s and t as presented in figure 52.

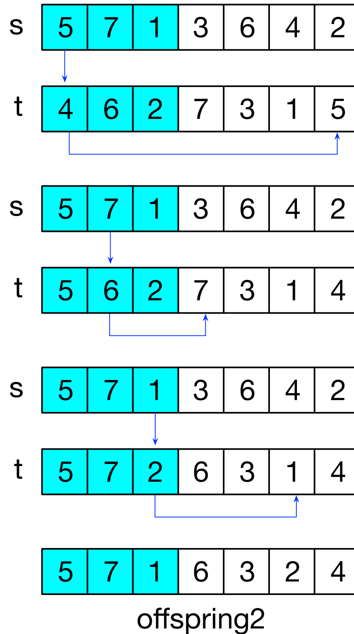


Figure 52: Creation process of the second child

As for the crossover operator, there exists a lot of mutation operator. The mutation operator chosen is the reversion operator because it corresponds to the processing carried out by the 2-opt algorithm which makes it possible to avoid "crossings" of edges (see section 25.1, figure 47).

28.2 Matlab code

Main program

```
%% TSP solving using genetic algorithm
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

clear
close all
X = rand(100,2);
n = length(X);

plot(X(:,1),X(:,2),'kx','MarkerSize',8,'LineWidth',2)
hold on

% Distance matrix
D = squareform(pdist(X,'euclidean'));

% Setting values for genetic algorithm
maxiter = 1000;        % Number of generations
crossover_prob = 0.9; % Crossover probability
mutation_prob = 0.06; % Mutation probability
popsize = 500;         % Size of the population
best_length = Inf;

% Generate population with random cycles
for i=1:popsize
    tour(i,:) = randperm(n);
end

next_gen = zeros(popsize,n);

% Genetic algorithm itself.
for  iter=1:maxiter
    % Calculate the length of the Hamiltonian cycles
    for i=1:popsize
        tlength(i) = tour_length(X,tour(i,:));
    end

    [min_length,ind]=min(tlength);
    best_tour=tour(ind,:);

    tournament_size=20;
    for k=1:popsize
        % Choosing parents for crossover operation bu using tournament ...
            approach.
        tournament_length=zeros(tournament_size,2);

        % First parent
        for i=1:tournament_size;
            random_tour = randi(popsize);
            tournament_length(i,1) = tlength(random_tour);
            tournament_length(i,2) = random_tour;
        end
        [~,ind] = min(tournament_length(:,1));
        parent1 = tour(tournament_length(ind,2),:);
```

```
        % Second parent
        for i=1:tournament_size;
            random_tour = randi(popsize);
            tournament_length(i,1) = tlength(random_tour);
            tournament_length(i,2) = random_tour;
        end
        [~,ind] = min(tournament_length(:,1));
        parent2 = tour(tournament_length(ind,2),:);

        % Child generation
        child = crossover(parent1,parent2,crossover_prob);
        child = mutate(child,mutation_prob);
        % New generation
        next_gen(k,:) = child;
    end
    % Assigning the created generation the current population.
    tour = next_gen;

    if min_length < best_length;
        best_length = min_length;
    end

    % Polygon plot
    if rem(iter,20) == 0
    cla
    tourdisp = [best_tour best_tour(1)];
    plot(X(tourdisp,1),X(tourdisp,2),'r','LineWidth',2);
    plot(X(:,1),X(:,2),'kx','MarkerSize',8,'LineWidth',2)
    pause(0)
    title(['Iteration number: ' num2str(iter)])
    end
end
% Add title
str = sprintf('Tour Length: %g',best_length);
title(str)
```

Functions

```
function mutated_tour = mutate(tour,prob)
    mutated_tour = tour;
    if rand <= prob;
        n = length(mutated_tour);
        index1 = randi(n);
        index2 = randi(n);
        if index1 < index2
            mutated_tour(index1:index2) = mutated_tour(index2:-1:index1);
        else
            mutated_tour(index1:-1:index2) = mutated_tour(index2:index1);
        end
    end
end
```

```
function cross_tour=crossover(tour1,tour2,prob)
    cross_tour = tour1;
    if rand <= prob;
```

```
        cp = randi(length(tour1));
        for cpc=1:cp
            if tour2(cpc)~=tour1(cpc) % if not same
                ind = find(tour1==tour2(cpc)); % then where where is it
                % swap:
                tour1(ind) = tour1(cpc); % send to that place
                tour1(cpc) = tour2(cpc);
            end
        end
        cross_tour = tour1;
    end
end
```

```
function L = tour_length(X,tour)
    tour = [tour tour(1)];
    L = sum(sqrt(diff(X(tour,1)).^2+diff(X(tour,2)).^2));
end
```

28.3 Explanations

The number of generations, corresponding here to the maximum number of iterations, is set to 1000, the size of population to 500, the probability of crossover to 0.9 and that of mutation to 0.06, the number of individuals selected for a tournament is set to 20.

29 Black hole algorithm

29.1 Method principle

The Black Hole algorithm is a nature-inspired algorithms [16]. It is based on the black hole phenomenon in the space. The gravitational power of the black hole is too high that even the light cannot escape from it. The gravity is so strong because matter has been squeezed into a tiny space. Anything that crosses the boundary of the black hole is swallowed by it and vanish.

In the Black Hole algorithm, like as other population-based algorithms a population of candidate solutions to the optimization problem is generated randomly. These candidate solutions are distributed in the problem space. After initialization, the fitness values of the candidate solutions are evaluated, and the best candidate among the population is chosen to be the black hole and the rest form the normal stars.

Then, the Black Hole algorithm evolves the candidate solutions towards the optimal solution via a simple mechanism. The black hole (best candidate) starts absorbing the stars (other candidates) around it, and all the stars start moving towards the black hole.

While the stars move towards the black hole, it is possible that a star reaches to a better location in the search space which has better fitness than the black hole position. In such a case, the black hole is replaced by that star, and then, the Black Hole algorithm continues with the new black hole and the stars start moving towards this new black hole.

During moving stars towards the black hole, there is the possibility of crossing the border of black hole (event horizon). Every star (candidate solution) that enters the range of the black hole will be sucked by the black hole. In such a case, every time a candidate (star) dies, another candidate solution (star) is born and distributed randomly in the search space. Once all the stars have moved to new locations, the next iteration takes place.

The radius of the event horizon in the Black Hole algorithm is calculated using the following equation:

$$R = \frac{f_{BH}}{\sum_{i=1}^{n} f_i}$$

where f_{BH} is the fitness value of the black hole, and f_i is the fitness value of the ith star, n being the number of stars (candidate solutions).

In the original method, the position updating of stars is given by:

$$x_i(t+1) = x_i(t) + \text{rand}.(x_{BH} - x_i(t))$$

where x_{BH} is the position of black hole and rand is an uniformly distributed random number in the interval $[0, 1]$.

For solving the TSP, a similar mechanism, corresponding to the attraction of the solutions (stars) towards the blackhole solution, for updating the solutions, must be chosen. A crossover operation between the stars and the black hole can play this role.

The order based crossover (OX2 operator) has been chosen. Let us explain how it works. As previously explained in the section dedicated to genetic algorithm, an Hamiltonian cycle is coded by a permutation vector of the first n integer for a problem with n vertices. This vector is considered as a chromosome. Given two parents, OX2 randomly picks two crossover points $cp1$ and $cp2$ (as integers between 1 and the size of the chromosome) and continues with the following procedure.

- Select the substring located between $cp1$ and $cp2$ from the blackhole chromosome.
- Produce a proto-child by copying the substring into its corresponding position.
- Delete the vertices which are already in the substring from the star chromosome. The resulted sequence of vertices contains the vertices that the proto-child needs.
- Place the vertices into the unfixed positions of the proto-child from left to right according to the order of the sequence to produce a child.

The figures 53 and 54 explain that kind of crossover.

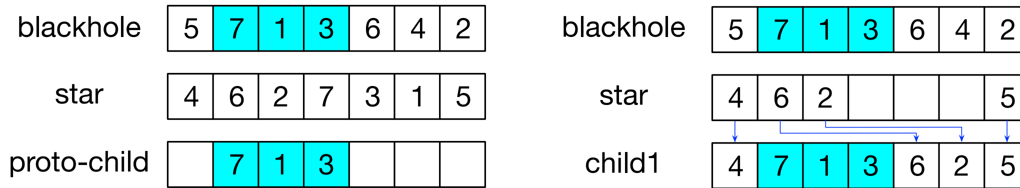


Figure 53: Generation of the first child ($cp1 = 2$ and $cp2 = 4$)

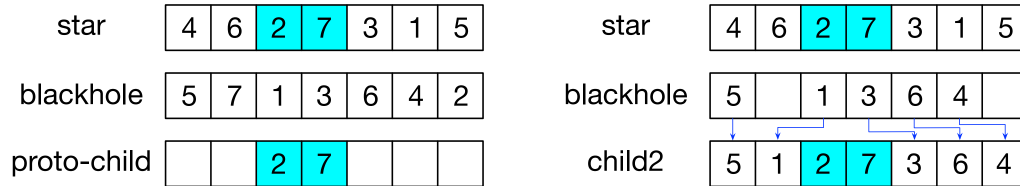


Figure 54: Generation of the second child ($cp1 = 3$ and $cp2 = 4$)

29.2 Matlab code

Inspired from Rahul (2021).
Python codes uploaded on https://github.com/Rahulk990/TSP_Using_AI.

Main program

```
%% TSP solving using black hole algorithm
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

clear
close all

X = rand(40,2);
n = length(X);
figure(1)
plot(X(:,1),X(:,2),'kx','MarkerSize',12,'LineWidth',2)
hold on

maxiter = 1000;
pop_size = 400;

% Initialization
tour = zeros(pop_size,n);
tlength = zeros(pop_size,1);

% Random tour
for i = 1:pop_size
    tour(i,:) = randperm(n);
    tlength(i) = tour_length(X,tour(i,:));
end
```

```
% Sorting of the solutions
[tlength,order] = sort(tlength);
tour = tour(order,:);

% Main iteration
for iter = 1:maxiter
    blackhole = tour(1,:);
    blackhole_length = tlength(1);
    % Moving stars towards the black hole
    for i = 2:pop_size
        newstar = move_closer(X,blackhole,tour(i,:));
        tour(i,:)=newstar;
        tlength(i)=tour_length(X,tour(i,:));
    end
    % Sorting of the solutions
    [tlength,order] = sort(tlength);
    tour = tour(order,:);

    % New blackhole
    blackhole = tour(1,:);
    blackhole_length = tlength(1);
    % Checking for event horizon
    blackhole_value = blackhole_length;
    sum_value = sum(tlength);

    % Death of stars
    radius = blackhole_value/sum_value;
    for j = 2:pop_size
        if abs(tlength(j)-blackhole_length) < radius/2
            tour(j,:) = randperm(n);
            tlength(j) = tour_length(X,tour(j,:));
        end
    end

    % Sorting of the solutions
    [tlength,order] = sort(tlength);
    tour = tour(order,:);
    % Polygon plot
    if rem(iter,40)==0 || iter < 20
        cla
        best_tour = tour(1,:);
        best_length = tlength(1);
        tourdisp = [best_tour best_tour(1)];
        plot(X(tourdisp,1),X(tourdisp,2),'r','LineWidth',2)
        plot(X(:,1),X(:,2),'kx','MarkerSize',8,'LineWidth',2)
        pause(0)
        title(['Iteration number : ' num2str(iter)])
    end
end
% Add title
str = sprintf('Tour length: %g',best_length);
title(str)
```

Functions

```
function min_tour = move_closer(X,blackhole,star)
    min_tour = star;
    min_length = tour_length(X,star);
    % Generation of two children
    child_star1 = OX2(X,blackhole,star);
    child_star2 = OX2(X,star,blackhole);
    length_child1 = tour_length(X,child_star1);
    length_child2 = tour_length(X,child_star2);

    % Selection of the best tour (star parent and 2 children)
    if length_child1 < min_length
        min_length = length_child1;
        min_tour = child_star1;
    elseif length_child2 < min_length
        min_length = length_child2;
        min_tour = child_star2;
    end
end
```

```
function new_star = OX2(X,blackhole,star)
    n = length(star);
    new_star = zeros(1,n);
    ind = sort(randperm(n,2));
    startpos = ind(1); endpos = ind(2);
    new_star(startpos:endpos) = blackhole(startpos:endpos);
    spos = [];
    cpstar = star;
    for i=startpos:endpos
        pos = find(star==blackhole(i));
        spos = [spos pos];
    end
    cpstar(spos) = [];
    newpos = 1:n;
    newpos([startpos:endpos]) = [];
    new_star(newpos) = cpstar;
end
```

```
function L = tour_length(X,tour)
    tour = [tour tour(1)];
    L = sum(sqrt(diff(X(tour,1)).^2+diff(X(tour,2)).^2));
end
```

29.3 Explanations

During one iteration, all stars are attracted by the black hole. This attraction consists of replacing the star with one of its two children obtained by OX2 crossover with the black hole if their fitness has improved (if the length of their corresponding Hamiltonian cycle has decreased). A star disappears and a new one is randomly created if its distance from the black hole is less than the event horizon characterized by a radius calculated as the ratio of the length of the Hamiltonian cycle represented by the star and the sum of the lengths of the Hamiltonian cycles for all stars.

The maximum number of iterations is set to 1000 and the size of the population to 400. As the proposed algorithm does not explicitly exploit the reverse operator, the final path frequently contains crossovers which can then be eliminated by applying the 2-opt heuristic.

30 Firefly algorithm

30.1 Method principle

The Firefly Algorithm (FA) is a metaheuristic inspired by the flashing behavior of fireflies. The primary purpose for a firefly's flash is to act as a signal system to attract other fireflies. FA uses the following three primary rules:

- All fireflies are unisex which means that they are attracted to other fireflies regardless of their sex.
- The degree of the attractiveness of a firefly is proportional to its brightness, thus for any two flashing fireflies, the less bright one will move toward the brighter one and their attractiveness will decrease as their distance increases. If there is no brighter one than a particular firefly, it will move randomly.
- The brightness or light intensity of a firefly is affected or determined by the landscape of the objective function to be optimized.

A solution representation for the TSP is a permutation representation. Here, a firefly represents one solution. In this representation, an element of an array represents a vertex and the index represents the order of a tour. Two adjacent elements thus indicate an edge between two vertices that lies along the tour.

The brightness of a firefly depends on the length of the cycle associated to a firefly. Since the purpose of the TSP is to find a cycle with the minimum length, a firefly which has less length cycle will have a higher light intensity (brighter) : $I = 1/cycle_length$; the light intensity of a firefly increases when the cycle length associated to a firefly decreases.

The attractiveness of a firefly is determined by its brightness, which is associated with the encoded objective function. The main form of the attractiveness function $\beta(d)$ can be any monotonic decreasing function:

$$\beta(d) = \beta_0 e^{-\gamma d^2}$$

where $\beta(d)$ is the attractiveness of a firefly when seen at distance d, d is the distance between two fireflies, β_0 is the brightness of a brighter firefly, and γ is a fixed light absorption coefficient.

Calculating attractiveness requires defining a distance between two fireflies. The distance between firefly i and firefly j can be defined as the number of different edges between them.

Firefly i	1	14	13	12	7	6	15	5	11	9	10	16	3	2	4	8
Firefly j	1	14	13	12	15	5	7	6	11	9	10	16	3	2	4	8

Let's consider the two previous fireflies. Three edges $12 - 15$, $5 - 7$, and $6 - 11$ in firefly j do not exist in firefly i. Hence, the number D of different edges between firefly i and firefly j is 3. Then, the distance d between two fireflies is calculated using:

$$d = 10D/n$$

n being the number of vertices. This equation scales d in the interval $[0, 10]$ as d will be used for the attractiveness calculation. Notice that the choice of this constant must be made in conjunction with the value of the γ parameter.

Indeed, the light absorption coefficient γ characterizes the variation of the attractiveness value of a firefly. Its value is very important in determining the speed of convergence and how the FA behaves. When $\gamma \to 0$, the attractiveness is quite constant and $\beta(d) = \beta_0$. In this case, the attractiveness of a firefly will not decrease when viewed by another. If $\gamma \to \infty$, the value of attractiveness of a firefly is close to zero when viewed by another firefly. No other fireflies can be seen, and each firefly roams in a completely random way. Therefore, this corresponds to a completely random search method.

The coefficient γ determines how much light intensity changes the attractiveness of a firefly over distances. The distance d scales in the interval $[1, 10]$ to handle the value of $e^{-\gamma d^2}$. If the distance is too high (more than 10), the value of $e^{-\gamma d^2}$ is close to zero, and the attractiveness of a firefly is also close to zero when viewed by another firefly, so the firefly will always move randomly.

Different movements reflecting the interaction of one firefly towards another have been defined. Here we have chosen a movement described by Gilang Kusuma Jati, Ruli Manurung and Suyanto in their paper entitled "Discrete Firefly Algorithm for Traveling Salesman Problem: A New Movement Scheme" [20]. The following presentation is copied from this paper. The idea for the new movement begins from the rules that state the distance between two fireflies must be reduced after a firefly moves toward another brighter firefly. Since the distance between two fireflies is calculated using the total number of different edges between them, then when a firefly moves toward another brighter firefly, the total number of different edges between them must also be reduced. The total number of different edges between them can be reduced by adding an edge that exists in the brighter firefly but does not exist in the less bright firefly in such a way that no similar edges between them that existed previously are removed.

The new movement scheme is illustrated by the following procedure. First, there are two fireflies, firefly i and another brighter firefly j. From all the edges that exist in firefly j but not in firefly i, randomly select one that will be added to firefly i, e.g., edge $x - y$ is the selected edge, where x and y are vertices in the TSP. Find the positions of vertices x and y in firefly i. For each vertex, we then build a **segment** around it, i.e., a maximally long sequence of edges that contains edges that are also found in firefly j. The procedure to build this segment is simple. Starting from a vertex, which we consider to be a segment of length 1, we keep extending the segment to the left and right, adding edges that are

found in firefly j, until it can no longer be extended. Let us call a segment built starting from node n as S_n.

Once the two segments S_x and S_y have been identified, we must merge them in such a way that vertices x and y appear adjacent to each other in firefly i, thus achieving the objective of adding edge $x - y$ to firefly i. There are four ways to merge these two segments, i.e., inserting S_x before S_y, inserting S_x after S_y, inserting S_y before S_x, and finally inserting S_y after S_x. Note that to ensure that edge $x - y$ appears in firefly i, some reverse of a segment might be required. Executing all four ways of merging these segments will yield four new fireflies (solutions).

The following figures illustrate this process through an example. Figure 55 shows two fireflies, firefly i and another brighter firefly j. The total number of different edges between them is seven, namely the edges $7 - 12$, $12 - 13$, $14 - 1$, $1 - 8$, $2 - 3$, $10 - 15$, and $5 - 6$ that exist in firefly j but do not exist in firefly i.

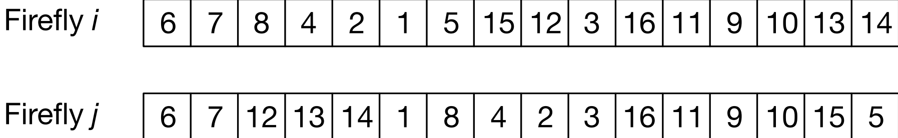


Figure 55: Two fireflies with seven different edges

Figure 56 shows the result of constructing segments in firefly i. The process begins by selecting one of the seven edges above that will be added to firefly i, say $2-3$ is the selected edge. We then locate the position of vertex 2 (marked with x) and vertex 3 (marked with y) in firefly i. We then construct a segment (edge sequences) for each node. Starting from node 2, we extend the segment to the left to include $4 - 2$ and $8 - 4$, since both these edges are also found in firefly j. However, we stop at node 8 since $7 - 8$ is not found in firefly j. Attempting to extend this segment to the right fails because the edge $2 - 1$ is also not found in firefly j. Likewise for node 3, we extend the segment to the right to include $3 - 16$, $16 - 11$, $11 - 9$, and $9 - 10$, since all of these edges are also found in firefly j. However, we stop at node 10 since $10 - 13$ is not found in firefly j. Attempting to extend this segment to the left fails because the edge $12 - 3$ is also not found in firefly j. In fact, it is guaranteed that each segment can only be extended in one direction, since the aim is to add the edge $x - y$ into firefly i, thus $x - y$ must not previously exist. Thus, S_x is $8 - 4 - 2$ and S_y is $3 - 16 - 11 - 9 - 10$.

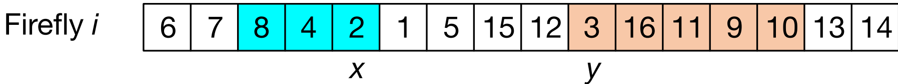


Figure 56: Firefly i with the segments of each vertex in the selected edge

Finally, the merging process of S_x and S_y is illustrated in figure 57. Fireflies i_1, i_2, i_3 and i_4 show the results of applying these four possibilities, yielding four new fireflies (solutions). Note that for fireflies i_2 and i_4, the segments S_x and S_y had to be reversed to ensure that

edge $x-y$ can be added. Note that in all of these solutions, edge $2-3$ has been successfully added to firefly i and has been done in such a way that no other edge belonging to firefly i that also appears in firefly j has been removed as a result. Thus, if distance is measured in terms of the number of different edges between two fireflies, the distance between them has now been decreased.

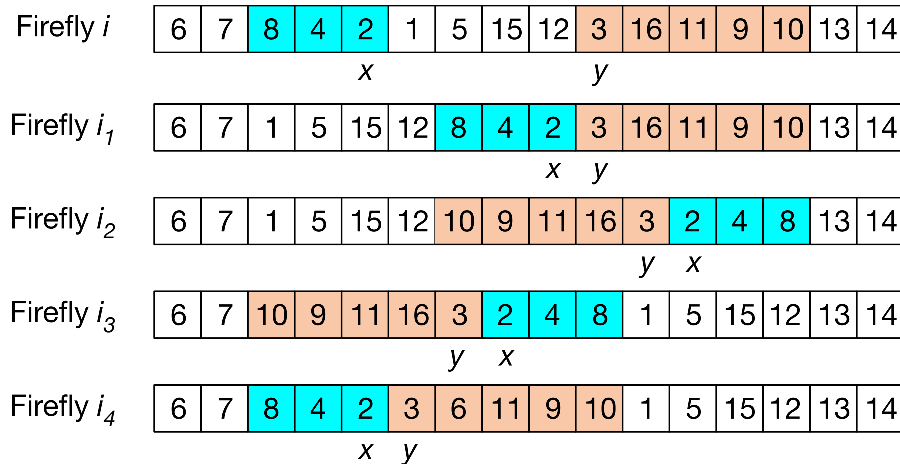


Figure 57: Merging process of two segments in firefly

30.2 Matlab code

Main program

```
%% TSP solving using Discrete Firefly Algorithm
%  Included in TSP20024 app
%  Didier Maquin (06/2025). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

% Part of this code has been generated by ChatGPT based on GPT-4-turbo
% (April 2024)

clear
close all;

% Problem description
n = 100;
X = rand(n, 2) * 100;
D = squareform(pdist(X));

% % Firefly algorithm parameters
nFireflies = 20;
maxIter = 300;
alpha = 0.2; % mutation probability
showInterval = 10;


% Initialization
```

```
for i = 1:nFireflies
    fireflies(i).tour = randperm(n);
    fireflies(i).cost = computeTourLength(fireflies(i).tour, D);
end

[~, bestIdx] = min([fireflies.cost]);
gBest = fireflies(bestIdx);

% Plot
figure('Name', 'Firefly Algorithm');
hPlot = plotTour(gBest.tour, X);
title(sprintf('Initial - Length %.2f', gBest.cost));
drawnow;

% Main loop
for iter = 1:maxIter
    for i = 1:nFireflies
        for j = 1:nFireflies
            if fireflies(j).cost < fireflies(i).cost
                newTour = edgeMove(fireflies(i).tour, fireflies(j).tour,D);
                % Mutation
                if rand < alpha
                    newTour = mutateSegmentReverse(newTour);
                end

                newCost = computeTourLength(newTour, D);

                if newCost < fireflies(i).cost
                    fireflies(i).tour = newTour;
                    fireflies(i).cost = newCost;

                    if newCost < gBest.cost
                        gBest.tour = newTour;
                        gBest.cost = newCost;
                    end
                end
            end
        end
    end

    % Periodic drawing
    if mod(iter, showInterval) == 0 || iter == 1 || iter == maxIter
        updatePlot(hPlot, gBest.tour, X);
        title(sprintf('Iter %d - Best length %.2f', iter, gBest.cost));
        drawnow;
    end
end
```

Functions

```
function L = computeTourLength(tour, D)
tour = [tour, tour(1)];
L = sum(D(sub2ind(size(D), tour(1:end-1), tour(2:end))));
end
```

```
function E = extractEdges(tour)
tour = [tour tour(1)];
E = sort([tour(1:end-1)', tour(2:end)'], 2); % unordered edge list
end
```

```
function tour = mutateSegmentReverse(tour)
idx = sort(randperm(length(tour), 2));
i = idx(1); j = idx(2);
tour(i:j) = tour(j:-1:i);
end
```

```
function h = plotTour(tour, coords)
coords = coords(:,:);
tour = [tour tour(1)];
h = plot(coords(tour,1), coords(tour,2), 'b-', 'LineWidth', 2);
hold on;
plot(coords(:,1), coords(:,2), 'ko', 'MarkerFaceColor', 'k');
axis equal; grid on;
end
```

```
function updatePlot(h, tour, coords)
tour = [tour tour(1)];
h.XData = coords(tour,1);
h.YData = coords(tour,2);
end
```

```
function tourNew = edgeMove(fireflies_i, fireflies_j,D)
tourNew = fireflies_i;
Ei = extractEdges(fireflies_i);
Ej = extractEdges(fireflies_j);
diffE = setdiff(Ej, Ei, 'rows');
if ~isempty(diffE)
    [a,~]=size(diffE);
    numedge = randi(a);
    x = diffE(numedge,1);
    [seqx, dirx] = genseq(fireflies_i,Ej,x);
    y = diffE(numedge,2);
    [seqy, diry] = genseq(fireflies_i,Ej,y);
    dx = find(fireflies_i==seqx(1));
    dy = find(fireflies_i==seqy(1));
    if dx < dy
        fireflies_i = circshift(fireflies_i,-dx+1);
        dy = dy-dx+1; dx = 1;
        seq1 = seqx;
        seq2 = fireflies_i(length(seqx)+1:dy-1);
        seq3 = fireflies_i(dy:dy+length(seqy)-1);
        seq4 = fireflies_i(dy+length(seqy):end);
    else
        fireflies_i = circshift(fireflies_i,-dy+1);
        dx = dx-dy+1; dy = 1;
```

```
        seq1 = seqy;
        seq2 = fireflies_i(length(seqy)+1:dx-1);
        seq3 = fireflies_i(dx:dx+length(seqx)-1);
        seq4 = fireflies_i(dx+length(seqx):end);
    end
    if dirx == 'l' && diry == 'r'
        fly(1).tour = [seq1 seq3 seq4 seq2];
        fly(2).tour = [flip(seq1) seq4 seq2 flip(seq3)];
        fly(3).tour = [flip(seq1) seq2 seq4 flip(seq3)];
        fly(4).tour = [seq1 seq3 seq2 seq4];
    end
    if dirx == 'r' && diry == 'r'
        fly(1).tour = [flip(seq1) seq3 seq2 seq4];
        fly(2).tour = [flip(seq1) seq3 seq4 seq2];
        fly(3).tour = [seq1 seq2 seq4 flip(seq3)];
        fly(4).tour = [seq1 seq4 seq2 flip(seq3)];
    end
    if dirx == 'r' && diry == 'l'
        fly(1).tour = [seq1 seq2 seq4 seq3];
        fly(2).tour = [seq1 seq4 seq2 seq3];
        fly(3).tour = [flip(seq1) flip(seq3) seq2 seq4];
        fly(4).tour = [flip(seq1) flip(seq3) seq4 seq2];
    end
    if dirx == 'l' && diry == 'l'
        fly(1).tour = [seq1 flip(seq3) seq2 seq4];
        fly(2).tour = [seq1 flip(seq3) seq4 seq2];
        fly(3).tour = [flip(seq1) seq2 seq4 seq3];
        fly(4).tour = [flip(seq1) seq4 seq2 seq3];
    end

    fly(1).cost = computeTourLength(fly(1).tour, D);
    fly(2).cost = computeTourLength(fly(2).tour, D);
    fly(3).cost = computeTourLength(fly(3).tour, D);
    fly(4).cost = computeTourLength(fly(4).tour, D);

    [L,ind]=min(vertcat(fly.cost));
    tourNew = fly(ind).tour;
end
end
```

```
function [seq,direction] = genseq(fireflies_i,Ej,vertex)
lfly = length(fireflies_i);
% Try to expand on the left
direction = 'l';
% Index of the initial vertex
indvertex = find(fireflies_i==vertex);
% Put this vertex on the right
fireflies_i = circshift(fireflies_i,lfly-indvertex);
ind = lfly;
edge = sort([fireflies_i(ind-1) fireflies_i(ind)],2);
while ismember(edge,Ej, 'rows')
    ind = ind-1;
    edge = sort([fireflies_i(ind-1) fireflies_i(ind)],2);
end
seq = fireflies_i(ind:end);
```

```
if seq == vertex % Expansion on the left is not possible
    % Try to expand on the right
    direction = 'r';
    % Index of the initial vertex
    indvertex = find(fireflies_i==vertex);
    % Put this vertex on the left
    fireflies_i = circshift(fireflies_i,-indvertex+1);
    ind = 1;
    edge = sort([fireflies_i(ind) fireflies_i(ind+1)],2);
    while ismember(edge,Ej, 'rows')
        ind = ind+1;
        edge = sort([fireflies_i(ind) fireflies_i(ind+1)],2);
    end
    seq = fireflies_i(1:ind);
end
end
```

30.3 Explanations

The proposed implementation is a simplified and modified version of the Firefly Algorithm proposed in [20]. The main building brick of the algorithm is the implementation of the new movement described previously. The parameters of the algorithm are the following: $nFireflies = 20$, the number of fireflies, $maxiter = 300$, the number of the iterations of the calculus, $alpha = 0.2$, the mutation probability.

The main program is quite simple. It begins by the initialization of the fireflies by random sequences of vertices. The firefly $gBest$ corresponding to the shortest length Hamiltonian cycle is then computed. The algorithm then consists of three nested loops. The outer loop repeats the calculation as many times as $maxIter$. All pairs of fireflies are then considered using two nested loops. Their lengths are compared and the "least bright" firefly (the one with the longer cycle length) is "attracted" to the brightest firefly (the one with the shorter cycle length). Compared to the original algorithm, there is therefore no brightness calculation nor of attraction function depending on the distance between two fireflies. The brightest firefly similarly attracts another less bright firefly (calling the `edgeMove` function). After that movement and depending of the mutation probability $alpha$, the firefly also undergoes a mutation. This is the reversal of a random subpath (calling the `mutateSegmentReverse` function). This mutation allows the solution domain to be explored while avoiding converging too quickly towards a local optimum. If this move generats a better solution than the previous one, it is stored in the corresponding firefly. The $gBest$ is then updated.

The functions `computeTourLength`, `mutateSegmentReverse` and `extractEdges` are explicit. The `genseq` function generates the segment (as defined in the method principle section) comprising the vertex $vertex$. This function needs as input a firefly $fireflies_i$, the list of edges of $fireflies_j$ and a $vertex$.

We first seek to extend the segment to the left. In order to simplify the management of indices (the vector represents a cycle and when we arrive at the first element of the vector, the next element is the last element of this vector) by circular permutation, we place the

starting vertex at the last position of the vector (completely to the right). As long as the adjacent edges belong to $fireflies_j$ we continue to extend the segment. If the resulting segment contains only the starting vertex (left extension failed), we try to extend it to the right. In the same way, we then place, by circular permutation, the starting vertex in the first position of the vector(completely to the left). We also remember the direction in which the segment was extended (`direction = 'l'` or `direction = 'r'`).

The `edgeMove` function implements the movement scheme from $fireflies_i$ to $fireflies_j$. The list of different edges between $fireflies_i$ and $fireflies_j$ is first generated. The movement consists to randomly draw a single edge $x - y$ from this list and insert it into $fireflies_i$. The two sequences beginning by vertex x and vertex y as well as their directions are computed. According to the respective orientations of the two segments, the four potential solutions are then generated.

Remember that a Hamiltonian cycle can be encoded by different vectors coming from each other by circular permutation. In order to simplify the construction of the vectors resulting from the movement, they are constructed by starting with the segment resulting from x or from y (depending on the position of the sequences in the vector $fireflies_i$). For example, the cycles shown in figure 57 will be generated according to figure 58.

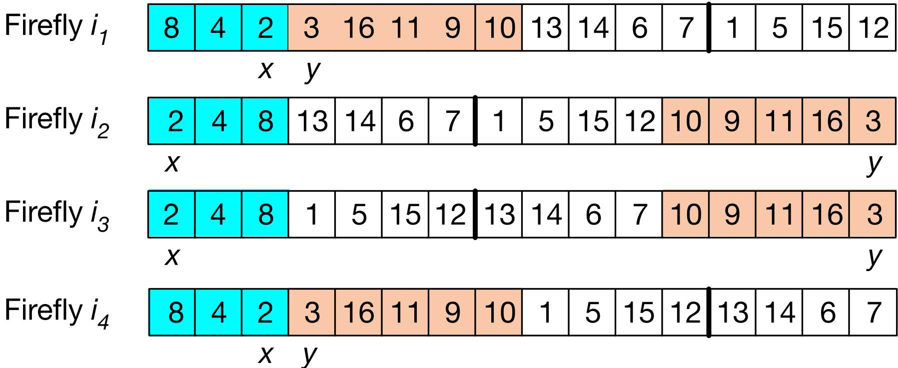


Figure 58: Merging process of two segments in firefly

After generating these four potential solutions, only the best of the four (shortest cycle) is retained to replace $fireflies_i$ (but the chosen solution can also be drawn at random). Choosing to insert only one edge during each move allows the convergence of the algorithm to be managed.

31 Tabu search

31.1 Method principle

Tabu search (TS) is a meta-heuristic search method employing local search methods used for mathematical optimization. It was created by Fred W. Glover in 1986 [13] and formalized in 1989 [14],[15].

Local (neighbourhood) searches take a potential solution to a problem and check its immediate neighbours (that is, solutions that are similar except for very few minor details) in the hope of finding an improved solution. Local search methods have a tendency to become stuck in suboptimal regions or on plateaus where many solutions are equally fit.

Tabu search enhances the performance of local search by relaxing its basic rule. First, at each step, worsening moves can be accepted if no improving move is available (like when the search is stuck at a strict local minimum). In addition, prohibitions (hence the term tabu) are introduced to discourage the search from coming back to previously visited solutions.

The implementation of tabu search uses memory structures that describe the visited solutions. If a potential solution has been previously visited within a certain short-term period, it is marked as “tabu” (forbidden) so that the algorithm does not consider that possibility repeatedly.

31.2 Matlab code

Main program

```
%% TSP solving using tabu search
%  Included in TSP20024 app
%  Didier Maquin (2024). TSP2024
%  (https://www.mathworks.com/matlabcentral/fileexchange/158496-tsp2024)
%  MATLAB Central File Exchange.

clear
close all
X = rand(100,2);
n = length(X);

plot(X(:,1),X(:,2),'kx','MarkerSize',8,'LineWidth',2)
hold on

move_list = create_list(n);     % List of all possible move
n_move = length(move_list);     % Number of different move

%% Tabu search parameters

maxiter = 100+n;                  % Maximum number of iterations
tabu_length = round(0.5*n_move); % Tabu Length

% Initial solution
current_tour = randperm(n);
current_length = tour_length(X,current_tour);

% Best tour
best_tour = current_tour;
best_length = current_length;

% Tabu counter of moves
tabu_count = zeros(n_move,1);
```

```
%% Main Loop
for iter=1:maxiter

    best_new_tour_length=inf;

    % Apply all moves in an random order
    move_order = randperm(n_move);
    for i=1:n_move
        if tabu_count(move_order(i))==0
            new_tour=current_tour;
            ind = move_list(move_order(i),:);
            i1 = ind(1);
            i2 = ind(2);
            new_tour(i1:i2)=current_tour(i2:-1:i1);

            if tour_length(X,new_tour)<=best_new_tour_length
                best_new_tour = new_tour;                       % tour
                best_new_tour_length = tour_length(X,new_tour); % length
                best_new_ai = move_order(i);                    % move index
            end
        end
    end

    % Update current tour
    current_tour = best_new_tour;
    current_length = best_new_tour_length;

    % Update tabu list
    for i=1:n_move
        if i==best_new_ai
            tabu_count(i)=tabu_length;              % Add to tabu list
        else
            tabu_count(i)=max(tabu_count(i)-1,0);   % Reduce tabu counter
        end
    end

    % Update best tour
    if current_length<=best_length
        best_tour = current_tour;
        best_length = current_length;
    end

    % Polygon plot
    cla
    tourdisp = [best_tour best_tour(1)];
    plot(X(tourdisp,1), X(tourdisp,2), 'r','LineWidth',2);
    plot(X(:,1), X(:,2),'kx','MarkerSize',8,'LineWidth',2);
    pause(0)
    title(['Iteration number: ' num2str(iter)])
end
% Add title
str = sprintf('Tour Length: %g',best_length);
title(str)
```

Functions

```
function move_list=create_list(n)
    % Create the list of all possible move (swap+reverse operations)
    move_list = zeros(n*(n-1)/2,2);
    c = 0;
    % Reverse moves (including swap moves)
    for i=1:n-1
        for j=i+1:n
            c=c+1;
            move_list(c,:) = [i j];
        end
    end
end
```

```
function L = tour_length(X,tour)
    tour = [tour tour(1)];
    L = sum(sqrt(diff(X(tour,1)).^2+diff(X(tour,2)).^2));
end
```

31.3 Explanations

For the symmetric Euclidean TSP problem, the reverse operator (which corresponds to the operation involved in the 2-opt algorithm) is well adapted to define the "neighbourhood" of a given solution solution. One will notice that the swap operation is only a special case of the reverse operation (when the two chosen indices are consecutive). In order to facilitate the method implementation, only the reverse operator will therefore be considered. The reader will also note that the reverse operator is involutive which is perfectly consistent with the fact of temporarily banning it so as not to explore a solution already explored recently.

To be able to manage a list of tabu moves, an exhaustive list of all possible reversion operations is first created. It is simply a matrix with $n \times (n-1)/2$ raws and two columns storing the indices allowing the chosen sequence to be reversed.

At each iteration, all possible movements are applied, in random order, to the current solution in order to seek a local optimal solution.

The moves allowing an improvement of the current solution are then included in the list of tabu move. These moves will no longer be used in future searches (at least for a certain number of iterations). Managing counters for all moves allows the management of the number of bans, here set at half of the total number of possible movements.

The maximum number of iterations of the main loop has been fixed to $100 + n$ where n is the chosen number of vertices.

References

[1] Berkeley lecture note.
(https://aswani.ieor.berkeley.edu/teaching/FA15/151/lecture_notes/ieor151_lec18.pdf), retrieved on March 2024.

[2] Yanping Bai, Wendong Zhang and Zhen Jin. An new self-organizing maps strategy for solving the traveling salesman problem. Chaos, Solitons and Fractals, 28:1082-1089, 2006.

[3] John Bartholdi III, Loren Platzman. An O(N log N) planar travelling salesman heuristic based on spacefilling curves. Operations Research Letters, 1(4):121:125,1982.

[4] Sumanta Basu. Tabu search implementation on Traveling Salesman Problem and its variations: A literature survey. American Journal of Operations Research, 2(2):19930, 2012.

[5] Guy Blelloch. Parallel and Sequential Data Structures and Algorithms–Lecture 4: Divide and Conquer Continued, 2012.
(http://www.cs.cmu.edu/afs/cs/academic/class/15210-f12/www/lectures/lecture04.pdf), retrieved on March 2024.

[6] Charles Bouillaget. Le problème du voyageur de commerce, juin 1984 (in french).
(https://www-almasty.lip6.fr/ bouillaguet/static/stuff/tipe.pdf), retrieved on May 2023.

[7] Lukasz Brocki and Danijel Korzinek. Kohonen Self-Organizing map for the Traveling Salesperson Problem. 7th International Symposium Mechatronics, Warsaw, Poland, September 19-21, 2007. In: Recent Advances in Mechatronics. Springer, Berlin, Heidelberg, pp. 116-119, 2007.

[8] G. Clarke and J. W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. Operations Research, 12(4):568-581, July-August 1964.

[9] Michael B. Dillencourt. Traveling salesman cycles are not always subgraphs of Delaunay triangulations or of minimum weights triangulations. Information Processing Letters, 24:339-342, 1987.

[10] Sheqin Dong, Fan Guo, Jun Yuan, Rensheng Wang, Xianlong Hong. A novel tour construction heuristic for Traveling Salesman Problem using LFF principle. 9th Joint International Conference on Information Sciences, JCIS 2006, Kaohsiung, Taiwan, ROC, October 8-11, 2006.

[11] Marco Dorigo, Vittorio Maniezzo and Alberto Colorni. Ant System: Optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics-Part B, 26(1):1-13, 1996.

[12] Arno Formella. Quasi-linear time heuristic to solve the Euclidean traveling salesman problem with low gap. Journal of Computational Science 82:102424, 2024.

[13] Fred Glover. Future paths for integer programming and links to artificial intelligence. Computers and Operations Research, 13(5): 533-549, 1986.

[14] Fred Glover. Tabu search – Part 1. ORSA Journal on Computing, 1(2): 190-206, 1989.

[15] Fred Glover. Tabu search – Part 2. ORSA Journal on Computing, 2(1): 4-32,1990.

[16] Abdolreza Hatamlou. Black hole: A new heuristic optimization approach for data clustering. Information Sciences 222:175-184, 2013.

[17] Ali Jazayeri, Hiroki Sayama. A polynomial-time deterministic approach to the traveling salesperson problem. International Journal of Parallel, Emergent and Distributed Systems, 35(4):454:460, 2020.

[18] Richard M. Karp. Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the Plane. Mathematics of Operations Research, 2(3):209-224, 1977.

[19] Natalio Krasnogor, Pablo Moscato, Michael G. Norman. A new hybrid heuristic for large geometric traveling salesman problems based on the delaunay triangulation. Anales del XXVII Simposio Brasileiro de Pesquisa Operacional, November 1995.

[20] Gilang Kusuma Jati, Ruli Manurung and Suyanto. Discrete firefly algorithm for traveling salesman problem: A new movement scheme. *In* Swarm Intelligence and Bio-Inspired Computation, chapter 13, pp. 295-312, Elsevier 2013.

[21] Jian Li. An improved dynamic programming algorithm for bitonic TSP. 2nd International Symposium on Computer, Communication, Control and Automation, ISCCCA 2013, 22-24 February 2013, Shijiazhuang, China. *In* Advances in Intelligent Systems Research, 36:24-27, Atlantis Press, 2014.

[22] Jonas Lundgren (2019). TSPSEARCH 1.0.0, MATLAB Central File Exchange. (https://www.mathworks.com/matlabcentral/fileexchange/71226-tspsearch), retrieved February 8, 2022.

[23] Stewart, William. A computational comparison of five heuristic algorithms for the Euclidean traveling salesman problem. In: Mulvey, J.M. (eds) Evaluating Mathematical Programming Techniques. Lecture Notes in Economics and Mathematical Systems, vol 199. Springer, Berlin, Heidelberg, 1982.

[24] Duc Truong Pham, Afshin Ghanbarzadeh, Ebubekir Koc, Sameh Otri, Sahra Rahim and Mb Zaidi. The Bees Algorithm. Technical Note, Manufacturing Engineering Centre, Cardiff University, UK, 2005.

[25] Fernando Guilherme Silvano Lobo Pimentel. Double-ended nearest and loneliest neighbour – a nearest neighbour heuristic variation for the travelling salesman problem. Revista de Ciências da Computação, 6:17-30, 2011.

[26] Loran Platzman, John Bartholdi III. Spacefilling curves and the planar Travelling Salesman Problem. Journal of the ACM, 36(4):719-737, 1989.

[27] Frano Rajic and Ivan Stresec. Advanced algorithms: Project TSP. Retrieved from https://github.com/istresec/kth-aa.

[28] Murat Sahin. Improvement of the bees algorithm for solving the Traveling Salesman Problems. Bilişim Teknolojileri Dergisi,15(1):65-74, 2022.

[29] Aravind Seshadri (2006). Traveling Salesman Problem (TSP) using Simulated Annealing 1.0.0.0, MATLAB Central File Exchange. (https://fr.mathworks.com/matlabcentral/fileexchange/9612-traveling-salesman-problem-tsp-using-simulated-annealing), retrieved on January 27, 2023.

[30] Kardi Teknomo. Q-learning by examples. https://people.revoledu.com/kardi/tutorial/ReinforcementLearning/ (retrieved on April 24, 2024).

[31] Christine Valenzuela and Antonia Jones. Evolutionary divide and conquer (I): A novel genetic approach to the TSP. Evolutionary Computation,1(4): 313–333, 1993.

[32] Diego Vicente (2018). som-tsp: Python codes uploaded on https://github.com/diego-vicente/som-tsp/. Software under MIT License.

[33] Jiaying Wang, Chenglong Xiao, Shanshan Wang, Yaqi Ruan. Reinforcement learning for the traveling salesman problem: Performance comparison of three algorithms. The Journal of Engineering, 2023:e12303, 2023.

[34] https://www.math.uwaterloo.ca/tsp/index.html.

[35] Sultan Zeybek, Asrul Harun Ismail, Natalia Hartono, Mario Caterino, Kaiwen Jiang. An improved vantage point bees algorithm to solve combinatorial optimization problems from TSPLIB. Macromolecular Symposia, 396(1):2000299, 2021.

Matlab Central License

Copyright (c) 2012, Jonas Lundgren
Copyright (c) 2007, Aravind Seshadri
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution
- Neither the name of none nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

MIT License

Copyright (c) 2018 Diego Vicente
Copyright (c) 2021 Rahul

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

GNU AFFERO GENERAL PUBLIC LICENSE
Version 3, 19 November 2007.

Copyright (C) 2007 Free Software Foundation, Inc. https://fsf.org/
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Fixfocus MATLAB function to fix an annoying, long-standing focus bug with selection dialog boxes and uifigure windows
Copyright (C) <2024> <Jorg Woehl>

https://fr.mathworks.com/matlabcentral/fileexchange/165961-fixfocus
https://github.com/JorgWoehl/fixfocus

This program is free software: you can redistribute it and/or modify it under the terms of the GNU Affero General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Affero General Public License for more details.

The GNU Affero General Public License is available at the web site

https://www.gnu.org/licenses.

Appendix 1 : Loading TSPLIB file

Description

The TSP2024 Matlab App is able to load TSPLIB file. TSPLIB is a library of sample instances for the TSP (and related problems) from various sources and of various types. It is maintained by Gerhard Reinelt at the Heidelberg University in Germany (http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html).

The TSPLIB file format is a text-based file format. Details about the file format can be found at http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf. Each file consists of a specification part and of a data part. The specification part contains information on the file format and on its contents. The data part contains explicit data. All entries in the specification part are of the form <keyword> : <value>, where <keyword> denotes an alphanumerical keyword and <value> denotes alphanumerical or numerical data. In the data part, each data section begins with the corresponding keyword. The length of the section is either implicitly known from the format specification, or the section is terminated by an appropriate end-of-section identifier.

An incomplete TSPLIB file may be read by TSP2024 App. The only mandatory keywords are:

- DIMENSION : the number of nodes (vertices)
- NODE_COORD_SECTION : node coordinates are given in this section. Each line is of the form <integer> <real> <real> where the first integer is the node number and the two following reals are the 2D coordinates. This section can optionally end with the keyword EOF.

Among the other keywords that may appear:

- If the keyword NAME is present, the name of the file is displayed as the title of the graphic window.
- If the keyword TYPE is present, it must be positioned to TSP otherwise an alert dialog is displayed indicating an invalid file.
- If the keyword EDGE_WEIGHT_TYPE is present, it must be positioned to EUC_2D (euclidean distances in 2-D) otherwise an alert dialog is displayed indicating an invalid file.
- if the keyword NODE_COORD_TYPE is present, it must be positioned to TWOD_COORDS (nodes are specified by coordinates in 2-D) otherwise an alert dialog is displayed indicating an invalid file.

All the other keywords are ignored.

If DIMENSION is greater than 150 an alert dialog is displayed to warn the user that not all methods will provide a satisfactory result.

Matlab code

```
function [n,X,Name,errflag]=read_TSPLIB_file(app,infile)
errflag = 0;
n = 0;
X = [];
Name = '';
fid = fopen(infile,'r');
if fid<0
    uialert(app.UIFigure,"Error while opening the file!","Invalid File");
    errflag = 1;
    return;
end
while feof(fid)==0
    temps = fgetl(fid);
    if strcmp(temps,'')
        continue;
    elseif strncmpi('NAME',temps,4)
        k = findstr(temps,':');
        Name = temps(k+1:length(temps));
        Name = Name(find(~isspace(Name)));
    elseif strncmpi('TYPE',temps,4)
        k = findstr(temps,':');
        TYP = temps(k+1:length(temps));
        TYP = TYP(find(~isspace(TYP)));
        if ~strncmpi('TSP',TYP,3)
            uialert(app.UIFigure,"TYPE must be positioned to TSP!","Invalid ...
                File");
            errflag = 1;
            break
        end
    elseif strncmpi('DIMENSION',temps,9)
        k = findstr(temps,':');
        d = temps(k+1:length(temps));
        n = str2double(d);
    elseif strncmpi('EDGE_WEIGHT_TYPE',temps,16)
        k = findstr(temps,':');
        EWS = temps(k+1:length(temps));
        EWS = EWS(find(~isspace(EWS)));
        if ~strncmpi('EUC_2D',EWS,6)
                    uialert(app.UIFigure,"EDGE_WEIGHT_TYPE must be ...
                        positioned to EUC_2D!","Invalid File");
            errflag = 1;
            break
        end
    elseif strncmpi('NODE_COORD_TYPE',temps,15)
        k = findstr(temps,':');
        NCT = temps(k+1:length(temps));
        NCT = NCT(find(~isspace(NCT)));
        if ~strncmpi('TWOD_COORDS',NCT,11)
            uialert(app.UIFigure,"NODE_COORD_TYPE must be positioned to ...
                TWOD_COORDS!","Invalid File");
            errflag = 1;
            break
        end
    elseif strncmpi('NODE_COORD_SECTION',temps,18)
        NodeCoord = fscanf(fid,'%g %g %g',[3 n]);
    end
end
```

```
fclose(fid);
if n >= 150
    uialert(app.UIFigure,"This app is designed for files containing up to a ...
        hundred points. Some methods will fail to provide a result for ...
        larger files.","Be careful");
end
if ~exist('NodeCoord')
    uialert(app.UIFigure,"NODE_COORD_SECTION is empty!","Invalid File");
    errflag = 1;
end
if errflag == 0
    X = NodeCoord(2:3,:)';
end
end
```

Appendix 2 : Algorithm, efficiency, algorithmic decidability

Notion of algorithm

The notion of algorithm, although old, was only defined and studied in the 20th century, when mathematicians had to think about the logical foundations of mathematics. Let us try in a few words to specify what this notion covers.

An algorithm can be defined as a process, a series of instructions allowing the resolution of a given problem in all its generality and in particular whatever the data of the problem..

More precisely, an algorithm must verify the following conditions:

1. it must be able to be written in a certain language (i.e. a set of words written in a defined alphabet);
2. data are submitted at the beginning (these are inputs);
3. the algorithm itself is a step-by-step process;
4. the action at each step is strictly determined by the algorithm, the inputs and the results obtained in the previous steps (note the strictly deterministic nature of this action);
5. the algorithm returns a clearly specified response (preferably the one we expect...), denoted output;
6. whatever the inputs, execution must complete in a finite number of steps.

This is a key concept in computer science: the computer can only deal with problems for which an algorithm exists. This definition raises two questions which we will try to answer in the following:

- is there an algorithm for any given problem?
- must an algorithm finish in a reasonable time interval and what is a reasonable time interval anyway?

Efficiency of algorithms

Undecidable problems – A question has intrigued mathematicians: is there an algorithm for any given problem?

The answer to this question is no: in the 1930s, several mathematicians (including Kurt Gödel, Alonzo Church, Alan Turing and Emil Post) showed the existence of problems which cannot be solved by algorithms (i.e. for which there is no solution method)[10].

[10] This is the case of the Hilbert's tenth problem. A polynomial $P(x, y, z, ...)$ is said to be Diophantine if and only if all its coefficients are in $\mathbb{Z}$. A Diophantine equation is an equation of the form $P(x, y, z, ...) = 0$ where P is a Diophantine polynomial. The formulation of the Hilbert's tenth problem is the following: is there a general algorithm for determining whether any Diophantine equation $P(x, y, z, ...) = 0$ admits an integer solution? The problem was solved by Yuri Matiasevich in 1970 and the answer is negative.

These problems were qualified as undecidable and, in a way, they mark the horizon of what a mathematician can treat.

Declaring a problem undecidable is both a great victory for the human mind (such a demonstration is often a mathematical feat) but also an acknowledgment of failure (it is the death knell of any research on this problem). This last point should be put into perspective because, if the problem in its generality is definitively closed, it can however be solved in certain particular cases.

Efficiency of algorithms – must an algorithm finish in a reasonable time interval?

From a theoretical point of view, having a finite number of steps is the only constraint imposed on an algorithm; but from a practical point of view, the answer to a given problem must be obtainable on a human scale: as such, the existence of an algorithm is not enough for the problem to be concretely solved. The efficiency of algorithms was initially studied in the mid-1960s by A. Cobham and J. Edwards. To evacuate the time factor, which is too subjective, the number of steps necessary to bring the algorithm to completion is involved. An algorithm is said to have polynomial complexity if there exist two fixed integers A and k such that from data of length n, the algorithm requires at most An^k steps.

Thus, the multiplication of two integers is of polynomial complexity. When we multiply two integers of n digits each, we perform n^2 multiplications of two one-digit integers.

Algorithms that are not polynomial complex are said to be exponentially complex. Thus an algorithm which requires 2^n, n^n or $n!$ steps for data of length n is of exponential complexity: the function which gives the number of steps is not necessarily a function exponential function in the usual sense.

The exhaustive algorithm of the traveling salesman problem is an algorithm with exponential complexity: this complexity results from the combinatorial explosion that we observe when we increase the number of cities. By definition, efficient algorithms are algorithms of polynomial complexity; inefficient algorithms are algorithms of exponential complexity.

Classification of different types of problems

The preceding approach makes it possible to initiate a classification of the types of problems that can meet a mathematician. First of all, there are undecidable problems (also called problems without proof), that is to say for which there is no algorithm. The other problems are therefore those for which an algorithm exists: a problem is said to be of **class P** if there exists for this problem a resolution algorithm of polynomial complexity (this is the case of the problem of calculating the GCD of two integers, solved by Euclid's algorithm).

If all the resolution algorithms are of the exponential type, the problem is said to be hard-proof (you will understand that the difficulty of this definition lies in the "all"!). Is the traveling salesman problem a hard-proof problem? The answer is far from simple. The only algorithm that we know, the exhaustive algorithm, is of the exponential type and

we do not know of any polynomial algorithm to date. But no one has demonstrated that it does not exist. This is indeed the difficulty of problems with difficult proofs! Strictly speaking, the traveling salesman problem cannot be considered as a hard-proof problem: however, it has every chance of being one.

There is another category of problems, encompassing the class P: it is the **class NP** (*non deterministic polynomial problem*). The traveling salesman problem is the prototype of this class. How to describe the NP class? In addition to the problems of class P, it includes problems that are all soluble (and therefore not undecidable) but of uncertain type: an inefficient resolution algorithm exists and it is not known whether it is possible to find an efficient algorithm for these problems. Another characteristic of these problems lies in the fact that if we have a solution, we can check whether it satisfies the problem by an efficient algorithm (of polynomial complexity).

The interest of the NP class therefore lies in two points. The first is that many problems for which we have not yet found an efficient algorithm turn out to be of the NP type. Let us cite an example, the factorization of an integer. It is a problem for which the only known algorithm is inefficient: it is the algorithm of exhaustive search of all the possible prime factors. For small integers, the answer is given quickly ($21 = 3 \times 7$); for larger integers, the calculation times grow exponentially (try factoring the whole number 7868595578623743171509654930779 to see). Perhaps there is an efficient algorithm; no one has found it yet. This problem is also of class NP. The solution is: $9876543219876543251 \times 796695301529$. If you are not convinced, do the multiplication: it will only take you a polynomial time!

The second point is that Stephen Cook showed in 1971 that it was highly improbable that we find for certain NP problems (**class NP-complete**) an efficient algorithm: in the sense that if we find an efficient algorithm for such a problem, we will deduce an efficient algorithm for any NP problem. However, believing that we will solve the case of all NP problems seems excessively optimistic, so we think that we will not be able to solve the case of an NP-complete problem. That said, the problem remains open expressed in the literature by the formula $P \stackrel{?}{=} NP$.

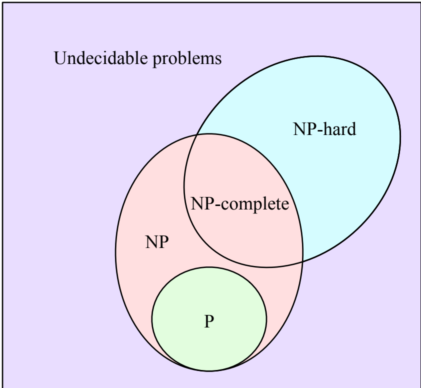


Figure 59: Classification of problems (under the hypothesis $P \neq NP$)

Figure 59 summarizes the problem classification:

- P is the set of algorithmic decision problems whose complexity is polynomial;
- NP is the set of algorithmic decision problems such that, if a solution is given, it is possible to verify this solution in polynomial time. So we have $P \subset NP$;
- the class of NP-complete problems gathers NP problems that are as difficult as any NP problem. These problems are said to be weakly equivalent[11] between them. This category is of particular importance because if someone succeeds in proving that only one problem of the class NP-complete is indeed in P, then all NP is included in P!
- the class of hard-proof problems (also called **class NP-hard**, which is ambiguous), gathers problems at least as hard as any NP problem. NP-complete and NP-hard problems are often confused. The difference is that an NP-hard problem need not be in NP (e.g. because it is such a hard problem that even testing it on a single solution cannot be accomplished in polynomial time).

[11] We say that a problem Q is *weakly reducible* to a problem P ($Q \propto P$) if there exists a transformation f which, for any instance q of problem Q, constructed in polynomial time in the size of q, an instance $p = f(q)$ of problem P of polynomial size in the size of q and such that the instance of Q has a solution if and only if the instance of P has a solution. The two problems P and Q are said to be *weakly equivalent* if P is reducible to Q and Q reducible to P.