Master Thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Application of Neural Networks for Routing Problems

Bc. Alan Drozen

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Field of study: Cybernetics and Robotics
Subfield: Robotics
January 2018

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student: Bc. Alan D r o z e n

Study programme: Cybernetics and Robotics

Specialisation: Robotics

Title of Diploma Thesis: Application of Neural Networks for Routing Problems

Guidelines:

1. Get acquainted with neural networks with a special attention to self organizing maps.
2. Implement selected neural networks and employ them for a chosen routing problem (e.g., Traveling Salesman Problem) in a polygonal domain.
3. Verify experimentally the proposed solution and describe and discuss obtained results.
4. Discuss possible extensions of the proposed solution to other routing problems in a polygonal domain or their subproblems.

Bibliography/Sources:

[1] E. M. Cochrane and J. E. Beasley: The co-adaptive neural network approach to the Euclidean travelling salesman problem. Neural Netw. 16, 10 (December 2003), 1499-1525.
[2] S. Somhom, A. Modares, T. Enkawa: A self-organising model for the travelling salesman problem. Journal of the Operational Research Society, 1997, 48 (9): 919-928.
[3] J. Šíma, R. Neruda: Teoretické otázky neuronových sítí. Vyd. 1. Praha: Matfyzpress, 1996.
[4] S. Ingram, T. Munzner, M. Olano: Glimmer: Multilevel MDS on the GPU, in Visualization and Computer Graphics, IEEE Transactions on , vol.15, no.2, pp.249-261, March-April 2009.
[5] A. Elad, R. Kimmel: On bending invariant signatures for surfaces, Pattern Analysis and Machine Intelligence, IEEE Transactions on , vol.25, no.10, pp.1285-1295, Oct. 2003.

Diploma Thesis Supervisor: RNDr. Miroslav Kulich, Ph.D.

Valid until: the end of the winter semester of academic year 2017/2018

L.S.

prof. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, April 15, 2016

Acknowledgements

I would like to express my thanks to my supervisor RNDr. Miroslav Kulich, Ph.D. for consultations and guidance. I wish to thank my family for their patience and support during the preparation of this thesis.

Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.

Declaration

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne ....

Podpis autora práce ....

Abstract

The thesis deals with the Traveling Salesman Problem in a polygonal domain using Self Organizing Maps. The task is transformed to the Traveling Salesman Problem in the Euclidean space of a higher dimension by the technique of the multidimensional scaling. Then it is solved using Self Organizing Maps procedures. Another method is based on the new non-Euclidean form of Self Organizing Maps, which was derived theoretically and implemented subsequently. Both methods were numerically compared concerning the speed of computation and the quality of solutions with various settings of parameters.

Keywords: Self-organising maps, multidimensional scaling, traveling salesman problem, polygonal domain, co-adaptive net, non-Euclidean SOM, Glimmer, TSP, MDS, CAN, SOM

Supervisor: RNDr. Miroslav Kulich, Ph.D.

Abstrakt

Tato práce se zabývá řešením problému obchodního cestujícího v polygonální doméně s využitím samoorganizujících se map. Pomocí muldimenzionálního škálování je úloha převedena na problém obchodního cestujícího v Euklidovském prostoru vyšší dimenze. Poté jsou k řešení využity standardní postupy samoorganizujících se map. Další metoda je založena na nové neeuklidovské formě samoorganizujících se map, jež byla nejprve odvozena teoreticky a následně implementována. Oba postupy byly numericky porovnány z hlediska rychlosti výpočtu a kvality řešení při různých nastaveních parametrů.

Klíčová slova: Samoorganizující se mapy, multidimenzionální škálování, problém obchodního cestujícího, polygonální doména, ko-adaptivní síť, neeuklidovský SOM, Glimmer, TSP, MDS, CAN, SOM

Překlad názvu: Aplikace neuronových sítí ve směrovacích problémech

Contents

Figures

Tables

Algorithms

Chapter 1

Introduction

The Traveling Salesman Problem (TSP) is the problem from the graph theory to find the shortest path through given guards (cities). Every guard has to be visited just once. In the general case, the guards are represented by vertices of some graph and distances between the guards are equivalent to the length of edges. When the guards are points in the Euclidean space, and their distances are defined as the Euclidean distances between these points, the problem is called the Euclidean Traveling Salesman Problem or the Traveling Salesman Problem in the Euclidean domain.

The TSP is the NP-hard problem, therefore it cannot be exactly solved in polynomial time. Many methods exist to solve the problem – some of them are exact, and the others try to quickly find just an approximate solution. The combinatorial heuristics, which are very popular now, are among them. Last but not least, some techniques are based on the usage of the Hopfield neural networks or the Self organizing maps.

The ordinary techniques based on the Self organizing maps work in the Euclidean space only. The objective of this thesis is to develop, implement, test and compare methods to solve the TSP in a space with polygonal boundaries and obstacles. The existing methods to solve the TSP in the Euclidean or the polygonal domain are described in chapter 2. The Glimmer MDS algorithm and the non-Euclidean SOM method are described in chapter 3. The numerical experiments are covered by chapter 4. Possible extensions to other routing problems are shortly discussed in chapter 5. Chapter 6 is the conclusion.

Chapter 2

State of the art

In sections 2.1 and 2.2 two classical methods to solve the Traveling Salesman Problem in the Euclidean domain using Self organizing maps (SOM) will be recalled. In the last section (2.3), the polygonal domain will be defined and two approaches to solve the TSP in the polygonal domain will be shortly discussed.

2.1 Basic SOM algorithm in Euclidean domain

A technique to use SOM to solve the TSP problem in the Euclidean domain was introduced in [10]. This method will be referred as Basic SOM in the following text. It uses the iterative process of moving neurons in the space of the guards from initial position to the final position when every neuron is near to some guard. The neurons are connected by a string in such a way, that string forms a loop. At the end of the iterative process, the order of neurons on the string determines the order of the guards in the path.

Consider the TSP task in the Euclidean domain. Denote the number of guards n, and denote these guards $\mathsf{G}_1, \dots, \mathsf{G}_n$. To use the Basic SOM method (for the pseudocode see Alg. 1), neurons have to be created first (lines 1–2). The number of the neurons used is set: $m = 3n$ (value $3n$ from [11]). Denote these neurons $\mathsf{N}_1, \dots, \mathsf{N}_m$. Their initial positions are equidistant points on a small circle around the centroid of the guards ($\mathsf{C} = \sum_k \mathsf{G}_k / n$). Other

Algorithm 1: Basic SOM

Input: Guards $\mathsf{G}_1, \dots, \mathsf{G}_n$ in Euclidean space
Output: Solution of TSP

1 $m \leftarrow 3n$ `// number of neurons = 3*number of guards`
2 initialize neuron positions
3 **while** *true* **do**
4     $error \leftarrow 0$
5     $inhibited \leftarrow \emptyset$ `// empty set of inhibited neurons`
6     $permutation \leftarrow$ random permutation of sequence $(1, \dots, n)$
7   **for** $k \leftarrow 1$ **to** n **do**
8         $l \leftarrow permutation[k]$
9   for G_l find winning not inhibited neuron N_i
10         $error \leftarrow \max(error, distance(\mathsf{G}_l, \mathsf{N}_i))$
11         $inhibited \leftarrow inhibited \cup \{i\}$
12   move neuron N_i and its neighbours towards G_l
13   **end**
14   **if** $error \leq max\_error$ **then break**
15   update parameters: $G \leftarrow G(1 - \alpha)$
16 **end**
17 construct *path*
18 **return** *path*

Parameter	Value
Initial value of gain G	10
Learning rate μ	0.6
Neighbourhood size d^*	$0.2m$
Gain change parameter α	0.03
Termination threshold max_error	0.1

Table 2.1: Basic SOM – parameters and proposed values [10], [11]

parameters of the algorithm are set – see Table 2.1, proposed values are from [10] and [11].

The main part of the algorithm is the loop (lines 3–16). At the beginning of every iteration, the set of inhibited neurons is emptied (line 5) and the guards are randomly permuted (line 6). The winning not inhibited neuron, i.e. the neuron with the shortest distance to the guard G_l among all not inhibited neurons, is found for the selected guard G_l (line 9). Then, the winning neuron N_i is added to the set of inhibited neurons (line 11), and the neuron N_i and neighbouring neurons are moved towards the guard G_l using the following equation:

$$\mathsf{N}_j^{\text{new}} = \mathsf{N}_j + \mu \exp\left(-\frac{\mathsf{d}^2_{card}(\mathsf{N}_i, \mathsf{N}_j)}{G^2}\right)(\mathsf{G}_l - \mathsf{N}_j), \tag{2.1}$$

where N_j is the position of the neuron N_j before the movement, and $\mathsf{N}_j^{\text{new}}$ is the position of the neuron after the movement.[1] Cardinal distance between the neurons N_i and N_j is denoted as $\mathsf{d}_{card}(\mathsf{N}_i, \mathsf{N}_j)$, and it is the minimal number of hops on the neural string to get from N_i to N_j:

$$\mathsf{d}_{card}(\mathsf{N}_i, \mathsf{N}_j) = \min(|i - j|, m - |i - j|). \tag{2.2}$$

The ratio of the movement, i.e. $\mu \exp(\mathsf{d}^2_{card}(\mathsf{N}_i, \mathsf{N}_j)/G^2)$, is the highest for the winning neuron N_i (the cardinal distance is zero), and the other neurons from the neighbourhood have this ratio smaller and smaller as they lie further on the string. The usual size of the neighbourhood d^* is $0.2m$ (using cardinal distance) thus the neuron N_j is moved only if $\mathsf{d}_{card}(\mathsf{N}_i, \mathsf{N}_j) < 0.2m$. The previous steps are repeated for every guard in the inner loop (lines 7–13).

Before the next iteration of the main loop, the parameter G is updated: $G = G(1 - \alpha)$ (line 15). Moreover, an error (the maximum of distances between the guards and their winning neurons) is calculated in every iteration (lines 4 and 10). If this error is less than the threshold *max_error*, the main loop is terminated (line 14). At this moment, the neurons are close to the guards, and the only step left to be done is to construct a path as a sequence of indices of the guards. The following procedure is used: find the winning neuron for the guard and save the index of the guard into the winning neuron. Repeat this for every guard. Then, the order of the neurons on the string determines the order of the guards in the route thus it can be constructed. If there are more guards than one with the same winning neuron (so their order is not exactly known), random order or some heuristics can be used.

2.2 Co-adaptive net algorithm (CAN) in Euclidean domain

Another approach that uses SOM to solve the TSP problem in the Euclidean domain is the Co-adaptive net algorithm (CAN). It was introduced in [2]. The CAN technique is similar to Basic SOM (see section 2.1).

For the pseudocode see Alg. 2. At the beginning, the input data (positions of the guards) are scaled to lie in the unit sized square ($[0, 1] \times [0, 1]$) (line 1). Scaling factor has to be the same in both dimensions not to distort relative distances between guards. Then parameters of the algorithm are set – see Table 2.2, proposed values are from [2]. The number of the neurons used is

[1] This notation is used on multiple places in this thesis to distinguish the original value of some variable and the new value of the same variable.

Algorithm 2: Co-adaptive net algorithm (CAN)

```
Input: Guards G_1, ..., G_n in Euclidean space
Output: Solution of TSP
scale guard positions to lie in [0,1] × [0,1] square
m ← 2.5n                 // number of neurons = 2.5*number of guards
initialize neuron positions
path_best ← (∞)                       // path with infinite length
while true do
    competition_phase ← (G ≥ G#)
    error ← 0
    ∀i : w_i ← 0                      // reset neuron-won counters
    neur_moved ← false
    permutation ← random permutation of sequence (1, ..., n)
    for k ← 1 to n do
        l ← permutation[k]
        for G_l find winning neuron N_i
        error ← error + distance(G_l, N_i)
        w_i ← w_i + 1                 // increment neuron-won counter
        if w_i = 1 then
            move neuron N_i and its neighbours towards G_l
        end
        if (w_i = 2) ∧ competition_phase then
            move neighbours of neuron N_i towards G_l
        end
    end
    w1count ← |{i : w_i = 1}|   // number of w_i that are equal to 1
    if w1count ≥ min(0.98n, n − 100) then
        construct path to path_temp
        if ||path_temp|| < ||path_best|| then path_best ← path_temp
    end
    if (error ≤ max_error) ∨ (G ≤ 0.01) ∨ (not neur_moved) then
      break
    if G > G#/2 then
        G ← G(1 − 2α)                 // update parameters
    else
        G ← G(1 − α)                  // update parameters
    end
end
if path was not constructed in the last iteration then
    construct path to path_temp
    if ||path_temp|| < ||path_best|| then path_best ← path_temp
end
return path_best
```

$m = 2.5n$ (line 2). The initial position of the neurons is the same as in the Basic SOM (points on the small circle around the centroid) (line 3).

Parameter	Value
Initial value of gain G	$n/3$
Learning rate $\mu = 1/R$, where R is learning rate from [2]	0.625
Maximal cardinal distance to search for a winning neuron C^*	250
Parameter β determining how often the full search will be applied	10
Maximal neighbourhood size D^*	200
Gain change parameter α	0.02
Competition to cooperation phase threshold $G^\#$	10
Termination threshold max_error	10^{-10}

Table 2.2: CAN – parameters and proposed values [2]

The main part of the algorithm is the loop (lines 5–34). At the beginning of every iteration, the guards are randomly permuted (line 10) and the counter w_i showing how many times the neuron N_i has won is set to zero (line 8) for every neuron. Then, the algorithm continues by the inner loop (lines 11–22): for every guard (G_l) find the winning neuron N_i (line 13), i.e. the neuron with the shortest distance to the guard G_l among all neurons with the cardinal distance smaller than C^* from the previous winner of the guard G_l. Every β-th iteration the full search among all neurons is used. Increment the counter w_i of the winner (Alg. 2, line 15). If the neuron N_i has won for the first time ($w_i = 1$) move it and neighbouring neurons towards the guard G_l using the equations (2.3) and (2.4). If the neuron N_i has won for the second time ($w_i = 2$) and if the algorithm is in the competition phase (i.e. $G \geq G^\#$) move the neighbours of the neuron N_i towards the guard G_l using the same equations, the neuron N_i is not moved in this case (Alg. 2, lines 16–21). The equations defining the neuron movement are:

$$\mathsf{N}_j^{\text{new}} = \mathsf{N}_j + \mu \exp\left(-\frac{\mathsf{d}_{card}(\mathsf{N}_i, \mathsf{N}_j)^2}{g_j^2}\right)(\mathsf{G}_l - \mathsf{N}_j), \tag{2.3}$$

where

$$g_j = G\Big(1 - \mathsf{d}(\mathsf{N}_j, \mathsf{G}_l)/\sqrt{2}\Big), \tag{2.4}$$

and $\mathsf{d}(\mathsf{N}_j, \mathsf{G}_l)$ is the cardinal distance between N_j and G_l. The neighbourhood of the neuron N_i is defined as:

$$S = \{\mathsf{N}_j : 0 < \mathsf{d}_{card}(\mathsf{N}_i, \mathsf{N}_j) < d^*\} \tag{2.5}$$

$$d^* = \min(2G + 1, D^*, m/2), \tag{2.6}$$

where G means its actual value (not the initial one).

Error – the sum of distances between the guards and their winning neurons – is calculated in every iteration (Alg. 2, lines 7 and 14). If this error is less than some threshold, the main loop is terminated. If the actual value of G is smaller than 0.01 or no neurons were moved in the last iteration the main loop is terminated too (line 28). In the case the main loop was not terminated, the parameter G is updated (lines 29–33):

$$G^{\text{new}} = \begin{cases} G(1-2\alpha) & \text{for } G > G^{\#}/2, \\ G(1-\alpha) & \text{otherwise}, \end{cases} \tag{2.7}$$

where G^{new} is the new value of variable G.

Whereas in the Basic SOM, the path is constructed only once at the end of the algorithm, in the CAN, paths are constructed in many iterations and join the competition for the shortest final path (Alg. 2, line 4 and lines 25 – 26 and 35 – 38). Because the path construction is not negligible in terms of computational difficulty, the condition that *w1count* (the number of neurons that has won exactly once in the last iteration) is at least $\min(0.98n, n-100)$ must be met before the algorithm constructs the path (lines 23–24). The process of path construction itself begins with pairing the winning neurons whose counter w_i is equal to one to their guards. These neurons are inhibited for further use. Then, the closest neuron is found for the first unpaired guard, they are paired and the neuron is inhibited. This step is repeated for every unpaired guard. At the end, the order of the neurons on the string determines the order of the guards in the path.

2.3 Polygonal domain

The polygonal domain is defined as a part of two-dimensional Euclidean space surrounded by polygonal boundaries and containing polygonal obstacles. The guards are represented by points in this space. The methods solving the TSP in Euclidean domain must be modified to be usable in the polygonal domain. In section 2.3.1, the approach that incorporates the Multidimensional scaling (MDS) to convert the TSP task from the polygonal domain to the Euclidean domain will be recalled. The technique using geodesic distances and geodesic moves is mentioned in section 2.3.2.

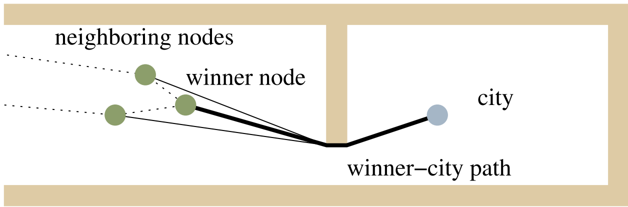
2.3.1 Multidimensional scaling (MDS) for TSP

The method used in [11] to solve the TSP problem in the polygonal domain will be shortly discussed in this section. For the pseudocode of the overall algorithm see Alg. 3. The procedure consist of calculation of the geodesic distances **E** between the guards – i.e. the lengths of the shortest paths from one guard to another which avoid the obstacles – (line 1) using the *VisiLibity* library [9] to calculate graph of visibility. Then the MDS algorithm (Stochastic forces or SMACOF) transforms these distances to positions of points (guards) in some higher-dimensional Euclidean space trying to approximate the specified distances **E** as much as possible (line 4). Recall that normal usage of the MDS algorithms is to transform data from some high-dimensional space to the one with lesser dimensions, whether in the approach [11] the MDS is used to transform in the opposite direction. The final step is to use ordinary SOM based methods (e.g. Basic SOM [10], CAN [2] or ORCSOM [12]) to solve the TSP in the Euclidean domain (line 5).

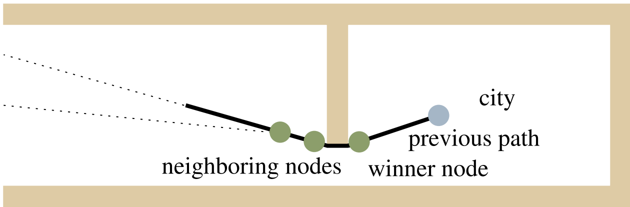
Some experiments with modifying the MDS and SOM algorithms to use other l_p norms than the l_2 norm were done in [11]. The main disadvantage of previous approach is that the MDS using Stochastic forces as implemented in [11] is slow and the SMACOF based MDS is unable to work with other norm than l_2 and is relatively slow too [11], [8].

Algorithm 3: Overall algorithm.

```
   Input: Map composed of polygons
   Input: Coordinates of guards
   Output: Solution of the TSP problem
1  compute geodesic distances E between guards
2  switch config do
3     case 1 do
4        use MDS algorithm (SMACOF, Stochastic forces, Glimmer) to
          place the guards to some Euclidean space according to E
5        use SOM algorithm (Basic SOM, CAN, ...) to solve the TSP in
          the Euclidean space
6        return path
7     end
8     case 2 do
9        use NESOM algorithm to solve the TSP in the non-Euclidean
          domain
10       return path
11    end
12 end
```



(a) : Before adaptation



(b) : After adaptation

Figure 2.1: Winner–guard(city) geodesic path – taken from [5].

2.3.2 Method of geodetic distances and movements (Fa-SOM)

Two methods to solve the TSP in the polygonal domain using the geodetic paths and distances are introduced in [5]: modified Basic SOM (referred to as modified SME or mSME in [5]) and modified CAN. To be used in the polygonal domain instead of the Euclidean domain some of the fundamental operations of the SOM methods has to be modified. Because the centroid of the guards can lie inside the obstacle, the initialization procedure is changed to use small circle around the first guard, around the guard nearest to the centroid or around the guard which has the smallest standard deviation of geodetic distances to the other guards. The convex hull of the guards can be used as the initial position of the neurons too. The neuron-guard distance computation is modified to return the length of the geodesic path – three variants with varying degree of accuracy are listed. Finally, the neuron movement procedure is changed to move the neurons using the geodesic path (see Fig. 2.1). Many optimization techniques are involved which leads to a great speedup of the algorithm.

Chapter 3

Own work

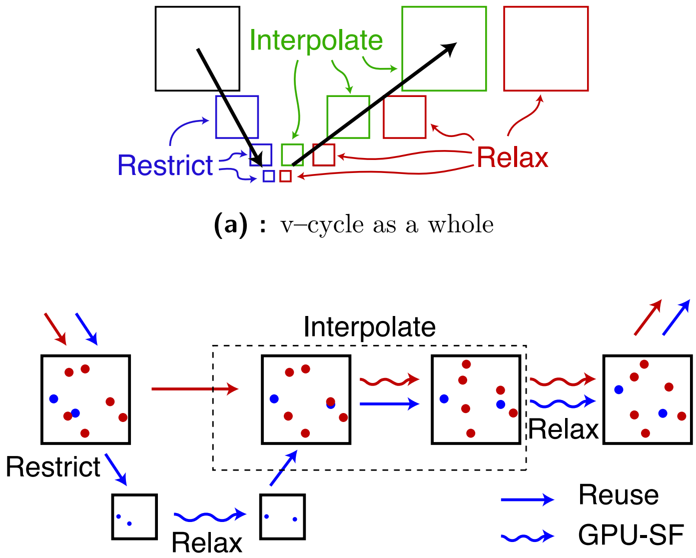
In the first part (section 3.1), the Glimmer algorithm and its use in the TSP will be showed. In the second part (section 3.2), the Non-Euclidean SOM method will be introduced.

3.1 Glimmer algorithm and its use in TSP

In section 3.1.1, the Glimmer algorithm will be described in the form as it was published in [8]. Incorporating of the algorithm to the TSP and its modifications to be able to calculate distances by the l_p norm instead of l_2 norm will be introduced in section 3.1.2.

3.1.1 Glimmer algorithm

The main principle of the Glimmer algorithm [8] is the usage of so called v-cycle (for pseudocode see Alg. 4). It is based on the fact that it is rather difficult to optimize positions of a high number of points when no information of appropriate initial position is known. In the first phase, the set of points is randomly restricted (lines 4–5) by some factor so long (line 1) that only a small number of points remain. This is the lowest level of the v–cycle (see Fig. 3.1), no previous information of the appropriate initial position is known.



(a) : v–cycle as a whole

(b) : v–cycle – one step in detail

Figure 3.1: The Glimmer algorithm v–cycle – taken from [8].

However, the number of points is low, so the process optimizing the positions of points trying to approximate the specified distances **E** as much as possible will take place relatively easily. After the optimization of this small subset using the Stochastic force technique (Alg. 4, line 2), the points are gradually returned. At the higher levels, the number of points is higher and higher, but the information of the appropriate point position from lower lever can be used – the "initial" position of the newly returned points must be, of course, interpolated from the positions of the points optimized at the previous level. This is done by the Stochastic force technique too with the exception that points optimized previously are fixed so that their positions are not messed up by the returned points (Alg. 4, line 6). Then the fixed points are relaxed, and the Stochastic force optimization performs again with all points on this level (Alg. 4, line 7).

At the beginning of the Glimmer algorithm (Alg. 5), the parameters must be set – see Table 3.1, proposed values are from [8]. The input distances are scaled (divided) so that the maximal distance in the distance matrix is one (Alg. 5, line 1). All points are randomly placed to the unit size hypercube $(0,1) \times (0,1) \times \cdots \times (0,1) = (0,1)^{\omega}$, where ω is the number of the dimensions (line 2). The main part of the algorithm, the v-cycle, is runned at line 3. Finally, the point positions are unscaled by the same factor as used at line 1.

The Stochastic force optimization process [8] is inspired by the behaviour of the physical system with n particles. The difference is that each particle interacts with few neighbours only instead of all other particles. The

Algorithm 4: Glimmer – v–cycle

```
Input: Points
Input: Matrix of distances E
Output: Modified points
if |points| ≤ MIN_SET_SIZE then                          // Lowest level?
    stoch_force (∅, points, E)
else
    subset ← restrict (points)                           // Restrict
    vcycle (subset, E)                                   // Process lower levels
    stoch_force (subset, points \ subset, E)             // Interpolate
    stoch_force (∅, points, E)                           // Relax
end
return points
```

Algorithm 5: Glimmer – the overall algorithm

```
  Input: Matrix of distances between points (guards) E
  Input: Number of dimensions ω of Euclidean space to place points into
  Output: Points (in the Euclidean space)
1 scale distances so that maximum of cell values of the matrix E is 1
2 place points randomly to unit hypercube
3 vcycle (points, E)                                     // Process v-cycle
4 unscale points by the same factor as in line 1
5 return points
```

neighbourhood of every point P_i is represented by two sets – the set of near points $\mathcal{V}_i$ and the set of random points $\mathcal{S}_i$. The algorithm is based on the iterative process (Alg. 6, lines 3–34). In each iteration, the set $\mathcal{S}_i$ is randomly generated, and then the points from the set $(\mathcal{V}_i \cup \mathcal{S}_i)$ nearest to P_i are placed to $\mathcal{V}_i$ and the rest to $\mathcal{S}_i$. This is done for every point P_i (lines 4–9). After it, the force is calculated for each point (lines 10–23). The force consists of two components. The spring force is repulsive if two points are too close, i.e. closer than the required distance $\mathbf{E}_{[i,j]}$, and attractive if they are too far. The damping force is repulsive if two points are approaching one another too fast and attractive if they are moving away two fast. This improves the stabilization of the system. The velocity (lines 24–28) and the position (lines 29–31) of each point is updated using the Euler integration formula finally. The speed of points are reduced and limited(lines 26–27) for better stabilization of the system.

Algorithm 6: Glimmer – stochastic force

Input: Fixed points $\mathcal{P}_{fixed}$ (coordinates of point P_k denoted $\boldsymbol{x_k}$)
Input: Free points $\mathcal{P}_{free}$ (coordinates of point P_l denoted $\boldsymbol{x_l}$)
Input: Matrix of distances $\mathbf{E}$
Output: Modified free points $\mathcal{P}_{free}$

$\forall i : \boldsymbol{v_i} \leftarrow 0$ `// clear velocities of points`
$\forall i : \mathcal{V}_i \leftarrow$ set of `V_SET_SIZE` randomly selected points `// near set`
while *true* **do**
  foreach $\mathsf{P}_i \in \mathcal{P}_{free}$ **do**
    $\mathcal{S}_i \leftarrow$ set of `S_SET_SIZE` randomly selected points `// random set`
    $\mathcal{Q} \leftarrow \mathcal{V}_i \cup \mathcal{S}_i$
    $\mathcal{V}_i \leftarrow$ `V_SET_SIZE` points from $\mathcal{Q}$ with the smallest original distances (see matrix $\mathbf{E}$) to P_i
    $\mathcal{S}_i \leftarrow \mathcal{Q} \setminus \mathcal{V}_i$
  end
  $\forall i : \boldsymbol{F_i} \leftarrow 0$ `// clear forces`
  foreach $\mathsf{P}_i \in \mathcal{P}_{free}$ **do**
    foreach $\mathsf{P}_j \in (\mathcal{V}_i \cup \mathcal{S}_i)$ **do**
      $\boldsymbol{w}_{spring} \leftarrow (\boldsymbol{x_j} - \boldsymbol{x_i}) \;/\; \|\boldsymbol{x_j} - \boldsymbol{x_i}\|$ `// direction vector`
      $F_{spring} \leftarrow (\|\boldsymbol{x_j} - \boldsymbol{x_i}\| - \mathbf{E}_{[i,j]}) \cdot$ `SPRINGFORCE` `// force size`
      $\boldsymbol{F_i} \leftarrow \boldsymbol{F_i} + F_{spring} \cdot \boldsymbol{w}_{spring}$ `// accumulate force`

      $\boldsymbol{w}_{damping} \leftarrow (\boldsymbol{x_j} - \boldsymbol{x_i}) \;/\; \|\boldsymbol{x_j} - \boldsymbol{x_i}\|$ `// direction vector`
      $\varphi \leftarrow \angle(\boldsymbol{w}_{damping} \,,\, \boldsymbol{v_j} - \boldsymbol{v_i})$ `// angle formed by vectors`
      $F_{damping} = \|\boldsymbol{v_j} - \boldsymbol{v_i}\| \cdot \cos(\varphi) \cdot$ `DAMPING` `// force size`
      $\boldsymbol{F_i} \leftarrow \boldsymbol{F_i} + F_{damping} \cdot \boldsymbol{w}_{damping}$ `// accumulate force`
    end
    $\boldsymbol{F_i} \leftarrow \boldsymbol{F_i} \;/\; |\mathcal{V}_i \cup \mathcal{S}_i|$ `// scale force by size of` $(\mathcal{V}_i \cup \mathcal{S}_i)$
  end
  foreach $\mathsf{P}_i \in \mathcal{P}_{free}$ **do**
    $\boldsymbol{v_i} \leftarrow \boldsymbol{v_i} + \Delta_{time} \cdot \boldsymbol{F_i}$
    $\boldsymbol{v_i} \leftarrow \boldsymbol{v_i} \cdot$ `FREENESS` `// reduce speed`
    limit $\boldsymbol{v_i}$ to specified maximal speed
  end
  foreach $\mathsf{P}_i \in \mathcal{P}_{free}$ **do**
    $\boldsymbol{x_i} \leftarrow \boldsymbol{x_i} + \Delta_{time} \cdot \boldsymbol{v_i}$ `// new position of` P_i
  end
  $\Delta stress \leftarrow$ calculate Δ of smoothed sparse stress
  if $\Delta stress < \varepsilon$ **then break**
end
return $\mathcal{P}_{free}$

Parameter	**Value**
Decimation factor `DEC_FACTOR`	8
Recursion termination condition `MIN_SET_SIZE`	100
Number of close neighbours `V_SET_SIZE`	14
Number of randomly chosen neighbours `S_SET_SIZE`	10
Spring force constant `SPRINGFORCE`	0.7
Damping force constant `DAMPING`	0.3
Freedom of movement constant `FREENESS`	0.85
Maximal speed limit (separately in each coordinate)	2.0
Time step size Δ_{time}	0.3
Termination threshold ε	10^{-4}
Which l_p norm to use (modified version of algorithm only)	2

Table 3.1: Glimmer – parameters and proposed values [8]

The naive approach to test whether to terminate the main loop (Alg. 6, lines 3–34) would be to calculate the value of the stress function defined as:

$$stress^2(points\ \mathcal{P}, \mathbf{E}) = \frac{\sum_{\mathsf{P}_i \in \mathcal{P}} \sum_{\mathsf{P}_j \in \mathcal{P}} \left(\|\boldsymbol{x_i} - \boldsymbol{x_j}\| - \mathbf{E}_{[i,j]} \right)^2}{\sum_{\mathsf{P}_i \in \mathcal{P}} \sum_{\mathsf{P}_j \in \mathcal{P}} \left(\mathbf{E}_{[i,j]} \right)^2}, \tag{3.1}$$

and then test the difference of the actual value and the value from previous iteration to some threshold. However, the asymptotic complexity of the stress calculation is $O(n^2)$, so it would be much slower then the rest of the iteration step. That is why, the sparse stress function is used (see [8]):

$$sparse\_stress^2(points\ \mathcal{P}, \mathbf{E}) = \frac{\sum_{\mathsf{P}_i \in \mathcal{P}} \sum_{\mathsf{P}_j \in \mathcal{V}_i \cup \mathcal{S}_i} \left(\|\boldsymbol{x_i} - \boldsymbol{x_j}\| - \mathbf{E}_{[i,j]} \right)^2}{\sum_{\mathsf{P}_i \in \mathcal{P}} \sum_{\mathsf{P}_j \in \mathcal{V}_i \cup \mathcal{S}_i} \left(\mathbf{E}_{[i,j]} \right)^2}, \tag{3.2}$$

where $\boldsymbol{x_k}$ denotes coordinates of P_k.

As stated in [8], the sparse stress value is so noisy, that it is inapplicable as the input to the termination threshold condition. Authors in [8] have been solved the problem so that they look at the sparse stress function value as it would be signal with a noise. They apply low-pass convolution filter of order 50 to smooth the behaviour of the function. Then, if the difference of the actual value and the previous value of the smoothed sparse stress is lesser than the parameter ε, the main loop is terminated (Alg. 6, lines 32–33).

3.1.2 Glimmer algorithm modifications

Usage of the Glimmer algorithm to solve the TSP in the polygonal domain is analogical to the approach shown in section 2.3.1. It is another MDS algorithm to choose in the middle step of the overall algorithm (Alg. 3, line 4).

In the previous text, the usage of the Glimmer algorithm with the l_2 norm was described. But former work [11] indicates that usage of other norms, especially l_∞, could bring some benefits. To run the Glimmer algorithm with other norms, the modifications listed below must be done. Recall the definition of the l_p norm:

$$\|\boldsymbol{z}\|_p = \left(\sum_{k=1}^{\omega} |z_k|^p \right)^{\frac{1}{p}}. \tag{3.3}$$

The first modification to cope with l_p norm is straightforward – the used norm in the sparse stress definition, see (3.2), is altered from $\|\boldsymbol{x_j} - \boldsymbol{x_j}\|_2$ to $\|\boldsymbol{x_j} - \boldsymbol{x_j}\|_p$:

$$sparse\_stress^2(points\ \mathcal{P}, \mathbf{E}) = \frac{\sum_{\mathsf{P}_i \in \mathcal{P}} \sum_{\mathsf{P}_j \in \mathcal{V}_i \cup \mathcal{S}_i} \left(\|\boldsymbol{x_i} - \boldsymbol{x_j}\|_p - \mathbf{E}_{[i,j]} \right)^2}{\sum_{\mathsf{P}_i \in \mathcal{P}} \sum_{\mathsf{P}_j \in \mathcal{V}_i \cup \mathcal{S}_i} \left(\mathbf{E}_{[i,j]} \right)^2}, \tag{3.4}$$

where $\boldsymbol{x_k}$ denotes the coordinates of P_k. This will ensure, that the points P_i and P_j will have zero contribution to summation in the numerator of the sparse stress if and only if the distance between them, measured by the l_p norm, is equal to the demanded original distance $\mathbf{E}_{[i,j]}$.

The second modification is similar norm replacement in the calculation of F_{spring} (Alg. 6, line 14):

$$F_{spring} = \left(\|\boldsymbol{x_j} - \boldsymbol{x_i}\|_p - \mathbf{E}_{[i,j]} \right) \cdot \texttt{SPRINGFORCE}. \tag{3.5}$$

This will ensure, that the spring force between two points P_i and P_j will be attractive if and only if the distance between them, measured by the l_p norm, is greater than the demanded original distance $\mathbf{E}_{[i,j]}$, and it will be repulsive if the distance is smaller.

The third modification changes the $\boldsymbol{w}_{spring}$ direction vector definition (Alg. 6, line 13). The simplest approach is to change the norm used in the vector normalization:

$$\boldsymbol{w}_{spring} = \frac{\boldsymbol{x_j} - \boldsymbol{x_i}}{\|\boldsymbol{x_j} - \boldsymbol{x_i}\|_p}. \tag{3.6}$$

The second way to set the direction vector ($\boldsymbol{w}_{spring\text{-}alt}$) is described below. Suppose that value of some function $N_p(\boldsymbol{z})$ should be reduced by the small movement of the vector $\boldsymbol{z}$. The usual way is to move the vector $\boldsymbol{z}$ in the direction of gradient of this function. So the normalized direction vector $\boldsymbol{w}$ is:

$$\boldsymbol{w} = \frac{\mathsf{grad}\, N_p(\boldsymbol{z})}{\|\mathsf{grad}\, N_p(\boldsymbol{z})\|_p}. \tag{3.7}$$

In the case of the Glimmer algorithm, the distance between points P_i and P_j measured by the norm l_p, i.e. $\|\boldsymbol{x_j} - \boldsymbol{x_i}\|_p$, should be lowered. Therefore the function $N_p(\boldsymbol{z})$ will be defined as:

$$N_p(\boldsymbol{z}) = \|\boldsymbol{z}\|_p. \tag{3.8}$$

The same formula written in another notation is:

$$N_p(\boldsymbol{x_j} - \boldsymbol{x_i}) = \|\boldsymbol{x_j} - \boldsymbol{x_i}\|_p, \tag{3.9}$$

where $\boldsymbol{x_j} - \boldsymbol{x_i} \equiv \boldsymbol{z}$. It follows from the definition of the gradient, that

$$\mathsf{grad}\, N_p(\boldsymbol{z}) = \left(\frac{\mathsf{d}\, N_p(\boldsymbol{z})}{\mathsf{d}\, z_1}, \frac{\mathsf{d}\, N_p(\boldsymbol{z})}{\mathsf{d}\, z_2}, \ldots, \frac{\mathsf{d}\, N_p(\boldsymbol{z})}{\mathsf{d}\, z_\omega} \right) = \tag{3.10}$$

$$= \left(\sum_{k=1}^{\omega} |z_k|^p \right)^{\frac{1}{p}-1} \left(|z_1|^{p-1} \,\mathsf{sgn}\, z_1 \,,\ |z_2|^{p-1} \,\mathsf{sgn}\, z_2 \,,\ \ldots \right). \tag{3.11}$$

Further from (3.3) and (3.11):

$$\|\mathsf{grad}\, N_p(\boldsymbol{z})\|_p = \left(\sum_{k=1}^{\omega} |z_k|^p \right)^{\frac{1}{p}-1} \left(\sum_{k=1}^{\omega} |z_k|^{p(p-1)} \right)^{\frac{1}{p}} \tag{3.12}$$

and from (3.7), (3.11) and (3.12):

$$\boldsymbol{w}_{spring\text{-}alt} = \frac{\mathsf{grad}\, N_p(\boldsymbol{z})}{\|\mathsf{grad}\, N_p(\boldsymbol{z})\|_p} = \tag{3.13}$$

$$= \left(\sum_{k=1}^{\omega} |z_k|^{p(p-1)} \right)^{-\frac{1}{p}} \left(|z_1|^{p-1} \,\mathsf{sgn}\, z_1 \,,\ |z_2|^{p-1} \,\mathsf{sgn}\, z_2 \,,\ \ldots \right). \tag{3.14}$$

This alternative normalized direction vector $\boldsymbol{w}_{spring\text{-}alt}$ can be used instead of $\boldsymbol{w}_{spring}$ (Alg. 6, line 13).

Discuss two special cases. First suppose that $p = 2$. We get:

$$\boldsymbol{w}_{\textit{spring-alt}} = \left(\sum_{k=1}^{\omega} |z_k|^2 \right)^{-\frac{1}{2}} \left(|z_1| \operatorname{sgn} z_1 \;,\; |z_2| \operatorname{sgn} z_2 \;,\; \ldots \right) = \tag{3.15}$$

$$= \left(\sum_{k=1}^{\omega} |z_k|^2 \right)^{-\frac{1}{2}} (z_1, z_2, \ldots) = \tag{3.16}$$

$$= \frac{\boldsymbol{z}}{\|\boldsymbol{z}\|_2} = \tag{3.17}$$

$$= \frac{\boldsymbol{x_j} - \boldsymbol{x_i}}{\|\boldsymbol{x_j} - \boldsymbol{x_i}\|_p} = \tag{3.18}$$

$$= \boldsymbol{w}_{\textit{spring}}. \tag{3.19}$$

It is obvious that for $p = 2$ there is no difference between choosing $\boldsymbol{w}_{\textit{spring}}$ and $\boldsymbol{w}_{\textit{spring-alt}}$.

For $p = \infty$, the limit has to be calculated (assume without loss of generity that $|z_1| > |z_k|, k \neq 1$):

$$\boldsymbol{w}_{\textit{spring-alt}} = \lim_{p \to \infty} \frac{\operatorname{grad} N_p(\boldsymbol{z})}{\|\operatorname{grad} N_p(\boldsymbol{z})\|_p} = \tag{3.20}$$

$$= \lim_{p \to \infty} \left(\sum_{k=1}^{\omega} |z_k|^{p(p-1)} \right)^{-\frac{1}{p}} \left(|z_1|^{p-1} \operatorname{sgn} z_1 \;,\; |z_2|^{p-1} \operatorname{sgn} z_2 \;,\; \ldots \right) = \tag{3.21}$$

$$= \lim_{p \to \infty} \left(\left(\sum_{k=1}^{\omega} |z_k|^{p(p-1)} \right)^{-\frac{1}{p}} |z_i|^{p-1} \operatorname{sgn} z_i \right)_{i=1,\ldots,\omega} = \tag{3.22}$$

$$= \lim_{p \to \infty} \left(\left(|z_i|^{-p(p-1)} \sum_{k=1}^{\omega} |z_k|^{p(p-1)} \right)^{-\frac{1}{p}} \operatorname{sgn} z_i \right)_{i=1,\ldots,\omega} = \tag{3.23}$$

$$= \lim_{p \to \infty} \left(\left(|z_i|^{-p(p-1)} \, |z_1|^{p(p-1)} \right)^{-\frac{1}{p}} \operatorname{sgn} z_i \right)_{i=1,\ldots,\omega} = \tag{3.24}$$

$$= \lim_{p \to \infty} \left(\left(\frac{|z_i|}{|z_1|} \right)^{(p-1)} \operatorname{sgn} z_i \right)_{i=1,\ldots,\omega} = \tag{3.25}$$

$$= \left(\operatorname{sgn} z_1,\; 0,\; 0, \ldots,\; 0 \right). \tag{3.26}$$

Thus for $p = \infty$ we obtain:

$$\boldsymbol{w}_{\textit{spring-alt}} = \frac{\operatorname{grad} N_\infty(\boldsymbol{x_j} - \boldsymbol{x_i})}{\|\operatorname{grad} N_\infty(\boldsymbol{x_j} - \boldsymbol{x_i})\|_\infty} = \tag{3.27}$$

$$= \left(0, \ldots, 0,\; \operatorname{sgn}\left((\boldsymbol{x_j} - \boldsymbol{x_i})_{[k]} \right),\; 0, \ldots, 0 \right), \tag{3.28}$$

where $(\boldsymbol{x_j} - \boldsymbol{x_i})_{[k]}$ is the element of the vector $\boldsymbol{x_j} - \boldsymbol{x_i}$ with the highest absolute value.

The Glimmer algorithm was modified to be able to use the l_p norm – both variants, i.e. the one using $\boldsymbol{w}_{spring}$ and the other using $\boldsymbol{w}_{spring\text{-}alt}$, will be tested and compared in chapter 4.

3.2 Non-Euclidean SOM (NESOM)

The Basic SOM and the CAN in the Euclidean domain (see sections 2.1 and 2.2) have representation of the guards and the neurons (using their Euclidean coordinates) and fundamental operations: initialization of neuron positions, the neuron movement towards the selected guard, the distance calculating, the path construction, etc. To create the non-Euclidean version of the algorithm, the non-Euclidean analogies of the representation and the fundamental operations have to be found.

The basic principle of the non-Euclidean SOM algorithm is introduced in section 3.2.1. The new representation of neurons is described in section 3.2.2. In the following sections, it is showed how to move a neuron (3.2.3), how to calculate distance (3.2.4) and how to speed up neuron movement (3.2.5) and the distance calculation (3.2.6). The transition from the Euclidean domain to the non-Euclidean domain is done in section 3.2.7, and the numerical stability of the proposed method is discussed in section 3.2.8. In later sections, it is shown how to initialize neuron positions (3.2.9) and how to construct a path (3.2.10). The path optimization by swapping is described in section 3.2.11. Finally, the overall non-Euclidean algorithm is introduced in section 3.2.12.

3.2.1 Basic principles

The proposed non-Euclidean algorithm is based on two principles. The first one is that when the TSP problem is solved (on a graph) the solution depends on the edge lengths (the distances between vertices) only. Also, when solving the TSP problem in the polygonal domain, the final solution should rely on the distances between guards only. It should not depend on the distance of an arbitrary point on the map to any other point on the map. The proposed algorithm should have the distance matrix only (the matrix of distances between guards) as the input.

The second principle takes an inspiration in the representation used in the Hopfield's network solution of TSP – it uses a matrix of size $n \times n$ for a problem containing n guards [7]. It starts with the matrix containing $1/n$ (plus small random disturbance) in every element, and it tries to end with the matrix containing just one value 1 in each column (and in each row). The first column specifies which guard will be visited as the first one, the second column specifies which guard will be visited as the second one, and so on. If elements of each column (with possible normalization of this columns to 1) are interpreted as coefficients of a linear combination of guards, we get that there are n points starting in the centroid of guards (with small random disturbance) and finishing each one of those n points at one guard. The Basic SOM and CAN networks behave very similarly [10], [2]. The difference is that every element of the matrix belongs to one neuron in the Hopfield's network, while every column of the matrix corresponds to one point (or neuron) in our interpretation. This way any state of the Hopfield's network containing n^2 neurons could be converted to the state of the SOM network comprising n neurons and vice versa. (In practice, the SOM network containing $2.5n$ or $3n$ neurons instead of n will be used, but it is not important now.)

3.2.2 Representation of neurons

The proposed new representation of neurons is described in this section. Consider solving of the TSP problem in the Euclidean domain. Guards are marked $\mathsf{G}_1, \ldots, \mathsf{G}_n$. Each of them lies in the ω-dimensional Euclidean space, and each has coordinates – denote them $\boldsymbol{g_k} = (g_{k,1}, g_{k,2}, \ldots, g_{k,\omega})^T$ (for guard G_k). The neurons $\mathsf{N}_1, \ldots, \mathsf{N}_m$ used in the neural network are also located in this Euclidean space. However, they will not be tracked by their coordinates. Instead, they will be expressed as a linear combination of individual guards. The coefficients of the linear combination are labelled $p_{i,l}$, and only such combinations that the sum of the coefficients of every combination will be equal to one are allowed:

$$\mathsf{N}_i = p_{i,1}\mathsf{G}_1 + p_{i,2}\mathsf{G}_2 + \cdots + p_{i,n}\mathsf{G}_n, \quad \text{where } \sum_{l=1}^{n} p_{i,l} = 1. \tag{3.29}$$

For coordinates, we get:

$$\boldsymbol{n_i} = \mathbf{G}\boldsymbol{p_i}, \tag{3.30}$$

where $\boldsymbol{n_i}$ are coordinates of neuron N_i, $\boldsymbol{p_i} = (p_{i,1}, \ldots, p_{i,n})^T$, and columns of the matrix $\mathbf{G}$ are made up of the vectors $\boldsymbol{g}$.

3.2.3 Movement of neurons

The movement of a neuron is one of the fundamental operations that has to be described in the proposed new representation. Common kind of neuronal movement among the SOM algorithms is to take some neuron (e.g. N_i) and move it towards the chosen guard (e.g. G_k) by a certain fraction of the distance between N_i and G_k. Denote this fraction γ. It must be fulfilled that $0 < \gamma < 1$. This movement can be characterized by the equation:

$$\begin{aligned}\mathsf{N}_i^{\text{new}} =& \mathsf{N}_i + \gamma(\mathsf{G}_k - \mathsf{N}_i) = \\ =& (1-\gamma)\mathsf{N}_i + \gamma \mathsf{G}_k,\end{aligned} \tag{3.31}$$

where N_i means the position of the neuron N_i before the movement, and $\mathsf{N}_i^{\text{new}}$ denotes the position of the neuron after the movement (similarly for other variables: $p_{i,l}$ for the value of the variable $p_{i,l}$ before the movement and $p_{i,l}^{\text{new}}$ for the value after it, and so on). Rewrite (3.31) to our notation: assume that

$$\mathsf{N}_i = \sum_{l=1}^{n} p_{i,l}\mathsf{G}_l \tag{3.32}$$

$$\mathsf{N}_i^{\text{new}} = \sum_{l=1}^{n} p_{i,l}^{\text{new}}\mathsf{G}_l, \tag{3.33}$$

from the equations (3.31), (3.32) and (3.33) we get

$$\sum_{l=1}^{n} p_{i,l}^{\text{new}}\mathsf{G}_l = \sum_{l=1}^{n}(1-\gamma)p_{i,l}\mathsf{G}_l + \gamma\mathsf{G}_k. \tag{3.34}$$

To meet the above equation for an arbitrary position of guards the following rule must hold true:

$$p_{i,l}^{\text{new}} = \begin{cases} (1-\gamma)p_{i,l} & \text{for } l \neq k \\ (1-\gamma)p_{i,l} + \gamma & \text{for } l = k \end{cases}. \tag{3.35}$$

The same formula written using vector notation is

$$\boldsymbol{p}_{\boldsymbol{i}}^{\text{new}} = (1-\gamma)\boldsymbol{p}_{\boldsymbol{i}} + \gamma(0,\dots,0,1,0,\dots,0), \tag{3.36}$$

where the value 1 is in the k-th element.

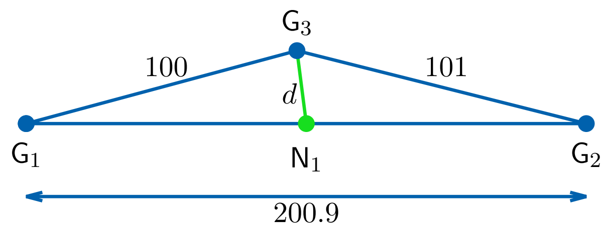


Figure 3.2: Problems with the simple distance function (see equation (3.37)): $d = 100.5$, but the real Euclidean distance is approx. 3.2. $\mathsf{N}_1 = (\mathsf{G}_1 + \mathsf{G}_2)/2$.

3.2.4 Distances

The calculation of the distance is another fundamental operation. Suppose the distance between neuron N_i and guard G_k is needed. The naive approach is to use the distance defined as

$$d(\mathsf{N}_i, \mathsf{G}_k) = \sum_{l=1}^{n} p_{i,l} d_{k,l}, \tag{3.37}$$

where $d_{k,l}$ is the distance between G_k and G_l.

The first disadvantage is that such distance function is distorted too much. Imagine three guards with distances $d_{1,2} = 200.9, d_{1,3} = 100, d_{2,3} = 101$ and a neuron with the linear combination coefficients $\boldsymbol{p_1} = (1/2, 1/2, 0)^T$. See Fig. 3.2. From the equation (3.37), we obtain $d = 0.5 d_{1,3} + 0.5 d_{2,3} = 100.5$ as the distance between the neuron and the guard G_3. However, the real Euclidean distance is approximately 3.2. This distortion caused significant problems with running the algorithm.

The second disadvantage is that some more advanced algorithms use the neuron–neuron distance too. Such distance function is not straightforwardly definable in a similar way as in the equation (3.37) if we want to maintain reasonable properties (for example, the distance between the neuron and the same neuron should be equal to zero). That is why more sophisticated approach is needed.

For the distance between two points in the Euclidean space holds true that:

$$d^2 = (\boldsymbol{x} - \boldsymbol{y})^T (\boldsymbol{x} - \boldsymbol{y}), \tag{3.38}$$

where $x = (x_1, x_2, \dots)^T$ and $y = (y_1, y_2, \dots)^T$ are coordinates of those two points. Thus the distance between two neurons N_i and N_j is

$$\begin{aligned} d^2(\mathsf{N}_i, \mathsf{N}_j) &= (\boldsymbol{n}_i - \boldsymbol{n}_j)^T(\boldsymbol{n}_i - \boldsymbol{n}_j) = \\ &= (\boldsymbol{p_i} - \boldsymbol{p_j})^T \mathbf{G}^T \mathbf{G} (\boldsymbol{p_i} - \boldsymbol{p_j}). \end{aligned} \tag{3.39}$$

From the fundamental assumptions

$$\sum_{l=1}^{n} p_{i,l} = 1 \tag{3.40}$$

$$\sum_{l=1}^{n} p_{j,l} = 1 \tag{3.41}$$

we obtain

$$\sum_{l=1}^{n} (p_{i,l} - p_{j,l}) = 0, \tag{3.42}$$

and so we can write:

$$\mathbf{H}(\boldsymbol{p_i} - \boldsymbol{p_j}) = 0, \tag{3.43}$$

where the matrix $\mathbf{H}$ is defined as:

$$\mathbf{H} = \begin{pmatrix} \boldsymbol{g}_1{}^T\boldsymbol{g}_1 & \boldsymbol{g}_1{}^T\boldsymbol{g}_1 & \dots & \boldsymbol{g}_1{}^T\boldsymbol{g}_1 \\ \boldsymbol{g}_2{}^T\boldsymbol{g}_2 & \boldsymbol{g}_2{}^T\boldsymbol{g}_2 & \dots & \boldsymbol{g}_2{}^T\boldsymbol{g}_2 \\ \vdots & \vdots & \vdots & \vdots \\ \boldsymbol{g}_n{}^T\boldsymbol{g}_n & \boldsymbol{g}_n{}^T\boldsymbol{g}_n & \dots & \boldsymbol{g}_n{}^T\boldsymbol{g}_n \end{pmatrix}. \tag{3.44}$$

Hence

$$\begin{aligned} d^2(\mathsf{N}_i, \mathsf{N}_j) =& (\boldsymbol{p_i} - \boldsymbol{p_j})^T \mathbf{G}^T \mathbf{G} (\boldsymbol{p_i} - \boldsymbol{p_j}) = \\ =& (\boldsymbol{p_i} - \boldsymbol{p_j})^T \mathbf{G}^T \mathbf{G} (\boldsymbol{p_i} - \boldsymbol{p_j}) - \frac{1}{2}(\boldsymbol{p_i} - \boldsymbol{p_j})^T \mathbf{H} (\boldsymbol{p_i} - \boldsymbol{p_j}) - \\ & - \frac{1}{2}(\boldsymbol{p_i} - \boldsymbol{p_j})^T \mathbf{H}^T (\boldsymbol{p_i} - \boldsymbol{p_j}) = \\ =& - \frac{1}{2}(\boldsymbol{p_i} - \boldsymbol{p_j})^T (-2\mathbf{G}^T\mathbf{G} + \mathbf{H} + \mathbf{H}^T)(\boldsymbol{p_i} - \boldsymbol{p_j}). \end{aligned} \tag{3.45}$$

Look at the matrix $(-2\mathbf{G}^T\mathbf{G} + \mathbf{H} + \mathbf{H}^T)$ in detail: the element on the k-th row and the l-th column is:

$$\begin{aligned} \left(-2\mathbf{G}^T\mathbf{G} + \mathbf{H} + \mathbf{H}^T\right)_{[k,l]} =& - 2\boldsymbol{g}_k{}^T\boldsymbol{g}_l + \boldsymbol{g}_k{}^T\boldsymbol{g}_k + \boldsymbol{g}_l{}^T\boldsymbol{g}_l = \\ =& (\boldsymbol{g}_k - \boldsymbol{g}_l)^T(\boldsymbol{g}_k - \boldsymbol{g}_l) = \\ =& d_{k,l}^2. \end{aligned} \tag{3.46}$$

Denote the matrix $(-2\mathbf{G}^T\mathbf{G} + \mathbf{H} + \mathbf{H}^T)$ as the matrix $\mathbf{D}$. It is the matrix of squares of distances between guards, it is symmetrical, and it has dimensions $n \times n$.

Finally, we see that the square of the distance between the neuron N_i and the neuron N_j is

$$d^2(\mathsf{N}_i, \mathsf{N}_j) = -\frac{1}{2}(\boldsymbol{p_i} - \boldsymbol{p_j})^T \mathbf{D}(\boldsymbol{p_i} - \boldsymbol{p_j}). \tag{3.47}$$

Note that $d^2(\mathsf{N}_i, \mathsf{N}_j)$ is always non-negative (when squares of the distances in the matrix $\mathbf{D}$ originate from Euclidean distances).

When the distance between the neuron N_i and the guard G_k is needed, the following method is used: create a virtual neuron with the linear combination coefficients $\boldsymbol{p_{\mathsf{G}_k}} = (0, \ldots, 0, 1, 0, \ldots, 0)^T$ (the value 1 is in the k-th element). Substitute $\boldsymbol{p_j}$ with $\boldsymbol{p_{\mathsf{G}_k}}$ in the equation (3.47). We obtain

$$\begin{aligned} d^2(\mathsf{N}_i, \mathsf{G}_k) &= -\frac{1}{2}(\boldsymbol{p_i} - \boldsymbol{p_{\mathsf{G}_k}})^T \mathbf{D}(\boldsymbol{p_i} - \boldsymbol{p_{\mathsf{G}_k}}) = \\ &= -\frac{1}{2}\boldsymbol{p_i}^T \mathbf{D}\boldsymbol{p_i} + \boldsymbol{p_{\mathsf{G}_k}}^T \mathbf{D}\boldsymbol{p_i} - \frac{1}{2}\boldsymbol{p_{\mathsf{G}_k}}^T \mathbf{D}\boldsymbol{p_{\mathsf{G}_k}} = \\ &= -\frac{1}{2}\boldsymbol{p_i}^T \mathbf{D}\boldsymbol{p_i} + (\mathbf{D}\boldsymbol{p_i})_{[k]} - \frac{1}{2}\mathbf{D}_{[k,k]} = \\ &= -\frac{1}{2}\boldsymbol{p_i}^T \mathbf{D}\boldsymbol{p_i} + (\mathbf{D}\boldsymbol{p_i})_{[k]}, \end{aligned} \tag{3.48}$$

as the square of the distance between the neuron N_i and the guard G_k.

3.2.5 Speedup of neuron movement and normalization

It follows from the equation (3.36) that every move of the neuron N_i leads to changing the entire vector $\boldsymbol{p_i}$ (which has n elements). The method for speeding up the neuron movement will be described in this section. First, a new variable f_i (neuron's factor) and a new vector $\widetilde{\boldsymbol{p_i}}$ will be introduced in such a way that the following equation applies:

$$\boldsymbol{p_i} = f_i \widetilde{\boldsymbol{p_i}}. \tag{3.49}$$

At the beginning, the variables f_i and the vectors $\widetilde{\boldsymbol{p_i}}$ will be set as follows: $f_i = 1$ and $\widetilde{\boldsymbol{p_i}} = \boldsymbol{p_i}$ (for every neuron). The variables f_i and the vectors $\widetilde{\boldsymbol{p_i}}$ will be used instead of the vectors $\boldsymbol{p_i}$ from then on.

When the movement of the neuron N_i towards the guard G_k is needed (see the equations (3.31), (3.35) and (3.36)), the following equations will be used:

$$f_i^{\text{new}} = (1 - \gamma) f_i \tag{3.50}$$

$$\widetilde{p}_{i,l}^{\text{new}} = \frac{p_{i,l}^{\text{new}}}{f_i^{\text{new}}} = \begin{cases} \frac{(1-\gamma)}{f_i^{\text{new}}} p_{i,l} = \widetilde{p}_{i,l} & \text{for } l \neq k \\ \frac{(1-\gamma)}{f_i^{\text{new}}} p_{i,l} + \frac{\gamma}{f_i^{\text{new}}} = \widetilde{p}_{i,l} + \frac{\gamma}{f_i^{\text{new}}} & \text{for } l = k, \end{cases} \tag{3.51}$$

where f_i denotes the value of the variable f_i before the movement and f_i^{new} after it (similarly for $\widetilde{p}_{i,l}$ and $p_{i,l}$). This way, the whole vector $\boldsymbol{p_i}$ containing n elements needs not to be changed, changing two values in memory (f_i and $\widetilde{p}_{ik}$) is enough.

As the algorithm runs and the neurons are moving, the values of f_i are getting lower and lower, and the values of $\widetilde{p}_{i,l}$ are getting higher and higher. However, common numerical types used in computers (e.g. `double`, `float`) have a limited range. Therefore at some moment, normalization is needed – the threshold condition used is: $f_i < 10^{-30}$. When this condition is met for some neuron (e.g. N_i), the values of the variables will be changed according to the following rules:

$$\widetilde{\boldsymbol{p_i}}^{\text{ren}} = f_i^{\text{orig}} \widetilde{\boldsymbol{p_i}}^{\text{orig}} \tag{3.52}$$

$$f_i^{\text{ren}} = 1, \tag{3.53}$$

where $\widetilde{\boldsymbol{p_i}}^{\text{orig}}$ denotes the vector $\widetilde{\boldsymbol{p_i}}$ before normalization and $\widetilde{\boldsymbol{p_i}}^{\text{ren}}$ after it (similarly for f_i). Soon, these rules will be expanded by normalizing the cache of distances.

3.2.6 Distance caching

Because the SOM algorithm (e.g. Basic SOM) searches for the neuron nearest to each guard, it needs to compute nm distances in every step of the main iterative process. If the distances were calculated using the equation (3.47), they would have the asymptotic complexity $O(n^2)$ for computation of every distance and thus $O(n^3 m)$ for every iteration. The algorithm would be too slow in such a case. Therefore two caching variables are introduced to speed up the distance computation – the matrices $\mathbf{C_{DP}}$ and $\mathbf{C_{PDP}}$. Their relevance will be apparent from the following text.

The matrix $\mathbf{C_{DP}}$ has dimensions $n \times m$ and is defined by:

$$\mathbf{C_{DP}} = \mathbf{D}\widetilde{\mathbf{P}}, \tag{3.54}$$

where columns of the matrix $\widetilde{\mathbf{P}}$ are made up of the vectors $\widetilde{\boldsymbol{p_i}}$. (The matrix $\widetilde{\mathbf{P}}$ has dimensions $n \times m$.)

The second caching matrix – the matrix $\mathbf{C_{PDP}}$ has dimensions $m \times m$ and is defined by:

$$\mathbf{C_{PDP}} = \widetilde{\mathbf{P}}^T \mathbf{D} \widetilde{\mathbf{P}} = \widetilde{\mathbf{P}}^T \mathbf{C_{DP}}. \tag{3.55}$$

Note that the matrix $\mathbf{C}_{\mathbf{PDP}}$ is symmetrical, that is why the algorithm does not have to compute and store the part of the matrix below the main diagonal.

When the distance between two neurons N_i and N_j is needed, we obtain from the equation (3.47):

$$\begin{aligned} d^2(\mathsf{N}_i, \mathsf{N}_j) &= -\frac{1}{2}(\boldsymbol{p_i} - \boldsymbol{p_j})^T \mathbf{D}(\boldsymbol{p_i} - \boldsymbol{p_j}) = \\ &= -\frac{1}{2}\boldsymbol{p_i}^T \mathbf{D}\boldsymbol{p_i} - \frac{1}{2}\boldsymbol{p_j}^T \mathbf{D}\boldsymbol{p_j} + \boldsymbol{p_j}\mathbf{D}\boldsymbol{p_i} = \\ &= -\frac{1}{2} f_i^2 \widetilde{\boldsymbol{p_i}}^T \mathbf{D}\widetilde{\boldsymbol{p_i}} - \frac{1}{2} f_j^2 \widetilde{\boldsymbol{p_j}}^T \mathbf{D}\widetilde{\boldsymbol{p_j}} + f_i f_j \widetilde{\boldsymbol{p_j}}\mathbf{D}\widetilde{\boldsymbol{p_i}} = \\ &= -\frac{1}{2} f_i^2 \mathbf{C}_{\mathbf{PDP}[i,i]} - \frac{1}{2} f_j^2 \mathbf{C}_{\mathbf{PDP}[j,j]} + f_i f_j \mathbf{C}_{\mathbf{PDP}[i,j]}. \end{aligned} \tag{3.56}$$

The calculation according to the previous formula has the asymptotic complexity $O(1)$.

When the neuron N_i moves towards the guard G_k, the cache has to be updated accordingly. Consider the equations (3.50) and (3.51) characterizing the movement of the neuron. The first one will not affect the cache at all. Denote $\delta = \gamma / f_i^{\text{new}}$, thus:

$$\widetilde{p}_{i,l}^{\text{new}} = \begin{cases} \widetilde{p}_{i,l} & \text{for } l \neq k \\ \widetilde{p}_{i,l} + \delta & \text{for } l = k. \end{cases} \tag{3.57}$$

Let the matrix $\mathbf{\Delta}$ be the matrix of dimensions $n \times m$ having zeroes at all cells with one exception – the element in the k-th row and the i-th column will be δ. Thus

$$\widetilde{\mathbf{P}}^{\text{new}} = \widetilde{\mathbf{P}} + \mathbf{\Delta} \tag{3.58}$$

$$\mathbf{C}_{\mathbf{DP}}^{\text{new}} = \mathbf{D}\widetilde{\mathbf{P}}^{\text{new}} = \mathbf{D}\widetilde{\mathbf{P}} + \mathbf{D}\mathbf{\Delta} = \mathbf{C}_{\mathbf{DP}} + \mathbf{D}\mathbf{\Delta} \tag{3.59}$$

$$\begin{aligned} \mathbf{C}_{\mathbf{PDP}}^{\text{new}} &= (\widetilde{\mathbf{P}}^{\text{new}})^T \mathbf{D}\widetilde{\mathbf{P}}^{\text{new}} = \\ &= (\widetilde{\mathbf{P}} + \mathbf{\Delta})^T \mathbf{D}(\widetilde{\mathbf{P}} + \mathbf{\Delta}) = \\ &= \widetilde{\mathbf{P}}^T \mathbf{D}\widetilde{\mathbf{P}} + \widetilde{\mathbf{P}}^T \mathbf{D}\mathbf{\Delta} + \mathbf{\Delta}^T \mathbf{D}\widetilde{\mathbf{P}} + \mathbf{\Delta}^T \mathbf{D}\mathbf{\Delta} = \\ &= \mathbf{C}_{\mathbf{PDP}} + \left(\mathbf{\Delta}^T \mathbf{D}\widetilde{\mathbf{P}}\right)^T + \mathbf{\Delta}^T \mathbf{D}\widetilde{\mathbf{P}} + \mathbf{\Delta}^T \mathbf{D}\mathbf{\Delta} = \\ &= \mathbf{C}_{\mathbf{PDP}} + \left(\mathbf{\Delta}^T \mathbf{C}_{\mathbf{DP}}\right)^T + \mathbf{\Delta}^T \mathbf{C}_{\mathbf{DP}} + \mathbf{\Delta}^T \mathbf{D}\mathbf{\Delta}. \end{aligned} \tag{3.60}$$

In other words, the i-th column and the i-th row of the matrix $\mathbf{C}_{\mathbf{PDP}}$ have to be updated this way:

$$\begin{aligned} \mathbf{C}_{\mathbf{PDP}[i,i]}^{\text{new}} &= \mathbf{C}_{\mathbf{PDP}[i,i]} + 2\delta \mathbf{C}_{\mathbf{DP}[k,i]} + \delta^2 \mathbf{D}_{[k,k]} = \\ &= \mathbf{C}_{\mathbf{PDP}[i,i]} + 2\delta \mathbf{C}_{\mathbf{DP}[k,i]} \\ \mathbf{C}_{\mathbf{PDP}[j,i]}^{\text{new}} &= \mathbf{C}_{\mathbf{PDP}[j,i]} + \delta \mathbf{C}_{\mathbf{DP}[k,j]} \quad \text{for } \forall j \in \{1, \ldots, i-1\} \\ \mathbf{C}_{\mathbf{PDP}[i,j]}^{\text{new}} &= \mathbf{C}_{\mathbf{PDP}[i,j]} + \delta \mathbf{C}_{\mathbf{DP}[k,j]} \quad \text{for } \forall j \in \{i+1, \ldots, m\}. \end{aligned} \tag{3.61}$$

This change has the asymptotic complexity $O(m)$. Moreover, the i-th column of the matrix $\mathbf{C}_{\mathbf{DP}}$ has to be updated in this manner:

$$\mathbf{C}^{\text{new}}_{\mathbf{DP}\,[l,i]} = \mathbf{C}_{\mathbf{DP}\,[l,i]} + \delta \mathbf{D}_{[l,k]} \quad \text{for } \forall l \in \{1, \ldots, n\}. \tag{3.62}$$

This update has the asymptotic complexity $O(n)$.

If some algorithms need to know distances between neurons and guards only (as the basic ones do), the equation (3.48) will be used for the distance calculation instead of (3.47), so the caching will become more simple. From (3.48) we obtain:

$$\begin{aligned} d^2(\mathsf{N}_i, \mathsf{G}_k) &= -\frac{1}{2}\boldsymbol{p_i}^T \mathbf{D} \boldsymbol{p_i} + (\mathbf{D}\boldsymbol{p_i})_{[k]} - \frac{1}{2}\mathbf{D}_{[k,k]} \\ &= -\frac{1}{2} f_i^2 \widetilde{\boldsymbol{p_i}}^T \mathbf{D} \widetilde{\boldsymbol{p_i}} + f_i (\mathbf{D}\widetilde{\boldsymbol{p_i}})_{[k]} \\ &= -\frac{1}{2} f_i^2 \mathbf{C}_{\mathbf{PDP}\,[i,i]} + f_i \mathbf{C}_{\mathbf{DP}\,[k,i]}. \end{aligned} \tag{3.63}$$

In such situation, the algorithm has to calculate and store the main diagonal of the matrix $\mathbf{C}_{\mathbf{PDP}}$ only, and the asymptotic complexity of the matrix $\mathbf{C}_{\mathbf{PDP}}$ update will become $O(1)$.

Finally, the previously established rules for normalization, see the equations (3.52) and (3.53), have to be extended. Applying (3.52) to the equations (3.54) and (3.55), we get:

$$\begin{aligned} \widetilde{\mathbf{P}}^{\text{ren}}_{[l,i]} &= f_i^{\text{orig}} \widetilde{\mathbf{P}}^{\text{orig}}_{[l,i]} && \text{for } \forall l \in \{1, \ldots, n\} \\ \mathbf{C}^{\text{ren}}_{\mathbf{DP}\,[l,i]} &= f_i^{\text{orig}} \mathbf{C}^{\text{orig}}_{\mathbf{DP}\,[l,i]} && \text{for } \forall l \in \{1, \ldots, n\} \\ \mathbf{C}^{\text{ren}}_{\mathbf{PDP}\,[i,i]} &= (f_i^{\text{orig}})^2 \mathbf{C}^{\text{orig}}_{\mathbf{PDP}\,[i,i]} && \\ \mathbf{C}^{\text{ren}}_{\mathbf{PDP}\,[j,i]} &= f_i^{\text{orig}} \mathbf{C}^{\text{orig}}_{\mathbf{PDP}\,[j,i]} && \text{for } \forall j \in \{1, \ldots, i-1\} \\ \mathbf{C}^{\text{ren}}_{\mathbf{PDP}\,[i,j]} &= f_i^{\text{orig}} \mathbf{C}^{\text{orig}}_{\mathbf{PDP}\,[i,j]} && \text{for } \forall j \in \{i+1, \ldots, m\} \\ f_i^{\text{ren}} &= 1 \end{aligned} \tag{3.64}$$

as the normalization rules for the neuron N_i.

For the asymptotic complexity of basic operations see Table 3.2. It is obvious from the third and the fourth row that the neuron movement is much more demanding in terms of computational complexity than the distance calculating.

Variant	Neuron –neuron distance	Neuron –guard distance	Neuron move incl. cache update	Normalization
no move speed-up no cache	$O(n^2)$	$O(n^2)$	$O(n)$	–
move speed-up no cache	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n)$
move speed-up full cache	$O(1)$	$O(1)$	$O(n+m)$	$O(n+m)$
move speed-up $\mathbf{C}_{\mathbf{PDP}}$ diag. only	–	$O(1)$	$O(n)$	$O(n)$

Table 3.2: Asymptotic complexity of basic operations.

3.2.7 Non-Euclidean distances and negative squares of distances

Until now, we assumed that the TSP problem in some hypothetical Euclidean space is being solved. However, the task is to solve the problem in the non-Euclidean domain. So a transition from the Euclidean domain to the non-Euclidean domain must be made.

This transition is simply done by assignment of squares of non-Euclidean distances to the matrix $\mathbf{D}$. (This distances can be obtained from the first part of the overall algorithm for example – see Alg. 3, line 1.) We expect the distances to meet the triangular inequality:

$$\forall i, j, k : d_{i,j} \leq d_{i,k} + d_{k,j}. \tag{3.65}$$

Consider whether the square of the distances (d^2) as defined in section 3.2.4 will always be non-negative. Imagine three arbitrary distances respecting the triangular inequality. Such distances are always Euclidean (a triangle can be constructed in some Euclidean space so that the lengths of the sides are equal to the specified distances). Therefore in situations, where only three individual distances from the matrix $\mathbf{D}$ has an effect in the computation of d^2 from the equation (3.47), the resulting d^2 will always be non-negative (because the calculation behaves like it were in the Euclidean space). Taking into account that the matrix $\mathbf{D}$ is symmetrical and has zeroes on the main diagonal, the previous eventualities correspond to the situations when at most three elements of the vector $(\boldsymbol{p_i} - \boldsymbol{p_j})$ are non-zero. It can be the trivial TSP task with $n \leq 3$ for example. Furthermore, it may be the case of calculating

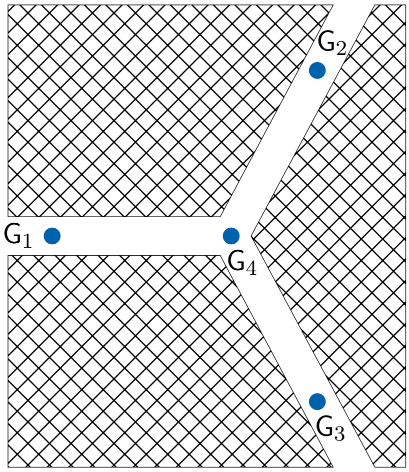


Figure 3.3: How a negative value of d^2 can occur. The hatched areas are obstacles. $\mathsf{N}_1 = (\mathsf{G}_1 + \mathsf{G}_2 + \mathsf{G}_3)/3$. Value d^2 is square of the distance between N_1 and G_4.

the distance between a neuron combined from three guards (the neuron which has three non-zero coefficients in its linear combination and the other coefficients are equal to zero) and one of those guards. Alternatively, it can be the situation of calculating the distance between a neuron with only two non-zero coefficients in its linear combination and arbitrary guard. (This may be the case in the late stage of the algorithm run when the neurons are very close to the guards, or they are near lines joining two guards. Then the negative values of d^2 occur sporadically.)

Several situations where can be proven that d^2 is always non-negative were discussed. However, in general, it is not guaranteed that d^2 is non-negative. See the situation in Fig. 3.3. There are four guards and one neuron in this arrangement. The hatched areas are obstacles, the distance between G_i and G_4 is 1 (for $i = 1, \ldots, 3$) and $\mathsf{N}_1 = (\mathsf{G}_1 + \mathsf{G}_2 + \mathsf{G}_3)/3$. The matrix of the squared distances $\mathbf{D}$ will be:

$$\mathbf{D} = \begin{pmatrix} 0 & 4 & 4 & 1 \\ 4 & 0 & 4 & 1 \\ 4 & 4 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}. \tag{3.66}$$

Consider the distance between the neuron N_1 and the guard G_4. From the equation (3.47), we get:

$$d^2(\mathsf{N}_1, \mathsf{G}_4) = -\frac{1}{2}\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}, -1\right)^T \mathbf{D} \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}, -1\right) = -\frac{1}{3}. \tag{3.67}$$

Thus d^2 can be negative in some cases. (Recall that the used distances meet the triangular inequality.) If the `sqrt` function is called somewhere in the algorithm to get d from d^2, the algorithm will crash.

There are two approaches how to deal with the situation. The first one is to detect the negative value and replace it with the zero value. The second option is the following technique. When the algorithm searches for the nearest neuron for some guard, it looks for the neuron with the minimal distance d between the neuron and the guard. This step can be equivalently replaced by looking for the neuron with the minimal square of distance d^2. Now, if there is any negative value between the values of d^2, it will be left, and it will win over non-negative values in the search for the minimum. However, none of the previous options gives much better results than the other.

3.2.8 Numerical stability

The numerical stability of the proposed method will be briefly discussed in this section. One of the fundamental assumptions is that the sum of the coefficients $p_{i,l}$ of every neuron is equal to one:

$$\sum_{l=1}^{n} p_{i,l} = 1. \tag{3.68}$$

However, as the algorithm runs and the neurons are moving, the values of $p_{i,l}$ are repeatedly changed. These changes are designed in such a way that the equation (3.68) is still valid – see (3.31) and (3.35). Nevertheless, this is only met theoretically. Running the algorithm on a real computer using data types like `double` and `float` rounding errors arise in each operation. If these rounding errors cumulated, the algorithm might crash. Consider whether this situation can occur. Assume that the equation (3.68) is not fully met:

$$\sum_{l=1}^{n} p_{i,l} = 1 + \varepsilon. \tag{3.69}$$

Explore what happens after the neuron N_i moves. (Suppose it moves towards the guard G_k.) From (3.35) we obtain:

$$\sum_{l=1}^{n} p_{i,l}^{\text{new}} = (1-\gamma)\sum_{l=1}^{n} p_{i,l} + \gamma = (1-\gamma)(1+\varepsilon) + \gamma = 1 + (1-\gamma)\varepsilon. \tag{3.70}$$

It follows from $0 < \gamma < 1$ that the rate of violation of the equation (3.68) after the move (which is $(1-\gamma)\varepsilon$) is smaller than the rate before the move (ε). Fortunately, earlier errors naturally disappear as the neuron moves thus the algorithm is numerically stable. The same is true when f_i and $\widetilde{\boldsymbol{p_i}}$ are used (see the equations (3.50) and (3.51)) instead of $\boldsymbol{p_i}$. Moreover, a similar principle applies to the calculation of the distances using cache in the equations (3.56) and (3.63).

3.2.9 Initial position of neurons

Before the SOM network is ready to run, the neurons have to be initialized. Two methods will be described in this section – the first places neurons near the centroid of the guards and the second one uses the `FastTSP` algorithm.

Algorithm 7: Initialization of neurons using the centroid of the guards (`centroid_init`)

Input: Matrix of distances between guards E
Output: Initial position of neurons including prepared distance cache
1 set all neurons to the centroid ($\forall i\ \forall l : \widetilde{p}_{i,l} \leftarrow 1/n$ and $\forall i : f_i \leftarrow 1$)
2 compute cache for one neuron
3 copy the previous result to the rest of the cache `// all neurons are equal`
4 *permutation* $\leftarrow$ random permutation of sequence $(1, \dots, m)$
5 **for** $k \leftarrow 1$ **to** n **do**
6     $i \leftarrow permutation[k]$
7   move neuron N_i by 1% towards guard G_k `// see sect.3.2.5`
8   update cache accordingly `// see sect.3.2.6`
9 **end**
10 run one iteration of SOM algorithm with special parameter settings (very small μ, very high G)
11 **return** *neurons, cache*

One way to initialize neurons in the SOM network in the Euclidean space is to place them on a small circle formed around the centre of gravity of the guards – see sections 2.1 and 2.2. Because in our representation, it would be hard to form a circle around some point another procedure will be used. (For the pseudocode of the entire initialization procedure see Alg. 7). First, all neurons are placed to the centroid ($f_i = 1$ and $\widetilde{p}_{i,l} = 1/n$ for every i and every l, see Alg. 7, lines 1–3). Then randomly selected n neurons (note that the total number of the neurons is m) are moved by 1% ($\gamma = 0.01$) towards the guards $\mathsf{G}_1, \dots, \mathsf{G}_n$ (always one neuron towards one guard). Random choose without repetition is used thus no neuron will move more than once, and every guard will be used just once (Alg. 7, lines 4–9).

The position of the neurons from the previous paragraph could be used as an initial position of the SOM run. However, the string of neurons (its projection to the original polygonal space respectively) intersects itself many times. As the SOM network has difficulties to get rid of some intersections of the neural string, passing these imperfections to the final route, it would be worthwhile to solve the problem another way. It can be fixed by running the first iteration of the SOM algorithm with special settings of the algorithm constants: very small μ ($\mu_{\text{c-init}} = 0.04$) and very high G ($G_{\text{c-init}} = 500$). Thanks to the large value of the parameter G, each winning neuron has big neighbourhood and this victorious neuron is moving to the selected guard with many of his neighbours. Because the value of the parameter μ is small, neurons change their position only a little in every move. Therefore resulting movement is smooth, and as individual guards are picked at random from different parts of the map, a string with no (or at worst with a few) self-intersections is created near the centroid (Alg. 7, line 10).

Algorithm 8: FastTSP

Input: Matrix $\mathbf{E}$ (dimensions $n \times n$) of distances between guards
Output: Solution of TSP as a sequence of indices of individual guards in the route

distances $\leftarrow$ () `// empty vector`
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow i+1$ **to** n **do**
    append triplet $(i, j, \mathbf{E}[i, j])$ to the end of *distances*
  end
end
sort *distances* by the third element of triplet (distance) from the shortest one to the longest one
edges $\leftarrow \emptyset$ `// empty set of edges`
$k \leftarrow 0$ `// number of edges`
$l \leftarrow 1$ `// index`
while $k < n$ **do**
  $edge \leftarrow (distances[l].i, distances[l].j)$
  if (*adding edge to edges will not create cycle* $\vee\ k = n-1$) $\wedge$ (*adding edge to edges will not create vertex with three or more incident edges*) **then**
    $edges \leftarrow edges \cup \{edge\}$
    $k \leftarrow k+1$
  end
  $l \leftarrow l+1$ `// go to the next triplet`
end
convert the set of *edges* to a *sequence* of indices of individual guards in the route
return *route (as the sequence of indices)*

Another way to initialize neurons uses the `FastTSP` algorithm[6] (greedy algorithm over the edge lengths). For the pseudocode of the `FastTSP` algorithm see Alg. 8 and for the pseudocode of entire initialization procedure see Alg. 9. At the beginning of the `FastTSP` algorithm, it sorts all distances between guards from the shortest one to the longest one (Alg. 8, lines 1–7). Then it starts with the empty set of edges, takes the shortest edge and joins this edge to the set. After this, the algorithm takes such shortest edge, which has not been used yet and whose adding to the set will create neither cycle nor vertex with three or more edges incident (with the exception that n-th edge can create a cycle). The previous step is repeated until there are n edges in the set (Alg. 8, lines 8–18). This set forms the path through all guards, each of them visited just once.

Algorithm 9: Initialization of neurons using the FastTSP algorithm (`FastTSP_init`)

Input: Matrix $\mathbf{E}$ of distances between guards
Output: Initial position of neurons including prepared distance cache

```
1  path ← FastTSP(E)                                    // see Alg. 8
2  do swap optimization: do_swaps(path)                 // see Alg. 12
3  set all neurons to the centroid (∀i ∀l : p̃_{i,l} ← 1/n and ∀i : f_i ← 1)
4  compute cache for one neuron
5  copy the previous result to the rest of the cache
                                                        // all neurons are equal
6  for i ← 1 to m do
7  |   l ← ⌈(n/m)i⌉             // index in path, notice ceil function
8  |   k ← path[l]
9  |   move neuron N_i by 99.99% towards guard G_k      // see sect.3.2.5
10 |   update cache accordingly                         // see sect.3.2.6
11 end
12 return neurons, cache
```

The `FastTSP` algorithm is done now (Alg. 9, line 1), and the path obtained is optimized by swapping – see section 3.2.11 (Alg. 9, line 2). This could shorten the path by removing some imperfections that the `FastTSP` leaves in the path (intersections with itself for example). Then, all neurons are placed to the centroid of the guards (Alg. 9, lines 3–5). Finally, the first m/n neurons are moved (i.e. the first three neurons for the case when $m = 3n$, etc.) by 99.99% ($\gamma = 0.9999$) to the first guard in the path, next m/n neurons to the second guard in the path and so on (Alg. 9, lines 6–11). It is the initial location of the neurons obtained by the `FastTSP_init` method.

The naive approach in the previous method would be to place the neurons straight at the positions of the selected guards. The reason to do it differently is that creating of the distance cache will be much faster. Filling up the cache of m different neurons means to compute the matrix $\mathbf{C_{DP}}$ from the equation (3.54) and the matrix $\mathbf{C_{PDP}}$ from (3.55). The first computation has the asymptotic complexity $O(n^2 m)$ and the second one has $O(nm^2)$.

Therefore calculating the cache of m different neurons has the asymptotic complexity $O\left(nm(n+m)\right)$.

To fill up the cache of m same neurons, the matrix $\mathbf{C_{DP}}$ has to be calculated from the equation (3.54) first. This time, the fact that the matrix $\widetilde{\mathbf{P}}$ has the same columns can be used thus the matrix $\mathbf{C_{DP}}$ will have identical columns too. It takes $O(n^2)$ to compute the first column of $\mathbf{C_{DP}}$ and $O(nm)$ to copy this column to the other columns. Then, the matrix $\mathbf{C_{PDP}}$ has to be calculated from the equation (3.55). The matrix $\mathbf{C_{PDP}}$ has all elements equal to each other because the matrix $\mathbf{C_{DP}}$ has identical columns and that the matrix $\widetilde{\mathbf{P}}^T$ has identical rows. It takes $O(n)$ to compute one cell of $\mathbf{C_{PDP}}$ and $O(m^2)$ to copy this cell to the rest of the matrix. After preparing the cache of m equal neurons, all m neurons have to be moved to the desired positions. From the previous results (see Table 3.2) we know that movement of the neuron has the asymptotic complexity $O(n+m)$. So the total asymptotic complexity will be:

$$
\begin{aligned}
&O(n^2) + O(nm) + O(n) + O(m^2) + mO(n+m) = \\
=&O(n^2 + nm + m^2) + O(nm + m^2) = \\
=&O(n^2 + 2nm + 2m^2) = \\
=&O\left((n+m)^2\right),
\end{aligned} \tag{3.71}
$$

in the case the full caching is being used (otherwise it will be even lesser). It is evident that this procedure has the smaller asymptotic complexity than the preparation of the distance cache of m different neurons.

3.2.10 Path construction

One of the fundamental operations of SOM network methods to solve TSP is to construct the path through all guards from the actual position of the neurons (some of them are close to guards, and some of them are not). First, the method of the SOM networks will be modified to be used in the non-Euclidean domain (`construct_path`), and then two new methods will be showed (`construct_path_alt` and `construct_path_alt_rand`).

The procedure used in the Euclidean domain has been described in the sections 2.1 and 2.2. In the non-Euclidean domain, the same procedure can be used with the exception that the square of distances will be minimised instead of the distances – it is better to cope with possibly negative d^2 (see section 3.2.7), and it is even faster (the `sqrt` function needs not to be called). Because this method consists of the finding the minimal distance among m neurons repeatedly for n guards and because one distance calculation has the asymptotic complexity $O(1)$ (see Table 3.2), the `construct_path` procedure has the asymptotic complexity $O(nm)$.

However, in our representation, there are two other methods to construct a tour. The first of these methods (`construct_path_alt`) is to find the maximum of each row of the matrix P (its columns are made up of the vectors $\boldsymbol{p_i}$). It means to find and select the neuron N_i with the maximal value $p_{i,k}$ for every guard G_k. Then save the index of the guard into the selected neuron. Repeat this for every guard. The rest of the procedure is the same as in the Euclidean domain (i.e. the order of the neurons in the string determines the order of guards in the path). The `construct_path_alt` has the asymptotic complexity $O(nm)$ (the matrix P has dimensions $n \times m$).

To see the principle behind the second new method, consider $0 \leq p_{i,l} \leq 1$ holds through the entire algorithm run (for every cell of the matrix P). Moreover, the sum of every column of the matrix P is equal to one through the algorithm run. (The previous statement can be proven using the fact, that at the beginning, the neurons are placed to the centroid ($\forall i \forall l : p_{i,l} = 1/n$), and after it, they are moved the way specified in section 3.2.3. Last but not least, the computation is numerically stable – see section 3.2.8.) Therefore the columns of the matrix P look similar to a probabilistic distribution.

When some neuron (e.g. N_i) is the winning neuron for some guard (e.g. G_k), it is very close to G_k in most cases (at least in the late stage of the algorithm run), thus $p_{i,k}$ is equal to one approximately, and the rest of the vector $\boldsymbol{p_i}$ has the elements nearly zeroed. Another neuron, which is close to another guard, has approximately zero in the k-th row, and so on. So, it makes sense to see other probability distribution in the rows of the matrix P (of course, the normalization of the rows must be done first). Look at examples in (3.72): at the beginning (P_1), nothing is known, while at the end (P_2), the order of the guards is known exactly: (G_2, G_3, G_1, ...)

$$\mathsf{P}_1 = \begin{pmatrix} 1/n & 1/n & 1/n & \dots \\ 1/n & 1/n & 1/n & \dots \\ 1/n & 1/n & 1/n & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \quad \mathsf{P}_2 = \begin{pmatrix} 0 & 0 & 1 & \dots \\ 1 & 0 & 0 & \dots \\ 0 & 1 & 0 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \tag{3.72}$$

See other examples in (3.73): more than one possible neurons ($\mathsf{N}_2, \mathsf{N}_3$) can be selected for the guard G_3 – the third row of the matrix P_3, but the order of the guards is still known exactly (G_2, G_3, G_1, ...). For P_4 there is uncertainty in the order: $(\mathsf{G}_2, \mathsf{G}_3, \mathsf{G}_1, \ldots)$ versus $(\mathsf{G}_2, \mathsf{G}_1, \mathsf{G}_3, \ldots)$.

$$\mathbf{P}_3 = \begin{pmatrix} 0 & 0 & 0 & 1 & \ldots \\ 1 & 0 & 0 & 0 & \ldots \\ 0 & 1 & 1 & 0 & \ldots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \quad \mathbf{P}_4 = \begin{pmatrix} 0 & 0 & 1 & 0 & \ldots \\ 1 & 0 & 0 & 0 & \ldots \\ 0 & 1 & 0 & 1 & \ldots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \tag{3.73}$$

The matrices $\mathbf{P}_2,\ldots,\mathbf{P}_4$ are theoretical examples only. In practice, the value of their cells will be somewhere between zero and one.

To summarize the previous paragraphs, the second new method to construct a tour (`construct_path_alt_rand`) is to normalize the sum of every row of the matrix $\mathbf{P}$ to one. Then select a random neuron using the first row as the probabilistic distribution for the guard G_1 and save the index of the guard into the selected neuron. Repeat this for every guard. The rest of the procedure is the same as in the Euclidean domain.

The method `construct_path_alt_rand` has an advantage over the method `construct_path_alt` when the order of the guards is not exactly determined, see the example matrix $\mathbf{P}_4$ above. If the method `construct_path_alt_rand` is called multiple times with the same (or similar) matrix $\mathbf{P}$ (e.g. it is called in the current iteration with some matrix $\mathbf{P}$, and it was also called in the previous iterations with the similar values in the matrix $\mathbf{P}$), it returns paths with different order of the guards probably. All these paths join the competition for the final shortest route – see section 3.2.12. On the contrary, the procedure `construct_path_alt` always returns the same route when it is repeatedly called with the same matrix $\mathbf{P}$. However, the `construct_path_alt` method has these advantages: it is more simple, faster, and in the late stage of the algorithm run, it has fewer situations with more than one guard with the same selected neuron. Finally, the benefits of the `construct_path_alt` procedure have shown to outweigh advantages of the `construct_path_alt_rand` procedure (the situation when the order of the guards is not exactly determined as in the $\mathbf{P}_4$ example is not so often, especially in the late stages) thus only `construct_path` and `construct_path_alt` methods are used in the NESOM algorithm.

3.2.11 Path optimization by swapping

The route constructed by one of the previously described methods (`FastTSP`, `construct_path`, `construct_path_alt` or `construct_path_alt_rand`) can contain some imperfections. To get rid of some of these defects, two optimization methods (`do_swap1` and `do_swap2`) are introduced. At the end of this section, the overall optimization procedure is described.

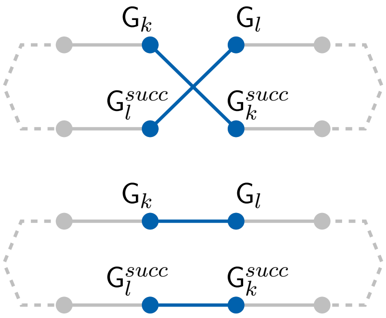


Figure 3.4: Simple swap. The upper half is the route before the swap, the lower half after the swap.

The first optimization method (`do_swap1`) uses simple swaps to eliminate the intersections in the route. For the pseudocode see Alg. 10. The method goes through all pairs of edges in the route (lines 5–8). Inside this loop, selected two edges form the swap (see Fig. 3.4), the path length difference of this swap is calculated (line 9), and the swap with the minimal difference is found (lines 10–14). If this swap shortens the route (i.e. the difference is less than zero, line 17), the swap is realized (lines 18–21).

Implementation note for comparing of the difference between the path length before the swap and the path length after the swap with zero (see Alg. 10, line 17): when zero is used as the threshold, there will appear an infinite loop of swaps there and back on some platforms. This problem is caused by rounding imprecision of `double` (`float`) type. It is necessary to use the threshold slightly smaller than zero (e.g. -10^{-5}).

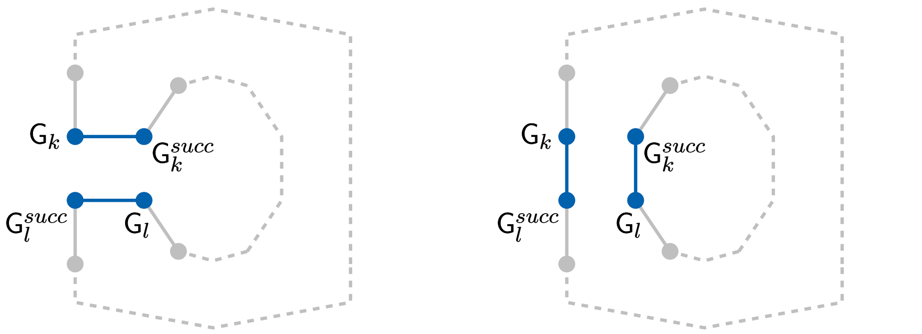
While the first swap method uses only one swap, the second method (`do_swap2`) utilizes a pair of swaps to try to reconnect part of the path to shorten it (see Fig. 3.5b). Notice that the swap used is of a different type than the swap in `do_swap1`. This swap disconnects the route into two independent cycles (see Fig. 3.5a), whereas the one in `do_swap1` does not. For the pseudocode see Alg. 11. In the first part of the algorithm, the list of all appropriate swaps is created (Alg. 11, lines 2–12). Note that the method demands the cardinal distance between edges to be at least two (line 4), and that even some swaps with positive route length difference (which would lengthen the path) are stored (line 8). Next, the list of swaps is sorted from the lowest length difference to the highest one (line 13).

After it, the procedure goes though all pairs of swaps (Alg. 11, lines 16 and 18). Swaps in the pair are tested to have no common edges (i.e. $\mathsf{G}_a \neq \mathsf{G}_c \wedge \mathsf{G}_a \neq \mathsf{G}_d \wedge \mathsf{G}_b \neq \mathsf{G}_c \wedge \mathsf{G}_b \neq \mathsf{G}_d$) (line 21) – any common edge would complicate further steps. Then, the right order of

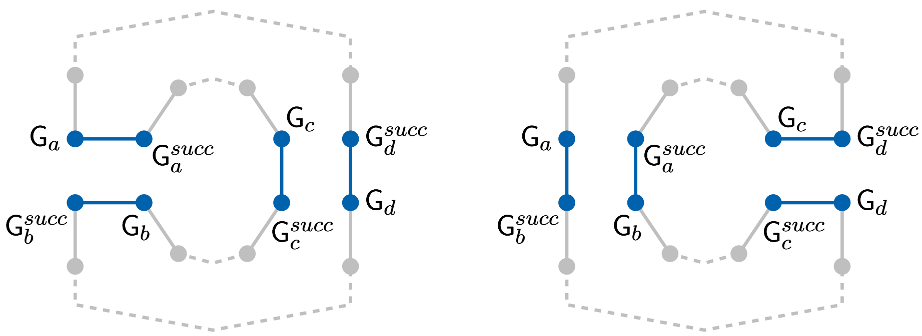
Algorithm 10: Do simple swap (`do_swap1`)

Input: Path through all guards (*path*) as a sequence of indices of individual guards
Input: Matrix of distances between guards **E**
Output: Boolean value indicating whether a swap was done
Output: Possibly modified *path*

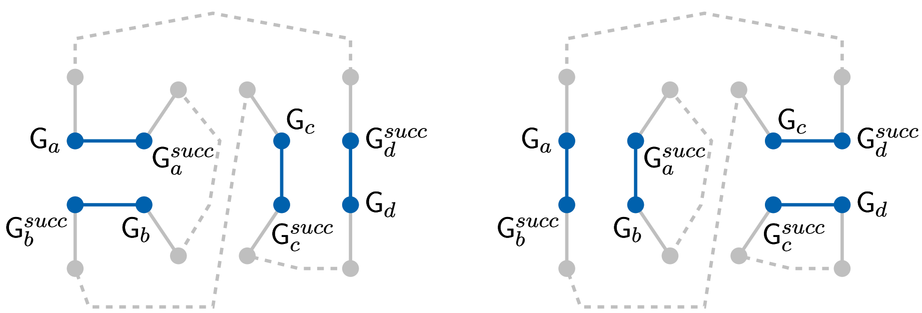
$was\_modified \leftarrow false$
$min\_diff \leftarrow 1$
$min\_k \leftarrow 0$
$min\_l \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow i + 1$ **to** n **do**
    $k \leftarrow path[i]$ `// index`
    $l \leftarrow path[j]$ `// index`
    $diff \leftarrow$ difference of the route length after changing of $\mathsf{G}_k - \mathsf{G}_k^{succ}$, $\mathsf{G}_l - \mathsf{G}_l^{succ}$ edges to $\mathsf{G}_k - \mathsf{G}_l$, $\mathsf{G}_k^{succ} - \mathsf{G}_l^{succ}$ edges – see Fig. 3.4
    if $diff < min\_diff$ **then**
      $min\_diff \leftarrow diff$
      $min\_k \leftarrow k$
      $min\_l \leftarrow l$
    end
  end
end
if $min\_diff < -10^{-5}$ **then**
  `// see implementation note in text on page 37`
  $was\_modified \leftarrow true$
  $k \leftarrow min\_k$
  $l \leftarrow min\_l$
  do swap ($\mathsf{G}_k - \mathsf{G}_k^{succ}$, $\mathsf{G}_l - \mathsf{G}_l^{succ}$ edges to $\mathsf{G}_k - \mathsf{G}_l$, $\mathsf{G}_k^{succ} - \mathsf{G}_l^{succ}$ edges, see Fig. 3.4) in the *path*
end
return *was_modified*, *path*



(a) : One half of reconnection (one swap). The left half is the route before the swap, the right half after the swap.



(b) : Reconnection – the correct case. The left half is the route before the reconnection, the right half after the reconnection.



(c) : Reconnection – the wrong case. The left half is the route before the reconnection, the right half after the reconnection.

Figure 3.5: Reconnection

Algorithm 11: Do reconnection (`do_swap2`)

Input: Path through all guards (*path*) as a sequence of indices of individual guards
Input: Matrix of distances between guards **E**
Output: Boolean value indicating whether a reconnection was done
Output: Possibly modified *path*

```
was_modified ← false
swaps ← ()                                                    // empty vector
for i ← 1 to n do
    for j ← i + 3 to min(n, n − 3 + i) do
        k ← path[i]                                                  // index
        l ← path[j]                                                  // index
        diff ← difference of the route length after changing of G_k − G_k^succ,
         G_l − G_l^succ edges to G_k − G_l^succ, G_l − G_k^succ edges – see Fig. 3.5a
        if diff < SWAP2_THRESH then                    // SWAP2_THRESH = 0.1
            append triplet (diff, k, l) to the end of swaps
        end
    end
end
sort swaps by the first element of triplet (diff) from the lowest one

min_diff ← 0,   min_i ← 0,   min_j ← 0
for i ← 1 to swaps.size do
    if 2 · swaps[i].diff ≥ min_diff then break
    for j ← i + 1 to swaps.size do
        diff ← swaps[i].diff + swaps[j].diff          // diff for both swaps
        if diff ≥ min_diff then break
        if swaps[i].k = swaps[j].k ∨ swaps[i].k = swaps[j].l ∨ swaps[i].l
         = swaps[j].k ∨ swaps[i].l = swaps[j].l then continue
        between_1 ← (swaps[i].k < swaps[j].k < swaps[i].l)
        between_2 ← (swaps[i].k < swaps[j].l < swaps[i].l)
        if between_1 ≠ between_2 ∧ diff < min_diff then
            min_diff ← diff
            min_i ← i
            min_j ← j
        end
    end
end
if min_diff < −10^−5 then
                         // see implementation note in text on page 37
    was_modified ← true
    do reconnection according to swaps[min_i] and swaps[min_j] (see
     Fig.3.5b)
end
return was_modified, path
```

the guards $\mathsf{G}_a, \ldots, \mathsf{G}_d$ in the route is verified to distinguish the correct case from the wrong case (lines 22–24). The correct case has one swap locked into the other one, so the order of the guards could be $\mathsf{G}_a, \mathsf{G}_c, \mathsf{G}_b, \mathsf{G}_d$ for example. See Fig. 3.5b. The wrong case has one complete swap (both of its edges) first and the other swap after it. The order of guards could be $\mathsf{G}_a, \mathsf{G}_b, \mathsf{G}_c, \mathsf{G}_d$ for example. The reconnection in the wrong case would disconnect the path into three independent cycles, see Fig. 3.5c. The previous steps are repeated for all pairs and the pair with the minimal sum of the differences (line 19) is found (lines 24–28). Thanks to the fact that the list of swaps is sorted from the lowest difference to the highest one, the cycles can be broken prematurely, when it is clear the present candidate for minimum will not be changed (lines 17 and 19). At the end, if the pair of swaps with the minimal sum of the differences shortens the route (i.e. the sum is less than zero, line 31), the reconnection is realized (Alg. 11, lines 32–33).

Algorithm 12: Swap optimization algorithm (`do_swaps`)

Input: Path through all guards as a sequence of indices of individual guards
Output: Boolean value indicating whether a swap or a reconnection was done
Output: Possibly modified *path*

```
1 was_modified ← false
2 while do_swap1(path) do  was_modified ← true
3 while do_swap2(path) do
4 |   was_modified ← true
5 |   while do_swap1(path) do
6 |   end
7 end
8 return was_modified, path
```

The overall route optimization procedure (`do_swaps`) using `do_swap1` and `do_swap2` methods (for the pseudocode see Alg. 12) removes all intersections in the route first (Alg. 12, line 2)). Then it repeatedly tries to find and realize a reconnection, that can shorten the path (lines 3–7). If any intersection emerges during the loop, it is removed immediately (line 5).

3.2.12 Basic SOM in non-Euclidean domain (NE-Basic SOM)

The fundamental operations of the SOM network in the non-Euclidean domain have been described in the previous sections. Now, the overall non-Euclidean Basic SOM algorithm can be introduced. It is based on the Basic SOM replacing the Euclidean representation and operations by their non-Euclidean analogies and adding several more advanced techniques (some of them inspired by the CAN).

For the pseudocode see Alg. 13. At the beginning, the input distances are scaled(divided) so that the maximal distance in the distance matrix is $\sqrt{2}$ (line 1). Scaling is generally good practice because it eases tuning of the algorithm parameters to various input data. The value $\sqrt{2}$ is inspired by the CAN that scales input data to fit the unit square and thus it allows the maximal distance of $\sqrt{2}$ after scaling. Next step is the initialization (lines 2–8). Whether to perform `centroid_init` or `FastTSP_init` (Alg. 13, line 4) depends on user's decision.

The main part of the algorithm is the loop of the iterative process (Alg. 13, lines 9–32). The permuting of the guards (lines 12 and 14), the neuron inhibiting (lines 11 and 17) and search for the winning neuron (line 15) is the same as in the Euclidean Basic SOM (see section 2.1) with the exception that the winning neuron is the neuron with the smallest square of the distance, not the distance itself (the square of the distance can have negative value in some cases, trying to calculate the square root of it would result in problems – see section 3.2.7). Movement of the neurons (Alg. 13, line 18) will be described later (Alg. 14).

Depending on user's decision (*cfg_running_findpath* and *cfg_running_findpath_alt*), the path can be constructed at the end of the algorithm run only (lines 33–34) as in the Basic SOM or during the iterative process as in the CAN (lines 20–27). Because in the earlier stages of the iterative process the neuron positions do not produce quality paths the threshold condition ($iteration\_count > 50$) must be met before the algorithm tries to produce a path (the purpose is to speed up the algorithm). Path construction itself has been described in section 3.2.10. Because the path construction has the asymptotic complexity $O(nm)$ (see section 3.2.10) which is less than the asymptotic complexity of the neuron movements $O(n^2m)$ (see the following text) and because the shortest path competition during the iterative process avoids the loss of quality solutions it shows to set $cfg\_running\_findpath = cfg\_running\_findpath\_alt = true$ to be a good practice.

The error calculation (Alg. 13, lines 10 and 16) is similar to the Euclidean Basic SOM (section 2.1). Because the square of the distance can be negative (see section 3.2.7), we maximize the square of the distance instead of the distance itself (and the initial value is not zero, but it is $-\infty$). The main iterative loop termination condition is modified correspondingly (line 28).

Algorithm 13: Non-Euclidean Basic SOM

Input: Matrix of distances between guards **E**
Output: Solution of TSP

scale distances so that maximum of cell values of the matrix **E** is $\sqrt{2}$
compute squares of the distances (the matrix **D**)
$m \leftarrow 3n$ `// number of neurons = 3*number of guards`
initialize neuron positions (`centroid_init` or `FastTSP_init`) `// see sect.3.2.9`
clear cache of small postponed moves ($\forall j\ \forall l : c_{jl} \leftarrow 0$) `// see Alg. 14`
$iteration\_count \leftarrow 0$
$path\_best \leftarrow (\infty)$ `// path with infinite length`
$path\_best\_len\_before\_swaps \leftarrow \infty$ `// see Alg. 15`
while *true* **do**
    $error\_sq \leftarrow -\infty$
    $inhibited \leftarrow \emptyset$ `// empty set of inhibited neurons`
    $permutation \leftarrow$ random permutation of sequence $(1, \dots, n)$
    for $k \leftarrow 1$ **to** n **do**
        $l \leftarrow permutation[k]$
        for G_l find winning not inhibited neuron N_i
        $error\_sq \leftarrow \max(error\_sq, distance\_squared(\mathsf{G}_l, \mathsf{N}_i))$
        $inhibited \leftarrow inhibited \cup \{i\}$
        `move_neurons` $(\mathsf{N}_i, \mathsf{G}_l)$ `// see Alg. 14`
    end
    if $cfg\_running\_findpath \wedge (iteration\_count > 50)$ **then**
        $path\_temp \leftarrow$ `construct_path()` `// see sect.3.2.10`
        `process_path` $(path\_temp)$ `// see Alg. 15`
    end
    if $cfg\_running\_findpath\_alt \wedge (iteration\_count > 50)$ **then**
        $path\_temp \leftarrow$ `construct_path_alt()` `// see sect.3.2.10`
        `process_path` $(path\_temp)$ `// see Alg. 15`
    end
    if $(error\_sq \leq max\_error^2) \wedge (error\_sq \geq 0)$ **then break**
    if $iteration\_count \geq max\_iterations$ **then break**
    $iteration\_count \leftarrow iteration\_count + 1$
    update parameters: $G \leftarrow G(1 - \alpha)$
end
$path\_temp \leftarrow$ `construct_path()` `// see sect.3.2.10`
`process_path` $(path\_temp)$ `// see Alg. 15`
if cfg_final_swaps **then** `do_swaps`$(path\_best)$ `// see sect.3.2.11`
return $path\_best$

Updating the gain parameter G (Alg. 13, line 31) is the same as in the Euclidean Basic SOM. Depending on user's decision (*cfg_final_swaps*), the final swap optimization by `do_swaps` (see Alg. 12) is possibly performed (Alg. 13, line 35).

Algorithm 14: Non-Euclidean Basic SOM subroutine `move_neurons`

Input: Winning neuron N_i
Input: Guard G_l
1 **for** $d\_cardinal \leftarrow -d^*$ **to** d^* **do** `// go through neighbourhood`
2 $\quad j \leftarrow i - d\_cardinal$ `// index of neuron from neighbourhood`
3 $\quad$ **if** $j < 1$ **then** $j \leftarrow j + m$
4 $\quad$ **if** $j > m$ **then** $j \leftarrow j - m$
5 $\quad$ **if** $f_j < 10^{-30}$ **then** normalize neuron j `// see eq.(3.64)`
6 $\quad \gamma \leftarrow \mu \exp(-d\_cardinal^{\,2}/G^2)$
7 $\quad$ **if** $\gamma <$ `IGNORE_MICROMOVE_THRESH` **then**
8 $\quad\quad$ **continue** `// IGNORE_MICROMOVE_THRESH = 1e-6`
9 $\quad$ **end**
10 $\quad \gamma \leftarrow \gamma + c_{jl}$ `// add postponed moves from the past`
11 $\quad$ **if** $\gamma <$ `SMALLMOVE_THRESH` **then** `// SMALLMOVE_THRESH = 1e-2`
12 $\quad\quad c_{jl} \leftarrow \gamma$ `// postpone move`
13 $\quad\quad$ **continue**
14 $\quad$ **end**
15 $\quad c_{jl} \leftarrow 0$ `// clear postponed moves`
16 $\quad$ move neuron N_j towards guard G_l by γ `// see sect.3.2.5`
17 $\quad$ update cache accordingly `// see sect.3.2.6`
18 **end**
19 **return**

The movement of the winning neuron and its neighbourhood towards the guard (see Alg. 14) is similar to the Euclidean Basic SOM (section 2.1): it goes through the neighbourhood (Alg. 14, lines 1–4), it calculates γ (line 6, compare with the equation (2.1)) and it moves the individual neurons (lines 16–17). There are two differences. The first one is that the normalization of the neuron is performed when the threshold condition ($f_i < 10^{-30}$) is met (line 5, see the equation (3.64)). The second difference is based on the fact, that the movement of the neurons is the most computationally demanding place in the algorithm: the move of one neuron has asymptotic complexity $O(n)$ (see Table 3.2, the last row). If it is called for all neurons from the neighbourhood (its size is $2d^* + 1 = 0.4m$), the asymptotic complexity of `move_neurons` subroutine is $O(nm)$ and the complexity of one adaptation step of the main iterative loop (Alg. 13, lines 13–19) is $O(n^2m)$. For lowering the computational difficulty, the moves with γ smaller than `IGNORE_MICROMOVE_THRESH` threshold are ignored completely (Alg. 14, lines 7–9). The moves with γ greater than `IGNORE_MICROMOVE_THRESH` but smaller than `SMALLMOVE_THRESH` threshold are postponed, and their γ are summed up in the variable c_{jl} (for the movement of the neuron N_j towards the guard G_l). When the sum exceeds

`SMALLMOVE_THRESH` threshold, the move is performed, and the c_{jl} variable is cleared (Alg. 14, lines 10–15). (Further speedup is achieved by caching the values of the $\mu \exp(-d\_cardinal\ ^2/G^2)$ function (line 6)).

Algorithm 15: Non-Euclidean Basic SOM subroutine process_path

```
   Input: path_temp
1  was_swaps ← false
2  path_temp_len_before_swaps ← ||path_temp||
           // save length of path_temp without swap optimization
3  if cfg_running_swaps_always ∨ (cfg_running_swaps ∧
   (path_temp_len_before_swaps < path_best_len_before_swaps)) then
4  |  was_swaps ← do_swaps (path_temp)          // see sect.3.2.11
5  end
6  if ||path_temp|| < ||path_best|| then
7  |  path_best ← path_temp
8  |  path_best_len_before_swaps ← path_temp_len_before_swaps
9  |  if was_swaps ∧ cfg_reinitialize_neurons then
10 |  |  reinitialize_neurons (path_temp)
   |  |                              // run Alg. 9 from line 3
11 |  end
12 end
13 return
```

The `process_path` subroutine (Alg. 15) consists of three main steps. The first one is the swap optimization of just created path (*path_temp*) (Alg. 15, lines 3–4). Depending on user's decision (*cfg_running_swaps* and *cfg_running_swaps_always*), the swap optimization could be done always, never or when the constructed path (*path_temp*) without swap optimization is shorter than the actual best path found (without swap optimization too). The second step is competition for the shortest final route (lines 6–8).

Consider the situation that the neural string (its projection to the original polygonal space respectively) has some self-intersection and the path constructed from the neuron positions has an intersection with itself too. Swap optimization removes the intersections from the path only, not from the string. In the next iteration, the neural string produces a new path with intersections again and so on. To transfer benefits of the path swap optimization to the neural string the last step of `process_path` subroutine was added (Alg. 15, lines 9–11). It reinitializes the neuron positions when enabled ($cfg\_reinitialize\_neurons = true$) and some swaps were done during the previous swap optimization process (of the new shortest path competition winner). To reinitialize neuron positions according to some path (`reinitialize_neurons`(*path*)) means to run Alg. 9 from line 3. It places the neural string so that it copies the specified path (compare with the

Parameter	**Value**
Initial value of gain G	40
Learning rate μ	0.6
Neighbourhood size d^*	$0.2m$
Gain change parameter α	0.03
Termination threshold *max_error*	10^{-3}
Maximum number of iterations *max_iterations*	180
Threshold of ignored moves `IGNORE_MICROMOVE_THRESH`	10^{-6}
Threshold of postponed moves `SMALLMOVE_THRESH`	0.01
Normalization threshold	10^{-30}
Threshold of stored swaps `SWAP2_THRESH` (`do_swap2` subroutine)	0.1
`centroid_init` first iteration gain $G_{\text{c-init}}$	500
`centroid_init` first iteration learning rate $\mu_{\text{c-init}}$	0.04

Table 3.3: NE-Basic SOM – parameters and proposed values

`FastTSP_init` method in section 3.2.9). The optimized path is transformed back to the neuron positions and the neural string shape this way.

During the development of the NE-Basic SOM algorithm, the proposed values of parameters of the algorithm were set – see Table 3.3. Some of parameter values were taken from the Basic SOM setting (see Table 2.1), some of them were developed using a large number of numerical tests.

Chapter 4

Experiments

The general arrangement of numerical experiments will be described in section 4.1. The specific settings of experiments and their results will be described in section 4.2 for the Glimmer algorithm, in section 4.3 for the NE-Basic SOM algorithm, and in section 4.4 for other algorithms. The overall comparison of selected algorithms will be discussed in section 4.5.

4.1 Implementation notes

The algorithms from previous chapters are implemented in C++. The source code is based on previous work of Roman Sushkov [11], Miroslav Kulich, Jan Faigl, Stephen Ingram [8] and others. The Concorde library [3] (version 03.12.19) is used to perform Chained Lin–Kernighan heuristic for purposes of evaluation and comparison (also referred to as L.K. in the following text). The Visrottree library developed in Intelligent and Mobile Robotics Group, Czech Technical University in Prague, is used to compute the graph of visibility.

The experiments were done on two different computational platforms:

- the local computer equipped with Intel(R) Core(TM) i7–6700HQ CPU running at 2.6GHz, 16GB DRAM (with speed 21GB/s), 32kB of L1 cache (169GB/s), 256kB of L2 cache (73GB/s) and 6MB of L3 cache (47GB/s) (speed measured by Memtest86 5.01). Operating system was Ubuntu Linux 16.04, kernel 4.10.0. The source code was compiled by G++

Map name	Number of guards n	Min. path length from 1000x L.K.	Min. path length from article [5]	Used as L_{opt} for PDM, PDB
map	17	2653	2650	2650
dense	53	17912	17910	17910
potholes	68	15455	15450	15450
jh2	80	20194	20190	20190
pb4	104	65459	65460	65459
ta_2	141	32801	32800	32800
h2_5	168	94595	94300	94300
jari	200	30476	–	30476
potholes	200	22525	–	22525
var_density	200	22330	–	22330
potholes	282	27734	27730	27730
pb_15	415	84184	83960	83960
h2_2	568	132943	131620	131620
ta_1	574	54348	54110	54110

Table 4.1: Maps and “optimal” path length

version 5.4 with the “-O3 -ffast-math -march=native” compiler settings. The experiments were done on one core of the CPU. Total consumed CPU time is: 163 hours, 254,000 runs.

- The cluster for parallel computing of the National Grid Infrastructure MetaCentrum (*https://www.metacentrum.cz*). Because it is heterogeneous platform, the results are comparable in terms of quality but not speed (the speed of individual computational nodes differs). The source code was compiled by G++ version 4.9.2 with the “-O3 -ffast-math -march=native” compiler settings. Total consumed CPU time is: 426 days, 4,259,000 runs.

Several working environments were employed as TSP tasks to measure quality and speed of individual algorithms and their parameter settings. To allow comparison with [5] and [11], the same environments were tested – see Table 4.1: the name of polygonal map and the number of guards are stated in the first two columns. The shortest route length obtained by the Concorde library (using L.K. heuristic) is shown in the third column. The procedure was ran 1000 times for every environment. The analogous value that was collected in [5] is in the fourth column. The smaller value from the third and the fourth column is stated in the last column and it is used as “optimal” path length L_{opt} for PDM and PDB calculations – see (4.1) and (4.2).

The quality of results is reported as the percentage deviation of the mean solution length to the optimal path length (PDM) and as the deviation of

Map	n	ΔPDM							
		l_∞ **6**	l_∞ **10**	l_2 **6**	l_2 **10**	l_3 **6**	l_3 **10**	l_8 **6**	l_8 **10**
map	17	-1.9	-0.8	0.0	0.0	1,7	0,8	17,7	7,2
dense	53	-3.1	-3.0	0.0	0.0	3,1	0,0	173,2	37,3
potholes	68	-1.0	-0.9	0.0	0.0	5,7	1,4	238,6	57,1
jh2	80	-0.5	-1.2	-0.1	0.0	8,4	0,7	280,0	75,8
pb4	104	-0.4	-0.6	0.0	0.0	3,7	1,5	322,8	101,1
ta_2	141	-0.9	-1.9	-0.3	0.0	13,6	5,7	309,4	152,5
h2_5	168	1.0	-0.2	0.2	0.0	17,8	4,2	264,1	126,9
jari	200	-0.6	-1.8	-0.2	0.0	21,7	3,8	309,4	167,4
potholes	200	-1.1	-0.9	0.0	0.0	19,5	2,7	309,0	176,4
var_density	200	-1.4	-0.9	0.0	0.0	20,8	2,7	354,5	190,3
potholes	282	-1.2	-1.0	0.0	0.0	30,1	3,9	390,6	247,5
pb_15	415	-2.8	-3.1	0.0	-0.1	26,3	6,2	791,8	517,7
h2_2	568	-0.8	-3.3	0.0	0.2	53,4	10,3	569,7	487,6
ta_1	574	-1.6	-1.7	0.0	-0.1	74,1	14,1	590,5	540,1
min		-3.1	-3.3	-0.3	-0.1	1,7	0,0	17,7	7,2
max		1.0	-0.2	0.2	0.2	74,1	14,1	791,8	540,1

Table 4.2: Glimmer: $\boldsymbol{w}_{spring}$ versus $\boldsymbol{w}_{spring\text{-}alt}$ direction vector usage. Stated ΔPDM = PDM($\boldsymbol{w}_{spring\text{-}alt}$ used) − PDM($\boldsymbol{w}_{spring}$ used) for l_∞, l_2, l_3 and l_8 norms and for number of dimensions $\omega = 6$ and 10.

the best solution length to the optimal path length (PDB):

$$\text{PDM} = 100(L_{\text{mean}} - L_{\text{opt}})/L_{\text{opt}}, \quad (4.1)$$
$$\text{PDB} = 100(L_{\text{best}} - L_{\text{opt}})/L_{\text{opt}}. \quad (4.2)$$

To express the difference between the results of two different algorithms on the same environment, the $\Delta\text{PDM} = \text{PDM}_2 - \text{PDM}_1$ value is used. The ratio of the CPU time consumed $\lambda = t_2/t_1$ is used to evaluate how many times one algorithm is faster than the other one.

4.2 Tests of Glimmer algorithm

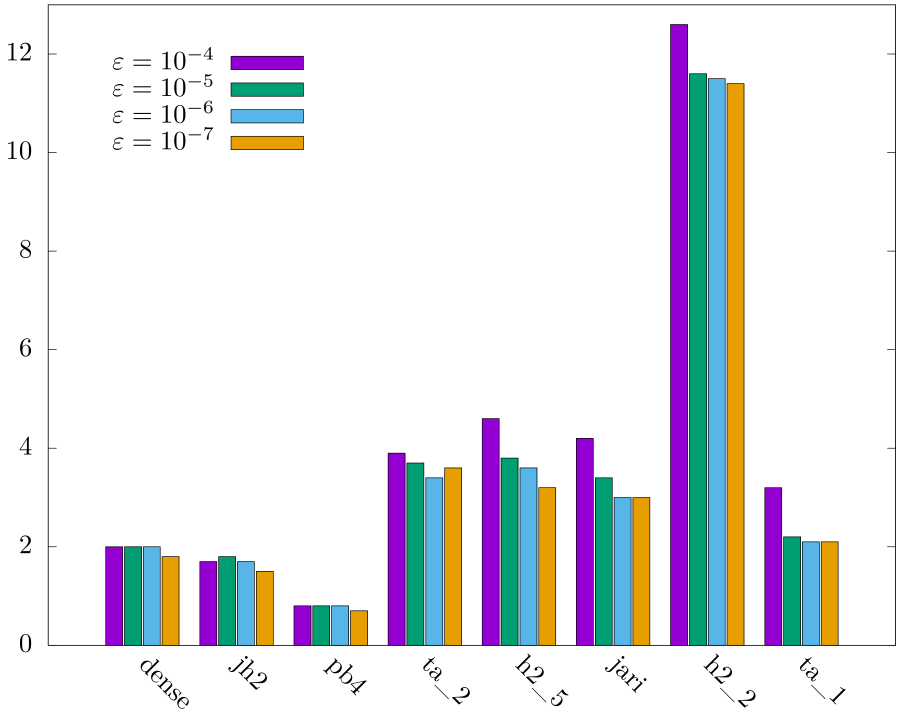
The purpose of the first test is to decide when use the $\boldsymbol{w}_{spring}$ direction vector and when use the $\boldsymbol{w}_{spring\text{-}alt}$ direction vector – for these two variants of the Glimmer algorithm modification see section 3.1.2. To eliminate any influence of possibly different behaviour of the sparse stress function with different l_p norms that would affect the number of iterations, the algorithm was modified to run exactly 3000 iterations in every step of the Glimmer v-cycle. The parameters of the Glimmer algorithm were set as in Table 3.1. The l_p norm was sequentially set to l_∞, l_2, l_3 and l_8. For each of these settings, the number of dimensions ω was set to 6 and 10. Both variants of the algorithm, i.e. using $\boldsymbol{w}_{spring}$ or $\boldsymbol{w}_{spring\text{-}alt}$ respectively, were run on all environments (Table 4.1) 30 times.

The quality of output of the MDS algorithm Glimmer is evaluated this way: the output of Glimmer is passed as an input to the Concorde library. The length of the path found is measured using the original undistorted distances $\mathbf{E}$. The previous steps are performed for each of the 30 runs of the test run and the L_{mean} value is calculated. Finally the PDM is computed from the equation (4.1).

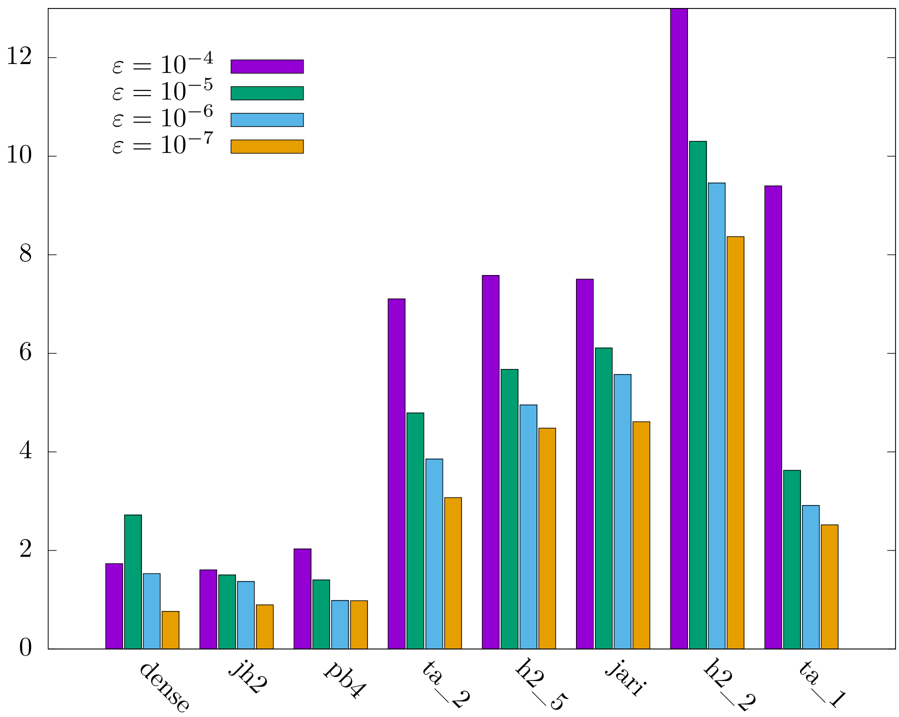
The differences between PDM when $\boldsymbol{w}_{spring\text{-}alt}$ vector was used and PDM when $\boldsymbol{w}_{spring}$ was used for various combinations of the l_p norm and ω settings are stated in Table 4.2. From the last columns follows, that for the l_3 and the l_8 norms it is much better to use the $\boldsymbol{w}_{spring}$ algorithm variant whether for $\omega = 10$ or $\omega = 6$. For the l_∞ norm, the results are opposite – it is better to use $\boldsymbol{w}_{spring\text{-}alt}$ variant. For the l_2 norm, there is no difference between $\boldsymbol{w}_{spring}$ and $\boldsymbol{w}_{spring\text{-}alt}$ variants which is in the agreement with the theoretical results (see section 3.1.2). Therefore, the direction vector $\boldsymbol{w}_{spring}$ will be used with the l_3 and the l_8 norms and the vector $\boldsymbol{w}_{spring\text{-}alt}$ with the l_∞ norm in all following tests.

The purpose of the second test is the comparison of speed and quality of the Glimmer algorithm for different l_p norms. To eliminate any influence of possibly different behaviour of the sparse stress function with different l_p norms that would affect the number of iterations, the algorithm was modified to run exactly 400 iterations in every step of the Glimmer v-cycle. The parameters of the Glimmer algorithm were set as in Table 3.1. The l_p norm was sequentially set to l_2, l_3, l_8 and l_∞ and the number of dimensions ω was set to 6 and 10. The algorithm was run on all environments (Table 4.1) 30 times. The CPU time consumed was measured from the start of the Glimmer algorithm to the end – i.e. geodetic distances $\mathbf{E}$ and L.K. computations were excluded. The quality of output was evaluated by L.K. as in the previous test.

The differences between PDM(l_p used) and PDM(l_2 used) and the ratios of CPU time consumed t_p/t_2 for l_3, l_8 and l_∞ norms and $\omega = 10$ are stated in Table 4.3. The results for $\omega = 6$ are similar. The table shows that the Glimmer algorithm using l_3 or l_8 norm is approximately 5 times slower than the variant using l_2, while the l_∞ norm is about 10% slower than l_2. There are no benefits in engaging either l_3 or l_8 norms – the differences of PDM are close to zero. Therefore, the l_3 and l_8 will be excluded from the following tests.



(a) : PDM for l_2 norm, $\omega = 6$



(b) : PDM for l_∞ norm, $\omega = 6$

Figure 4.1: PDM for l_2 and l_∞ norm, $\omega = 6$, various ε, selected environments.

Map	n	ΔPDM l_3	t_p/t_2 l_3	ΔPDM l_8	t_p/t_2 l_8	ΔPDM l_∞	t_p/t_2 l_∞
map	17	-0.4	3.9	0.1	3.9	1.3	1.1
dense	53	0.0	4.9	1.1	4.9	-0.8	1.1
potholes	68	0.1	5.1	0.5	5.1	0.9	1.1
jh2	80	-0.2	5.2	-0.6	5.2	-0.7	1.1
pb4	104	0.0	4.7	0.2	4.6	0.1	1.1
ta_2	141	0.0	4.8	-0.3	4.8	-0.9	1.1
h2_5	168	-0.3	4.9	-0.5	4.9	1.9	1.1
jari	200	0.2	5.1	0.5	5.1	0.9	1.1
potholes	200	0.1	5.1	0.7	5.1	0.2	1.1
var_density	200	0.0	5.0	0.5	5.1	0.1	1.1
potholes	282	0.1	5.1	0.6	5.1	0.3	1.1
pb_15	415	0.2	5.3	0.7	5.3	-1.2	1.1
h2_2	568	-0.2	5.4	-1.6	5.3	-2.2	1.1
ta_1	574	0.1	5.4	0.4	5.3	-0.6	1.1
min		-0.4	3.9	-1.6	3.9	-2.2	1.1
max		0.2	5.4	1.1	5.3	1.9	1.1

Table 4.3: Glimmer: speed and quality of different l_p norms in comparison with l_2. Stated $\Delta\text{PDM} = \text{PDM}(l_p \text{ used}) - \text{PDM}(l_2 \text{ used})$ and ratios of CPU time consumed $\lambda = t_p/t_2$ for l_3, l_8 and l_∞ norms and for number of dimensions $\omega = 10$.

The purpose of the third test is to set appropriate value of the Glimmer algorithm termination threshold ε. The parameters of the Glimmer algorithm were set as in Table 3.1. The ε was sequentially set to 10^{-4}, 10^{-5}, 10^{-6} and 10^{-7}. The l_p norm was set to l_∞ and l_2 and the number of dimensions ω was set to 6 and 10. The algorithm was run on all environments (Table 4.1) 100 times. The quality of output was evaluated by L.K. as in the first test.

The resulting values of PDM for $\omega = 6$ for selected environments are shown at Fig. 4.1. The values for $\omega = 10$ are similar. It follows from the results, that the value proposed in [8] (10^{-4}) is too high for the purpose of solving the TSP task. The significant decrease of PDM is achieved by changing ε from 10^{-4} to 10^{-5}. For l_2 norm, further reduction of ε brings only a small improvement to the PDM. Thus appropriate value will be $\varepsilon = 10^{-5}$ when using l_2 norm. For l_∞ norm, the situation is more complicated: the setting of ε to 10^{-6} instead of 10^{-5} should be considered. But decreasing of the ε threshold increases the number of iterations and thus it slows down the whole algorithm. Moreover, other tests showed that changing ε to 10^{-6} improves the quality of solution only negligibly when the Glimmer MDS algorithm is used with Basic SOM or CAN. The ε threshold will be set to 10^{-5} for all further tests unless stated otherwise.

Map	n	PDM *false*	PDM *true*	ΔPDM	t *false*	t *true*	λ
map	17	0.1	0.1	0.0	0.001	0.001	0.989
dense	53	4.9	4.2	-0.7	0.013	0.013	0.979
potholes	68	2.5	2.5	-0.1	0.021	0.020	0.967
jh2	80	0.9	1.0	0.1	0.027	0.027	0.990
pb4	104	0.0	0.1	0.0	0.051	0.050	0.986
ta_2	141	2.0	1.7	-0.3	0.093	0.086	0.920
h2_5	168	0.3	0.3	0.0	0.179	0.142	0.797
jari	200	0.7	0.6	-0.1	0.204	0.169	0.829
potholes	200	4.1	3.4	-0.7	0.256	0.176	0.688
var_density	200	3.4	3.2	-0.3	0.178	0.156	0.873
potholes	282	4.2	3.6	-0.6	0.459	0.342	0.746
pb_15	415	0.8	0.9	0.1	1.545	0.944	0.611
h2_2	568	1.2	0.8	-0.3	2.739	1.608	0.587
ta_1	574	3.8	3.4	-0.4	3.227	1.650	0.511
min		0.0	0.1	-0.7	0.001	0.001	
max		4.9	4.2	0.1	3.227	1.650	

Table 4.4: NE-Basic SOM – neuron reinitialization test.
Stated PDM_{false}, PDM_{true}, $\Delta\text{PDM} = PDM_{true} - \text{PDM}_{false}$,
the CPU time consumed t_{false}, t_{true} and their ratio $\lambda = t_{true}/t_{false}$.

4.3 Tests of NE-Basic SOM algorithm

To decide whether to use the reinitialization of neuron positions (see section 3.2.12 and Alg. 15, line 9), i.e. wheter to set the value of parameter *cfg_reinitialize_neurons* to *true*, the numerical experiment was done. The NE-Basic SOM algorithm was used with the parameters set according to Table 3.3, with the initialization procedure `centroid_init` and with this configuration:
cfg_running_findpath = *true*
cfg_running_findpath_alt = *true*
cfg_running_swaps = *true*
cfg_running_swaps_always = *false*
cfg_final_swaps = *true*.
The parameter *cfg_reinitialize_neurons* was set to *false* and *true* respectively. The test was run on all environments (Table 4.1) 100 times. The CPU time consumed was measured from the start of the NE-Basic SOM algorithm to the end. The quality of output was evaluated by PDM.

The PDM_{false} values, the PDM_{true} values and their difference $\Delta\text{PDM} = \text{PDM}_{true} - \text{PDM}_{false}$ are stated in Table 4.4. The CPU time consumed t_{false}, t_{true} and their ratio $\lambda = t_{true}/t_{false}$ are stated too.

It follows from the fifth column of the table, that setting the parameter *cfg_reinitialize_neurons* to *true* will improve the quality of solutions only slightly. However, the main change is obvious from the last column – the algorithm runs faster. This speedup is negligible for n small, but the algorithm runs almost twice faster for n large. It confirms our assumption from section 3.2.12 that the benefits of swap optimization of path can be transferred to the neuron string by the procedure of neuron reinitialization and that this method can work in practice.

4.4 Other tests

Besides the tests already described, other test were done too. The test (l_2 and l_∞ used, $\omega = 6$ and 10, $\varepsilon = 10^{-6}$, the CAN termination threshold *max_error* sequentially increased from 10^{-10}, the other parameters set as in Tables 3.1 and 2.2, 100 runs on all environments with the Glimmer & CAN algorithms) showed that the appropriate value of *max_error* threshold is 10^{-3}. The progress of experiment was observed not only by the PDM of solutions, but mainly by monitoring the difference between the number of CAN iterations realized and the iteration number, when the winning path was discovered. This is made possible by the fact, that the CAN algorithm finds shortest solutions in the first part of iteration process only. For example, the winning path is found at the latest in the 155-th iteration even if the CAN runs 397 iterations total for `ta_1` environment, $\omega = 10$, l_2 norm. It fundamentally simplifies the process of *max_error* threshold setting. The *max_error* threshold will be set to 10^{-3} for all further tests.

Another test (l_2 and l_∞ used, $\omega = 6$ and 10, $\varepsilon = 10^{-5}$ and 10^{-6}, the CAN termination threshold *max_error* $= 10^{-3}$, the other parameters as in Tables 3.1, 2.1 and 2.2, 100 runs on all environments with the Glimmer & Basic SOM algorithms and the Glimmer & CAN algorithms, evaluated by PDM) showed that it is sufficient to set number of dimensions ω to 6. Changing ω to 10 brings very small decrease of PDM of solutions only. Moreover, it follows from the results that the Glimmer & Basic SOM combined algorithm is significantly outperformed with the combination of the Glimmer & CAN both in speed and quality.

4.5 Overall comparison

Based on the results of preliminary experiments, the following numerical tests were selected for the overall comparison of speed and solution quality of individual algorithms:

- the Glimmer & CAN combined algorithm test with the following settings: l_2 norm used, the number of dimensions $\omega = 6$, the Glimmer termination threshold $\varepsilon = 10^{-5}$, the CAN termination threshold $max\_error = 10^{-3}$, the other parameters set as in Tables 3.1 and 2.2.
- The Glimmer & CAN combined algorithm test using l_∞ norm. The rest of the settings as in the previous experiment.
- The SMACOF & CAN combined algorithm test with the following settings: the SMACOF as implemented in [11], the number of dimensions $\omega = 6$, the Glimmer termination threshold $\varepsilon = 10^{-5}$, the other parameters set as in Table 3.1.
- The NE-Basic SOM algorithm with the Gain change parameter α set to 0.06, the other parameters set as in Table 3.3, `centroid_init` used.
  $cfg\_running\_findpath = false$
  $cfg\_running\_findpath\_alt = false$
  $cfg\_running\_swaps = false$
  $cfg\_running\_swaps\_always = false$
  $cfg\_reinitialize\_neurons = false$
  $cfg\_final\_swaps = true$.
  This variant should be the fastest one among three selected NE-Basic SOM variants. It will be referred to as the Fast.
- The NE-Basic SOM algorithm with the Gain change parameter α set to 0.03, the other parameters set as in Table 3.3, `centroid_init` used.
  $cfg\_running\_findpath = true$
  $cfg\_running\_findpath\_alt = true$
  $cfg\_running\_swaps = false$
  $cfg\_running\_swaps\_always = false$
  $cfg\_reinitialize\_neurons = false$
  $cfg\_final\_swaps = false$.
  This variant does not use any swap optimization, therefore it will be referred to as the Pure variant.
- The NE-Basic SOM algorithm with the Gain change parameter α set to 0.03, the other parameters set as in Table 3.3, `centroid_init` used.
  $cfg\_running\_findpath = true$
  $cfg\_running\_findpath\_alt = true$
  $cfg\_running\_swaps = true$

$cfg\_running\_swaps\_always = false$
$cfg\_reinitialize\_neurons = true$
$cfg\_final\_swaps = true.$
This variant should give the best results among the three selected NE-Basic SOM variants. It will be referred to as the Best.

- The Chained Lin–Kernighan heuristic using the Concorde library was run for purposes of comparison.

Each test was run on all environments (Table 4.1) 100 times. The CPU time consumed was measured from the end of the initial computation of the geodetic distances **E** to the end of the whole algorithm. The quality of output was evaluated by PDM and PDB – see equations (4.1) and (4.2).

The results are stated in Table 4.5 (PDM values), 4.6 (PDB values) and 4.7 (the CPU time consumed). To allow the comparison with [5], the resulting values of the mSME algorithm are attached in the last column. The values of PDM are directly comparable because the experiments were done in the same environments as in [5]. When comparing the PDB values, attention must be paid to the fact that the PDB values in this work are based on 100 runs while the PDB values from [5] are base on 20 runs only.

The values of the CPU time consumed can not be directly compared, because the speed of computer used in [5] differs from the speed of the local computer which was used in this thesis. For approximate comparison, the CPU times consumed by the Concorde library runs on the same environments were evaluated and the estimate was made, that the computer used in [5] was 3.5 to 4.5 times slower. Therefore, the value $t_{adapt}/4$, i.e. the time consumed by the algorithm with the exclusion of the initialization part divided by 4, was stated in the last column of Table 4.7.

The results show that the Glimmer & CAN combined algorithm with the l_2 norm outperforms the same algorithm with the l_∞ norm both in quality and speed: $\Delta\text{PDM} = \text{PDM}_\infty - \text{PDM}_2$ equals 1.8 to 13.1 for individual environments and $\lambda = t_\infty/t_2$ equals 1.2 to 2. The SMACOF & CAN combined algorithm is similar to the Glimmer & CAN l_2 combined algorithm in quality of solutions on nine environments, but is worse ($\Delta\text{PDM} = 1.6$ to 3.6) on other four environments and significantly worse on `h2_5`: $\Delta\text{PDM} = 9.6$. The SMACOF & CAN algorithm is three times faster on the environment with the least guards (`map`), the speeds are similar on two other environments, `dense` and `pb4`, and the SMACOF & CAN is 2 to 9 times slower on the remaining eleven environments. The PDM of the Glimmer & CAN l_2 ranges from 2.1 to 16.2, which may be acceptable. The PDM of the Glimmer & CAN l_∞ and the SMACOF & CAN reach up to 25.0, and worse the run of the SMACOF & CAN can take up to 22 seconds.

The Fast variant of NE-Basic SOM is faster ($\lambda = 2.2$ to 3.6) than the Best variant. On the other side, the Best variant has better result ΔPDM = 3.0 for the `dense` environment and slightly better results ΔPDM = 0.1 to 1.6 for the others. The Pure variant works slower than the Fast variant ($\lambda = 2.1$ to 2.5), and the quality of its results is similar (ΔPDM = −1.2 to 1.2). It shows that even variant with no swap optimizations can work well but slower. The Fast variant has PDM equal 0.1 to 7.1 and the Best variant 0.1 to 4.2 which should be acceptable.

The speed ratio λ varies when comparing the mSME algorithm with the Fast variant. It shows that mSME is 7.4 to 13 times slower for the environments with the lowest number of guards, 2.9 to 7.5 times slower for the middle environments and 1.1 to 2.4 times slower for the environments with the highest number of guards. The Fast variant returns significantly better results for the `map` (ΔPDM = 9.0) and for the `dense` (ΔPDM = 5.0) environments. It get slightly better results for the others: ΔPDM = 0.5 to 1.9. The comparison of the mSME with the Best variant shows similar behaviour: $\lambda = t_{adapt}/4/t_{best}$ decreases from 4.6 to 0.3 as n increases. The Best variant returns better results: ΔPDM = 10.2 for `map`, ΔPDM = 8.0 for `dense` and ΔPDM = 0.6 to 3.1 for the others.

The Glimmer & CAN l_2 algorithm is slower than the Fast variant for the tested environments: λ decreases from 23 for the smallest n to 4 for the highest n. It returns significantly worse solutions: ΔPDM = 14.9 for `h2_2` and ΔPDM = 0.3 to 8.6 for the others.

To summarize the previous results, it can be stated that the Fast variant of the NE-Basic SOM algorithm outperforms the mSME algorithm [5] when the number of guards n is smaller, i.e. n is lesser than approximately 500. Their performance will be similar for higher n and better performance of the mSME algorithm can be expected for even higher n.

Map	n	Glimmer CAN $l_2, \omega = 6$	Glimmer CAN $l_\infty, \omega = 6$	SMACOF CAN $\omega = 6$	NE-Basic SOM Fast	NE-Basic SOM Pure	NE-Basic SOM Best	Lin.Kern.	Fa-SOM mSME
map	17	5.6	17.8	6.3	1.3	0.1	0.1	0.1	10.3
dense	53	9.5	14.9	10.2	7.1	7.7	4.2	0.0	12.1
potholes	68	4.2	6.0	4.0	3.9	4.0	2.5	0.0	5.6
jh2	80	2.8	7.6	3.2	1.2	1.3	1.0	0.0	1.8
pb4	104	2.1	5.7	2.1	0.1	0.2	0.1	0.0	0.7
ta_2	141	11.1	21.7	11.7	2.5	2.6	1.7	1.2	3.3
h2_5	168	7.0	14.7	10.8	0.6	1.2	0.3	0.7	2.3
jari	200	5.7	13.0	9.3	1.1	1.5	0.6	0.6	–
potholes	200	6.1	9.0	6.3	5.0	6.2	3.4	0.0	–
var_density	200	5.8	8.8	6.3	4.1	4.0	3.2	0.0	–
potholes	282	5.4	8.1	5.4	4.6	5.2	3.6	0.1	6.6
pb_15	415	5.5	11.4	7.3	1.4	1.6	0.9	0.4	1.8
h2_2	568	16.2	25.1	25.8	1.4	2.1	0.8	2.4	2.8
ta_1	574	10.5	23.6	12.2	4.3	5.0	3.4	0.5	6.0
min		2.1	5.7	2.1	0.1	0.1	0.1	0.0	0.7
max		16.2	25.1	25.8	7.1	7.7	4.2	2.4	12.1

Table 4.5: The overall comparison – PDM
PDM values of the mSME algorithm taken from [5].

Map	n	Glimmer CAN $l_2, \omega = 6$	Glimmer CAN $l_\infty, \omega = 6$	SMACOF CAN $\omega = 6$	NE-Basic SOM Fast	NE-Basic SOM Pure	NE-Basic SOM Best	Lin.Kern.	Fa-SOM mSME
map	17	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.0
dense	53	5.7	5.6	5.9	2.0	4.6	1.3	0.0	7.3
potholes	68	1.8	2.3	1.8	1.7	2.2	0.3	0.0	3.5
jh2	80	0.5	3.4	0.5	0.1	0.1	0.1	0.0	0.4
pb4	104	0.3	1.4	0.7	0.0	0.0	0.0	0.0	0.0
ta_2	141	7.9	15.5	7.4	1.2	1.5	1.0	0.0	2.4
h2_5	168	4.4	8.7	7.9	0.1	0.2	0.0	0.3	1.2
jari	200	3.1	8.1	7.5	0.4	0.6	0.1	0.1	–
potholes	200	4.3	4.9	4.3	2.8	4.5	1.6	0.0	–
var_density	200	2.3	4.9	2.9	1.8	1.7	1.1	0.0	–
potholes	282	3.8	4.8	3.6	2.7	2.8	1.7	0.0	4.0
pb_15	415	3.7	6.7	5.4	0.6	1.2	0.4	0.3	1.4
h2_2	568	13.1	19.4	20.6	0.7	1.5	0.3	1.3	1.7
ta_1	574	6.5	13.4	8.2	3.2	4.1	2.4	0.4	4.9
min		0.1	0.1	0.1	0.0	0.0	0.0	0.0	0.0
max		13.1	19.4	20.6	3.2	4.6	2.4	1.3	7.3

Table 4.6: The overall comparison – PDB
PDB values of the mSME algorithm taken from [5].

Map	n	$t_{\text{MDS}} + t_{\text{CAN}}$ Glimmer CAN $l_2, \omega = 6$	$t_{\text{MDS}} + t_{\text{CAN}}$ Glimmer CAN $l_\infty, \omega = 6$	$t_{\text{MDS}} + t_{\text{CAN}}$ SMACOF CAN $\omega = 6$	t NE-Basic SOM Fast	t NE-Basic SOM Pure	t NE-Basic SOM Best	t Lin.Kern.	$t_{adapt}/4$ Fa-SOM mSME
map	17	0.015	0.030	0.005	0.001	0.001	0.001	0.001	0.005
dense	53	0.052	0.092	0.042	0.004	0.011	0.013	0.006	0.059
potholes	68	0.087	0.131	0.370	0.007	0.017	0.020	0.009	0.074
jh2	80	0.094	0.151	0.194	0.009	0.023	0.027	0.015	0.082
pb4	104	0.213	0.366	0.186	0.017	0.040	0.050	0.044	0.074
ta_2	141	0.309	0.480	0.659	0.032	0.072	0.086	0.089	0.092
h2_5	168	0.477	0.724	1.754	0.043	0.097	0.142	0.138	0.320
jari	200	0.510	0.788	1.742	0.059	0.138	0.169	0.098	–
potholes	200	0.499	0.783	2.581	0.058	0.136	0.176	0.054	–
var_density	200	0.510	0.786	1.452	0.059	0.139	0.156	0.058	–
potholes	282	0.794	1.211	5.611	0.118	0.277	0.342	0.084	0.286
pb_15	415	1.520	1.905	3.101	0.262	0.576	0.944	0.405	0.365
h2_2	568	2.529	3.387	22.250	0.487	1.095	1.608	0.631	1.070
ta_1	574	2.103	2.701	12.043	0.516	1.145	1.650	0.415	0.550
max		2.529	3.387	22.250	0.516	1.145	1.650	0.631	1.070

Table 4.7: The overall comparison – CPU time consumed [s]
t_{adapt} values of the mSME algorithm taken from [5].

Chapter 5

Extensions for other routing problems

The extensions of the proposed solutions to other routing problems in a polygonal domain will be discussed in this chapter.

The multi-robotic scenario or Multiple Traveling Salesman Problem (mTSP) is the first one, and it differs from the standard TSP by engaging more than one salesman. Moreover, the path of every salesman must begin and finish at the special point called depot [4]. It can be solved by Self Organizing Maps involving more than one neuron string. Some of fundamental operations must be modified, e.g. when the winning neurons are searched those ones from shorter strings are preferred [4]. Both methods mentioned in chapter 3 can be used to solve the mTSP in the polygonal domain. The method using the Glimmer algorithm is straightforward – the task is transformed by multidimensional scaling from the polygonal domain to the Euclidean domain, and then it is solved by ordinary Euclidean SOM procedure for mTSP. The non-Euclidean SOM (NESOM) technique can be used too. However, it should be noted that to calculate the actual lengths of individual neuron strings, the neuron-neuron distances are demanded, which is something the NE-Basic SOM does not need. Therefore, the caching of the matrix $\mathbf{C_{PDP}}$ must be extended to the complete version (see section 3.2.6).

The zookeeper problem can be described as the task to find the shortest path to visit the specified parts of frontier of polygonal obstacles instead of guards with some simplification [4]. The NESOM technique must be modified to be able to find such a point on the specified line which is closest to specified neuron. Moreover, it must be modified to be able to move the neuron toward arbitrary point on the line. The former can be effectively done using the $\mathbf{C_{DP}}$ and $\mathbf{C_{PDP}}$ cache, the latter can be done by composing two basic moves.

For the other problems, e.g. the Watchman route problem or the Safari routing problem, possible solution by methods used in this thesis are not straightforward.

Chapter 6

Conclusions

Two methods for solving the traveling salesman problem in the polygonal domain were described in this thesis. The method using multidimensional scaling – the Glimmer algorithm – to transform the TSP task in the polygonal domain to the TSP task in the Euclidean domain to be used with SOM methods afterwards was described and implemented. The Glimmer algorithm was modified to run with norms other than l_2. Two variants with different direction vectors were introduced. The Glimmer algorithm was successfully used with the Basic SOM and CAN algorithms and it outperforms the previously used MDS procedure SMACOF, with a few exceptions. However, the expected profit from the use of other norms has not been achieved.

The completely new non-Euclidean form of SOM technique was introduced. It was implemented for the Basic SOM algorithm, but it can by implemented for the CAN algorithm easily. The main disadvantage of the NE-Basic SOM method could be its higher asymptotical complexity in comparison with the other algorithms.

The results of numerical experiments were compared with the result from [5] and it has been shown that for the number of guards smaller (n is lesser than 500 approximately) the other SOM based methods are outperformed by the NE-Basic SOM algorithm. It is expected that the mSME algorithm would outperform the others for higher n, but it was not tested experimentally.

The methods mentioned above could be improved in various ways. The non-Euclidean version of CAN algorithm (NECAN) could be completed, tuned up and tested. The efficiency and the speed of the swap optimization routines could be increased – e.g. by storing the list of previously found swaps or by using some heuristic instead of searching for the best swap possible. The changes of the μ parameter while running the algorithm as in [1] could be tested. The combined method Glimmer & Basic SOM & NE-Basic SOM using the non-Euclidean method for the distance corrections only and working with the sparse matrix of the distance corrections $\mathbf{\Delta D}$ could be completed and tested for lowering the asymptotic complexity, i.e. to make method usable for even higher values of n.

Appendix A

Bibliography

[1] Yanping Bai, Wendong Zhang, and Zhen Jin. "An new self-organizing maps strategy for solving the traveling salesman problem". In: *Chaos, Solitons & Fractals* 28 (May 2006), pp. 1082–1089. ISSN: 0960-0779.

[2] E. M. Cochrane and J. E. Beasley. "The co-adaptive neural network approach to the Euclidean travelling salesman problem". In: *Neural Networks* 16.10 (Dec. 2003), pp. 1499–1525. ISSN: 0893-6080.

[3] *Concorde TSP Solver*. URL: `http://www.math.uwaterloo.ca/tsp/concorde.html`.

[4] J. Faigl. "Multi-goal path planning for cooperative sensing". Doctoral Thesis. Czech Technical University in Prague, Feb. 2010.

[5] J. Faigl. "On the performance of self-organizing maps for the non-Euclidean Traveling Salesman Problem in the polygonal domain". In: *Information Sciences* 181.19 (Oct. 2011), pp. 4214–4229. ISSN: 0020-0255.

[6] J. Gross and J. Yellen. *Handbook of graph theory*. CRC Press, 2004. ISBN: 1-58488-090-2.

[7] J. J. Hopfield and D. W. Tank. "Neural computation of decisions in optimization problems". In: *Biological Cybernetics* 52.3 (July 1985), pp. 141–152. ISSN: 0340-1200.

[8] S. Ingram, T. Munzner, and M. Olano. "Glimmer: Multilevel MDS on the GPU". In: *IEEE Transactions on Visualization and Computer Graphics* 15.2 (Mar. 2009), pp. 249–261. ISSN: 1077-2626.

[9] K. J. Obermeyer and Contributors. *The VisiLibity Library*. URL: `http://www.VisiLibity.org`.

[10] S. Somhom, A. Modares, and T. Enkawa. "A self-organising model for the travelling salesman problem". In: *Journal of the Operational Research Society* 48.9 (Sept. 1997), pp. 919–928. ISSN: 0160-5682.

[11] R. Sushkov. "Self-Organizing Structures for the Travelling Salesman Problem in a Polygonal Domain". Bachelor's Thesis. Czech Technical University in Prague, May 2015.

[12] Junying Zhang et al. "An overall-regional competitive self-organizing map neural network for the Euclidean traveling salesman problem". In: *Neurocomputing* 89 (2012), pp. 1–11.

Appendix B

CD Content

Directory name	**Description**
data	Output of numerical experiments
maps	Maps used for testing
progs	Source code
scr_local	Auxiliary scripts