# Satura rādītājs

Satura rādītājs ..... 1
1. Policy iteration and value iteration ..... 3
1.1. Problem formulation ..... 3
1.1.1. The grid, states and actions ..... 3
1.1.2. State transitions – the potential result of performing actions ..... 3
1.1.3. The state transition rewards ..... 4
1.1.4. Finding the optimal policy – problem solution ..... 5
1.1.5. Available input data ..... 6
1.2. Policy evaluation ..... 6
1.2.1. Algorithm „evaluate_policy_det” ..... 6
1.2.2. Examples ..... 7
1.3. Policy iteration. Algorithm „iterate_policy_det” ..... 8
1.4. Value iteration. Algorithm „iterate_values_det” ..... 8
1.5. Summary of lab works programming task ..... 9
2. Monte Carlo method ..... 10
2.1. The definition of the problem ..... 10
2.1.1. A grid, states and actions ..... 10
2.1.2. State transitions – the potential result of actions ..... 10
2.1.3. State transition rewards ..... 10
2.1.4. Optimal policy ..... 11
2.1.5. Available input data ..... 11
2.2. Generating an episode. Algorithm „generate_episode_soft” ..... 11
2.3. Monte Carlo control algorithm „monte_carlo_det” ..... 13
2.4. Summary of the programming work required for the lab work ..... 14
3. TD (temporal difference) algoritmi. Laboratorijas darbs ..... 15
3.1. Problēmas formulējums ..... 15
3.2. Algoritms epizodes viena soļa aprēķināšanai „next_state” ..... 15
3.3. On-policy algoritms optimālas politikas iegūšanai „Sarsa” ..... 16
3.4. Off-policy algoritms optimālas politikas iegūšanai „Q-learning” ..... 17
3.5. Laboratorijas darba veicamo programmēšanas darbu kopsavilkums ..... 17
4. Genetic algorithms ..... 18
3.1. The definition of the problem ..... 18
3.1.1. Modeling of the boolean 'NOT AND' function ..... 18
*3.1.2.* Available data ..... 18
3.2. Genetic algorithm realization ..... 19
3.2.1. Awaited result ..... 19
3.2.2. Presenting the solution ..... 19
3.2.2.1. Algorithm for turning a number in series of bits „number_to_bits” ..... 20
3.2.2.2. Algorithm „bits_to_number” turns series of bits in to a number ..... 20
3.2.2.3. Algorithm to turn the whole solution to bits and back ..... 21
3.2.3. Checking the validity of the solution ..... 21
3.2.4. Partitioning of probabilities and choice of individuals relevant to the probability partitions ..... 23
3.2.5. Genetic opperands ..... 26
3.2.6. Summary of the genetic algorithm ..... 28
3.3. The summary of programming tasks in the labwork ..... 29
5. Naive Bayes classifier ..... 30
1.6. Problem formulation ..... 30

1.6.1. Credit risk estimation examples ..... 30
*1.6.2.* Available data ..... 31
1.7. The principle of operation of Naive Bayes classifier ..... 32
1.8. Naive Bayes classifier realization ..... 32
1.8.1. Classification of one sample ..... 32
1.8.2. Testing classifier with training samples ..... 33
1.9. The summary of programming tasks in the labwork ..... 33
6. Ant colony optimization for TSP ..... 34
1.10. Problem formulation – Travelling Salesman Problem ..... 34
1.11. Realization of algorithm ..... 35
1.11.1. Generation of tour ..... 35
1.11.2. The summary of ant colony optimization algorithm ..... 38
1.12. The summary of programming tasks in the labwork ..... 39
7. Support vector machine (*SVM*) ..... 40
7.1. Definition of two linear problems for a SVM ..... 40
7.1.1. Problem #1. Modeling the Boolean function ‘AND’ ..... 40
7.1.2. Problem #2. The distribution of linear points in a plane ..... 41
7.2. Realization of the iterative SVM algorithm in the case of a linear problem ..... 41
7.2.1. Running a single example with a linear perceptron ..... 42
7.2.2. Training a linear perceptron on a set of examples ..... 42
7.2.3. Summary of iterative linear SVM algorithm ..... 44
7.2.4. System configuration and obtainable result ..... 44
7.3. Definition of a non-linear problem for a SVM ..... 45
7.4. Realization of an iterative SVM algorithm in the case of a linear problem ..... 47
7.4.1. Realization of the conversion function ..... 47
7.4.2. Summary of the non-linear iterative SVM algorithm ..... 50
7.4.3. Configuration of a non-linear system and obtainable result ..... 51
8. Partial observability and POMDP. Tiger’s problem ..... 53
8.1. Problēmas formulējums ..... 53
8.2. Ticamības stāvokļu koka būvēšana ..... 54
8.3. Ticamības stāvokļu koka apmācīšana ..... 58

# 1. Policy iteration and value iteration

## *1.1. Problem formulation*

### 1.1.1. The grid, states and actions

A 5x5 grid is given, through which the so-called agent is traveling.

Every cell of the grid is represented by a state $s_i$.

$$S = \{s_i\}_{i=1}^{n},$$

where $n$=25.
The set of states $S$:

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

From every state $s_i$ there are four moves possible (four actions) – {up, down, left, right}.

$$A = \{a_i\}_{i=1}^{m},$$

where $m$=4.
The set of possible actions $A$:

| 1 | ↑ | up |
|---|---|---|
| 2 | ↓ | down |
| 3 | ← | left |
| 4 | → | right |

### 1.1.2. State transitions – the potential result of performing actions

A set of transitions $T$ is given, which determines to which state does the agent move while applying action $a$ to the state $s$. Essentially those are the state transitions which determines the environment of the action. If there are 25 states and 4 actions possible on every state then there are 100 records in the set of state transitions where every record is in form: {state, action, next state}:

$$T = \{\langle s, a, s'\rangle \mid s, s' \notin S, a \in A\}$$ or defining this as a function,

$$T = S \times A \rightarrow S$$

There are 3 different cases of state transitions in the given task:

**A. Usual case.** When performing an action agent moves to the nearby cell according to the type of action, for instance, moving right from state 13 leads to state 14:

$$\langle s_{13}, \rightarrow, s_{14}\rangle$$

**B. Outer case.** The exception occurs when there is a chance of moving out of the grid when performing the corresponding move – then the existing state persists. For an instance,

$\langle s_{11}, \leftarrow, s_{11} \rangle$

**C. Special cases.**

There are two special cases.

Any action from the state 2 leads to state 22, but from state 4 – leads to state 14.

$\langle s_2, \{\uparrow, \downarrow, \leftarrow, \rightarrow\}, s_{22} \rangle$

$\langle s_4, \{\uparrow, \downarrow, \leftarrow, \rightarrow\}, s_{14} \rangle$

**Thereby the total scheme of the grid looks like this:**

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

### 1.1.3. The state transition rewards

Every state transition has a certain reward which is obtained by the agent when performing the given action (more precisely, performing the action of the given state). The reward can also be with a minus sign which can be interpret as a consumption of resources when performing an action:

$R = \{\langle s, a, r \rangle \mid s \notin S, a \in A, r \notin \Re\}$ or defining this as a function,

$R = S \times A \rightarrow \Re$

*Reinforcement learning* basics consists of choosing that kind of strategy or policy which provides the highest total of the rewards in a longer period of time. For every state transition there is a reward being defined and in our task there are 100 of them.

**A. Usual case.** When performing an usual action, agent does not get any reward, respectively $r$=0. For instance,

$\langle s_{13}, \rightarrow, 0 \rangle$

**B. Outer case.** If there is a chance of moving out of the grid when performing the corresponding move then reward (penalty in this case) is -1, for instance,

$\langle s_{11}, \leftarrow, -1 \rangle$

**C. Special cases.**

There are 2 special cases:

Performing an action from state 2 obtains a reward of 10, but from state 4 – reward of 5:

$\langle s_2, \{\uparrow, \downarrow, \leftarrow, \rightarrow\}, 10 \rangle$

$$\langle s_4, \{\uparrow, \downarrow, \leftarrow, \rightarrow\}, 5 \rangle$$

**The total scheme of the grid including rewards looks like this:**

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

### 1.1.4. Finding the optimal policy – problem solution

**Policy** means a strategy which determines which action should be performed in which state.

Optimal policy means a strategy (which to be guided from) where the total reward in a longer period of time is the highest.

In the next example a randomly generated policy is shown (1-up, 2-down, 3-left, 4-right)

| 1 | 1 | 3 | 2 | 1 |
|---|---|---|---|---|
| 1 | 3 | 2 | 4 | 2 |
| 2 | 4 | 3 | 1 | 3 |
| 4 | 1 | 3 | 2 | 4 |
| 3 | 3 | 2 | 2 | 1 |

It turns out that the optimal policy is not dependant only from the state transitions and rewards which have been defined for this problem previously, but also from the so-called **discount rate** $\gamma$, which in the case of positive rewards is similar to inflation. Discount rate $\gamma$ is a real number between [0;1] and the less $\gamma$ is, the less effect previously collected rewards make so it is more efficient to get rewards more often than getting just a valuable ones not so often.

- Thereby the problem is determined by 3 components:

1. the model of state transitions $T$,
2. the rewards of state transitions $R$,
3. discount rate $\gamma$.

If $\gamma$ is close to 0 then policy is affected almost only by the current reward $r$, if close to 1, then fully by the next state notional value too.

**The example of optimal policy No. 1, $\gamma$ = 0.5.**

| 4 | 1 | 3 | 1 | 3 |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

**The example of optimal policy No. 2, γ = 0.9.**

| | | | | |
|---|---|---|---|---|
| 4 | 1 | 3 | 1 | 3 |
| 1 | 1 | 1 | 3 | 3 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

In the first example the discount rate is smaller and that is why the main goal is to achieve the reward faster – it doesn't matter it is state 2 or 4, while in the second example a higher tendency is to get directly to the state 2 (from states 9 and 10 we need to move left not up).

### 1.1.5. Available input data

There are 25 states total (1..25) and 4 actions total (1-up, 2-down, 3-left, 4-right).

**T.txt –** table 25x4 where there are states located in the lines, actions and value in the line *i* located in the collumns, and the column *k* stands for the state transition t(*i*,*k*) (corresponds the *T* in the section 1.1.2).

**R.txt –** rewards corresponding to the state transitions (corresponding the *R* in the section **Error! Reference source not found.**).

**P.txt –** randomly created policy for self-control (corresponds to random policy in the beginning of the section 1.1.4).

## *1.2. Policy evaluation*

### 1.2.1. Algorithm „evaluate_policy_det”

To obtain the optimal policy we should be able to evaluate the current policy.

The next pseudo code shows the evaluation of a policy.

Policy evaluation stands for obtaining a table of values where there is one value allocated to each state.

```
Procedure evaluate_policy_det (T, R, π, γ, θ) returns V
        S() – states
        T() – state transitions
        R() – rewards
        π() – policy
        γ – discount rate [0..1]
        θ – convergence rate, 0 < θ << 1
        V() – policy evaluation
Begin
        V := 0
        Do
                W := V  % fixes the current evaluations
                Δ := 0  % change of the matrix of values. When it becomes small enough we can stop
                Forall s in S do
                        a := π(s)  % action corresponding the policy
                        s2 := T(s,a)  % next state
                        r := R(s,a)  % the reward of the next state
                        V(s) := r + γ * W(s2)  % the value of the current state
                        Δ := max(Δ,|V(s)-W(s)|)  % maximal change of some state
                Endforall
        While Δ ≥ θ
End
```

### 1.2.2. Examples

Two of the algorithm ***evaluate_policy_det*** results shown here (section 1.2.1) result examples

*T*=„T.txt"; *R*=„R.txt"; *π*=„P.txt"; *θ*=0.1 (see available data files in the section 1.1.5)

**Example #1. γ=0.5**

| -1.9375 | 9.5625 | 4.8125 | 5.0000 | -1.9375 |
|---|---|---|---|---|
| -0.9375 | -0.4375 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | -0.9375 | -1.9375 |
| -1.9375 | -0.9375 | -1.9375 | -1.9375 | -0.9375 |

**Example #1. γ=0.9**

| -9.1137 | 2.7863 | 2.5963 | 5.0000 | -9.1137 |
|---|---|---|---|---|
| -8.1137 | -7.2137 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | -8.1137 | -9.1137 |
| -9.1137 | -8.1137 | -9.1137 | -9.1137 | -8.1137 |

## *1.3. Policy iteration. Algorithm „iterate_policy_det”*

Obtainment of an optimal determined policy is realized by algorithm ***iterate_policy_det***.

```
Procedure iterate_policy_det (T, R, γ, θ) returns π
        A() – actions
        S() – states
        T() – state transitions
        R() – rewards
        γ – discount rate [0..1]
        θ – convergence rate, 0 < θ << 1
        π() – policy
        evaluate_policy_det – determined policy evaluation
Begin
        Generates a random determined policy  π
        Do
                V := evaluate_policy_det (T, R, π, γ, θ)
                policy_stable := True
                Forall s In S Do
                        oldp := π(s)
                          % the action a at which the given state value would be the maximum of all possible
                        maxval := Null
                        Forall a In A Do
                                s2 := T(s,a)  % next state
                                r := R(s,a)  % next state value
                                v := r + γ * V(s2)  % current state value at selected action a
                                If maxval = Null Or v > maxval Then
                                        maxval := v
                                        π(s) := a
                                Endif
                        Enforall
                        If π(s) ≠ oldp Then
                                policy_stable := False
                        Endif
                Endforall
        While Not policy_stable
End
```

Examples for obtaining optimal policies at $T$=„T.txt”; $R$=„R.txt”; $\pi$=„P.txt”; $\theta$=0.1 are shown in section 1.1.4.

## *1.4. Value iteration. Algorithm „iterate_values_det”*

More compact optimal determined policy obtainment is realized by algorithm ***iterate_values_det*** (comparing with policy iteration there is no need to use a separate and resource consuming policy evaluation step).

```
Procedure iterate_values_det (T, R, γ, θ) returns π
        A() – actions
        S() – states
        T() – state transitions
        R() – rewards
        V() – state values
        γ – discount rate [0..1]
        θ – convergence rate, 0 < θ << 1
        π() – policy
Begin
        V := 0
        Do
                Δ := 0  % change of the matrix of values. When it becomes small enough we can stop
                Forall s In S Do
                        vs_old := V(s)  % current state value
                        V(s) := Null
                        Forall a In A Do
                                s2 := T(s,a)  % next state
                                r := R(s,a)  % next state value
                                v := r + γ * V(s2)  % current state value at selected action a
                                If V(s) = Null Or v > V(s) Then
                                        V(s) := v
                                        π(s) := a
                                Endif
                        Enforall
                        Δ := max(Δ,|V(s)-vs_old|)  % maximal change of some state
                Endforall
        While Δ ≥ θ
End
```

Examples for obtaining optimal policies at $T$=„T.txt”; $R$=„R.txt”; $\pi$=„P.txt”; $\theta$=0.1 are shown in section 1.1.4.

## 1.5. Summary of lab works programming task

- Load files $T$=„T.txt”; $R$=„R.txt”; $\pi$=„P.txt” in the computers memory.
- Implement algorithm ***evaluate_policy_det*** (section 1.2.1) and test it on both examples at section 1.2.2.
- Implement algorithm ***iterate_policy_det*** (section 1.3) and test it on both examples about obtaining optimal policies at section 1.1.4.
- Implement algorithm ***iterate_values*** (section 1.4) and test it on both examples about obtaining optimal policies at section 1.1.4.

# 2. Monte Carlo method

## 2.1. The definition of the problem

### 2.1.1. A grid, states and actions

A 3×4 grid is given, wherein one of the nodes is empty (between 5 and 6), and states 4 and 7 are defined as final states (framed with a bold line border).

Let $S$ be the set of states, which are not final states, and $S^+$ – the set of all states, including final states.

State set $S^+$:

| 1 | 2 | 3 | **4** |
|---|---|---|---|
| 5 | | 6 | **7** |
| 8 | 9 | 10 | 11 |

From each state $s_i$ four moves (four actions) can be made – {up, down, left, right}, as it was in the previous lab work.

### 2.1.2. State transitions – the potential result of actions

State transitions can only be made from regular (non-final) states. If a node in the chosen direction exists then the transition to it happens, else stay in the same state or node, akin to the previous lab work:

$$\langle s_6, \uparrow, s_3 \rangle$$

$$\langle s_9, \downarrow, s_9 \rangle$$

The middle filled-in node is also an obstacle and an attempt to move to it will result in staying in the same node:

$$\langle s_5, \rightarrow, s_5 \rangle$$

Getting to the final states is possible in three ways: from states 3 and 4 – to the right, and from state 11 – upwards.

### 2.1.3. State transition rewards

Whether an attempt to go outside of the grid is made or not, moving to a nonterminal (non-final) state results in a negative reward: -1, for example:

$$\langle s_6, \uparrow, -1 \rangle$$

But the big deal of the task is in the transitions to the final states.

**State 4 is a „good" state.** When transitioning to it a positive reward of 100 is given:

$$\langle s_3, \rightarrow, 100 \rangle$$

**State 7 is a „bad” state.** When transitioning to it a negative reward of -9 is given:

$$\langle s_6, \rightarrow, -9 \rangle$$

$$\langle s_{11}, \uparrow, -9 \rangle$$

**This is what the complete scheme of the grid looks like when taking rewards into account:**

| | | | |
|---|---|---|---|
| -1 | -1 | -1 | 100 |
| -1 | | -1 | -9 |
| -1 | -1 | -1 | -1 |

#### 2.1.4. Optimal policy

An optimal policy means getting the biggest reward, upon reaching a final state. Finding the optimal policy is also influenced by the discount factor γ.

**Example of optimal policy γ = 0.5.**

| | | | |
|---|---|---|---|
| 4 | 4 | 4 | |
| 1 | | 1 | |
| 1 | 4 | 1 | 3 |

#### 2.1.5. Available input data

The set has 11 states (1..11), of which 9 are nonterminal, and 4 actions (1 - up, 2 - down, 3 - left, 4 - right).

**T.txt** – a 11×4 table, where the rows are states, the columns are actions and the value in row *i* and column *k* denotes a state transition t(*i*,*k*).

**R.txt** – the corresponding rewards of state transitions.

**F.txt** – final states (marked with a 1).

**P.txt** – a randomly generated policy for self-control.

### *2.2. Generating an episode. Algorithm „generate_episode_soft”*

The *Monte Carlo* method includes a generation of episodes mechanism that takes an existing policy into account. The episode begins with a freely chosen nonterminal state and ends with a final state.

An episode is basically a set of three, where each of the three is as follows: *<action, received reward, next state>*.

To ensure that the generation of episodes doesn't get stuck in a cycle (the policy is determined and in the beginning there can be a sub-optimal policy), and ensure that there would be a statistically large enough probability that the episode goes through all states and all possible

actions in the states, the so called *ε-soft* principle is used, meaning that with a set probability ($\varepsilon$) in the given state an action that does not conform to the policy is chosen:

An example of a generated episode with $\varepsilon = 0.1$ (episodes can be **much** longer!):

| 0 | 0 | 10 |
|---|---|---|
| 2 | -1 | 10 |
| 2 | -1 | 10 |
| 2 | -1 | 10 |
| 4 | -1 | 11 |
| 1 | -9 | 7 |

The episode is generated with the policy (2 1 4 0 1 1 0 1 3 2 4)

**Procedure** ***generate_episode_soft*** **(***T*, *R*, *π*, *ε***) returns** *Ep*

*T()* – state transitions
*R()* – rewards
*π()* – policy
*ε* – *softening* factor, $0 < \varepsilon << 1$
*Ep()* – episode

**Begin**

*s* := choose a nonterminal state at random
*Ep* := <<0,0,*s*>>
**While** *s* is not terminal
  *a0* := *π*(*s*) {policy in the given state}
  *a* := choose an action from $a_1..a_n$ with a probability distribution $p_1..p_n$, where
    $p_i := \varepsilon$, if $a_i \neq a0$
    $p_i := 1-\varepsilon*(n-1)$, if $a_i = a0$
  *s2* := *T*(*s*,*a*)
  *r* := *R*(*s*,*a*)
  *Ep* := *Ep* + <*a*,*r*,*s2*>
  *s* := *s2*
**Endwhile**

**End**

## 2.3. Monte Carlo control algorithm „monte_carlo_det"

The algorithm ***monte_carlo_det*** realizes the acquisition of the optimal determined policy.

```
Procedure monte_carlo_det (T, R, γ, ε, maxit) returns π
    S() – nonterminal states
    T() – state transitions
    R() – rewards
    γ – discount factor [0..1]
    ε – policy softening factor, 0 < ε << 1
    maxit – maximum number of iterations
    Q() – action values of states
    Returns() – the resulting rewards of state actions
    π() – policy
    generate_episode_soft – generation of episodes (section Error! Reference source not found.)
Begin
    Generate a determined policy π at random
    Forall (s, a) Do
        Q(s,a) := 0  % can be other freely chosen values
        Returns(s,a) := []  % empty string
    Endforall
    Do maxit times
        Ep := generate_episode_soft (T, R, π, ε)
        For e:=2 To size(Ep) Do  % the first element contains only the start state
            s := Ep(e-1,3)
            a := Ep(e,1)
            If (s,a) is the first occurrence in Ep Then
                r := 0
                For f := size(Ep) Downto e Do
                    r := r * γ + Ep(f,2)
                Endif
                Returns(s,a) := (Returns(s,a) Union {r}) %  Add r to Returns(s,a)
                Q(s,a) := average(Returns(s,a))
            Endif
        Endforall
        Forall s In S Do  % in all nonterminal states
             % the action a, with the highest Q value, when s is a fixed value
            π(s) := argmax(a,Q(s,a))
        Endforall
    Enddo
End
```

An example for acquiring an optimal policy with *T*=„T.txt"; *R*=„R.txt" ; *π*=„P.txt" ; *F*=„F.txt"; *ε*=0.1; *γ*=0.9 is shown in section **Error! Reference source not found.**

Example of an action evaluation table Q:

| | ⇑ | ⇓ | ⇐ | ⇒ |
|---|---|---|---|---|
| 1. | 56.700324 | 43.651243 | 50.385249 | 65.254062 |
| 2. | 61.023008 | 60.682764 | 49.445314 | 79.313545 |
| 3. | 74.169516 | 59.260233 | 63.143447 | 100 |
| 4. | 0 | 0 | 0 | 0 |
| 5. | 53.261263 | 34.059217 | 37.536579 | 47.382969 |
| 6. | 80.867575 | 48.084442 | 60.802438 | -9 |
| 7. | 0 | 0 | 0 | 0 |
| 8. | 42.843857 | 39.079461 | 32.372455 | 24.434825 |
| 9. | 36.584147 | 29.349683 | 34.113061 | 49.056515 |
| 10. | 63.207617 | 41.743727 | 33.236223 | 36.884705 |
| 11. | -9 | 34.900828 | 43.33167 | 21.217676 |

## 2.4. Summary of the programming work required for the lab work

- Load files *T*=„T.txt”; *R*=„R.txt”; *π*=„P.txt”; F=„F.txt” into the computer memory.
- Implement algorithm ***generate_episode_soft*** (section **Error! Reference source not found.**) and test it with parameter ε=0.1.
- Implement algorithm ***monte_carlo_det*** (section 2.3) and test it with parameters ε=0.1, γ=0.9, *maxit*=1000.

# 3. TD (temporal difference) algoritmi. Laboratorijas darbs

## 3.1. Problēmas formulējums

Tā pati problēma, kas aprakstīta nodaļā **Error! Reference source not found.**

## 3.2. Algoritms epizodes viena soļa aprēķināšanai „next_state”

Algoritms viena soļa aprēķināšanai ***next_state*** atbilst viena soļa aprēķināšanai epizodē, kas aprakstīta nodaļā **Error! Reference source not found.**.

```
Function next_state (s, ε) Returns a, r, s2
        s – sākuma stāvoklis
        T() – stāvokļu pārejas
        R() – atlīdzības
        π() – politika
        ε – mīkstināšanas (soft) faktors, 0 < ε << 1
        a – izvēlētā darbība
        r – iegūtā atlīdzība
        s2 – nākošais stāvoklis
Begin
        rnd := nejaušs skaitlis robežās 0..1
        If rnd < ε Then
                a := izvēlēties darbību nejauši starp visām iespējamām
        Else
                a := π(s)  % politika dotajā stāvoklī
        Endif
End
```

Darbību vērtējumu tabulas $Q$ piemērs:

| | ⇑ | ⇓ | ⇐ | ⇒ |
|---|---|---|---|---|
| 1. | 3.3891817 | -0.60174394 | 2.580622 | 35.8232 |
| 2. | 20.842085 | 12.184189 | 7.2605488 | 66.686393 |
| 3. | 55.488228 | 22.20272 | 16.191977 | 99.991536 |
| 4. | 0 | 0 | 0 | 0 |
| 5. | 10.030075 | -1.2971339 | -1.6344344 | -2.0346993 |
| 6. | 63.579861 | 13.406996 | 13.350152 | -4.6953279 |
| 7. | 0 | 0 | 0 | 0 |
| 8. | -0.705524 | -1.654783 | -1.1973656 | 0.023690834 |
| 9. | 0.27263845 | -1.6602663 | -1.2339563 | 20.367975 |
| 10. | 45.504665 | 3.1392824 | 0.58727142 | 3.2718419 |
| 11. | -3.0951 | -0.72839782 | 16.328932 | -0.019517445 |

### 3.3. On-policy algoritms optimālas politikas iegūšanai „Sarsa”

Optimālas determinētas politikas iegūšanu realizē algoritms ***sarsa***.

```
Function sarsa (T, R, γ, ε, maxit, α) Returns π
    S() – neterminālie stāvokļi
    T() – stāvokļu pārejas
    R() – atlīdzības
    γ – diskonta faktors [0..1]
    ε – politikas mīkstināšanas faktors, 0 < ε << 1
    α – apmācības koeficients, 0 < α << 1
    maxit – maksimālais iterāciju skaits
    Q() – stāvokļu darbību vērtības
    π() – politika
    next_state – nākošā stāvokļa izrēķināšana (nodaļa 3.2)
Begin
    Uzģenerē determinētu politiku π uz labu laimi (in random)
    Forall (s, a): Q(s,a) := 0  % var arī citas vērtības, brīvi pēc izvēles
    Do maxit times
        s :=  izvēlas neterminālu sākuma stāvokli uz labu laimi
        a, r, s2 :=  next_state (s, ε)
        While s is not terminal
            If s2 is terminal Then
                Q(s,a) := Q(s,a) + α * (r - Q(s,a))
                s := s2
            Else
                a2, r2, s3 :=  next_state (s2, ε)
                Q(s,a) := Q(s,a) + α * (r + γ * Q(s2,a2) - Q(s,a))
                a := a2
                r := r2
                s := s2
                s2 := s3
            Endif
        Endwhile
        Forall s In S Do  % pa visiem neterminālajiem stāvokļiem
            π(s) := argmax(a,Q(s,a))
        Endforall
    Enddo
End
```

## 3.4. Off-policy algoritms optimālas politikas iegūšanai „Q-learning"

Optimālas determinētas politikas iegūšanu realizē algoritms ***q_learning***.

```
Function q_learning (T, R, γ, ε, maxit, α) Returns π
    S() – neterminālie stāvokļi
    T() – stāvokļu pārejas
    R() – atlīdzības
    γ – diskonta faktors [0..1]
    ε – politikas mīkstināšanas faktors, 0 < ε << 1
    α – apmācības koeficients, 0 < α << 1
    maxit – maksimālais iterāciju skaits
    Q() – stāvokļu darbību vērtības
    π() – politika
    next_state – nākošā stāvokļa izrēķināšana (nodaļa 3.2)
Begin
    Uzģenerē determinētu politiku π uz labu laimi (in random)
    Forall (s, a): Q(s,a) := 0  % var arī citas vērtības, brīvi pēc izvēles
    Do maxit times
        s :=  izvēlas netermināļu sākuma stāvokli uz labu laimi
        While s is not terminal
            a, r, s2 :=  next_state (s, ε)
            If s2 is terminal Then
                Q(s,a) := Q(s,a) + α * (r - Q(s,a))
            Else
                Q(s,a) := Q(s,a) + α * (r + γ * max(Q(s2,_)) - Q(s,a))
            Endif
            s := s2
        Endwhile
        Forall s In S Do  % pa visiem neterminālajiem stāvokļiem
            π(s) := argmax(a,Q(s,a))
        Endforall
    Enddo
End
```

## 3.5. Laboratorijas darba veicamo programmēšanas darbu kopsavilkums

- Ielādēt datora atmiņā failus $T$=„T.txt"; $R$=„R.txt"; $\pi$=„P.txt"; F=„F.txt"
- Realizēt palīgalgoritmu ***next_state*** (nodaļa 3.2), izceļot to no algoritma ***generate_episode_soft*** (nodaļa 2.2).
- Realizēt algoritmu ***sarsa*** (nodaļa 3.3) un notestēt ar parametriem $\varepsilon$=0.1, $\gamma$=0.9, $\alpha$=0.1, *maxit*=200..500.
- Realizēt algoritmu ***q_learning*** (nodaļa 3.4) un notestēt ar parametriem $\varepsilon$=0.1, $\gamma$=0.9, $\alpha$=0.1, *maxit*=200..500

# 4. Genetic algorithms

## *3.1. The definition of the problem*

### 3.1.1. Modeling of the boolean 'NOT AND' function

There are 4 examples given, that describe the logical function ***NOT ($x_1$ AND $x_2$)***.

| $x_1$ | $x_2$ | d |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Model the function in ‘NOT AND’ form

$y = F(x_1, x_2) = w_0 + w_1x_1 + w_2x_2$.

That means, that such coefficients $w_0$, $w_1$, $w_2$ should be found, that for each pair of parameters $<x_1, x_2>$ the value of function $F(x_1, x_2)$ would close in to *NOT ($x_1$ AND $x_2$).*

For the sake of simplifying the process lets do some changes – we will limit $F(x_1, x_2)$ to the interval [0..1]:

$$F(x_1, x_2) = \begin{cases} 1; & NET > 1 \\ 0; & NET < 0 \\ NET; & else \end{cases}, \text{ where}$$

$$NET = w_0 + w_1x_1 + w_2x_2$$

### *3.1.2.* Available data

*x.txt* and *d.txt* – instructing examples

From the data 2 parts can be set apart:

- *D* – awaited result (*d.txt*):

| |
|---|
| 1 |
| 1 |
| 1 |
| 0 |

- *X* – input data (*x.txt*):

| | |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

*p.txt*, *f.txt* and *w.txt* – data for interim result checking

## 3.2. Genetic algorithm realization

### 3.2.1. Awaited result

One of the perfect results (where $F(x_1, x_2) = d$) is:

$w_0 = 2$

$w_1 = -1$

$w_2 = -1$

<table>
<tr><th>$x_1$</th><th>$x_2$</th><th>$w_0$</th><th>$w_1$</th><th>$w_2$</th><th>NET</th><th>$F(x_1,x_2)$</th><th>$d$</th></tr>
<tr><td>0</td><td>0</td><td rowspan="4">2</td><td rowspan="4">-1</td><td rowspan="4">-1</td><td>2</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
</table>

### 3.2.2. Presenting the solution

Presenting the solution is a vital aspect in the realization of genetic algorithm and the success fo a genetic algorithm is often dependent on the success of the presentation.

For a genetic algorithm the solution is usually presented in the form of bits, so that it`s easier to apply the so called genetic operands during the operation of the algorithm.

The solution of our problem are the 3 coefficients.

Let us code each of the coefficients as 8 bits, which makes it a total of 24 bits.

Lets code the numbers in the range of [-2, +2].

Coding a single number:

- the most left bit codes the sign +/-,
- the others code the powers of two.

If there is a series of bits $<b_1, ..., b_n>$

then it represents the number:

$$s = \begin{cases} \sum_{i=2}^{n} b_i 2^{2-i}; & b_1 = 0 \\ -\sum_{i=2}^{n} b_i 2^{2-i}; & b_1 = 1 \end{cases}$$

or better

$$s = \pm\left( b_2 + \frac{b_3}{2} + \frac{b_4}{4} + \frac{b_5}{8} ... \right)$$

### *3.2.2.1. Algorithm for turning a number in series of bits „number_to_bits”*

Algorithm ***number_to_bits*** is meant to turn a real number in the range of [-2..+2] in to a series of bits.

```
Procedure number_to_bits (n, bitcount) Returns Bits
        n – number to chnage in bit form
        bitcount – how many bits will the number be turned in to
        Bits() – a series of bits (e.g. array) in the length of bitcount
Begin
        Bits := {0...}  % creates an array with length of bitcount and fills with zeros
        If n < 0 Then  % if a negative number
                Bits(1) := 1  % the first bit becomes a one
                n = -n
        Endif
        factor := 1  % power of two, at first 2 in the power of zero
        For k := 2 To bitcount Do
                If factor <= n Then
                        Bits(k) = 1
                        n := n – factor;
                Endif
                factor := factor / 2
        Endfor
End
```

### *3.2.2.2. Algorithm „bits_to_number” turns series of bits in to a number*

Algorithm ***bits_to_number*** is meant for turning a series of bits in to a real number in the range of [-2..+2].

```
Procedure bits_to_number (Bits, bitcount) Returns n
        Bits() – a series of bits (e.g. array) in the length of bitcount
        bitcount –  how many bits will the number be turned in to
        n – number to chnage in bit form
Begin
        n := 0
        factor := 1  % power of two, at first 2 in the power of zero
        For k := 2 To bitcount Do
                n := n + factor * Bits(k)
                factor := factor / 2
        Endfor
        If Bits(1) = 1 Then  % if the sign bit is 1
                n = -n
        Endif
End
```

#### *3.2.2.3. Algorithm to turn the whole solution to bits and back*

Algorithm ***solution_to_bits*** turns a series of numbers in to a series of bits

```
Procedure solution_to_bits (x, bitcount) Returns Bits
    x – a series of numbers to turn in to bit form
    bitcount – to how many bits including the sign will the number be turned
    Bits() – series of bits that represent x in the length of bitcount * |x|
    number_to_bits – single number conversion to  aseries of bits
Begin
    Bits := Empty
    For i := 1 To |x| Do
        Bits := concatenate (Bits, number_to_bits (x(i), bitcount))
    Endfor
End
```

Algorithm ***bits_to_solution*** turns a series of bits in to a series of nubers

```
Procedure bits_to_solution (Bits, numcount, bitcount) Returns x
    Bits() – a series of bits bitcount*numcount, that represents the solution
    numcount – how many numbers the solution contains
    bitcount – to how many bits including the sign will the number be turned
    x() – solution in the form of numbers
    bits_to_number – converting the series of bits in to a number
Begin
    For n := 1 To numcount Do
         % get a sequence of 'bitcount' bits
        bitportion := get interval of Bits from ((n-1)*bitcount+1) to (n*bitcount)
         % transform it  to a number
        x(n) := bits_to_number (bitportion, bitcount)
    Endfor
End
```

**Example to check the *solution_to_bits* and *bits_to_solution functions:***

Solution in the form of numbers:

| 0.1875 | -1.5469 | 0.9219 |
|---|---|---|

Corresponding series of bits:

| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

### 3.2.3. Checking the validity of the solution

One of the key component in genetic algorithms is the so called fitness function. To check the fitness (validity) of a solution, the solution itself should be first obtained.

Function ***run_model*** runs a model with input series x and gets the result as output number y.

```
Function run_model (x, w) Returns y
    x() – input data: x1, x2
    w() – model coefficients: w0, w1, w2
    y – answer provided by the model
Begin
    NET := w(0)
    For i := 1 To ||x|| Do  % for all  x`s
        NET := NET + x(i) * w(i)
    Endfor
    If NET < 0 Then
        y := 0
    Elseif NET > 1 Then
        y := 1
    Else
        y := NET
    Endif
End
```

In this case it is easier to instead of the fitness function get the error function, which is the exact opposite.

The error function means that all samples X are being run on all models, that are represented by the coefficient matrix W. Then the obtained results Y should be compared with awaited results D, geting a series of values E, that contains a single number for each model.

(Fitness function, that will be calculated later, will be the opposite of the error function).

```
Function error_function (X, W, D) Returns E
    X() – input data for all samples (1st dimension – samble, 2nd dimension – input)
    W() – parallel model coefficients (1st dimension – model number, 2nd dimension – coefficient)
    D() – awaited results (single number for each sample)
    E() – model error (single number for each model)
    run_model – running a single model on a single sample
Begin
    For i := 1 To ||W||Do  % for all models
        For k := 1 To ||X|| Do  % for all samples
            Y(k) := run_model (X(k), W(i))
        Enfor
         % model evaluation is the difference of the worst square sample and and the expected
         % (since small mistake means good evaluation, the max is the worst!)
        E(i) := max((Y-D)^2)
    Endfor
End
```

**Example for checking the *error_function* (3 input data files are used):**

X (*x.txt*) =

| | |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

W (*w.txt*) =

| | | |
|---|---|---|
| 1.5 | -0.5 | -0.5 |
| 2 | -1 | -1 |
| 1.5 | -0.25 | -0.5 |
| -0.3125 | -0.1406 | -0.4219 |
| 0.0313 | 0.375 | 0.0938 |

D (*x.txt*) =

| |
|---|
| 1 |
| 1 |
| 1 |
| 0 |

E (obtained results) =

| |
|---|
| 0.25 |
| 0 |
| 0.5625 |
| 1 |
| 0.93838 |

### 3.2.4. Partitioning of probabilities and choice of individuals relevant to the probability partitions

In genetic algorithms in the case of double genetic opperand individuals (solutions) for futher manipulation are taken by chance. The probability partiotioning is not even when choosing individuals, but individuals, that have been evaluated with a better mark have a higher chance of being picked.

If we know the value F() for an individual, then in general case the probability Pr() for picking each individual is calculated with the following formula:

$$Pr(i) = \frac{F(i)}{\sum_k F(k)}$$

Unfortunately such a formula only works when the result of fitness function is directly not invertedly dependent on the individual to be evaluated (in other words: the higher result represents the better quality). Because of the fact, that for each model we have calculated E() - the error function, we now need to get the fitness from the error. That can be achieved by raising the number e in the power of a negative number (error).

$$F(i) = \exp(-E(i))$$

When fitness value F is obtained, the probability partitioning is obtained by the above given formula:

```
Function compute_probabilities (E) Returns P
        E() – model error calculated by error function
        F() – model fitness
        P() – obtained cumulative probability series
Begin
         % calculating fitness from error
        For i := 1 To ||E|| Do
                F(i) := exp(-E(i))
        Endfor
         % probability calculation
        P := F / sum(F)
         % obtaining the cumulative probability
        For k := 2 To ||P||-1 Do  % in all probabilities except the outer ones
                P(k) := P(k) + P(k-1)
        Endfor
         % theoretically the above loop can also set this (last) element,
         % but this is used to avoid rounding mistakes:
        P(||P||) := 1  % the last element is set to 1
End
```

**Example for validation of *compute_probabilities* function (uses a single input data file).**

Given is a series of error values *E (e.txt)* =

| |
|---|
| 0.3906 |
| 0.3713 |
| 0.3713 |
| 0.3713 |
| 0.3906 |
| 1.0000 |
| 1.0000 |
| 1.0000 |
| 0.3713 |
| 1.0000 |

The obtained series of cumulative probabilities *P*:

| |
|---|
| 0.121173 |
| 0.244708 |
| 0.368242 |
| 0.491776 |
| 0.612949 |
| 0.678829 |
| 0.744708 |
| 0.810587 |
| 0.934121 |
| 1 |

When the series of cumulative probabilities is obtained, the element(here the solution or individual) choice at random is easily practicable according to the given probability partition (sometimes called the Monte Carlo method or Roulette method).:

The following algorithm describes the creation of a series of a certain size, using the given set according to the probability partition with the Roulette method (actually the main part of the

Roulette method is the obtaining of cumulative probabilities, but the roulette choice according to this series is fairly simple):

```
Function selection (Population, P, n) Returns Selection
        Population() – set of solutions
        P() – series of cumulative probabilities, that describes the elements of Population
        n – size of the needed selection
        Selection() – obtained selection
Begin
        Selection := Empty  % at the beginning the selection is an empty set
        While ||Selection|| < n
                % roulette choice
                r := random number limited by [0, 1].
                num := 1
                While r > P(num)
                        num := num + 1
                Endwhile
                Add Population(num) To Selection
        Endwhile
End
```

**Example for checking the *selection* function:**

Given is a series of potential solutions W (*w.txt*) =

| | | |
|---|---|---|
| 1.5 | -0.5 | -0.5 |
| 2 | -1 | -1 |
| 1.5 | -0.25 | -0.5 |
| -0.3125 | -0.1406 | -0.4219 |
| 0.0313 | 0.3750 | 0.0938 |

And an according cumulative probability series *P (p.txt)* =

| |
|---|
| 0.1 |
| 0.2 |
| 0.3 |
| 0.4 |
| 1.0 |

(to which the probabilities corresponds [0.1, 0.1, 0.1, 0.1, 0.6])

Run the selection function on this series 5 times with n=2. So from the range of five elements two will be chosen each time, which brings us to a total of 10. Statistically the most common should be the last individual (starts with 0.0313):

| | | |
|---|---|---|
| 0.0313 | 0.375 | 0.0938 |
| 2 | -1 | -1 |
| 0.0313 | 0.375 | 0.0938 |
| -0.3125 | -0.1406 | -0.4219 |
| 2 | -1 | -1 |
| 0.0313 | 0.375 | 0.0938 |
| -0.3125 | -0.1406 | -0.4219 |
| 1.5 | -0.5 | -0.5 |
| 0.0313 | 0.375 | 0.0938 |
| -0.3125 | -0.1406 | -0.4219 |

### 3.2.5. Genetic opperands

In this labwork 2 genetic opperands will be used: crossing and mutation of a single point.

The function ***single_point_crossover*** carries out the crossing of a point if 2 already chosen base individuals are given.

```
Function single_point_crossover (parent1, parent2) Returns child
    parent1, parent2 – individuals that the crossing affects
    bitcount – number of bits for the coding of model coefficients
    child – The individual obtained with the crossing
    solution_to_bits – tranforming the series of numbers in to a series of bits
    bits_to_solution – transforming the series of bits in to a series of numbers
Begin
    p1 := solution_to_bits (parent1, bitcount)
    p2 := solution_to_bits (parent2, bitcount)
    r := get random number from interval [1..||p1||-1]
    c := Concatenate (p1(1..r), p2(r+1..||p1||))
    child := bits_to_solution (c, bitcount)
End
```

**Example for evaluation of *single_point_crossover* function:**

Given is a series of potential solutions W (*w.txt*) =

| | | |
|---|---|---|
| 1.5 | -0.5 | -0.5 |
| 2 | -1 | -1 |
| 1.5 | -0.25 | -0.5 |
| -0.3125 | -0.1406 | -0.4219 |
| 0.0313 | 0.3750 | 0.0938 |

Run the *single_point_crossover* function on the top 2 individuals 5 times, which leads us to having one side form one parent and the other from the other one in the solution. Example:

| | | |
|---|---|---|
| 1.546875 | -1 | -1 |
| 1.984375 | -1 | -1 |
| 1.5 | -0 | -1 |
| 1.5 | -0.5 | -1 |
| 1.5 | -0.5 | -1 |

The ***mutation*** function changes a single bit in the chromosome of a fixed individual.

```
Function mutation (oldsolution) Returns newsolution
    oldsolution – solution before the mutation
    bitcount – the number of bits for coding the model coefficients
    newsolution – solution after the mutation
    solution_to_bits – turning a series of numbers in to a series of bits
    bits_to_solution – turning a series of bits in to a series of numbers
Begin
    s := solution_to_bits (oldsolution, bitcount)
    r := get random number from interval [1..||s||]
    s(r) := 1 - s(r)
    newsolution := bits_to_solution (s, bitcount)
End
```

**Example for validating the *mutation* function:**

Given is a series of potential solutions W (*w.txt*) =

| | | |
|---|---|---|
| 1.5 | -0.5 | -0.5 |
| 2 | -1 | -1 |
| 1.5 | -0.25 | -0.5 |
| -0.3125 | -0.1406 | -0.4219 |
| 0.0313 | 0.3750 | 0.0938 |

Run the *mutation* function on each solution getting 3 changed numbers in each solution as a result (the other 2 stay the same). Example:

| | | |
|---|---|---|
| 1.5 | -0.515625 | -0.5 |
| 1.984375 | -1 | -0 |
| 1.5 | -0.25 | -0.625 |
| -0.3125 | 0.125 | -0.421875 |
| 0.03125 | 0.375 | 1.09375 |

### 3.2.6. Summary of the genetic algorithm

```
Function GA (X, D) Returns Solution
        X() – input samples
        D() – awaited results
        popsize – size of the created population
        crossrate – count of the individuals in the population, that are changed(using the crossing) in
                each step
        mutrate – individual count, that will be subjected to mutation in each step
        maxit – the maximum amount of iterations
        bitcount – number of bits for coding the model coefficient
        ε – maximal allowed mistake for a model
        Solution – result
Begin
        ncount – the number of coefficients in a single solution (in our case 3, because the length of the
                input is 2)
        W(popsize, ncount) – an array of coefficients or solutions that represent the population
        W := Fills with random values in the range of [-0.5, 0.5]
        E := error_function (X, W, D)  % obtaining the model mistake
        it := 0  % the generation counter
        While it < maxit And min(E) > ε
                it := it + 1
                P := compute_probabilities (F)  % calculates the partition of the cumulative probability
                 % (A) automatically switches to the next generation (popsize-crossrate) elements
                WNEXT := selection (W, P, popsize - crossrate)
                 % (B) the remaining elements are obtained in the result of crossing
                WPARENTS := selection (W, P, crossrate×2)  % choosing the parents
                While WPARENTS ≠ Empty  % for each pair of parents
                        p1, p2 := two elements of WPARENTS are chosen
                        child := single_point_crossover (p1, p2)
                        Add child to WNEXT
                        Remove p1 and p2 from WPARENTS
                Endwhile
                 % (C1) Mutating a certain number of individuals
                WMUTATION := chooses random elements from WNEXT in the count of mutrate
                WMUTATION2 := Empty
                Forall e In WMUTATION Do
                        emut := mutation (e)
                        Add emut to WMUTATION2
                Endforall
                 % (C2)Changes the starting elements with the mutated ones
                WNEXT := WNEXT – WMUTATION  % operating with the set
                WNEXT := WNEXT Union WMUTATION2  % operating with the set
                 % (D) te next generation is evaluated and fixed
                W := WNEXT
                E := error_function (X, W, D)  % Obtains the model error
        Endwhile
        Solution := W(minindex (E), _)  % The best solution from the whole population
End
```

## *3.3. The summary of programming tasks in the labwork*

First lesson:

- Practicate the solution conversion to a series of bits and back (paragraph 3.2.2) with the parameter *bitcount*=8.
- Practicate the fitness function (paragraph 3.2.3).
- Practicate the probability partition (paragraph 3.2.4)

Second lesson:

- Practicate denetic opperands (paragraph **Error! Reference source not found.**)
- Practicate the genetic algorithm as a whole(paragraph 3.2.6) with paramters *popsize*=10, *crossrate*=4, *mutrate*=5, *maxit*=1000, *bitcount*=8, $\varepsilon$=0.1

# 5. Naive Bayes classifier

## 1.6. Problem formulation

### 1.6.1. Credit risk estimation examples

There are 14 examples given that allow estimation of credit risk by 4 attributes (2nd column) – low, medium or high. Attributes that are available are credit history, existing debt, recommendations and income (columns 3rd to 6th ).

| | *estimation* | *attributes* | | | |
|---|---|---|---|---|---|
| | **RISC** | **HIST** | **DEBT** | **RECM** | **INCOME** |
| 1. | HIGH | Bad | HIGH | No | Low |
| 2. | HIGH | UNKNWN | HIGH | No | MEDIUM |
| 3. | MEDIUM | UNKNWN | Low | No | MEDIUM |
| 4. | HIGH | UNKNWN | Low | No | Low |
| 5. | LOW | UNKNWN | Low | No | HIGH |
| 6. | LOW | UNKNWN | HIGH | YES | HIGH |
| 7. | HIGH | Bad | Low | No | Low |
| 8. | MEDIUM | Bad | Low | YES | HIGH |
| 9. | LOW | GOOD | Low | No | HIGH |
| 10. | LOW | GOOD | HIGH | YES | HIGH |
| 11. | HIGH | GOOD | HIGH | No | Low |
| 12. | MEDIUM | GOOD | HIGH | No | MEDIUM |
| 13. | LOW | GOOD | HIGH | No | HIGH |
| 14. | HIGH | Bad | HIGH | No | MEDIUM |

We'll code given samples with numbers like this:

- We'll code attributes as numbers (including target attribute) 1..$n$: <risc, hist, debt, recm, income>.
- Attribute values we'll code as numbers 1..$n_a$, where $n_a$ – the amount of attribute values:
  - values(hist) = <bad, unknown, good>
  - values(debt) = <low, high>
  - values(recm) = <no, yes>
  - values(income) = <low, medium, high>

Thereby given table can be overwritten like this (see files *x.txt* and *d.txt*):

| | *d.txt* | *x.txt* | | | |
|---|---|---|---|---|---|
| | **RISC** | **HIST** | **DEBT** | **RECM** | **INCOME** |
| 1. | 3 | 1 | 2 | 1 | 1 |
| 2. | 3 | 2 | 2 | 1 | 2 |
| 3. | 2 | 2 | 1 | 1 | 2 |
| 4. | 3 | 2 | 1 | 1 | 1 |
| 5. | 1 | 2 | 1 | 1 | 3 |
| 6. | 1 | 2 | 2 | 2 | 3 |
| 7. | 3 | 1 | 1 | 1 | 1 |
| 8. | 2 | 1 | 1 | 2 | 3 |
| 9. | 1 | 3 | 1 | 1 | 3 |
| 10. | 1 | 3 | 2 | 2 | 3 |
| 11. | 3 | 3 | 2 | 1 | 1 |
| 12. | 2 | 3 | 2 | 1 | 2 |
| 13. | 1 | 3 | 2 | 1 | 3 |
| 14. | 3 | 1 | 2 | 1 | 2 |

### *1.6.2.* Available data

*Desired values D* (*d.txt*) =

| |
|---|
| 3 |
| 3 |
| 2 |
| 3 |
| 1 |
| 1 |
| 3 |
| 2 |
| 1 |
| 1 |
| 3 |
| 2 |
| 1 |
| 3 |

*Input examples E* (*x.txt*) =

| | | | |
|---|---|---|---|
| 1 | 2 | 1 | 1 |
| 2 | 2 | 1 | 2 |
| 2 | 1 | 1 | 2 |
| 2 | 1 | 1 | 1 |
| 2 | 1 | 1 | 3 |
| 2 | 2 | 2 | 3 |
| 1 | 1 | 1 | 1 |
| 1 | 1 | 2 | 3 |
| 3 | 1 | 1 | 3 |
| 3 | 2 | 2 | 3 |
| 3 | 2 | 1 | 1 |
| 3 | 2 | 1 | 2 |
| 3 | 2 | 1 | 3 |
| 1 | 2 | 1 | 2 |

## *1.7. The principle of operation of Naive Bayes classifier*

Naive Bayes classifier operates according to formula:

$$v = \operatorname*{argmax}_{v_j \in V} P(v_j) \prod_i P(a_i \mid v_j),$$

where $P(a_i|v_j)$ – the probability that the attribute value $a_i$ is valid on condition that classified (target) value is $v_j$.

## *1.8. Naive Bayes classifier realization*

### 1.8.1. Classification of one sample

In case of Naive Bayes classifier system is not built and trained, which is used to recognize samples:

In recognition of each sample all training samples are used, that way training phase and usage phase merges in one. This principle simplifies algorithm but increases the amount of needed resources for usage of this classifier because in recogntition of each new sample it is necessary to go through all other training samples.

Function ***naive_bayes*** classifies one given sample:

```
Function naive_bayes (ex) returns decision
        ex – sample to recognize
        E() – training samples (see chapter 1.6.2)
        D() – training sample correct answers (see chapter 1.6.2)
        A() – attribute list, practically its tables' E columns
        decision() – attribute values (for each attribute exactly one)
Begin
        Forall Distinct currdec In D Do  % for all possible values of D
                % sample count with given decision currdec:
                C1 := count ({d in D where d = currdec})
                % count of all samples
                C2 := count (D)
                % decisions' currdec proportion in all samples:
                prob(currdec) := C1 / C2
                Forall currattr In A Do  % for all possible attributes
                        % sample count with given decision currdec and sample's ex given value currattr :
                        C3 := count ({(d,e) in (D,E) where d = currdec and e(currattr) = ex(currattr)})
                        % given decision's approximation of probability:
                        prob(currdec) := prob(currdec) * (C3 / C1)
                Endforall
        Endforall
        decision := maxindex (prob)
End
```

Example.

| Bad | Low | No | HIGH |
|---|---|---|---|

Or numerically

| 1 | 1 | 1 | 3 |
|---|---|---|---|

returns;

2 = MEDIUM

### 1.8.2. Testing classifier with training samples

To make sure how correctly classifier works with same training samples E, each of them is processed with classifier and results are checked with correct answers D.

**Procedure** ***test_naive_bayes***
*E()* – training samples (see chapter 1.6.2)
*D()* – correct answers for training samples (see chapter 1.6.2)
***naive_bayes*** – Naive Bayes classifier
**Begin**
**Forall** (*ex*,*d*) **In** (*E*,*D*)
*rez* := ***naive_bayes*** (*ex*)
**Print** *d*, *res*
**Enforall**
**End**

With given test samples ***test_naive_bayes*** returns this result (from desired answers only one sample's classification differs):

| **Desired value** | **Calculated value** |
|---|---|
| 3 | 3 |
| 3 | 3 |
| 2 | 2 |
| 3 | 3 |
| 1 | 1 |
| 1 | 1 |
| 3 | 3 |
| 2 | 2 |
| 1 | 1 |
| 1 | 1 |
| 3 | 3 |
| 2 | 3 |
| 1 | 1 |
| 3 | 3 |

## *1.9. The summary of programming tasks in the labwork*

1) Download files „x.txt” and „d.txt” to get tables *D* and *E* (see chapter 1.6.2)
2) Implement algorithm ***naive_bayes*** (chapter 0) and test it with some sample that's not in the training samples.

Check algorithm with training samples (chapter 1.8.2).

# 6. Ant colony optimization for TSP

## *1.10. Problem formulation – Travelling Salesman Problem*

Travelling salesman problem in its classical variant means:

Given:

- *n* cities,
- known distance from each city to each other,

Task:

- find shortest path in which each city is visited exactly once and it returns to starting city.

In the following table the problem is defined with *n*=20. Each city has its coordinates in plane (sample taken from [Abraham, Guo and Liu, 2006]:

| s | $x_s$ | $y_s$ |
|---|---|---|
| **1** | 5.2940 | 1.5580 |
| **2** | 4.2860 | 3.6220 |
| **3** | 4.7190 | 2.7740 |
| **4** | 4.1850 | 2.2300 |
| **5** | 0.9150 | 3.8210 |
| **6** | 4.7710 | 6.0410 |
| **7** | 1.5240 | 2.8710 |
| **8** | 3.4470 | 2.1110 |
| **9** | 3.7180 | 3.6650 |
| **10** | 2.6490 | 2.5560 |
| **11** | 4.3990 | 1.1940 |
| **12** | 4.6600 | 2.9490 |
| **13** | 1.2320 | 6.4400 |
| **14** | 5.0360 | 0.2440 |
| **15** | 2.7100 | 3.1400 |
| **16** | 1.0720 | 3.4540 |
| **17** | 5.8550 | 6.2030 |
| **18** | 0.1940 | 1.8620 |
| **19** | 1.7620 | 2.6930 |
| **20** | 2.6820 | 6.0970 |

Distance *d* or weigth between two states *s1 and s2* can be calculated using one of the metrics for example Euclidean distance:

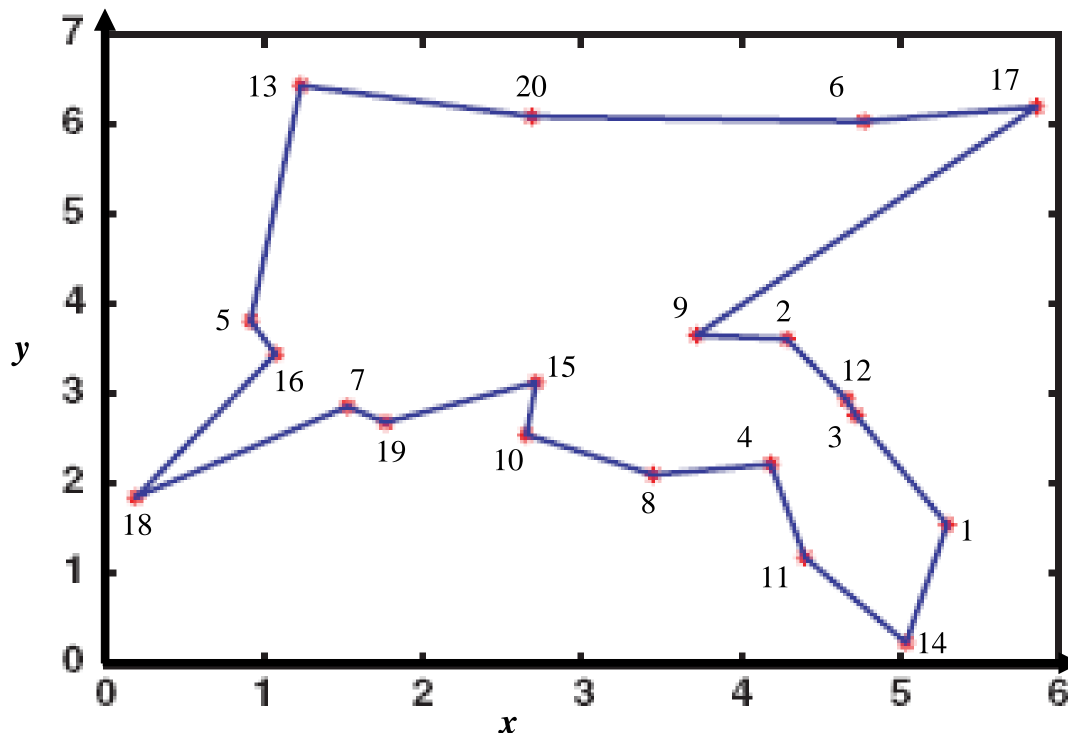
$$\mathrm{d}(s1, s2) = \sqrt{(x_{s1} - x_{s2})^2 + (y_{s1} - y_{s2})^2}$$

Tours *T* length *L* is equal to sum of all distances in tour:

$$L = \sum_{i=2}^{\|T\|} \mathrm{d}(i-1, i)$$

Good enough route here would be: 1 → 14 → 11 → 4 → 8 → 10 → 15 →19 → 7 → 18 → 16 → 5 → 13 → 20 → 6 → 17 → 9 → 2 → 12 → 3 → 1 (with total distance or weigth of 24.5222),

illustrated with drawing:




## 1.11. Realization of algorithm

### 1.11.1. Generation of tour

Tour is one ant's (agent's) full path through graph vertices visiting each exactly once and returning to starting vertice.

When solving the problem with ant colony's optimization algorithm we need opposite concept to distance - visibility$\eta$:

$$\eta(s1, s2) = \frac{1}{\mathrm{d}(s1, s2)}$$

In tour generation one of the most important elements is to choose next step (next vertice to visit in graph, in case of TSP city) For that we need to calculate probability for each possible vertice. Taking into account the estimated probability distribution we choose the next state:

```
Function next_state (s, S, N, τ, α, β) Returns statenext
        s – state from which we need to choose next step
        S() – set of all states (known distance between any two states)
        N() – subset of S containing states that are available for selection of next step
        τ – pheromone values for each transition from one state to another ||S|| × ||S||
        α – pheromone factor (to what extent we take into account fact that the edge is often visited)
        β – distance factor (to what extent we take into account fact that two states are close to each
            other)
        η(·,·) – visibility – inverse value of distance: η(s1,s2) = 1 ÷ d(s1,s2)
Begin
          % getting probabilty distribution
        For i := 1 To ||N|| Do  % for all available states (from given state)
                s2 := N(i)
                P(i) := τ(s,s2)^α * η(s,s2)^β
        Endfor
        P := P / sum(P)
          % the cumulative calculation for string of probabilties
        For i:=2 To ||P||-1 Do
                P(i) := P(i) + P(i-1)
        Endfor
        P(||P||) := 1
          % selection of state
        r := random number between [0, 1].
        num := 1
        While r > P(num)
                num := num + 1
        Endwhile
        statenext := N(num)  % returns next state
End
```

**Example for checking function *next_state* #1:**

Set of all states S(length 20) with distances between states is given (*tsp.txt*, see above)

Set of available states N (*n.txt*) is given =

| |
|---|
| 4 |
| 8 |
| 10 |

Given matrix τ with size ||S||×||S||, filled with 1.

s = 3; α = 2; β = 2

Call function *next_state* and record intermediate results (1) for probability distribution (P, before cumulation), (2) cumulative string of probabilties after cumulation.

Notice that states in N have different distances from 3rd state and probabilty to be chosen is higher for closer ones.

Result example.

Probability distribution:

| | | |
|---|---|---|
| 0.705943 | 0.199372 | 0.0946855 |

Cumulative string of probabilities:

| | | |
|---|---|---|
| 0.705943 | 0.905314 | 1 |

**Example for checking function *next_state* #2:**

Given same input data.

Call function *next_state* 20 times and record chosen next state.

Result, for example:

4 4 4 8 8 4 4 10 4 4 8 8 4 4 4 4 4 10 4 4

(on average statistically 4, rarest 10).

Tour is a string of states in which each state is represented once except starting state which is also the end state.

**Function *generate_tour*** **(*S*, *τ*, *α*, *β*) Returns** *episode*
*S* – set of all states (known distance between any two states)
*N* – subest of S iwt states that are avilable for selection of next step
τ – pheromone values for each transition from one state to another ||*S*|| × ||*S*||
α – pheromone factor (to what extent we take into account fact that the edge is often visited)
β – distance factor (to what extent we take into account fact that two states are close to each other)
*episode* – generated string of states
***next_state*** – calculation of next state
**Begin**
*startstate* := selection of starting state for episode (usually always the same)
*N* := *S* \ {*startstate*} *% available states for next step*
*episode* := <*startstate*>
*s* := *startstate*
**While** *N* **Is Not Empty**
*s* := ***next_state*** (*s*, *S*, *N*, *τ*, *α*, *β*)
Append *s* to *episode*
*N* := *N* \ {*s*}
**Endwhile**
Append *startstate* to *episode* *% episode starts and ends with the starting state*
**End**

**Example for checking function *generate_tour*:**

Given set of all states S with distances between states (*tsp2.txt*).

| s | $x_s$ | $y_s$ |
|---|---|---|
| **1** | 4.7190 | 2.7740 |
| **2** | 4.1850 | 2.2300 |
| **3** | 3.4470 | 2.1110 |
| **4** | 2.6490 | 2.5560 |

Given matrix τ with size ||S||×||S||, filled with 1.

α = 2; β = 2

Call function *generate_tour* 10 times (starting with state 1) and check generated episodes.

Result example (10 different tours, tour ends with starting state):

| 1 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 1 |
| 1 | 2 | 3 | 4 | 1 |
| 1 | 4 | 3 | 2 | 1 |
| 1 | 3 | 2 | 4 | 1 |
| 1 | 2 | 3 | 4 | 1 |
| 1 | 2 | 3 | 4 | 1 |
| 1 | 3 | 4 | 2 | 1 |
| 1 | 2 | 3 | 4 | 1 |
| 1 | 4 | 3 | 2 | 1 |

### 1.11.2. The summary of ant colony optimization algorithm

In ant colony algorithm pheromone is a concept that represents popularity of edge during search in graph.

```
Procedure ant_colony_optimization (maxit, popsize, α, β, ρ, Q, τ0) Returns best_tour
    S() – set of all states
    maxit – total count of campaigns
    popsize – ant count in experiment
    α – pheromone factor (to what extent we take into account fact that the edge is often visited)
    β – distance factor (to what extent we take into account fact that two states are close to each
        other)
    ρ – pheromone evaporation factor
    Q – pheromone change calculation coefficient
    τ() – pheromone table with size ||S|| × ||S|| (one value for each transition of states)
    τ0 – starting value of pheromones
    L(·) – tour's length
    generate_tour – generation of one tour for one ant
Begin
    τ := whole pheromone table is filled with small values τ0
    For it:=1 To maxit  % for all epoches
        % (A) generates tours for all ants, T
        For p:=1 To popsize  % for all ants
            T(p) := generate_tour (S, τ, α, β)
        Endfor
        % (B) applies pheromone evaporation to all transitions of states
        Forall s,s2 ∈ S Do % for all pairs of points
            τ(s,s2) := (1 – ρ) · τ(s,s2)
        Endforall
        % (C) complements pheromone with last generated tours' information
        For p:=1 To popsize  % for all ants (tours made by ants)
            Forall (s,s2) In T(p)  % for all others in tour
                τ(s,s2) := τ(s,s2) + Q / L(T(p))
            Endforall
        Endfor
    Endfor
    best_tour := shortest tour encountered (with smallest L)
End
```

## *1.12. The summary of programming tasks in the labwork*

3) Download file „tsp.txt”, with coordinates of each state (chapter **Error! Reference source not found.**)
4) Realize generation of one step ***next_state*** (chapter 1.11.1).
5) Realize generation of episode ***generate_tour*** (chapter 1.11.1).
6) Realize ant colony algorithm ***ant_colony_optimization*** with parametres *popsize*=20, $\alpha$=1.3, $\beta$=2.5, *maxit*=100, $\rho$=0.5, *Q*=0.5, $\tau_0$=0.001
7) Execute algorithm 50 times, with very high likelihood there will be solution (tour) with length $L < 26$.

The best tour I have gotten (with given configuration) is
{1,14,11,4,8,10,15,7,19,18,16,5,13,20,6,17,9,2,12,3,1} with length L= 24.7954.

# 7. Support vector machine (*SVM*)

Support vector machine (Vapnik, 1995) (from now on SVM) is used for solving the problem of classification, and in its classic form is used specifically for binary classification (namely, into two classes), which will also be used here. In this lab work a simplified version of iterative SVM is offered, the idea of which has been borrowed from (Roobaert, 2000) and (Vishwanathan and Murty, 2002).

The lab work consists of two parts:

1) The creation of an iterative SVM for solving linear problems (sections 7.1 and 7.2)

2) Conversion of the SVM for solving non-linear problems (sections **Error! Reference source not found.** and 7.4)

In this SVM one linear perceptron neuron is used as the base learning system.

## *7.1. Definition of two linear problems for a SVM*

### 7.1.1. Problem #1. Modeling the Boolean function 'AND'

4 examples are given, describing a logical function ***($x_1$ AND $x_2$)***. A characteristic of SVM is such that the desired values, which correspond to these two classes are {-1, 1}. (**NB!** In this examples notice the little bit „strange" order of the examples – positive values first, followed by negative values)

| $x_1$ | $x_2$ | d |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 1 | -1 |
| 1 | 0 | -1 |
| 0 | 0 | -1 |

Model the function in 'AND' form.

$$\mathrm{F}(x_1, x_2) = \begin{cases} 1; & NET > 0 \\ -1; & else \end{cases}, \text{ where}$$

$$NET = w_0 + w_1 x_1 + w_2 x_2$$

Additional condition.

Model the function in a more „convincing" way, meaning that NET values are not close to 0:

$$\mathrm{F}(x_1, x_2) = \begin{cases} 1; & NET > 1-\xi \\ -1; & NET < -1+\xi \end{cases}, \text{ where}$$

$\xi$ (greek letter *xi*) – slack rate. With $\xi$=1 the definition would correspond more to the initial version.

**Available data.**

*posand.txt* and *negand.txt* – examples of positive and negative learning

- *POS* – input data (*posand.txt*):

| | |
|---|---|
| 1 | 1 |

- *NEG* – input data (*negand.txt*):

| | |
|---|---|
| 0 | 1 |
| 1 | 0 |
| 0 | 0 |

The input information can be defined differently:

- *D* – desired result:

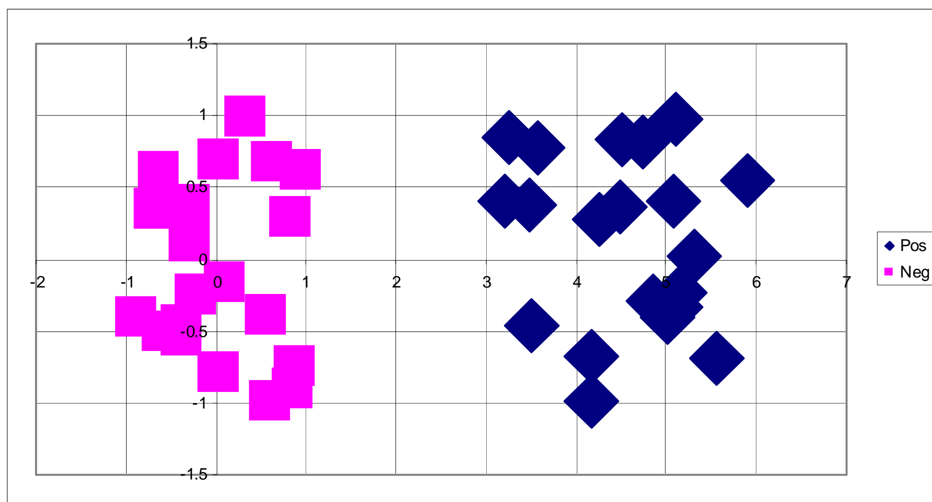
| |
|---|
| 1 |
| -1 |
| -1 |
| -1 |

*E* – input data:

| | |
|---|---|
| 1 | 1 |
| 0 | 1 |
| 1 | 0 |
| 0 | 0 |

### 7.1.2. Problem #2. The distribution of linear points in a plane

40 learning examples are given (*pos1a.txt* un *neg1a.txt* – positive and negative learning examples). For problem #2 only the input data graph is given:




Model the function the same as in the case of AND.

## 7.2. Realization of the iterative SVM algorithm in the case of a linear problem

To create the linear algorithm, three modules have to be built:

1) Running the perceptron ***run_perceptron***

2) Training the perceptron ***train_perceptron***

3) Common module ***svm_simple***, which calls both of the previous

### 7.2.1. Running a single example with a linear perceptron

The function ***run_perceptron*** runs the neural network W with a single example *X*.

```
Function run_perceptron (X, W) Returns y
        X – input example: x1.. xn
        W – model weight: w0..wn
        y – given result of the model
Begin
        NET := W(0)
        For i := 1 To |X| Do  % for all X
                NET := NET + X(i) * W(i)
        Endfor
        If NET < -1 Then y := -1
        Elseif NET > 1 Then y := 1
        Else y := NET
        Endif
End
```

Test data for the function ***run_perceptron***:

W ($w_0$, $w_1$, $w_2$) =

| | | |
|---|---|---|
| -2.5 | 1.9 | 1.7 |

*X(4)* – input data ($x_1$, $x_2$):

| | |
|---|---|
| 1 | 1 |
| 0 | 1 |
| 1 | 0 |
| 0 | 0 |

Calling the function ***run_perceptron*** 4 times for each of the input data, 4 results are given:
1
-0.8
-0.6
-1

### 7.2.2. Training a linear perceptron on a set of examples

The function ***train_perceptron*** fully trains a neural network on a given set of examples (*E*, *D*).

```
Function train_perceptron (E, D) Returns W
        E – input data
        D – the correct values of input examples
        W– obtainable weights
        MaxEpochs– maximum number of network training epochs
        ε – maximum network error
        LearningRate –learning coefficient
Begin
        W – initializes the weights with values [-0.3, 0.3]
        epoch := 0
        neterror := ∞
        While neterror > ε And epoch < MaxEpochs Do  % training cycle
                epoch := epoch + 1
                neterror := 0
                For k := 1 To |E| Do  % for each example (here only for support vectors!)
                        y := run_perceptron (E(k), W)  % ordinary network output
                        δ := D(k) – y  % error of example or difference from correct answer
                        W(0) := W(0) + LearningRate * δ  % additional weight training
                        For i := 1 To |E(k)| Do  % for weights
                                W(i) := W(i) + LearningRate * δ * E(k,i)
                        Endfor
                        neterror := neterror + δ * δ
                Endfor
                neterror := sqrt (neterror)
        Endwhile
End
```

Test data and configuration for the function ***train_perceptron***:

*E* – input data:

| | |
|---|---|
| 1 | 1 |
| 0 | 1 |
| 1 | 0 |
| 0 | 0 |

*D* – desired answers:

| |
|---|
| 1 |
| -1 |
| -1 |
| -1 |

MaxEpochs = 1000

ε = 0.01

LearningRate = 0.1

**Example of obtainable string of weights:**

W ($w_0$, $w_1$, $w_2$) =

| | | |
|---|---|---|
| -2.9 | 1.9 | 1.9 |

Approximate number of epochs necessary for convergence with parameters of 150.

### 7.2.3. Summary of iterative linear SVM algorithm

The algorithm ***svm_simple*** is a simplified iterative version of the SVM algorithm.

```
Procedure svm_simple (E, D) Returns W
        E – input data
        D – correct values of input examples
        W– obtainable weights
        SVMERR – error of the machine
        SlackRate – slack coefficient
        train_perceptron – training a neural network on all examples
        run_perceptron – running a neural network once on a single example
Begin
         % A) determines the initial list of support vectors
        S := {sp,sn}  % from E two closest points in opposite classes are chosen (using Euclidean
                                                       metric)
        SD – the correct answers of the support vectors
        Loop Forever
                 % B) train the perceptron and evaluate the result
                W := train_perceptron (S, SD)  % train (only) on the support vectors
                For e := 1 To |E| Do  % run the network on all examples, not only support vectors
                        y := run_perceptron (E(e), W)  % ordinary network output
                        SVMERR(e) := 1 – SlackRate – y * D(e);
                Endfor
                 % C) find the worst example from those, which are not support vectors and add to them
                If max(SVMERR) > 0 And |S|<|E| Do  % If there are bad examples and there are still
                                                       support vectors
                        e := maxindex_i (SVMERR(i)) Where E(i) Not In S
                        S := S + E(e)
                        SD – update according to new S
                Else
                        Return
                Endif
        Endloop
End
```

The running conditions are in the next section.

### 7.2.4. System configuration and obtainable result

The algorithm svm_simple is run with such parameters:

MaxEpochs = 1000

ε = 0.01

LearningRate = 0.1

SlackRate = 0.5

For both problems 3 support vectors suffice.

**Solving Problem #1**

Obtained string of weights W ($w_0$, $w_1$, $w_2$) =

| -3 | 2 | 2 |
|---|---|---|

Used support vectors (3 from 4 possible):

1110

The decoded support vectors (using files *posand.txt* and *negand.txt*):

(1,1)

(0,1)

(1,0)

Approximate number of epochs necessary for both SVM iterations – 200.

**Solving Problem #2**

Obtained string of weights W ($w_0$, $w_1$, $w_2$) =

| -1.7829466 | 0.86973058 | -0.03622049 |
|---|---|---|

Used support vectors (3 of 40 possible, one positive, but 2 negative examples, as can be seen):

00000000100000000000 00010010000000000000

The decoded support vectors (using files *pos1a.txt* and *neg1a.txt*):

Left-most positive one:

(3.212471537, 0.408951459),

Right-most negatives ones:

(0.935840634, 0.615499651)

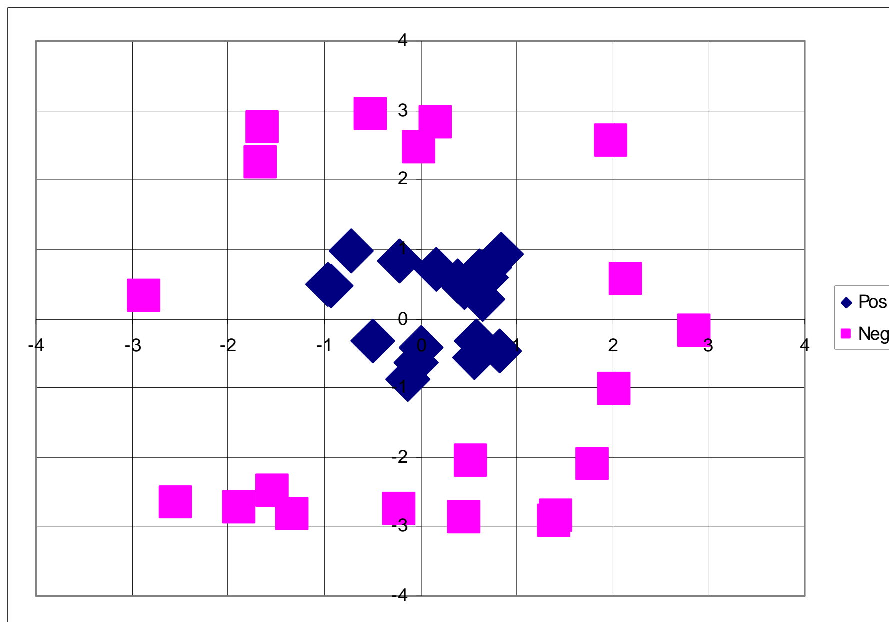
(0.863338348, -0.906680607)

Approximate number of epochs necessary for both SVM iterations – 70.

## 7.3. Definition of a non-linear problem for a SVM

**Problem #3. Distribution of non-linear points in a plane**

40 examples are given (*pos2a.txt* and *neg2a.txt*), where the positive examples are not separable from the negative examples in a linear way:




Model the function as before, but with additional conditions against the structure of the training system.

Since both classes are not seprable in a linear way, the input example has to be converted to a linearly separable plane of points before training.

For this objective *kernel functions* are usually used un thus it can be said that the conversion happens from the *input space* to the *kernel space* and the training happens on data from the kernel space.

The kernel function Κ (greek letter *kappa*) is an element of the conversion mechanism, which has two parameters, both vectors – one of which is a support vector.

One of the most used types of kernels is the radial base function kernel (*RBF kernel*), which is defined like this:

$$\mathrm{K}(x, y) = \exp\left(-\frac{\|x - y\|^2}{\sigma^2}\right),$$

where σ – slope coefficent, ||x-y|| – Euclidean distance:

$$\|x - y\| = \sqrt{\sum_i (x_i - y_i)^2}$$

To convert the example input, the kernel function has to be called the number of times of defined support vectors.

The conversion function Φ in our case with 2 inputs looks like this:

$$\Phi(x) = \Phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} \mathrm{K}(s_1, x) \\ \mathrm{K}(s_2, x) \\ \cdots \\ \mathrm{K}(s_m, x) \end{pmatrix},$$

where $x$ – convertable example, $s_1,...,s_m$ – support vectors.

## *7.4. Realization of an iterative SVM algorithm in the case of a linear problem*

The difference from non-linear case is the introduction of a **conversion function**, which converts input data in a linearly separable form.

### 7.4.1. Realization of the conversion function

The function ***phi_mapping*** converts an example into an example, which is in the length of the number of support vectors.

```
Function phi_mapping (x, S) Returns y
        x – input example: x1.. xn
        S – String of support vectors: S0..Sm
        y – converted example
Begin
        y.resize (|S|)
        For i := 1 To |S| Do  % for each support vector
                y(i) := K(Si, x)
        Endfor
End

Function K (a, b) Returns y
        a – input example #1
        b – input example #2
        σ (sigma) – RBF slope coefficent
Begin
        NET := 0
        For i := 1 To |a| Do  % for each element of the examples
                NET := NET + (ai – bi)^2
        Endfor
        y := exp (-NET / (2 * σ^2))
End
```

Testing data for the function ***phi_mapping***:

**Problem #1: modeling AND**

σ = **0.3**

$X$ – input data (*posand.txt* + *negand.txt*):

| | |
|---|---|
| 1 | 1 |
| 0 | 1 |
| 1 | 0 |
| 0 | 0 |

These support vectors $S$ are given (first 3):

(1,1)

(0,1)

(1,0)

Calling the function ***phi_mapping*** 4 times for each of the input data, gives these 4 results:

| | | |
|---|---|---|
| 1 | 0.0038659201 | 0.0038659201 |
| 0.0038659201 | 1 | 1.4945339e-005 |
| 0.0038659201 | 1.4945339e-005 | 1 |
| 1.4945339e-005 | 0.0038659201 | 0.0038659201 |

**Problem #3: Separation of non-linear points**

σ = 3

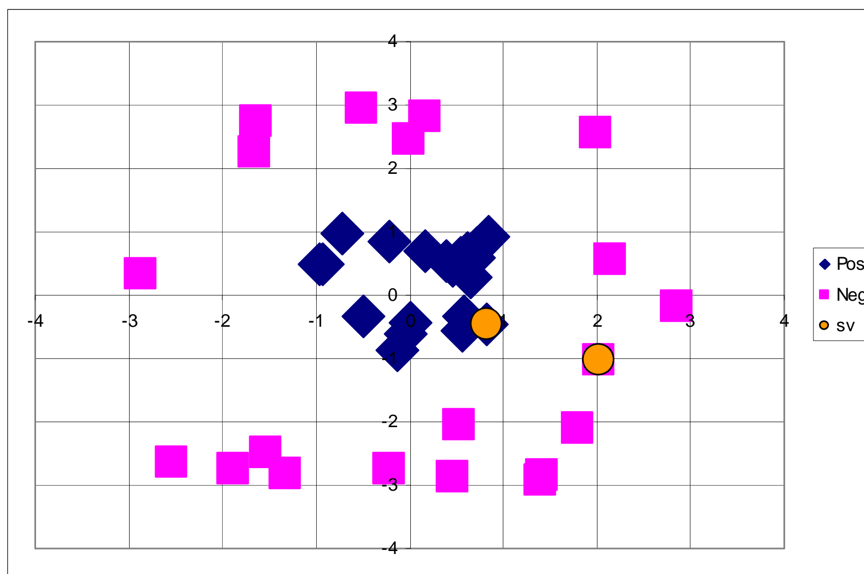
*X* – input data (*pos2a.txt* + *neg2a.txt*):

These support vectors *S* are given (#13 un #35):

(0.831363001, -0.473996589)

(2.021344674, -1.016343516)

The corresponding graph for the input data, specifically highlighting (the two) support vectors:

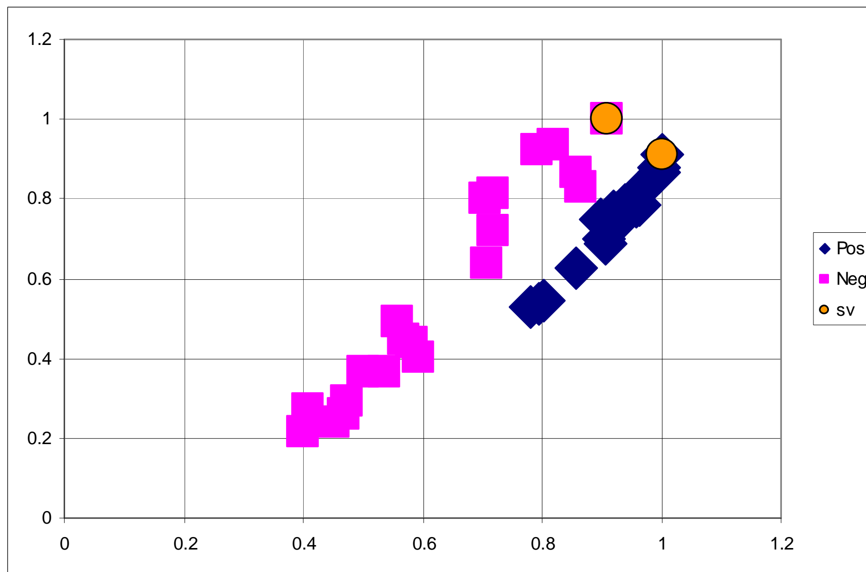



Calling the function ***phi_mapping*** 40 times for each of the input data, yields 40 results (support vectors are highlighted in yellow):

| | |
|---|---|
| 0.79481 | 0.5382 |
| 0.77974 | 0.52986 |
| 0.93404 | 0.76686 |
| 0.89674 | 0.75023 |
| 0.93424 | 0.75388 |
| 0.9036 | 0.70072 |
| 0.94634 | 0.77534 |
| 0.80192 | 0.54624 |
| 0.96337 | 0.78393 |

| | |
|---|---|
| 0.93864 | 0.78583 |
| 0.99544 | 0.87854 |
| 0.95677 | 0.78169 |
| 1 | 0.90936 |
| 0.96674 | 0.81988 |
| 0.99538 | 0.86823 |
| 0.92084 | 0.76726 |
| 0.90699 | 0.6868 |
| 0.85583 | 0.62674 |
| 0.94156 | 0.77203 |
| 0.92651 | 0.76385 |
| 0.86565 | 0.83115 |
| 0.71705 | 0.81266 |
| 0.79213 | 0.92524 |
| 0.40927 | 0.27167 |
| 0.46813 | 0.25972 |
| 0.55659 | 0.49066 |
| 0.70675 | 0.64062 |
| 0.45139 | 0.23985 |
| 0.58332 | 0.43801 |
| 0.70368 | 0.79955 |
| 0.59318 | 0.40384 |
| 0.82021 | 0.93446 |
| 0.47225 | 0.29252 |
| 0.56987 | 0.44853 |
| 0.90936 | 1 |
| 0.40076 | 0.21644 |
| 0.50062 | 0.36477 |
| 0.85602 | 0.86821 |
| 0.5355 | 0.36591 |
| 0.71931 | 0.72005 |

The corresponding graph after using the conversion function, specifically highlighting support vectors:




## 7.4.2. Summary of the non-linear iterative SVM algorithm

To convert the linear and non-linear SVM, a few small changes have to be made, assuming that a conversion function has been made. The essence of the first 3 (of 4) changes is that in the place of the (ordinary) examples transformed examples are used.

**Change #1**

In the function ***train_perceptron*** the line:

y := ***run_perceptron*** (*E*(k), *W*) % ordinary network output

is changed to

y := ***run_perceptron*** (***phi_mapping*** (*E*(*k*), *S*), *W*) % ordinary network output,

where *S* – string of support vectors

**Change #2**

In the function ***train_perceptron*** the line:

*W*(*i*) := *W*(*i*) + *LearningRate* * δ * *E*(*k*,*i*)

is changed to

*W*(*i*) := *W*(*i*) + *LearningRate* * δ * ***phi_mapping*** (*E*(*k*), *S*)(*i*),

where *S* – string of support vectors

**Change #3**

In the function ***svm_simple*** the line:

*y* := ***run_perceptron*** (*E*(*e*), *W*) % ordinary network output

is changed to

y := ***run_perceptron*** (***phi_mapping*** (*E*(*e*), *S*), *W*) % ordinary network output

**Change #4**

In the function ***svm_simple*** in the beginning before setting the values of the weights this is added:

*W*.***resize*** (|*S*|+1) *% because weights are with numbers 0..|S|*

### 7.4.3. Configuration of a non-linear system and obtainable result

**Solving Problem #3**

MaxEpochs = 1000

ε = 0.1

LearningRate = 0.5

SlackRate = 0.95

σ = 3.0

5 support vectors suffice here but one more weight is needed, because kernel functions are used for input data conversion.

Obtained string of weights W ($w_0$, $w_1$, $w_2$, $w_3$, $w_4$, $w_5$) =

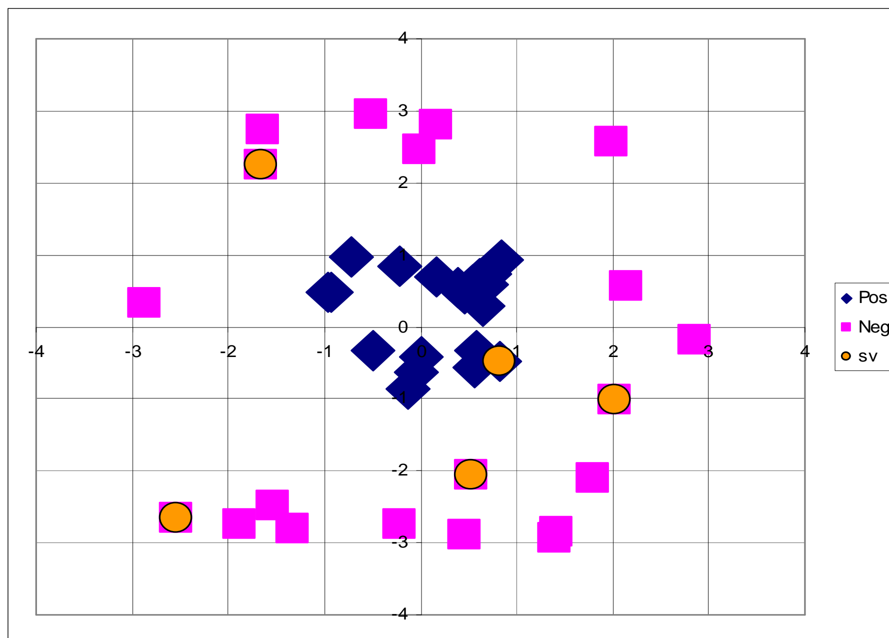
| -12.342 | 12.0777 | -2.242 | -1.0837 | 6.0707 | 0.91796 |
|---|---|---|---|---|---|

Used support vectors (5 of 40 possible, one of the positive ones, 2 of the negative ones, as can be seen):

00000000000010000000 10011000000000100000

Approximate number of epochs necessary for SVM iterations – 200.

Input data graph with all 5 support vectors, for insight:




Aleksandrs Polocks generously offered the generated result of his program, including the non-linear separation line (support vectors are squares with no filling):

# 8. Partial observability and POMDP. Tiger's problem

## 8.1. Problēmas formulējums

Viena uzdevuma instance (epizode) ir tāda, ka aģentam priekšā ir divas durvis, kur aiz vienām no durvīm atrodas tīģeris ar vienādu varbūtību būt aiz katrām no durvīm.

Speciāli pievienojot beigu stāvokli (problēmu var noformulēt ar un bez beigu stāvokļa), var teikt, ka problēmai ir trīs stāvokļi:

$$S = \{s_{TL}, s_{TR}, s_{end}\},$$

kur *TL* – tiger left, *TR* – tiger right.
Tātad katra epizode sākas ar stāvokļa izvēli, katra (izņemot beigu stāvokli) ar varbūtību 50%, bet aģents nezina, kurš stāvoklis ir izvēlēts.
Aģentam ir pieejamas trīs darbības:

$$A = \{a_{OLEFT}, a_{ORIGHT}, a_{LISTEN}\},$$

kur OLEFT – open left, ORIGHT – open right.

Izvēloties atvērt durvis, epizode beidzas, bet izvēloties klausīties, epizode turpinās un tiek iegūts viens no diviem novērojumiem (sk. tabulu 8.1):

$$Z = \{z_{TL}, z_{TR}\},$$

kur *TL* – tiger left, *TR* – tiger right.

Tabula 8.1. Darbības un epizodes.

| Darbība | |
|---|---|
| $a_{OLEFT}$ = veru kreisās durvis | Pāriet uz stāvokli $S_{fin}$: epizode beidzas |
| $a_{ORIGHT}$ = veru labās durvis | Pāriet uz stāvokli $S_{fin}$: epizode beidzas |
| $a_{LISTEN}$ = klausos | Paliek tai pašā stāvoklī ($S_{TL}$ vai $S_{TR}$) |

Ja aģents atver durvis, aiz kurām ir tīģeris, tas saņem sodu (-100), ja atver otras durvis, tad atlīdzību (+10) un, bet ja klausās (listen), tad tas maksā (-1) (sk. tabulu 8.2).

Tabula 8.2. Atlīdzības tīģera problēmai.

| Darbība / stāvoklis | $S_{TL}$ = Tīģeris IR pa kreisi | $S_{TR}$ = Tīģeris IR pa labi |
|---|---|---|
| $a_{OLEFT}$ = veru kreisās durvis | -100 | +10 |
| $a_{ORIGHT}$ = veru labās durvis | +10 | -100 |
| $a_{LISTEN}$ = klausos | -1 | -1 |

Papildus problēma ir tāda, ka, izvēloties klausīšanos, aģents tikai ar 85% varbūtību dzird tīģeri pareizajā pusē, tātad ar 15% varbūtību dzird nepareizajā (sk. tabulu 8.3).

Tabula 8.3. Varbūtība dzirdēt tīģeri aiz sienas.

| Novērojums / stāvoklis | $S_{TL}$ = Tīģeris IR pa kreisi | $S_{TR}$ = Tīģeris IR pa labi |
|---|---|---|
| $z_{TL}$ = Izklausās pa kreisi | 85% | 15% |
| $z_{TR}$ = Izklausās pa labi | 15% | 85% |

Aģenta uzdevums ir izrēķināt savas rīcības politiku ilgtermiņā, – lai daudzu epizožu gaitā tiktu iegūta maksimālā summārā atlīdzība. Pēc būtības, tas nozīmē, ka aģentam ir jāsaprot, cik reizes būtu jāklausās, pirms atvērt durvis (2, 3, vai vairāk), turklāt situāciju sarežģī tas, ka aģents pareizi dzird tikai ar varbūtību 85%, un tāpēc ir iespējams, ka vairākas reizes pēc kārtas klausoties, tiks iegūti dažādi novērojumi.

## 8.2. Ticamības stāvokļu koka būvēšana

Ņemot vērā to, ka vides stāvokļi nav tiešā veida novērojami, RL algoritmi uz tiem tiešā veidā nav izmantojumi. Tiks uz būvēts speciāls ticamības stāvokļu (*belief states*) koks. Problēmas vienkāršības dēļ mēs varam atļauties būvēt pilnu koku, tādu, kas satur visas iespējamās politikas līdz fiksētam dziļumam (horizontu skaitam), kas vispārīgā gadījumā nav iespējams, tomēr dod zināmu priekšstatu par problēmas specifiku.

Ticamības stāvokļu koks vienkāršākajā nozīmē reprezentē visas iespējamās gājienu un tā rezultātā radušos novērojumu secības.

Ticamības stāvokļi aizstās stāvokļus, un katrā ticamības stāvoklī tiks izrēķināts, ar kādu varbūtību tajā iespējams katrs no “īstajiem” stāvokļiem. Turklāt ticamības stāvokļi (izņemot tos, kas būs koka lapas) saturēs norādes uz “nākošajiem” ticamības stāvokļiem, no dotā ticamības stāvokļa izdarot attiecīgo darbību $a$ un fiksējot novērojumu $z$.

Tāpat kā “parasto” stāvokļu gadījumā, arī katrs ticamības stāvoklis saturēs darbību vērtību (*action values*) komplektu, kuru uzstādīšana arī būs apmācības uzdevums.

Nākamajā attēlā redzams ticamības stāvokļu koka fragments ar augstumu 3 (tiek rēķināti maksimums 3 gājieni uz priekšu).

Viss apmācības process būtu iedalāms 3 fāzēs:

1) Ticamības stāvokļu koka tehniska izveidošana iepriekš noteiktajā dziļumā.
2) Katra ticamības stāvokļa ticamības vērtību izrēķināšana katram stāvoklim (*Beliefs*) un paša ticamības stāvokļa iestāšanās varbūtības izrēķināšana (*Probability*).
3) Darbību vērtību izrēķināšana (*Action-values*)

Pirmās divas fāzes tiks izdarītas ar kopēju procedūru, kur koka veidošanā tiks izmantota potenciāli veidojamā mezgla varbūtība – ja tā ir pārāk maza (zem noteikta sliekšņa), tad koks šajā virzienā dziļumā vairs netiek veidots. Līdz ar to koka apjoms tiek ierobežots divos veidos:

1) Iepriekš noteiktais koka maksimālais augstums;
2) Pārāk maza jaunveidojamā mezgla (ticamības stāvokļa) iestāšanās varbūtība.

Gan ticamības vērtības (*Beliefs*), gan mezglu varbūtības tiek rēķinātas pēc līdzīgas formulas, tāpēc to rēķināšanai izmantosim kopīgu jēdzienu RawBeliefs (ticamības vērtībām ir lokāla jēga – kāda ir katra stāvokļa varbūtība, pieņemot, ka **ir izdevies tikt** līdz dotajam ticamības stāvoklim, bet mezglu varbūtībām ir globāla jēga – kāda varbūtība ir **vispār** līdz šim ticamības stāvoklim tikt).

| | tiger left | tiger right | end |
|---|---|---|---|
| *Beliefs* | 0.5 | 0.5 | 0 |

| | open left | open right | listen |
|---|---|---|---|
| *Action values* | 0 | 0 | 0 |

| | |
|---|---|
| *Probability* | 1.0 |
| *Level* | 0 |
| *Steps left* | 3 |

| next belief state | | | | |
|---|---|---|---|---|
| open left | open right | listen | | |
| end | end | tiger left | tiger right | end |

*Pašā sākumā (0-tajā līmenī) divi pamatstāvokļi (ne-beigu stāvokļi) ir ar vienādu varbūtību*

*Ja tiek izvēlēta darbība ‘listen’ un tiek novērots ‘tiger right’, nonākam nākamajā ticamības stāvoklī*

| | tiger left | tiger right | end |
|---|---|---|---|
| *Beliefs* | | | |

| | open left | open right | listen |
|---|---|---|---|
| *Action values* | 0 | 0 | 0 |

| | |
|---|---|
| *Probability* | |
| *Level* | 1 |
| *Steps left* | 2 |

| next belief state | | | | |
|---|---|---|---|---|
| open left | open right | listen | | |
| end | end | tiger left | tiger right | end |

*Beliefs*

*Action values*

*Probability*

| | |
|---|---|
| *Level* | 2 |
| *Steps left* | 1 |

Tehniskais problēmas formulējums vairākdimensiju masīvu formā – stāvokļu pāreju shēma T, atlīdzību shēma R un novērojumu shēma O (salīdzināt ar problēmas formulējumu sākumā, ir nelielas atšķirības novērojumu shēmas aprakstā) – tiks izmantots tālākajos algoritmu aprakstos.

```
Description of the Tiger's problem
    # stāvokļi s: 0-tiger left, 1-tiger right, 2-end
    # darbības a: 0-open left, 1-open right, 2-listen
    # novērojumi z: 0-tiger left, 1-tiger right, 2-end
    InitialBelief = [0.5, 0.5, 0]
    Terminal state: 2
    # T(s,a,s') – kāda varbūtība, ka stāvoklī s izdarot darbību a, nonāksim stāvoklī s'.
    T[0] = [
        [0, 0, 1],
        [0, 0, 1],
        [1, 0, 0]
    ]
    T[1] = [
        [0, 0, 1],
        [0, 0, 1],
        [0, 1, 0]
    ]
    T[2] = [
        [0, 0, 1],
        [0, 0, 1],
        [0, 0, 1]
    ]
    # O(s',a,z) – kāda varbūtība, atrodoties stāvoklī s', nonākot tur ar darbību a, novērot z.
    O[0] = [
        [0, 0, 1],
        [0, 0, 1],
        [0.85, 0.15, 0]
    ]
    O[1] = [
        [0, 0, 1],
        [0, 0, 1],
        [0.15, 0.85, 0]
    ]
    O[2] = [
        [0, 0, 1],
        [0, 0, 1],
        [0, 0, 1]
    ]
    # R(s,a) – kāda ir atlīdzība, stāvoklī s izvēloties darbību a
    R = [
        [-100, 10, -1],
        [10, -100, -1],
        [0, 0, 0]
    ]
```

Ticamības stāvokļu koka izveide sākas ar saknes mezgla izveidi, kuram uzstāda sākotnējās, iepriekš zināmās ticamības vērtības (šeit [0.5, 0.5, 0]), bet mezgla varbūtība ir 1.0 (t.i., sākumā vienmēr būsim šajā ticamības stāvoklī). Pēc tam koks tiek būvēts rekursīvi, katrā nākamajā mezglā izrēķinot ticamības vērtības un varbūtību pēc tā vecāka mezgla (*parent node*) attiecīgajām vērtībām, kā arī stāvokļu pārejām (T) un novērojumu shēmas (O).

Algoritms ***create_belief_tree*** ticamības stāvokļu koka izveidošanai

```
Function create_belief_tree (maxheight)
    maxheight – maksimālais koka augstums
    InitialBelief – sākotnējās ticamības vērtības, dotas kā problēmas sastāvdaļa
Begin
    Root = new node  % izveido koka sakni
    Root.Level = 0 % līmenis
    Root.StepsLeft = maxheight % cik darbības no šī mezgla iespējamas
    Root.ActionValues = [0 for each action]
    create_subtree (Root,InitialBelief)  % izveido visu koku
    return Root
End

Procedure create_subtree (Node,Bel)
    Epsilon – mazākā iespējamā pieļaujamā varbūtība, kurai būvē mezglu kokā
Begin
    Node.RawBelief =Bel % sākotnējās ticamības vērtības
    Node.Probability = sum(Node.RawBelief)
    Node.Belief = Node.RawBelief / Node.Probability
    If Root. StepsLeft > 1 Then  % nav lapa
        Forall possible actions a Do
            Forall possible observations z Do
                CandidateBelief := child_belief(z,Node.Belief,a)
                If sum(CandidateBelief) > Epsilon Then
                    s = argmax(CandidateBelief)
                    If max(CandidateBelief) 1- Epsilon or s is not terminal state Then
                        newnode = new node
                        newnode.Level = Node.Level+1
                        newnode. StepsLeft = Node. StepsLeft -1
                        newnode.ActionValues = [0 for each action]
                        Node.Children[a][z] = newnode
                        create_subtree (newnode,CandidateBelief)
                    Endif
                Endif
            Endforall
        Endforall
    Endif
End

Function child_belief (z,parentbelief,a)
    T – stāvokļu pāreju shēma
    O – novērojumu shēma
Begin
    Forall possible_states sprim Do
        p = 0
        Forall possible states s Do
            p += T[s][a][sprim] * parentbelief[s]
        Endforall
        Belief[sprim] = O[sprim][a][z] * p
    Endforall
    return Belief
End
```

Lai būtu vieglāk vizualizēt iegūtos rezultātus, katrā mezglā vēlams glabāt papildus informāciju, piemēram, ceļu no saknes līdz dotajam mezglam.

## 8.3. Ticamības stāvokļu koka apmācīšana

Kad ir uzbūvēts ticamības stāvokļu koks, tad var laist klasiskos RL algoritmus, lai veiktu apmācību uz ticamības stāvokļiem. No klasiskā RL algoritma viedokļa, epizode, pārvietojoties pa vidi, izpaudīsies kā pārvietošanās no koka saknes dziļāk kokā (bet ne obligāti līdz kādai koka lapai).

**Apmācības rezultāts** ir *ActionValues* vērtības katrā mezglā:

Algoritms ***train_tiger_problem*** aģenta apmācīšanai dots zemāk pseidokodā.

```
Function train_tiger_problem (maxlevel)
        maxheight =4 vai 6 – maksimālais koka augstums
        maxcount=1000 – maksimālais epizožu skaits
        γ=0.9 – diskonta likme
        α=0.05 – learning rate (apmācības koeficients)
Begin
        Root := create_belief_tree (maxheight)
        Do maxcount Times # pa visām epizodēm
                Forall Node in the belief tree (started at Root) Do
                        Forall possible actions a from Node Do
                                believed_reward = compute_believed_reward(Node.Belief,a)
                                If Node is terminal Then  % mezgls ir lapa kokā
                                        Node.ActionValues[a] += α * (believed_reward -
                                                Node.ActionValues[a])
                                Else  % iekšējs mezgls
                                        valueprim = 0
                                        Forall observations z in Node.Children[a] Do
                                                nextnode = Node.Children[a][z]
                                                valueprim += nextnode.Probability *
                                                        max(nextnode.ActionValues)
                                        Endforall
                                        Node.ActionValues[a] += α * (γ * valueprim + believed_reward -
                                                Node.ActionValues[a])
                                Endif
                        Endforall
                Endforall
        Enddo
        (Vizualizēt ticamības koku un ActionValues katrā mezglā)
End

Function compute_believed_reward(belief,a)
        R – atlīdzību shēma
Begin
        br = 0.0
        Forall possible states s Do
                br += R[s][a] * belief[s]
        Endforall
        return br
End
```

Apmācīts ticamības stāvokļu koks ar 3 līmeņiem (augstums = 4).

Formāts:

<hierarhija> **a** <darbība> **z** <novērojums> [<ticamība>] <varbūtība> [<darbību vērtības>] <līmenis> <soļi līdz beigām>

```
| a None z None [0.50 0.50 0.00] 1.000 [-45.00 -45.00 1.74] 0 4
__| a 2 z 0 [0.85 0.15 0.00] 0.500 [-83.50 -6.50 3.04] 1 3
____| a 2 z 0 [0.97 0.03 0.00] 0.745 [-96.68 6.68 5.86] 2 2
______| a 2 z 0 [0.99 0.01 0.00] 0.829 [-99.40 9.40 -1.00] 3 1
______| a 2 z 1 [0.85 0.15 0.00] 0.171 [-83.50 -6.50 -1.00] 3 1
____| a 2 z 1 [0.50 0.50 0.00] 0.255 [-45.00 -45.00 -1.90] 2 2
______| a 2 z 0 [0.85 0.15 0.00] 0.500 [-83.50 -6.50 -1.00] 3 1
______| a 2 z 1 [0.15 0.85 0.00] 0.500 [-6.50 -83.50 -1.00] 3 1
__| a 2 z 1 [0.15 0.85 0.00] 0.500 [-6.50 -83.50 3.04] 1 3
____| a 2 z 0 [0.50 0.50 0.00] 0.255 [-45.00 -45.00 -1.90] 2 2
______| a 2 z 0 [0.85 0.15 0.00] 0.500 [-83.50 -6.50 -1.00] 3 1
______| a 2 z 1 [0.15 0.85 0.00] 0.500 [-6.50 -83.50 -1.00] 3 1
____| a 2 z 1 [0.03 0.97 0.00] 0.745 [6.68 -96.68 5.86] 2 2
______| a 2 z 0 [0.15 0.85 0.00] 0.171 [-6.50 -83.50 -1.00] 3 1
______| a 2 z 1 [0.01 0.99 0.00] 0.829 [9.40 -99.40 -1.00] 3 1
```