Satura rādītājs

Satura rādītājs 1
1. Politiku iterācija un vērtību iterācija 3
1.1. Problēmas formulējums........ 3
1.1.1. Režģis, stāvokļi un darbības........ 3
1.1.2. Stāvokļu pārejas – darbību veikšanas potenciālais rezultāts........ 3
1.1.3. Stāvokļu pāreju atlīdzības 4
1.1.4. Optimālas politikas iegūšana – problēmas risinājums 5
1.1.5. Pieejamie ieejas dati 6
1.2. Politikas novērtēšana........ 6
1.2.1. Algoritms „evaluate_policy_det” 6
1.2.2. Piemēri 7
1.3. Politiku iterācija. Algoritms „iterate_policy_det”........ 8
1.4. Vērtību iterācija. Algoritms „iterate_values_det” 9
1.5. Laboratorijas darba veicamo programmēšanas darbu kopsavilkums........ 9
2. Monte Carlo metode........ 10
2.1. Problēmas formulējums........ 10
2.1.1. Režģis, stāvokļi un darbības........ 10
2.1.2. Stāvokļu pārejas – darbību veikšanas potenciālais rezultāts........ 10
2.1.3. Stāvokļu pāreju atlīdzības 10
2.1.4. Optimāla politika........ 11
2.1.5. Pieejamie ieejas dati 11
2.2. Epizodes ģenerēšana. Algoritms „generate_episode_soft” 11
2.3. Monte Carlo kontroles algoritms „monte_carlo_det” 13
2.4. Laboratorijas darba veicamo programmēšanas darbu kopsavilkums........ 14
3. TD (temporal difference) algoritmi........ 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. Ģenētiskie algoritmi 18
4.1. Problēmas formulējums........ 18
4.1.1. Būla funkcijas ‘NOT AND’ modelēšana 18
4.1.2. Pieejamie dati 18
4.2. Ģenētiskā algoritma realizācija 19
4.2.1. Iegūstamais rezultāts 19
4.2.2. Risinājuma reprezentācija 19
4.2.2.1. Algoritms „number_to_bits” skaitļa pārveidošanai bitu virknē........ 20
4.2.2.2. Algoritms „bits_to_number” bitu virknes pārveidošanai skaitlī........ 20
4.2.2.3. Algoritms visa risinājuma pārveidošanai bitos un atpakaļ........ 21
4.2.3. Risinājumu derīguma noskaidrošana 21
4.2.4. Varbūtību sadalījuma iegūšana un indivīdu izvēle atbilstoši varbūtību sadalījumam 23
4.2.5. Ģenētiskie operatori 26
4.2.6. Ģenētiskā algoritma kopsavilkums 28
4.3. Laboratorijas darba veicamo programmēšanas darbu kopsavilkums........ 29
5. Naivais Beijesa klasifikators 30
5.1. Problēmas formulējums........ 30

5.1.1. Kredītriska novērtēšanas piemēri ... 30
5.1.2. Pieejamie dati ... 31
5.2. Naivā Beijesa klasifikatora darbības princips ... 31
5.3. Naivā Beijesa klasifikatora realizācija ... 32
5.3.1. Viena parauga klasificēšana ... 32
5.3.2. Klasifikatora testēšana uz apmācības paraugiem ... 33
5.4. Laboratorijas darba veicamo programmēšanas darbu kopsavilkums ... 33
6. Skudru koloniju optimizācija TSP gadījumā ... 34
6.1. Problēmas formulējums – Travelling Salesman Problem ... 34
6.2. Algoritma realizācija ... 35
6.2.1. Tūres ģenerēšana ... 35
6.2.2. Skudru kolonijas algoritma kopsavilkums ... 38
6.3. Laboratorijas darba veicamo programmēšanas darbu kopsavilkums ... 39
7. Atbalsta vektoru mašīna (*support vector machine*, *SVM*) ... 40
7.1. Divu lineāru problēmu formulējums priekš SVM ... 40
7.1.1. Problēma #1. Būla funkcijas ‘AND’ modelēšana ... 40
7.1.2. Problēma #2. Lineāra punktu sadalīšana plaknē ... 41
7.2. Iteratīvā SVM algoritma realizācija lineāras problēmas gadījumā ... 41
7.2.1. Viena parauga darbināšana ar lineāru perceptronu ... 42
7.2.2. Lineāra perceptrona apmācīšana uz paraugu kopas ... 42
7.2.3. Lineārā iteratīvā SVM algoritma kopsavilkums ... 44
7.2.4. Sistēmas konfigurācija un iegūstamais rezultāts ... 44
7.3. Nelineāras problēmas formulējums priekš SVM ... 45
7.4. Iteratīvā SVM algoritma realizācija lineāras problēmas gadījumā ... 47
7.4.1. Pārveidošanas funkcijas realizācija ... 47
7.4.2. Nelineārā iteratīvā SVM algoritma kopsavilkums ... 50
7.4.3. Nelineārās sistēmas konfigurācija un iegūstamais rezultāts ... 51
8. Daļēja novērojamība un POMDP. Tīģera problēma ... 53
8.1. Problēmas formulējums ... 53
8.2. Ticamības stāvokļu koka būvēšana ... 54
8.3. Ticamības stāvokļu koka apmācīšana ... 58

1. Politiku iterācija un vērtību iterācija

1.1. Problēmas formulējums

1.1.1. Režģis, stāvokļi un darbības

Dots režģis (*grid*) 5×5, pa kuru pārvietojas t.s. aģents.

Katru režģa šūnu reprezentē viens stāvoklis (*state*) s_i.

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

kur n=25.

Stāvokļu kopa 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

No katra stāvokļa s_i var izdarīt četrus gājienus (četras darbības, *actions*) – {uz augšu, uz leju, pa kreisi, pa labi}.

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

kur m=4.

Iespējamo darbību kopa A:

1	↑	uz augšu
2	↓	uz leju
3	←	pa kreisi
4	→	pa labi

1.1.2. Stāvokļu pārejas – darbību veikšanas potenciālais rezultāts

Dota pāreju (*transitions*) kopa T, kas nosaka, uz kādu stāvokli pāriet aģents, stāvoklī s piemērojot darbību a. Pēc būtības tieši stāvokļu pārejas nosaka darbības vidi (*environment*). Tā kā ir 25 stāvokļi, un katrā stāvoklī iespējamas 4 darbības, tad stāvokļu pāreju kopā T šajā gadījumā ir 100 ieraksti, kur katrs ieraksts ir trijnieks formā: {stāvoklis, darbība, nākamais stāvoklis}:

$$T = \{\langle s, a, s' \rangle \mid s, s' \notin S, a \in A\}$$ vai, definējot to kā funkciju,

$$T = S \times A \to S$$

Dotajā uzdevumā ir 3 dažādi gadījumi stāvokļu pārējām:

A. Parastais gadījums. Izdarot gājienu aģents nonāk blakus šūnā atbilstoši darbības tipam, piemēram, no stāvokļa 13 ejot pa labi, nonāk stāvolī 14:

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

B. Malējais gadījums. Izņēmums ir tad, ja veicot attiecīgo gājienu, draud iziešana ārpus režģa – tad esošais stāvoklis saglabājas, piemēram,

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

C. Speciālie gadījumi.
Ir divi speciālgadījumi:
No stāvokļa 2, izdarot jebkuru gājienu, nonāk stāvoklī 22, bet no stāvokļa 4 – stāvoklī 14:

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

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

Tādējādi kopējā režģa shēma izskatās šādi:

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. Stāvokļu pāreju atlīdzības

Katrai stāvokļu pārejai ir noteikta atlīdzība (*reward*), kuru iegūst aģents, veicot doto darbību (precīzāk, izpildot doto stāvokļa pāreju). Atlīdzība var būt arī ar mīnusa zīmi, ko var interpretēt, kā resursu patēriņu, veicot darbību:

$R = \{\langle s, a, r \rangle \mid s \notin S, a \in A, r \notin \Re\}$ vai, definējot to kā funkciju,

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

Reinforcement learning pamatā ir izvēlēties tādu darbību veikšanas stratēģiju jeb politiku (*policy*), kas nodrošinātu pēc iespējas lielāku iegūto atlīdzību kopsummu ilgākā laika periodā. Katrai stāvokļu pārejai tiek definēta viena atlīdzība, un mūsu uzdevumā tādu kopā ir 100.

A. Parastais gadījums. Izdarot parastu gājienu, aģents nesaņem nekādu atlīdzību, resp. r=0, piemēram:

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

B. Malējais gadījums. Ja veicot attiecīgo gājienu, draud iziešana ārpus režģa – tad atlīdzība (resp., sods) ir -1, piemēram,

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

C. Speciālie gadījumi.
Ir divi speciālgadījumi:
No stāvokļa 2, izdarot jebkuru gājienu, dabū atlīdzību 10, bet no stāvokļa 4 – atlīdzību 5:

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

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

Kopējā režģa shēma, ietverot atlīdzības, izskatās šādi:

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

10 5 -1 0

1.1.4. Optimālas politikas iegūšana – problēmas risinājums

Politika (*policy*) nozīmē stratēģiju, kas nosaka, kādu darbību kurā stāvoklī veikt.

Optimāla politika ir tāda, pēc kuras vadoties, summārā atlīdzība ilgākā laika periodā ir vislielākā.

Nākošajā piemērā nejauši parādīta uzģenerēta politika (1-uz augšu, 2-uz leju, 3-pa kreisi, 4-pa labi):

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

Izrādās, ka optimāla politika nav atkarīga tikai no stāvokļu pārejām un atlīdzībām, kuras šai problēmai iepriekš tika definētas, bet ir vēl arī t.s. **diskonta faktors** (*discount rate*) γ, kas pozitīvu atlīdzību gadījumā būtu kaut kas līdzīgs inflācijai. Diskonta faktors γ ir reāls skaitlis robežās [0; 1]. Jo γ ir mazāks, jo mazāku ietekmi sastāda iepriekš savāktās atlīdzības, un jācenšas tikt nevis pie lielākām atlīdzībām, bet gan pie kaut kādām atlīdzībām biežāk.

Tādējādi problēmu nosaka 3 lietas:

1. stāvokļu pāreju modelis T,
2. stāvokļu pāreju atlīdzības R,
3. diskonta faktors γ.

Ja γ ir tuvu nullei, tad politiku ietekmē gandrīz tikai kārtējā atlīdzība r, ja tuvu vienam, tad pilnā mērā arī – nākošā stāvokļa nosacītā vērtība.

Optimālas politikas piemērs Nr. 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

Optimālas politikas piemērs Nr. 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

Pirmajā piemērā diskonta faktors ir mazāks, tāpēc galvenais ir ātrāk tikt pie atlīdzības – vienalga, vai stāvoklī 2 vai 4, savukārt 2. piemērā ir augstākā tendence tikt tieši uz stāvokli 2 (no stāvokļiem 9 un 10 jā iet pa kreisi, nevis uz augšu).

1.1.5. Pieejamie ieejas dati

Kopā ir 25 stāvokļi (1..25) un 4 darbības (1-uz augšu, 2 uz leju, 3-pa kreisi, 4-pa labi).

T.txt – tabula 25×4, kur pa rindām ir stāvokļi, pa kolonnām ir darbības un vērtība rindā *i* un kolonnā *k* apzīmē stāvokļu pāreju t(*i*,*k*) (atbilst *T* nodaļā 1.1.2).

R.txt – stāvokļu pārejām atbilstošās atlīdzības (atbilst *R* nodaļā 1.1.3).

P.txt – nejauši iegūta politika paškontrolei (atbilst nejaušajai politikai nodaļas 1.1.4 sākumā).

1.2. Politikas novērtēšana

1.2.1. Algoritms „evaluate_policy_det"

Lai iegūtu optimālu politiku, būtu jāprot novērtēt esoša politika.

Nākošais pseidokods parāda politikas novērtēšanu.

Politikas novērtējums nozīmē vērtību tabulas iegūšanu, kur katram stāvoklim tiek piešķirta viena vērtība.

Procedure ***evaluate_policy_det*** **(**T, R, π, γ, θ**) returns** V

    S() – stāvokļi
    T() – stāvokļu pārejas
    R() – atlīdzības
    π() – politika
    γ – diskonta faktors [0..1]
    θ – konverģences faktors, $0 < \theta << 1$
    V() – politikas novērtējums

Begin

    $V := 0$
    Do
        $W := V$ % nofiksē esošos vērtējumus
        $\Delta := 0$ % vērtību matricas izmaiņa, kad tā paliks pietiekoši maza, var beigt
        Forall s **in** S **do**
            $a := \pi(s)$ % darbība atbilstoši politikai
            $s2 := T(s,a)$ % nākošais stāvoklis
            $r := R(s,a)$ % nākošā stāvokļa atlīdzība
            $V(s) := r + \gamma * W(s2)$ % aktuālā stāvokļa vērtība
            $\Delta := \max(\Delta,|V(s)-W(s)|)$ % maksimālā izmaiņa kādam stāvoklim
        Endforall
    While $\Delta \geq \theta$

End

1.2.2. Piemēri

Te parādīti divi algoritma ***evaluate_policy_det*** (nodaļa 1.2.1) rezultātu piemēri.

T=„T.txt”; R=„R.txt”; π=„P.txt”; θ=0.1 (sk. pieejamos datu failus nodaļā 1.1.5)

Piemērs #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

Piemērs #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. Politiku iterācija. Algoritms „iterate_policy_det”

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

```
Procedure iterate_policy_det (T, R, γ, θ) returns π
        A() – darbības
        S() – stāvokļi
        T() – stāvokļu pārejas
        R() – atlīdzības
        γ – diskonta faktors [0..1]
        θ – konverģences faktors, 0 < θ << 1
        π() – politika
        evaluate_policy_det – determinētas politikas novērtējums
Begin
        Uzģenerē determinētu politiku π uz labu laimi (in random)
        Do
                V := evaluate_policy_det (T, R, π, γ, θ)
                policy_stable := True
                Forall s In S Do
                        oldp := π(s)
                          % tā darbība a, pie kuras dotā stāvokļa vērtība būtu maksimāli iespējamā:
                        maxval := Null
                        Forall a In A Do
                                s2 := T(s,a)  % nākošais stāvoklis
                                r := R(s,a)  % nākošā stāvokļa atlīdzība
                                v := r + γ * V(s2)  % aktuālā stāvokļa vērtība pie izvēlētas darbības 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
```

Piemēri optimālu politiku iegūšanai pie T=„T.txt”; R=„R.txt”; π=„P.txt”; θ=0.1 parādīti nodaļā 1.1.4.

1.4. Vērtību iterācija. Algoritms „iterate_values_det”

Kompaktāku optimālas determinētas politikas iegūšanu realizē algoritms ***iterate_values_det*** (salīdzinot ar politiku iterāciju, nav nepieciešams atsevišķs un resursu ietilpīgs politikas novērtēšanas solis).

```
Procedure iterate_values_det (T, R, γ, θ) returns π
        A() – darbības
        S() – stāvokļi
        T() – stāvokļu pārejas
        R() – atlīdzības
        V() – stāvokļu vērtības
        γ – diskonta faktors [0..1]
        θ – konverģences faktors, 0 < θ << 1
        π() – politika
Begin
        V := 0
        Do
                Δ := 0  % vērtību matricas izmaiņa, kad tā paliks pietiekoši maza, var beigt
                Forall s In S Do
                        vs_old := V(s)  % aktuālā stāvokļa vērtība
                        V(s) := Null
                        Forall a In A Do
                                s2 := T(s,a)  % nākošais stāvoklis
                                r := R(s,a)  % nākošā stāvokļa atlīdzība
                                v := r + γ * V(s2)  % aktuālā stāvokļa vērtība pie izvēlētas darbības 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|)  % maksimālā izmaiņa kādam stāvoklim
                Endforall
        While Δ ≥ θ
End
```

Piemēri optimālu politiku iegūšanai pie T=„T.txt”; R=„R.txt”; π=„P.txt”; θ=0.1 parādīti nodaļā 1.1.4.

1.5. Laboratorijas darba veicamo programmēšanas darbu kopsavilkums

- Ielādēt datora atmiņā failus T=„T.txt”; R=„R.txt”; π=„P.txt”
- Realizēt algoritmu ***evaluate_policy_det*** (nodaļa 1.2.1) un notestēt to uz abiem piemēriem nodaļā 1.2.2.
- Realizēt algoritmu ***iterate_policy_det*** (nodaļa 1.3) un notestēt to uz abiem piemēriem optimālu politiku iegūšanai, kas parādīti nodaļā 1.1.4.
- Realizēt algoritmu ***iterate_values_det*** (nodaļa 1.4) un notestēt to uz abiem piemēriem optimālu politiku iegūšanai, kas parādīti nodaļā 1.1.4.

2. Monte Carlo metode

2.1. Problēmas formulējums

2.1.1. Režģis, stāvokļi un darbības

Dots režģis (*grid*) 3×4, kur viens lauciņš ir tukšs (starp 5 un 6), bet 4. un 7. stāvoklis nodefinēti kā beigu stāvokļi (ierāmēti ar treknu līniju).

Par S sauksim to stāvokļu kopu, kas nav beigu stāvokļi, bet par S^+ –visu stāvokļu kopu, ieskaitot beigu stāvokļus.

Stāvokļu kopa S^+:

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

No katra stāvokļa s_i var izdarīt četrus gājienus (četras darbības, *actions*) – {uz augšu, uz leju, pa kreisi, pa labi}, tāpat, kā iepriekšējā laboratorijas darbā

2.1.2. Stāvokļu pārejas – darbību veikšanas potenciālais rezultāts

Stāvokļu pārejas ir tikai no parastajiem (ne-beigu) stāvokļiem. Līdzīgi, kā iepriekšējā laboratorijas darbā – ja izvēlētajā virzienā eksistē lauciņš, tad uz to notiek pāreja, ja ne, tad paliek uzvietas:

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

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

Arī vidējais aizkrāsotais lauciņš nozīmē šķērsli, un mēģinājums iet uz to, nozīmēs palikt uz vietas:

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

Uz beigu stāvokļiem var nonākt 3 veidos: no 3. un 4. stāvokļa – pa labi, bet no 11. – uz augšu.

2.1.3. Stāvokļu pāreju atlīdzības

Neatkarīgi no tā, vai notiek mēģinājums iziet ārpus laukuma vai nē, pārejot uz neterminālo (ne-beigu) stāvokli atlīdzība ir negatīva: -1, piemēram:

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

Tomēr uzdevuma „sāls” ir pārejās uz beigu stāvokļiem.

4. stāvoklis ir „labais” stāvoklis. Pārejot uz to, atlīdzībā tiek iegūts 100:

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

7. stāvoklis ir „sliktais” stāvoklis. Pārejot uz to, atlīdzībā tiek iegūts -99:

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

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

Kopējā režģa shēma, ņemot vērā atlīdzības, izskatās šādi:

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

2.1.4. Optimāla politika

Optimāla politika nozīmē, iegūt vislielāko atlīdzību skaitu, sasniedzot (kādu) beigu stāvokli. Optimālās politikas atrašanu ietekmē arī diskonta faktors γ.

Optimālas politikas piemērs γ = 0.5.

4	4	4	
1		1	
1	4	1	3

2.1.5. Pieejamie ieejas dati

Kopā ir 11 stāvokļi (1..11), no kuriem 9 ir neterminālie un 4 darbības (1-uz augšu, 2 uz leju, 3-pa kreisi, 4-pa labi).

T.txt – tabula 11×4, kur pa rindām ir stāvokļi, pa kolonnām ir darbības un vērtība rindā *i* un kolonnā *k* apzīmē stāvokļu pāreju t(*i*,*k*).

R.txt – stāvokļu pārejām atbilstošās atlīdzības.

F.txt – beigu stāvokļi (atzīmēti ar 1).

P.txt – nejauši iegūta politika paškontrolei.

2.2. Epizodes ģenerēšana. Algoritms „generate_episode_soft”

Monte Carlo metode ietver sevī tādu mehānismu kā epizožu ģenerēšanu, balstoties uz esošu politiku. Epizode sākas ar brīvi izvēlētu neterminālo stāvokli, bet beidzas ar beigu stāvokli.

Epizode būtībā ir trijnieku kopums, kur katrs trijnieks ir *<darbība, iegūtā atlīdzība, nākošais stāvoklis>*.

Lai nodrošinātu to, ka epizodes ģenerēšana neieciklojas (politika ir determinēta, un sākumā var būt neoptimāla politika), kā arī lai statistiski būtu pieteikoši liela varbūtība, ka epizode iet cauri visiem stāvokļiem un visām iespējamām darbībām stāvokļos, tad epozodes ģenerēšanā tiek pielietots t.s. *ε-soft* princips, resp., ar noteiktu varbūtību (resp. *ε*), dotajā stāvoklī tiek izvēlēta kāda no darbībām, kas neatbilst politikai:

Ģenerētas epizodes piemērs pie ε=0.1 (epizodes var būt **daudz** garākas!):

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

Epizode ģenerēta pie politikas (2 1 4 0 1 1 0 1 3 2 4)

Procedure ***generate_episode_soft*** **(***T*, *R*, π, ε**) returns** *Ep*
  T() – stāvokļu pārejas
  R() – atlīdzības
  π*()* – politika
  ε – mīkstināšanas (*soft*) faktors, $0 < \varepsilon << 1$
  Ep() – epizode
Begin
  s := izvēlēties neterminālo stāvokli uz labu laimi (*in random*)
  Ep := <<0,0,*s*>>
  While *s* is not terminal
    a0 := π(*s*) {politika dotajā stāvoklī}
    a := izvēlēties darbību no $a_1..a_n$ pie varbūtību sadalījuma $p_1..p_n$, kur
      $p_i := \varepsilon$, ja $a_i \neq a0$
      $p_i := 1-\varepsilon*(n-1)$, ja $a_i = a0$
    s2 := *T*(*s*,*a*)
    r := *R*(*s*,*a*)
    Ep := *Ep* + <*a*,*r*,*s2*>
    s := *s2*
  Endwhile
End

2.3. Monte Carlo kontroles algoritms „monte_carlo_det”

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

```
Procedure monte_carlo_det (T, R, γ, ε, maxit) returns π
        S() – neterminālie stāvokļi
        T() – stāvokļu pārejas
        R() – atlīdzības
        γ – diskonta faktors [0..1]
        ε – politikas mīkstināšanas faktors, 0 < ε << 1
        maxit – maksimālais iterāciju skaits
        Q() – stāvokļu darbību vērtības
        Returns() – Stāvokļu darbību iegītās atlīdzības
        π() – politika
        generate_episode_soft – epizodes ģenerēšana (nodaļa 2.2)
Begin
        Uzģenerē determinētu politiku π uz labu laimi (in random)
        Forall (s, a) Do
                Q(s,a) := 0  % var arī citas vērtības, brīvi pēc izvēles
                Returns(s,a) := []  % tukša virkne
        Endforall
        Do maxit times
                Ep := generate_episode_soft (T, R, π, ε)
                For e:=2 To size(Ep) Do  % pirmais elements satur tikai sākuma stāvokli
                        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}) %  Pievienot r pie Returns(s,a)
                                Q(s,a) := average(Returns(s,a))
                        Endif
                Endforall
                Forall s In S Do  % pa visiem neterminālajiem stāvokļiem
                          % Tā darbība a, pie kuras fiksētam s vērtība Q ir vislielākā
                        π(s) := argmax(a,Q(s,a))
                Endforall
        Enddo
End
```

Piemērs optimālas politikas iegūšanai pie T=„T.txt”; R=„R.txt” ; π=„P.txt” ; F=„F.txt”; ε=0.1; γ=0.9 parādīts nodaļā 2.1.4

Darbību vērtējumu tabulas *Q* piemērs:

	⇑	⇓	⇐	⇒
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. Laboratorijas darba veicamo programmēšanas darbu kopsavilkums

- Ielādēt datora atmiņā failus *T*=„T.txt”; *R*=„R.txt”; π=„P.txt”; F=„F.txt”
- Realizēt algoritmu ***generate_episode_soft*** (nodaļa 2.2) un notestēt ar parametru ε=0.1.
- Realizēt algoritmu ***monte_carlo_det*** (nodaļa 2.3) un notestēt to ar parametriem ε=0.1, γ=0.9, *maxit*=1000.

3. TD (temporal difference) algoritmi

3.1. Problēmas formulējums

Tā pati problēma, kas aprakstīta nodaļā 2.1

3.2. Algoritms epizodes viena soļa aprēķināšanai „next_state”

Algoritms viena soļa aprēķināšanai ***next_state*** atbilst viena soļa aprēķināšanai epizodē, kas aprakstīta nodaļā 2.2.

Function *next_state* (s, ε) **Returns** *a, r, s2*
- *s* – sākuma stāvoklis
- *T()* – stāvokļu pārejas
- *R()* – atlīdzības
- $\pi()$ – politika
- ε – mīkstināšanas (*soft*) faktors, $0 < \varepsilon << 1$
- *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* := $\pi(s)$ % 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”; *π*=„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 ε=0.1, γ=0.9, α=0.1, *maxit*=200..500.
- Realizēt algoritmu ***q_learning*** (nodaļa 3.4) un notestēt ar parametriem ε=0.1, γ=0.9, α=0.1, *maxit*=200..500

4. Ģenētiskie algoritmi

4.1. Problēmas formulējums

4.1.1. Būla funkcijas ‘NOT AND’ modelēšana

Doti 4 piemēri, apraksta loģisku funkciju ***NOT*** $\boldsymbol{(x_1}$ ***AND*** $\boldsymbol{x_2)}$.

x_1	x_2	d
0	0	1
0	1	1
1	0	1
1	1	0

Modelēt funkciju ‘NOT AND’ formā

$y = \mathrm{F}(x_1, x_2) = w_0 + w_1x_1 + w_2x_2$.

Tas nozīmē, ka jāatrod tādi koeficienti w_0, w_1, w_2, lai katram paraugam $<x_1, x_2>$ funkcija $\mathrm{F}(x_1, x_2)$ pēc iespējas tuvotos vērtībai *NOT* $(x_1$ *AND* $x_2)$.

Procesa vienkāršošanai izdarīsim nelielas izmaiņas – ierobežosim $\mathrm{F}(x_1, x_2)$ intervālā [0..1]:

$$\mathrm{F}(x_1, x_2) = \begin{Bmatrix} 1; & NET > 1 \\ 0; & NET < 0 \\ NET; & else \end{Bmatrix}$$, kur

$$NET = w_0 + w_1x_1 + w_2x_2$$

4.1.2. Pieejamie dati

x.txt un *d.txt* – apmācības piemēri

No šiem datiem var izšķirt 2 daļas:

- *D* – vēlamais rezultāts (*d.txt*):

1
1
1
0

- *X* – ievaddati (*x.txt*):

0	0
0	1
1	0
1	1

p.txt, *f.txt* un *w.txt* – dati starprezultātu pārbaudei

4.2. Ģenētiskā algoritma realizācija

4.2.1. Iegūstamais rezultāts

Viens no ideālajiem rezultātiem (kur $F(x_1, x_2) = d$) ir šāds:

$w_0 = 2$

$w_1 = -1$

$w_2 = -1$

x_1	x_2	w_0	w_1	w_2	NET	$F(x_1,x_2)$	d
0	0	2	-1	-1	2	1	1
0	1				1	1	1
1	0				1	1	1
1	1				0	0	0

4.2.2. Risinājuma reprezentācija

Risinājuma reprezentācija ir svarīgs aspekts ģenētiskā algoritma realizācijā, un no veiksmīgas risinājuma reprezentācijas bieži vien ir atkarīga ģenētiskā algoritma realizācijas veiksme.

Ģenētiskajā algoritmā risinājumus parasti reprezentē bitu formā – lai algoritma darbības laikā tam būtu vieglāk piemērot t.s. ģenētiskos operatorus.

Mūsu problēmas risinājums – tie ir 3 koeficienti.

Katru no tiem kodēsim ar 8 bitiem, tādējādi viss risinājums būs 24 biti.

Kodēsim skaitļus tā, lai tie būtu robežās [-2, +2].

Viena skaitļa kodēšana:

- vecākais bits kodē zīmi +/-,
- pārējie divnieka pakāpes.

Ja man ir bitu virkne $<b_1, ..., b_n>$

Tad tas apzīmē skaitli:

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

jeb

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

4.2.2.1. Algoritms „number_to_bits” skaitļa pārveidošanai bitu virknē

Algoritms ***number_to_bits*** domāts reāla skaitļa [-2..+2] pārveidošanai par bitu virkni.

```
Procedure number_to_bits (n, bitcount) Returns Bits
        n – skaitlis, kuru pārveido bitu formā
        bitcount – cik bitos (ieskaitot zīmi) tiks pārveidots skaitlis
        Bits() – bitu virkne (piemēram, masīvs) garumā bitcount
Begin
        Bits := {0...}  % izveido masīvu garumā bitcount un aizpilda ar 0
        If n < 0 Then  % ja negatīvs skailis
                Bits(1) := 1  % pirmais bits kļūst par viennieku
                n = -n
        Endif
        factor := 1  % divnieka pakāpe, sākumā 0-tā
        For k := 2 To bitcount Do
                If factor <= n Then
                        Bits(k) = 1
                        n := n – factor;
                Endif
                factor := factor / 2
        Endfor
End
```

4.2.2.2. Algoritms „bits_to_number” bitu virknes pārveidošanai skaitlī

Algoritms ***bits_to_number*** domāts bitu virknes pārveidošanai reālā skaitlī [-2..+2].

```
Procedure bits_to_number (Bits, bitcount) Returns n
        Bits() – bitu virkne (piemēram, masīvs) garumā bitcount
        bitcount – cik bitos (ieskaitot zīmi) tiks pārveidots skaitlis
        n – skaitlis, kuru pārveido bitu formā
Begin
        n := 0
        factor := 1  % divnieka pakāpe, sākumā 0-tā
        For k := 2 To bitcount Do
                n := n + factor * Bits(k)
                factor := factor / 2
        Endfor
        If Bits(1) = 1 Then  % ja zīmes bits
                n = -n
        Endif
End
```

4.2.2.3. Algoritms visa risinājuma pārveidošanai bitos un atpakaļ

Algoritms ***solution_to_bits*** pārveido skaitļu virkni par bitu virkni

```
Procedure solution_to_bits (x, bitcount) Returns Bits
        x – skaitļu virkne, kuru pārveido bitu formā
        bitcount – cik bitos (ieskaitot zīmi) tiks pārveidots skaitlis
        Bits() – bitu virkne, kas reprezentē x, garumā bitcount * |x|
        number_to_bits – viena skaitļa pārveidošana bitu virknē
Begin
        Bits := Empty
        For i := 1 To |x| Do
                Bits := concatenate (Bits, number_to_bits (x(i), bitcount))
        Endfor
End
```

Algoritms ***bits_to_solution*** pārveido bitu virkni par skaitļu virkni

```
Procedure bits_to_solution (Bits, numcount, bitcount) Returns x
        Bits() – bitu virkne bitcount*numcount, kas reprezentē risinājumu
        numcount – cik skaitļu ir risinājumā
        bitcount – cik bitos tiks pārveidots skaitlis
        x() – risinājums skaitļu formā
        bits_to_number – bitu virknes pārveidošana skaitlī
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
```

Piemērs funkciju *solution_to_bits* un *bits_to_solution pārbaudei:*

Risinājums skaitliskā formā:

0.1875	-1.5469	0.9219

Atbilstošā bitu virkne:

0	0	0	0	1	1	0	0	1	1	1	0	0	0	1	1	0	0	1	1	1	0	1	1

4.2.3. Risinājumu derīguma noskaidrošana

Viena no izšķirošajām komponentēm ģenētiskajos algoritmos ir t.s. derīguma funkcija (*fitness function*). Lai pārbaudītu risinājuma derīgumu, pats risinājums vispirms ir jāiegūst (atbilstoši 4.1.1).

Funkcija ***run_model*** darbina modeli ar ieeju virkni *x* un iegūst rezultātu kā vienu skaitli *y*.

```
Function run_model (x, w) Returns y
        x() – ieejas dati: x1, x2
        w() – modeļa koeficienti: w0, w1, w2
        y – modeļa sniegtais rezultāts
Begin
        NET := w(0)
        For i := 1 To ||x|| Do  % pa visiem x
                NET := NET + x(i) * w(i)
        Endfor
        If NET < 0 Then
                y := 0
        Elseif NET > 1 Then
                y := 1
        Else
                y := NET
        Endif
End
```

Šajā gadījumā vieglāk iegūt nevis derīguma funkciju *F*, bet gan kļūdas funkciju *E*, kas ir apgriezta derīguma funkcijai.

Kļūdas funkcija nozīmē darbināt visus paraugus *X* uz visiem modeļiem, kurus reprezentē koeficientu matrica *W*, tad noteiktā veidā iegūtos rezultātus *Y* salīdzināt ar vēlamo rezultātu *D*, iegūstot vērtību virkni *E*, kas satur vienu skaitli katram modelim.

(Derīguma funkcija, kuru aprēķinās vēlāk, būs pretēja kļūdas funkcijai).

```
Function error_function (X, W, D) Returns E
        X() – ieejas dati visiem paraugiem (1. dimensija – paraugs, 2. dimensija – ieejas)
        W() – paralēlo modeļu koeficienti (1. dimensija – modeļa numurs, 2. dimensija – koeficients)
        D() – vēlamās atbildes (viens skaitlis katram paraugam)
        E() – modeļu kļūda (viens skaitlis katram modelim)
        run_model – viena modeļa darbināšana uz viena parauga
Begin
        For i := 1 To ||W||Do  % pa visiem modeļiem
                For k := 1 To ||X|| Do  % pa visiem paraugiem
                        Y(k) := run_model (X(k), W(i))
                Enfor
                  % modeļa novērtējums ir sliktākā kvadrātiskā parauga starpība no vēlamā
                  % (tā kā maza kļūda nozīmē labu vērtējumu, tad "sliktākais" ir max, nevis min!)
                E(i) := max((Y-D)^2)
        Endfor
End
```

Piemērs funkcijas *error_function* pārbaudei (izmanto 3 ieejas datu failus):

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 (iegūstamais rezultāts) =

0.25
0
0.5625
1
0.93838

4.2.4. Varbūtību sadalījuma iegūšana un indivīdu izvēle atbilstoši varbūtību sadalījumam

Ģenētiskajos algoritms divu ģenētisko operatoru gadījumā tiek uz labu laimi izvēlēti indivīdi (t.i., risinājumi) tālākām manipulācijām. Izvēloties indivīdus, varbūtību sadalījums nav vienmērīgs, bet lielākā iespēja izvēlēties ir tādus indivīdus, kuri ir labāk novērtēti.

Ja mums ir zināms indivīdu vērtējums *F*(), tad vispārīgā gadījumā varbūtību *Pr*() katra indivīda izvēlei rēķina pēc formulas:

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

Diemžēl šāda formula darbojas gadījumā, kad derīguma funkcijas rezultāts ir tiešs, nevis apgriezts attiecībā pret novērtējamā indivīda kvalitāti (resp., augstākam rezultātam atbilst labāka kvalitāte). Tā kā mums katram modelim ir izrēķināts nevis derīgums *F*, bet gan kļūda *E*, no kļūdas jāiegūst derīgums. To var veikt, piemēram, kāpinot skaitli e negatīvā pakāpē „kļūda”:

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

Kad iegūtas derīguma vērtības *F*, varbūtību sadalījuma iegūšana notiek pēc augstāk parādītās formulas:

```
Function compute_probabilities (E) Returns P
        E() – modeļu kļūda, ko izrēķinājusi kļūdas funkcija
        F() – modeļu derīgums
        P() – iegūtā kumulatīvā varbūtību virkne
Begin
         % derīguma iegūšana no kļūdas
        For i := 1 To ||E|| Do
                F(i) := exp(-E(i))
        Endfor
         % varbūtību aprēķins
        P := F / sum(F)
         % kumulatīvo varbūtību iegūšana
        For k := 2 To ||P||-1 Do  % pa visām varbūtībām, neskaitot malējās
                P(k) := P(k) + P(k-1)
        Endfor
         % teorētiski iepriekšējais cikls māk uzstādīt arī šo (pēdējo) elementu,
         % bet šis ir, lai izvairītos no noapaļošanas kļūdām:
        P(||P||) := 1  % beidzamo elementu tieši uzliek 1
End
```

Piemērs funkcijas *compute_probabilities* pārbaudei (izmanto vienu ieejas datu failu).

Dota kļūdu vērtību virkne *E (e.txt) =*

0.3906
0.3713
0.3713
0.3713
0.3906
1.0000
1.0000
1.0000
0.3713
1.0000

Iegūtā kumulatīvo varbūtību virkne *P*:

0.121173
0.244708
0.368242
0.491776
0.612949
0.678829
0.744708
0.810587
0.934121
1

Kad ir iegūta kumulatīvo varbūtību virkne, tad ir viegli realizējama elementa (šeit risinājuma jeb indivīda) izvēle uz labu laimi atbilstoši dotajam varbūtību sadalījumam (to mēdz saukt par *Monte Carlo* jeb ruletes metodi):

Nākošais algoritms apraksta noteikta lieluma izlases izveidošanu no dotās kopas atbilstoši varbūtību sadalījumam, izmantojot ruletes metodi (patiesībā kumulatīvo varbūtību virknes

iegūšana ir ruletes metodes galvenā sastāvdaļa, bet pati ruletes izvēle atbilstoši šai virknei ir ļoti vienkārša):

```
Function selection (Population, P, n) Returns Selection
        Population() – risinājumu kopums
        P() – kumulatīvā varbūtību virkne, kas apraksta Population elementus
        n – iegūstamās izlases lielums
        Selection() – iegūtā izlase
Begin
        Selection := Empty  % sākumā izlase ir tukša kopa
        While ||Selection|| < n
                % ruletes izvēle
                r := gadījuma skaitlis robežās [0, 1].
                num := 1
                While r > P(num)
                        num := num + 1
                Endwhile
                Add Population(num) To Selection
        Endwhile
End
```

Piemērs funkcijas *selection* pārbaudei:

Dota potenciālo risinājumu virkne 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

un tai atbilstošā kumulatīvo varbūtību virkne *P (p.txt) =*

0.1
0.2
0.3
0.4
1.0

(kam atbilst varbūtības attiecīgi [0.1, 0.1, 0.1, 0.1, 0.6])

Palaist funkciju *selection* uz šo virkni 5 reizes ar *n*=2, tātad no piecu elementu klāsta katrreiz tiks izvēlēti divi, kopā 10. Visbiežāk vidēji statistiski būtu jābūt pēdējam indivīdam (kas sākas ar 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

4.2.5. Ģenētiskie operatori

Šajā laboratorijas darbā tiks izmantoti 2 ģenētiskie operatori: viena punkta krustošana un mutācija.

Funkcija ***single_point_crossover*** realizē viena punkta krustošana, ja doti divi jau izvēlēti bāzes indivīdi.

```
Function single_point_crossover (parent1, parent2) Returns child
        parent1, parent2 – indivīdi, kuriem piemēro krustošanu
        bitcount – bitu skaits modeļa koeficientu kodēšanai
        child – krustošanas rezultātā iegūtais indivīds
        solution_to_bits – skaitļu virknes pārveidošana bitu virknē
        bits_to_solution – bitu virknes pārveidošana skaitļu virknē
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
```

Piemērs funkcijas *single_point_crossover* pārbaudei:

Dota potenciālo risinājumu virkne 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

Palaist funkciju *single_point_crossover* uz diviem augšējiem indivīdiem 5 reizes, rezultātā iegūstot, ka katrā iegūtajā risinājumā viena puse ir no viena vecāka otra no otra, piemēram:

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

Funkcija ***mutation*** veic izmaiņu vienam fiksēta indivīda hromosomas bitam.

```
Function mutation (oldsolution) Returns newsolution
        oldsolution – risinājums pirms mutācijas
        bitcount – bitu skaits modeļa koeficientu kodēšanai
        newsolution – risinājums pēc mutācijas
        solution_to_bits – skaitļu virknes pārveidošana bitu virknē
        bits_to_solution – bitu virknes pārveidošana skaitļu virknē
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
```

Piemērs funkcijas *mutation* pārbaudei:

Dota potenciālo risinājumu virkne 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

Palaist funkciju *mutation* uz katru no risinājumiem, rezultātā iegūstot, ka katrā risinājumā viens no 3 skaitļiem ir pamainīts (pārējie divi “paliek uz vietas”), piemēram:

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

4.2.6. Ģenētiskā algoritma kopsavilkums

```
Function GA (X, D) Returns Solution
        X() – ieejas paraugi
        D() – vēlamās atbildes
        popsize – veidojamās populācijas lielums
        crossrate – indivīdu skaits populācijā, kurus katrā solī nomainīs (izmantojot krustošanu)
        mutrate – indivīdu skaits populācijā, kurus katrā solī pakļaus mutācijai
        maxit – maksimālais iterāciju (soļu) skaits
        bitcount – bitu skaits modeļa koeficientu kodēšanai
        ε – maksimālā pieļaujamā modeļa kļūda
        Solution – rezultāts
Begin
        ncount – koeficientu skaits vienam risinājumam (mūsu gadījumā – 3, jo ieejas garums ir 2)
        W(popsize, ncount) – koeficientu jeb risinājumu masīvs, kas reprezentē populāciju
        W := aizpilda ar gadījuma vērtībām intervālā [-0.5, 0.5]
        E := error_function (X, W, D)  % modeļa kļūdas noskaidrošana
        it := 0  % paaudžu skaitītājs
        While it < maxit And min(E) > ε
                it := it + 1
                P := compute_probabilities (F)  % aprēķina kumulatīvo varbūtību sadalījumu
                  % (A) automātiski pāriet uz nākošo paaudzi (popsize-crossrate) elementi
                WNEXT := selection (W, P, popsize - crossrate)
                  % (B) atlikušos elementus iegūst krustošanas rezultātā
                WPARENTS := selection (W, P, crossrate×2)  % vecāku izvēle
                While WPARENTS ≠ Empty  % pa visiem vecāku pāriem
                        p1, p2 := izvēlas divus elementus no WPARENTS
                        child := single_point_crossover (p1, p2)
                        Add child to WNEXT
                        Remove p1 and p2 from WPARENTS
                Endwhile
                  % (C1) mutācijas veikšana noteiktam skaitam indivīdu
                WMUTATION := izvēlas uz labu laimi elementus no WNEXT skaitā mutrate
                WMUTATION2 := Empty
                Forall e In WMUTATION Do
                        emut := mutation (e)
                        Add emut to WMUTATION2
                Endforall
                  % (C2) nomaina sākotnējos elementus ar mutētajiem
                WNEXT := WNEXT – WMUTATION  % kopas operācija
                WNEXT := WNEXT Union WMUTATION2  % kopas operācija
                  % (D) tiek nofiksēta un novērtēta nākošā paaudze
                W := WNEXT
                E := error_function (X, W, D)  % modeļa kļūdas noskaidrošana
        Endwhile
        Solution := W(minindex (E), _)  % labākais risinājums no visas populācijas
End
```

4.3. Laboratorijas darba veicamo programmēšanas darbu kopsavilkums

Pirmajā nodarbībā:

- Realizēt risinājuma pārveidošanu bitu virknē un atpakaļ (nodaļa 4.2.2) ar parametru *bitcount*=8.
- Realizēt derīguma funkciju (nodaļa 4.2.3).
- Realizēt varbūtību sadalījuma iegūšanu (nodaļa 4.2.4)

Otrajā nodarbībā:

- Realizēt ģenētiskos operatorus (nodaļa 4.2.5)
- Realizēt ģenētisko algoritmu kopumā (nodaļa 4.2.6) ar parametriem *popsize*=10, *crossrate*=4, *mutrate*=5, *maxit*=1000, *bitcount*=8, ε=0.1

5. Naivais Beijesa klasifikators

5.1. Problēmas formulējums

5.1.1. Kredītriska novērtēšanas piemēri

Doti 14 piemēri, kas ļauj pēc 4 atribūtiem novērtēt kredītrisku (2. kolonna) – zems, vidējs vai augsts. Atribūti, kas ir pieejami par katru gadījumu ir kredītvēsture, esošās parādsaistības, rekomendācijas un ienākumi (3.-6. kolonna).

	vērtējums	*atribūti*			
	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

Dotos paraugus kodēsim skaitliski šādi:

- Atribūtus kodēsim kā skaitļus (ieskaitot mērķa atribūtu) 1..*n*: <risc, hist, debt, recm, income>.
- Atribūtu vērtības kodēsim kā skaitļus 1..n_a, kur n_a – atribūta vērtību skaits:
    - values(hist) = <bad, unknown, good>
    - values(debt) = <low, high>
    - values(recm) = <no, yes>
    - values(income) = <low, medium, high>

Tādējādi doto tabulu var pārrakstīt šādi (sk. failus *x.txt* un *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

5.1.2. Pieejamie dati

Vēlamās vērtības D (*d.txt*) =

3
3
2
3
1
1
3
2
1
1
3
2
1
3

Ieejas paraugi 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

5.2. Naivā Beijesa klasifikatora darbības princips

Naivais Beijesa klasifikators (*Naïve Bayes classifier*) darbojas pēc formulas:

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

kur $P(a_i|v_j)$ – varbūtība, ka ir spēkā atribūta vērtība a_i pie nosacījuma, ka klasificētā (mērķa) vērtība ir v_j.

5.3. Naivā Beijesa klasifikatora realizācija

5.3.1. Viena parauga klasificēšana

Naivā Beijesa klasifikatora gadījumā netiek būvēta un apmācīta sistēma, ar kuru pēc tam tiek atpazīti paraugi:

Katra parauga atpazīšanā piedalās visi apmācības paraugi, līdz ar to apmācīšanās fāze un izmantošanas fāzē saplūst vienā. Šāds princips konkrētajā gadījumā vienkāršo algoritmu, bet palielina nepieciešamos resursus šāda klasifikatora izmantošanā, jo katra jauna parauga atpazīšanā atkal un atkal „jāizskrien” cauri visiem apmācības paraugiem.

Funkcija ***naive_bayes*** klasificē vienu tai padotu paraugu:

```
Function naive_bayes (ex) returns decision
    ex – paraugs, kas būtu jāatpazīst
    E() – apmācības paraugi (sk. nodaļu 5.1.2)
    D() – apmācības paraugu pareizās atbildes (sk. nodaļu 5.1.2)
    A() – atribūtu saraksts, praktiski tās ir tabulas E kolonnas
    decision() – atribūtu vērtības (katram atribūtam tieši viena)
Begin
    Forall Distinct currdec In D Do  % pa visām iespējamām vērtībām D
          % paraugu skaits ar doto lēmumu currdec:
        C1 := count ({d in D where d = currdec})
          % visu paraugu skaits
        C2 := count (D)
          % lēmuma currdec proporcija visos paraugos:
        prob(currdec) := C1 / C2
        Forall currattr In A Do  % pa visiem iespējamiem atribūtiem
              % paraugu skaits ar doto lēmumu currdec un parauga ex doto currattr vērtību:
            C3 := count ({(d,e) in (D,E) where d = currdec and e(currattr) = ex(currattr)})
              % dotā lēmuma varbūtības precizēšana:
            prob(currdec) := prob(currdec) * (C3 / C1)
        Endforall
    Endforall
    decision := maxindex (prob)
End
```

Piemērs.

Bad	Low	No	HIGH

jeb skaitliski

1	1	1	3

atgriež;

2 = MEDIUM

5.3.2. Klasifikatora testēšana uz apmācības paraugiem

Lai pārliecinātos, cik korekti klasifikators strādā uz pašiem apmācības paraugiem *E*, katru no viņiem laiž cauri klasifikatoram un rezultātu salīdzina ar pareizajām atbildēm *D*.

```
Procedure test_naive_bayes
        E() – apmācības paraugi (sk. nodaļu 5.1.2)
        D() – apmācības paraugu pareizās atbildes (sk. nodaļu 5.1.2)
        naive_bayes – Naivais Beijesa klasifikators
Begin
        Forall (ex,d) In (E,D)
                rez := naive_bayes (ex)
                Print d, res
        Enforall
End
```

Uz dotajiem testa piemēriem ***test_naive_bayes*** atgriež šādu rezultātu (no vēlamajām atbildēm atšķiras tikai viena parauga klasifikācija):

Vēlamā vērtība	**Izrēķinātā vērtība**
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

5.4. Laboratorijas darba veicamo programmēšanas darbu kopsavilkums

- Ielādēt datora atmiņā failus „x.txt” un „d.txt”, iegūstot tabulas *D* un *E* (sk. nodaļu 5.1.2)
- Realizēt algoritmu ***naive_bayes*** (nodaļa 0) un notestēt ar kādu piemēru, kas nav apmācības paraugos.
- Pārbaudīt algoritmu uz apmācības paraugiem (nodaļa 5.3.2).

6. Skudru koloniju optimizācija TSP gadījumā

6.1. Problēmas formulējums – Travelling Salesman Problem

Komivojažiera problēma (*traveling salesman problem*) klasiskajā variantā nozīmē šādu problēmas uzstādījumu.

Dots:

- *n* pilsētas,
- ir zināms attālums no katras pilsētas uz katru,

Uzdevums:

- atrast īsāko maršrutu, tādu, kura laikā tieši vienu reizi tiktu pārstaigātas visas pilsētas un atgrieztos sākotnējā pilsētā.

Sekojošajā tabulā problēma definēta pie *n*=20, uzrādot katras pilsētas koordinātes plaknē (piemērs ņemts no [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

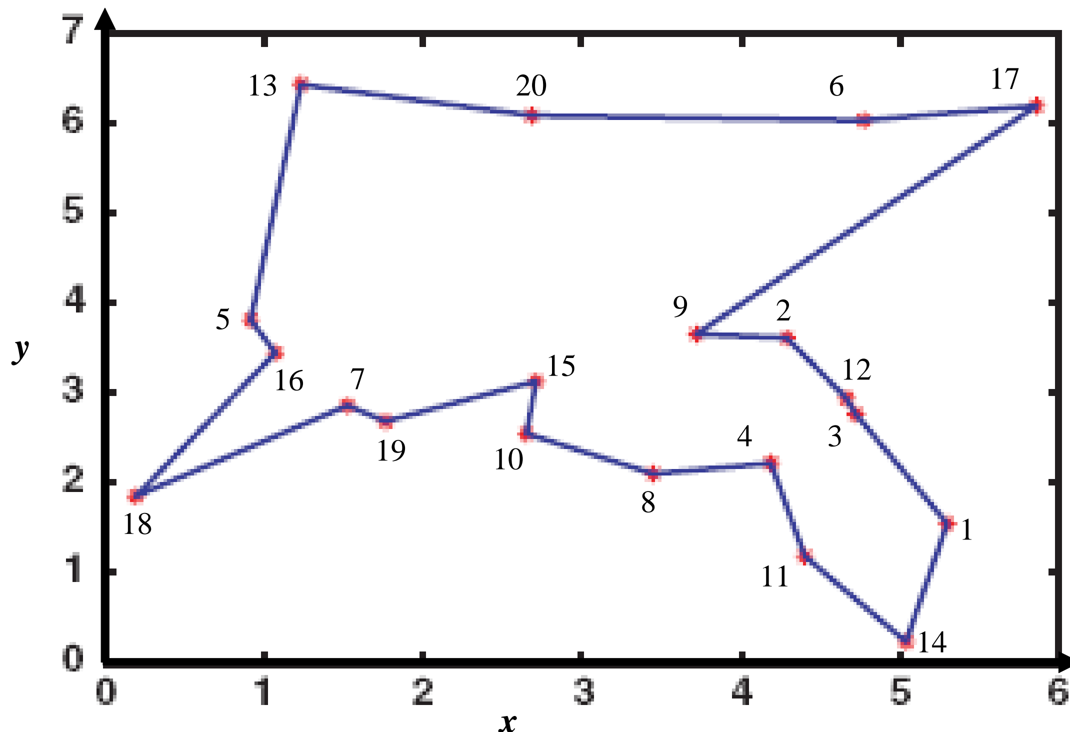
Attālumu *d* (*distance*) jeb svaru starp diviem stāvokļiem *s1* un *s2* var aprēķināt, lietojot kādu no metrikām, piemēram, Eiklīda attālumu:

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

Tūres *T* kopējais garums *L* vienāds ar visu attālumu kopsummu tūrē:

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

Pietiekoši labs maršruts šeit būtu: 1 → 14 → 11 → 4 → 8 → 10 → 15 →19 → 7 → 18 → 16 → 5 → 13 → 20 → 6 → 17 → 9 → 2 → 12 → 3 → 1 (ar kopējo garumu jeb svaru 24.5222), kuru ilustrē zīmējums:



6.2. Algoritma realizācija

6.2.1. Tūres ģenerēšana

Tūre (*tour*, jeb epizode *Reinforcement learning* terminoloģijā) ir vienas skudras (aģenta) veikta pilna grafa virsotņu apstaigāšana, atgriežoties sākotnējā virsotnē.

Risinot problēmu ar skudru kolonijas optimizācijas algoritmu, ir nepieciešams attālumam pretējs jēdziens – redzamība (*visibility*) η:

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

Tūres ģenerēšanas viens no svarīgākajiem elementiem ir izvēlēties nākošo soli (resp., nākošo virsotni grafā, jeb TSP gadījumā, pilsētu, uz kuru iet). Šim nolūkam ir jāizrēķina varbūtības katrai iespējamai virsotnei.
Ņemot vērā aprēķināto varbūtību sadalījumu, notiek nākošā stāvokļa noteikšana:

Function ***next_state*** **(***s*, *S*, *N*, *τ*, *α*, *β***) Returns** *statenext*
s – stāvoklis, no kura jāizvēlas nākošais solis
S() – visu stāvokļu kopa (zināms attālums starp jebkuriem diviem stāvokļiem)
N() – *S* apakškopa, kas pieejama nākošā soļa izvēlei
τ – feromona vērtības katrai pārejai no viena stāvokļa uz otru ||*S*|| × ||*S*||
α – feromona faktors (cik lielā mērā ņem vērā to, ka šī šķautne bieži apstaigāta)
β – distances faktors (cik lielā mērā ņem vērā to, ka divi stāvokļi ir tuvu viens otram)
η(·,·) – redzamība (*visibility*) – apgriezta vērtība attālumam: η(*s1*,*s2*) = 1 ÷ d(*s1*,*s2*)
Begin
% varbūtību sadalījuma iegūšana
For *i* := 1 **To** ||*N*|| **Do** *% pa visiem pieejamajiem stāvokļiem (no dotā stāvokļa)*
s2 := *N*(*i*)
P(*i*) := τ(*s*,*s2*)^*α* * η(*s*,*s2*)^β
Endfor
P := *P* / ***sum***(*P*)
% kumulatīvās varbūtību virknes aprēķins
For *i*:=2 **To** ||*P*||-1 **Do**
P(*i*) := *P*(*i*) + *P*(*i*-1)
Endfor
P(||*P*||) := 1
% stāvokļa izvēle
r := gadījuma skaitlis robežās [0, 1].
num := 1
While *r* > *P*(*num*)
num := *num* + 1
Endwhile
statenext := *N*(*num*) *%atgriež nākamo stāvokli*
End

Piemērs funkcijas *next_state* pārbaudei #1:

Dota visu stāvokļu virkne *S* (garumā 20) ar attālumiem starp stāvokļiem (*tsp.txt*, sk. augstāk)

Dota pieejamo stāvokļu virkne *N* (*n.txt*) =

4
8
10

Dota matrica τ ar izmēru ||S||×||S||, aizpildīta ar 1.

s = 3; α = 2; β = 2

Palaist funkciju *next_state* un fiksēt starprezultātus (1) varbūtību sadalījumu (P, pirms kumulācijas), (2) kumulatīvo varbūtību virkni pēc kumulācijas.

Ievērojiet, ka N sastopamie stāvokļi ir dažādā attālumā no 3. stāvokļa, un tuvākajiem ir lielāka varbūtība tikt izvēlētiem.

Rezultāta piemērs.

Varbūtību sadalījums:

0.705943	0.199372	0.0946855

Kumulatīvā varbūtību virkne:

0.705943	0.905314	1

Piemērs funkcijas *next_state* pārbaudei #2:

Doti tie paši ievaddati.

Palaist funkciju *next_state* 20 reizes un fiksēt izvēlēto nākamo stāvokli.

Rezultāts, piemēram:

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

(vidēji statistiski visbiežāk 4, visretāk 10).
Tūre ir stāvokļu virkne, kurā vienu reizi pārstāvēti visi stāvokļi, izņemot sākuma stāvokli, kas ir arī beigu stāvoklis.

Function ***generate_tour*** **(*S*, *τ*, *α*, *β*)** **Returns** *episode*
- *S* – visu stāvokļu kopa (zināms attālums starp jebkuriem diviem stāvokļiem)
- *N* – *S* apakškopa, kas pieejama nākošā soļa izvēlei
- τ – feromona vērtības katrai pārejai no viena stāvokļa uz otru ||*S*|| × ||*S*||
- α – feromona faktors (cik lielā mērā ņem vērā to, ka šī šķautne bieži apstaigāta)
- β – distances faktors (cik lielā mērā ņem vērā to, ka divi stāvokļi ir tuvu viens otram)
- *episode* – uzģenerētā stāvokļu virkne
- ***next_state*** – nākošā stāvokļa izrēķināšana

Begin

startstate := izvēlas epizodes sākuma stāvokli (parasti vienmēr vienu un to pašu)
N := *S* \ {*startstate*} *% pieejamie stāvokļi nākošajam solim*
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* *% epizode sākas un beidzas ar sākuma stāvokli*

End

Piemērs funkcijas *generate_tour* pārbaudei:

Dota visu stāvokļu virkne *S* ar attālumiem starp stāvokļiem (*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

Dota matrica τ ar izmēru ||S||×||S||, aizpildīta ar 1.

$\alpha = 2$; $\beta = 2$

Palaist funkciju *generate_tour* 10 reizes (sākot ar stāvokli 1) un pārbaudīt ģenerētās epizodes.

Rezultāta piemērs (10 dažādas tūres, tūre beidzas ar sākuma stāvokli):

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

6.2.2. Skudru kolonijas algoritma kopsavilkums

Skudru koloniju algoritmā feromons (*pheromone*) ir jēdziens, kas reprezentē noteiktas grafa šķautnes popularitāti grafa apstaigāšanā.

```
Procedure ant_colony_optimization (maxit, popsize, α, β, ρ, Q, τ0) Returns best_tour
    S() – visu stāvokļu kopa
    maxit – kopējais veicamo kampaņu skaits
    popsize – skudru skaits eksperimentā
    α – feromona faktors (cik lielā mērā ņem vērā to, ka šī šķautne bieži apstaigāta)
    β – distances faktors (cik lielā mērā ņem vērā to, ka divi stāvokļi ir tuvu viens otram)
    ρ – feromona iztvaikošanas faktors
    Q – feromona izmaiņu aprēķināšanas koeficients
    τ() – feromona tabula izmērā ||S|| × ||S|| (viena vērtība katrai stāvokļu pārejai)
    τ0 – feromona sākuma vērtība
    L(·) – tūres (ceļa) garums
    generate_tour – vienas tūres ģenerēšana vienai skudrai
Begin
    τ := visu feromona tabulu aizpilda ar nelielām vērtībām τ0
    For it:=1 To maxit  % pa visām epohām
          % (A) ģenerē tūres visām skudrām, T
        For p:=1 To popsize  % pa visām skudrām
            T(p) := generate_tour (S, τ, α, β)
        Endfor
          % (B) piemēro feromona iztvaikošanu visām stāvokļu pārejām
        Forall s,s2 ∈ S Do % pa visiem punktu pāriem
            τ(s,s2) := (1 – ρ) · τ(s,s2)
        Endforall
          % (C) papildina feromonu pēc pēdējo ģenerēto tūru informācijas
        For p:=1 To popsize  % pa visām skudrām (t.i. skudru veiktajām tūrēm)
            Forall (s,s2) In T(p)  % pa visām pārejām tūrē
                τ(s,s2) := τ(s,s2) + Q / L(T(p))
            Endforall
        Endfor
    Endfor
    best_tour := īsākā sastaptā tūre (ar mazāko L)
End
```

6.3. Laboratorijas darba veicamo programmēšanas darbu kopsavilkums

- Ielādēt datora atmiņā failu „tsp.txt”, kas nosaka katra stāvokļa koordinātes (nodaļa 6.1)
- Realizēt viena soļa ģenerēšanu ***next_state*** (nodaļa 6.2.1).
- Realizēt epizodes ģenerēšanu ***generate_tour*** (nodaļa 6.2.1).
- Realizēt skudru kolonijas algoritmu ***ant_colony_optimization*** ar parametriem *popsize*=20, α=1.3, β=2.5, *maxit*=100, ρ=0.5, Q=0.5, τ_0=0.001
- Laižot algoritmu 50 reizes, ar ļoti lielu varbūtību parādīsies risinājums (tūre) ar garumu $L < 26$.
- Labākā tūre, ko man pašam pagaidām ir izdevies iegūt (ar doto konfigurāciju), ir {1,14,11,4,8,10,15,7,19,18,16,5,13,20,6,17,9,2,12,3,1} ar garumu L= 24.7954.

7. Atbalsta vektoru mašīna (*support vector machine, SVM*)

Atbalsta vektoru mašīna (Vapnik, 1995) (turpmāk SVM) paredzēta klasifikācijas problēmas risināšanai, un savā klasiskajā formā – tieši binārajai klasifikācijai (t.i., divās klasēs), kāda tiks veikta arī šeit. Laboratorijas darbā piedāvāts vienkāršots iteratīvais SVM variants, kur ideja aizgūta no (Roobaert, 2000) un (Vishwanathan and Murty, 2002).

Laboratorijas darbs sastāv no divām daļām:

1) Iteratīva SVM izveide lineāru problēmu atrisināšanai (nodaļas 7.1 un 7.2)

2) SVM pārveidošana nelineāru problēmu risināšanai (nodaļas 7.3 un 7.4)

Par bāzes apmācošo sistēmu šajā SVM tiek lietots viens lineārs perceptrona neirons.

7.1. Divu lineāru problēmu formulējums priekš SVM

7.1.1. Problēma #1. Būla funkcijas ‘AND’ modelēšana

Doti 4 piemēri, apraksta loģisku funkciju *(x_1 **AND** x_2)*. SVM īpatnība ir tā, ka vēlamās vērtības, kas atbilst divām klasēm ir {-1, 1}. (**NB!** Šajos piemēros ievērojiet nedaudz „dīvaino” paraugu secību – vispirms pozitīvie, tad negatīvie)

x_1	x_2	d
1	1	1
0	1	-1
1	0	-1
0	0	-1

Modelēt funkciju ‘AND’ formā

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

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

Papildus nosacījums.

Modelēt funkciju „pārliecinošāk”, t.i, lai NET vērtības nebūtu tuvu 0:

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

ξ (grieķu burts *xi*) – tolerances koeficients (*slack rate*), pie ξ=1 formulējums atbilstu sākotnējam variantam.

Pieejamie dati.

posand.txt un *negand.txt* – pozitīvie un negatīvie apmācības piemēri

- *POS* – ievaddati (*posand.txt*):

1	1

- *NEG* – ievaddati (*negand.txt*):

0	1
1	0
0	0

Ievadinformāciju var pārformulēt citādi:

- *D* – vēlamais rezultāts:

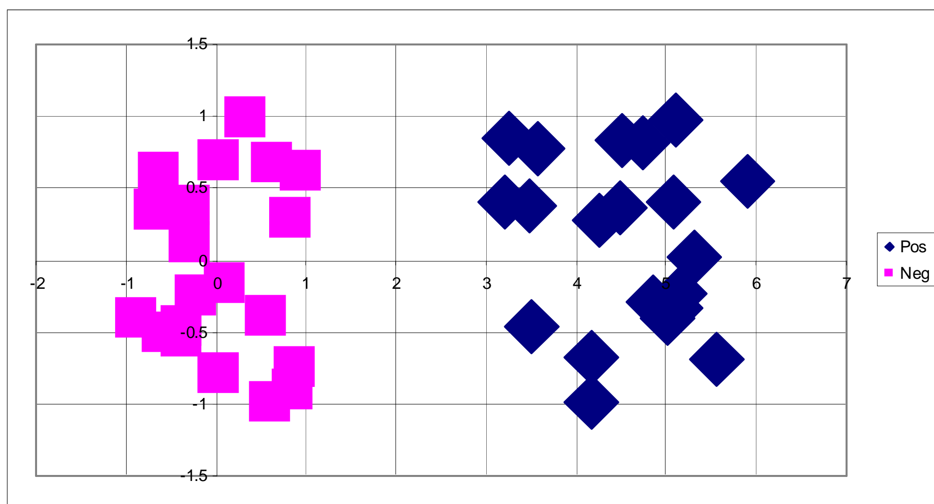
1
-1
-1
-1

E – ievaddati:

1	1
0	1
1	0
0	0

7.1.2. Problēma #2. Lineāra punktu sadalīšana plaknē

Doti 40 apmācības piemēri (*pos1a.txt* un *neg1a.txt* – pozitīvie un negatīvie apmācības piemēri). Problēmai #2 dots tikai ievaddatu grafiks:



Modelēt funkciju tāpat kā AND gadījumā.

7.2. Iteratīvā SVM algoritma realizācija lineāras problēmas gadījumā

Lai izveidotu lineāro algoritmu, jāuztaisa trīs moduļi:

1) Perceptrona darbināšana ***run_perceptron***

2) Perceptrona apmācīšana ***train_perceptron***

3) Kopējais modulis ***svm_simple***, kas izsauc abus iepriekšējos

7.2.1. Viena parauga darbināšana ar lineāru perceptronu

Funkcija ***run_perceptron*** veic neironu tīkla *W* darbināšanu ar vienu paraugu *X*.

```
Function run_perceptron (X, W) Returns y
        X – ieejas paraugs: x1.. xn
        W – modeļa svari: w0..wn
        y – modeļa sniegtais rezultāts
Begin
        NET := W(0)
        For i := 1 To |X| Do  % pa visiem 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
```

Testa piemērs funkcijas ***run_perceptron*** pārbaudei:

W (w_0, w_1, w_2) =

-2.5	1.9	1.7

X(4) – ievaddati (x_1, x_2):

1	1
0	1
1	0
0	0

Izsaucot funkciju ***run_perceptron*** 4 reizes uz katru no ievaddatiem, sanāk šādi 4 rezultāti:
1
-0.8
-0.6
-1

7.2.2. Lineāra perceptrona apmācīšana uz paraugu kopas

Funkcija ***train_perceptron*** veic pilnu neironu tīkla apmācību uz doto paraugu kopu (*E*, *D*).

```
Function train_perceptron (E, D) Returns W
    E – ieejas paraugi
    D – ieejas paraugu pareizās vērtības
    W– iegūstamie svari
    MaxEpochs– maksimālais tīkla apmācības epohu skaits
    ε – maksimālā tīkla kļūda
    LearningRate – apmācības koeficients
Begin
    W – inicializē svarus ar gadījuma vērtībām [-0.3, 0.3]
    epoch := 0
    neterror := ∞
    While neterror > ε And epoch < MaxEpochs Do  % apmācības cikls
        epoch := epoch + 1
        neterror := 0
        For k := 1 To |E| Do  % pa visiem paraugiem (šeit tikai pa atbalsta vektoriem!)
            y := run_perceptron (E(k), W)  % parastā tīkla izeja
            δ := D(k) – y  % parauga kļūda jeb diference pret pareizo atbildi
            W(0) := W(0) + LearningRate * δ  % papildus svara apmācība
            For i := 1 To |E(k)| Do  % pa svariem
                W(i) := W(i) + LearningRate * δ * E(k,i)
            Endfor
            neterror := neterror + δ * δ
        Endfor
        neterror := sqrt (neterror)
    Endwhile
End
```

Testa piemērs un konfigurācija funkcijas ***train_perceptron*** pārbaudei:

E – ievaddati:

1	1
0	1
1	0
0	0

D – vēlamās atbildes:

1
-1
-1
-1

MaxEpochs = 1000

ε = 0.01

LearningRate = 0.1

Iegūstamās svaru virknes piemērs:

W (w_0, w_1, w_2) =

-2.9	1.9	1.9

Aptuvenais konverģencei nepieciešamais epohu skaits pie šādiem parametriem – 150.

7.2.3. Lineārā iteratīvā SVM algoritma kopsavilkums

Algoritms ir ***svm_simple*** ir SVM algoritma vienkāršota iteratīvā versija.

```
Procedure svm_simple (E, D) Returns W
        E – ieejas paraugi
        D – ieejas paraugu pareizās vērtības
        W– iegūstamie svari
        SVMERR – mašīnas kļūda
        SlackRate – tolerances koeficients
        train_perceptron – neironu tīkla pilna apmācība uz visiem paraugiem
        run_perceptron – neironu tīkla vienreizēja darbināšana ar vienu paraugu
Begin
         % A) nosaka sākotnējo atbalsta vektoru sarakstu
        S := {sp,sn}  % no E izvēlas divus tuvākos punktus pretējās klasēs (pēc Eiklīda metrikas)
        SD – atbalsta vektoru pareizās atbildes
        Loop Forever
                 % B) apmāca percpetronu un novērtē rezultātu
                W := train_perceptron (S, SD)  % apmāca (tikai) uz atbalsta vektoriem
                For e := 1 To |E| Do  % darbina tīklu uz visiem paraugiem, ne tikai atbalsta vektoriem
                        y := run_perceptron (E(e), W)  % parastā tīkla izeja
                        SVMERR(e) := 1 – SlackRate – y * D(e);
                Endfor
                 % C) atrod sliktāko paraugu starp tiem, kas nav atbalsta vektori un pievieno tiem
                If max(SVMERR) > 0 And |S|<|E| Do  % Ja ir sliktie paraugi un ir vēl atb. vektori
                        e := maxindex_i (SVMERR(i)) Where E(i) Not In S
                        S := S + E(e)
                        SD – atjauno atbilstoši jaunajam S
                Else
                        Return
                Endif
        Endloop
End
```

Palaišanas nosacījumi nākošajā sadaļā.

7.2.4. Sistēmas konfigurācija un iegūstamais rezultāts

Algoritms svm_simple tiek darbināts ar šādiem parametriem:

MaxEpochs = 1000

ε = 0.01

LearningRate = 0.1

SlackRate = 0.5

Abu problēmu gadījumā pietiek ar 3 atbalsta vektoriem.

Problēmas #1 risinājums

Iegūtā svaru virkne W (w_0, w_1, w_2) =

-3	2	2

Izmantotie atbalsta vektori (3 no 4 iespējamiem):

1110

Atbalsta vektoru atšifrējums (sk. failus *posand.txt* un *negand.txt*):

(1,1)

(0,1)

(1,0)

Aptuvenais nepieciešamo epohu skaits abās SVM iterācijās – 200.

Problēmas #2 risinājums

Iegūtā svaru virkne W (w_0, w_1, w_2) =

-1.7829466	0.86973058	-0.03622049

Izmantotie atbalsta vektori (3 no 40 iespējamiem, kā redzams viens no pozitīvajiem, bet 2 no negatīvajiem paraugiem):

00000000100000000000 00010010000000000000

Atbalsta vektoru atšifrējums (sk. failus *pos1a.txt* un *neg1a.txt*):

Tas pozitīvais, kas novietots visvairāk pa kreisi:

(3.212471537, 0.408951459),

Tie negatīvie, kas novietoti vairāk pa labi:

(0.935840634, 0.615499651)

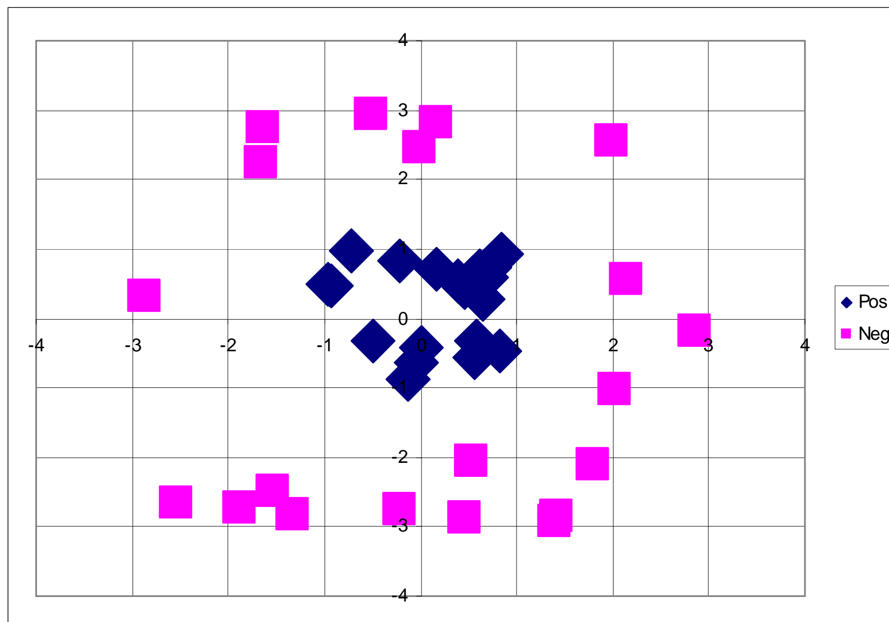
(0.863338348, -0.906680607)

Aptuvenais nepieciešamo epohu skaits abās SVM iterācijās – 70.

7.3. Nelineāras problēmas formulējums priekš SVM

Problēma #3. Nelineāra punktu sadalīšana plaknē

Doti 40 piemēri (*pos2a.txt* un *neg2a.txt*), kur pozitīvie piemēri nav nodalāmi no negatīvajiem lineārā veidā:



Modelēt funkciju kā iepriekš, bet ar papildus nosacījumiem pret apmācošās sistēmas struktūru.

Tā abas klases nav lineāri atdalāmas, tad pirms apmācības ievades paraugs jāpārveido uz lineāri atdalāmu punktu telpu.

Šim nolūkam parasti izmanto t.s. kodola funkcijas (*kernel functions*) un tādējādi var teikt kā pārveidošana notiek no ievades telpas (*input space*) uz kodolu telpu (*kernel space*) un amācība pēc tam notiek uz datiem no kodolu telpas.

Kodola funkcija K (grieķu burts *kappa*) ir pārveidošanas mehānisma elements, kuram ir divi parametri, kas abi ir vektori – viens no kuriem ir atbalsta vektors.

Viens no biežāk lietotajiem kodolu veidiem ir radiālo bāzes funkciju kodols (*RBF kernel*), kuru definē šādi:

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

kur σ – slīpuma koeficients, ||x-y|| – Eiklīda attālums:

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

Lai pārveidotu ievadīto paraugu, kodola funkcija ir jāizsauc tik reizes, cik atbalsta vektori ir definēti.

Pārveidošanas funkcija Φ mūsu gadījumā ar 2 ieejām izskatās šādi:

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

kur *x* – pārveidojamais paraugs, $s_1,\ldots,s_m$ – atbalsta vektori.

7.4. Iteratīvā SVM algoritma realizācija lineāras problēmas gadījumā

Nelineārā algoritma galvenā atšķirība ir **pārveidošanas funkcijas** ieviešana, kas pārveido ieejas datus lineāri atdalāmā formā.

7.4.1. Pārveidošanas funkcijas realizācija

Funkcija ***phi_mapping*** veic viena parauga pārveidošanu par paraugu, kas ir garumā atbalsta vektoru skaits.

```
Function phi_mapping (x, S) Returns y
        x – ieejas paraugs: x1.. xn
        S – Atbalsta vektoru virkne: S0..Sm
        y – pārveidotais paraugs
Begin
        y.resize (|S|)
        For i := 1 To |S| Do  % pa visiem atbalsta vektoriem
                y(i) := K(Si, x)
        Endfor
End

Function K (a, b) Returns y
        a – ieejas paraugs #1
        b – ieejas paraugs #2
        σ (sigma) – RBF funkcijas slīpuma koeficients
Begin
        NET := 0
        For i := 1 To |a| Do  % pa visiem paraugu elenentiem
                NET := NET + (ai – bi)^2
        Endfor
        y := exp (-NET / (2 * σ^2))
End
```

Testa piemērs funkcijas ***phi_mapping*** pārbaudei:

Problēma #1: AND modelēšana

σ = **0.3**

X – ievaddati (*posand.txt* + *negand.txt*):

1	1
0	1
1	0
0	0

Doti šādi atbalsta vektori *S* (pirmie 3):

(1,1)

(0,1)

(1,0)

Izsaucot funkciju ***phi_mapping*** 4 reizes uz katru no ievaddatiem, sanāk šādi 4 rezultāti:

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

Problēma #3: Nelineāra punktu atdalīšana

$\sigma = 3$

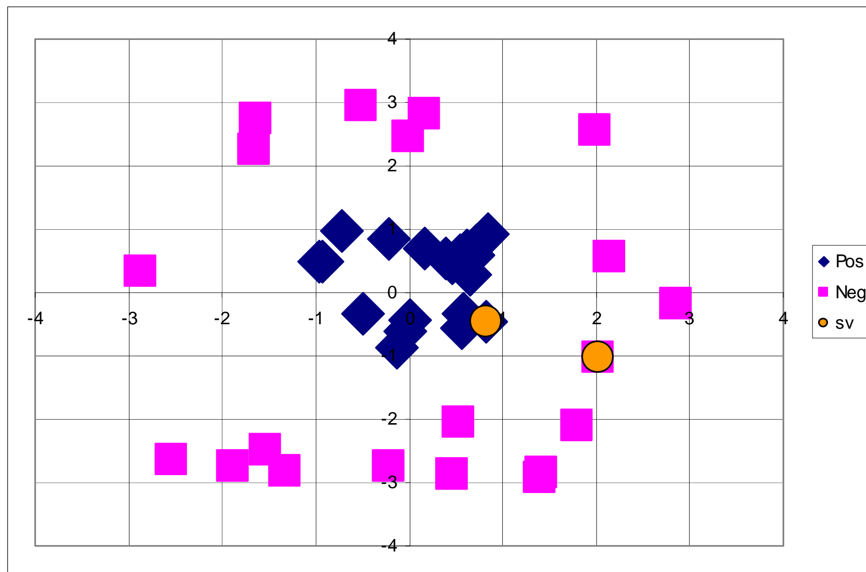
X – ievaddati (*pos2a.txt* + *neg2a.txt*):

Doti šādi atbalsta vektori *S* (#13 un #35):

(0.831363001, -0.473996589)

(2.021344674, -1.016343516)

Atbilstošais grafiks ievaddatiem, atsevišķi izceļot (pirmos divus) atbalsta vektorus:

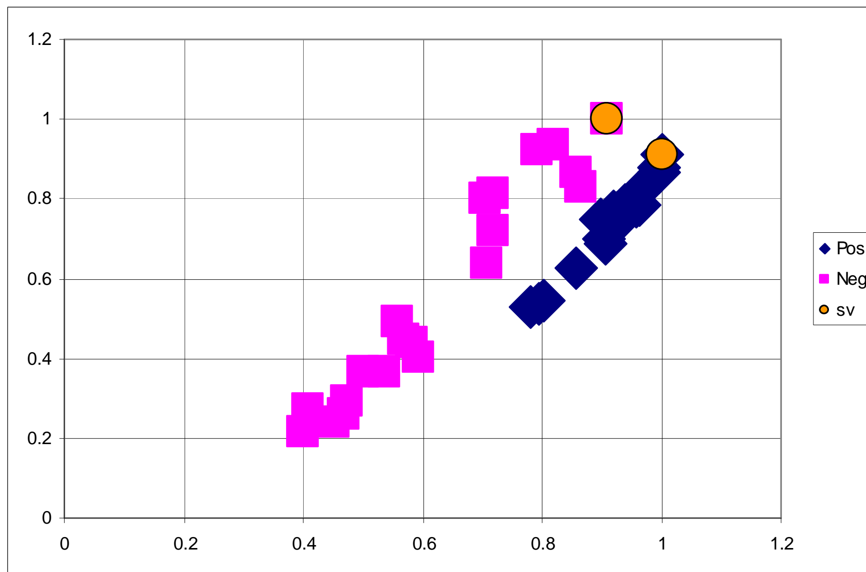


Izsaucot funkciju ***phi_mapping*** 40 reizes uz katru no ievaddatiem, sanāk šādi 40 rezultāti (ar dzeltenu atzīmēti atbalsta vektori):

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

Atbilstošais grafiks, pēc pārveidošanas funkcijas pielietošanas, atsevišķi izceļot atbalsta vektorus:



7.4.2. Nelineārā iteratīvā SVM algoritma kopsavilkums

Lai pārveidotu lineāro SVM un nelineāro, jāveic dažas nelielas izmaiņas pieņemot, ka ir izveidota pārveidošanas funkcija. Pirmo 3 (no 4) izmaiņu būtība ir, ka (parasto) paraugu vietā tiek izmantoti transformētie paraugi.

Izmaiņa #1

Funkcijā ***train_perceptron*** rindiņu:

y := ***run_perceptron*** (*E*(k), *W*) % parastā tīkla izeja

nomaina par

y := ***run_perceptron*** (***phi_mapping*** (*E*(*k*), *S*), *W*) % parastā tīkla izeja,

kur *S* – atbalsta vektoru virkne

Izmaiņa #2

Funkcijā ***train_perceptron*** rindiņu:

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

nomaina par

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

kur *S* – atbalsta vektoru virkne

Izmaiņa #3

Funkcijā ***svm_simple*** rindiņu:

y := ***run_perceptron*** (*E*(*e*), *W*) *% parastā tīkla izej*

nomaina par

y := ***run_perceptron*** (***phi_mapping*** (*E*(*e*), *S*), *W*) % parastā tīkla izeja

Izmaiņa #4

Funkcijas ***svm_simple*** pašā sākumā pirms svaru vērtību uzstādīšanas pievienot:

W.***resize*** (|*S*|+1) *% jo svari ir ar numuriem 0..|S|*

7.4.3. Nelineārās sistēmas konfigurācija un iegūstamais rezultāts

Problēmas #3 risinājums

MaxEpochs = 1000

$\varepsilon = 0.1$

LearningRate = 0.5

SlackRate = 0.95

$\sigma = 3.0$

Šeit pietiek ar 5 atbalsta vektoriem, bet vajag par vienu svaru vairāk, jo tiek izmantotas kodola funkcijas ievaddatu pārveidošanai.

Iegūtā svaru virkne W ($w_0, w_1, w_2, w_3, w_4, w_5$) =

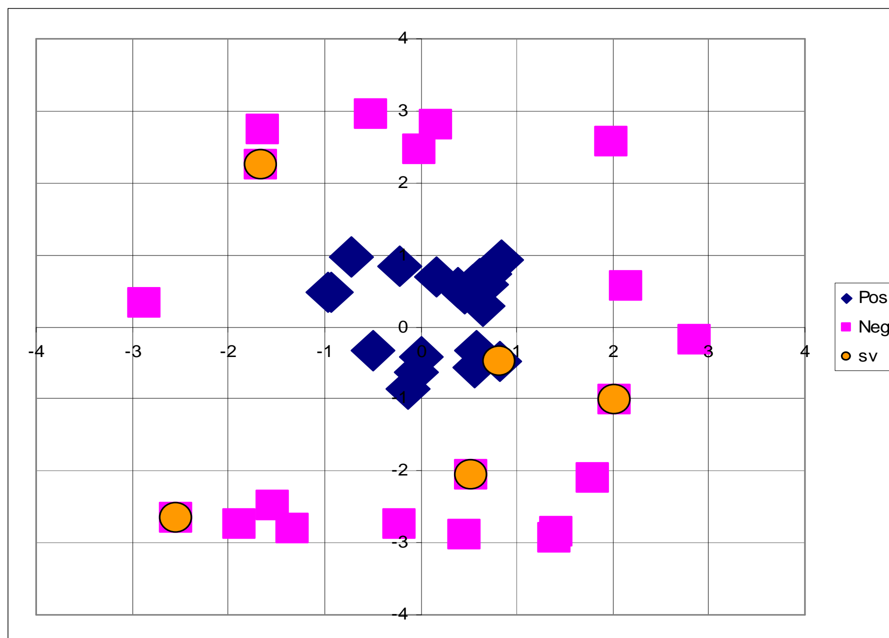
-12.342	12.0777	-2.242	-1.0837	6.0707	0.91796

Izmantotie atbalsta vektori (5 no 40 iespējamiem, kā redzams viens no pozitīvajiem, bet 2 no negatīvajiem paraugiem):

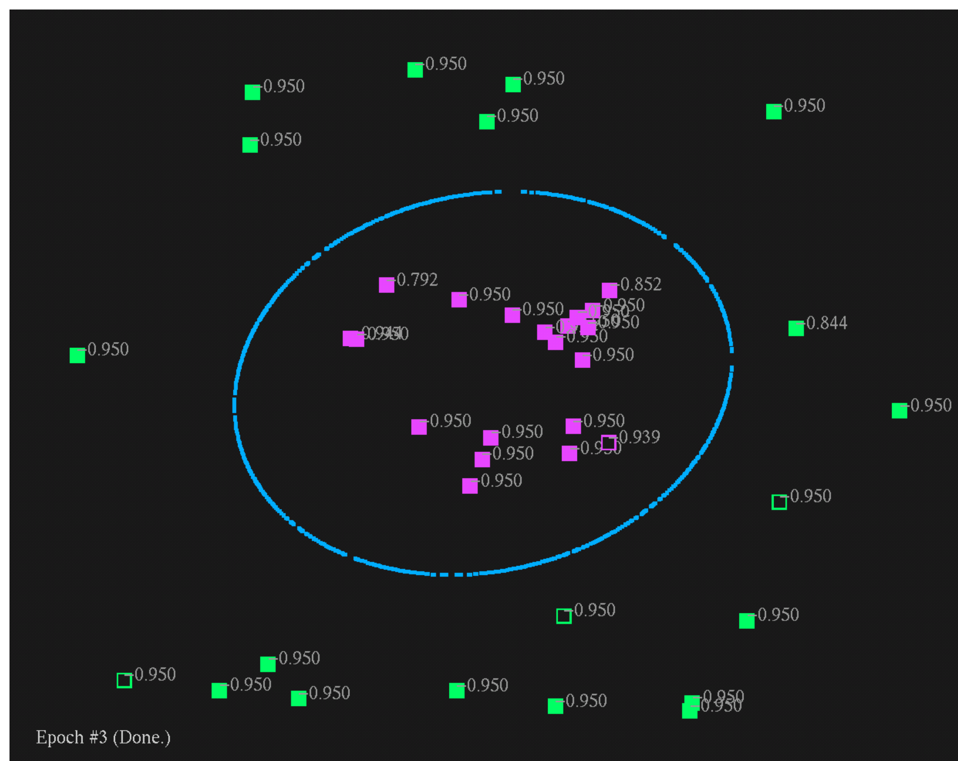
00000000000010000000 10011000000000100000

Aptuvenais nepieciešamo epohu skaits SVM iterācijās – 200.

Ieskatam ievaddatu grafiks ar visiem 5 atbalsta vektoriem:



Aleksandrs Polocks laipni piedāvāja savu programmas ģenerēto rezultātu, turklāt ar visu nelineāro atdalīšanas līniju (atbalsta vektori ir tie kvadrātiņi, kas ar tukšo vidu):


0.950
0.950
0.950
0.950
0.950
0.950
0.792
0.950
0.852
0.950
0.950
0.844
0.950
0.950
0.950
0.950
0.950
0.950
0.939
0.950
0.950
0.950
0.950
0.950
0.950
0.950
0.950
0.950
0.950
0.950
0.950
0.950
Epoch #3 (Done.)

8. Daļēja novērojamība un POMDP. Tīģera problēma

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).

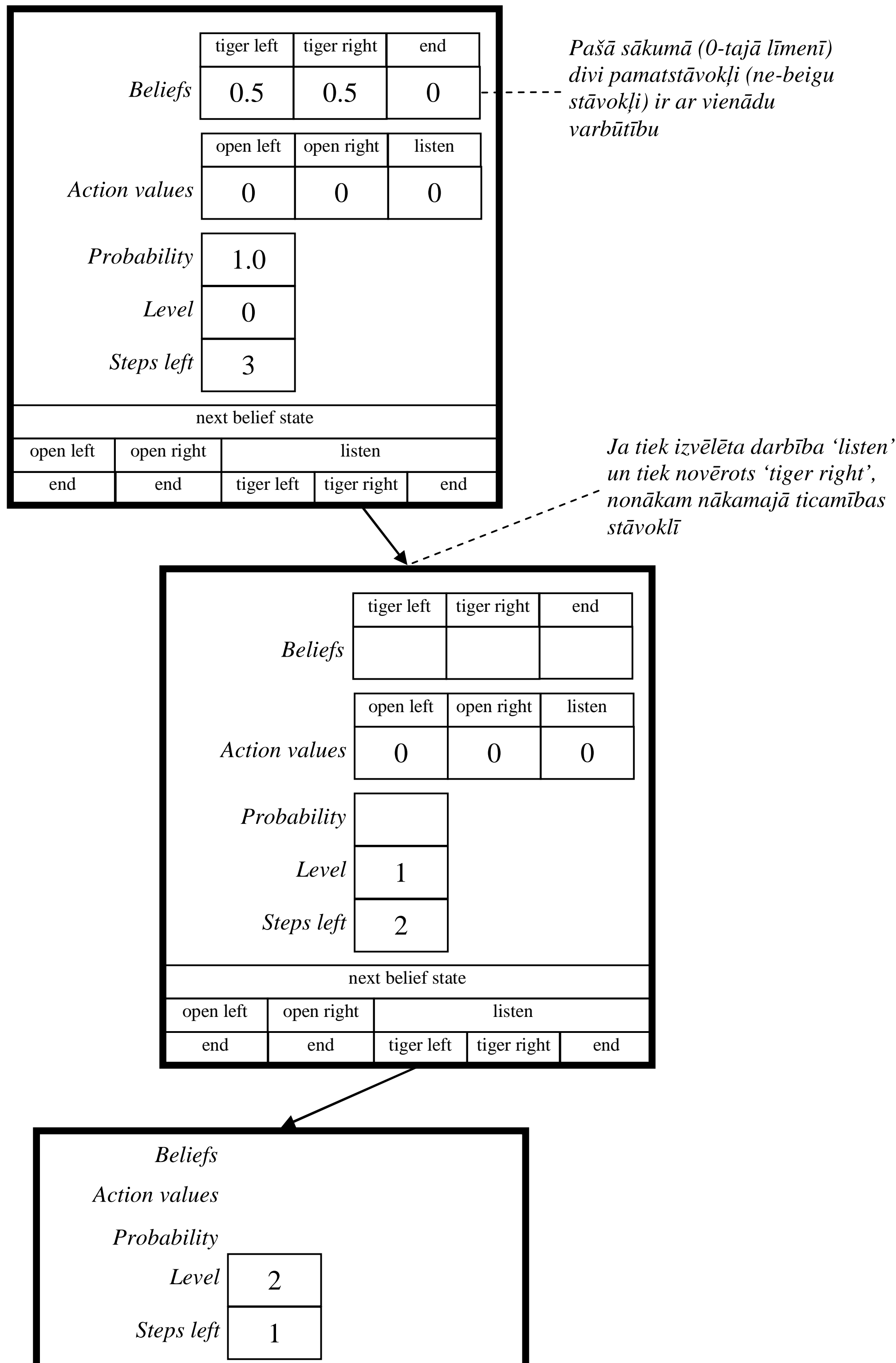
Viss apmācības process būtu iedalāms 3 fāzēs:

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).



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
```