LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

MĀKSLĪGĀ INTELEKTA IZSTRĀDE KĀRŠU SPĒLEI „ZOLE”

BAKALAURA DARBS

Autors: **Gustavs Venters**
Studenta apliecības Nr.: gv11017
Darba vadītājs: pētnieks Mg. dat. Kaspars Balodis

RĪGA 2015

ANOTĀCIJA

Bakalaura darbā tiek pētīts, kā automatizēt kāršu spēles Zole spēlēšanu un kādas ir iespējas izveidot mākslīgā intelekta Zoles spēlētāju. Zoles spēle tiek analizēta ar datorzinātnes un spēļu teorijas līdzekļiem. Tiek pētīti spēlei piemēroti algoritmi un aģentu risinājumi, kas tos izmanto. Zole ir stohastiska, nepilnas informācijas spēle, kas padara to sarežģītu datora analīzei.

Bakalaura darba gaitā tiek secināts, ka laba aģenta izveidošanai nepieciešams apvienot Zolei raksturīgu heiristiku izmantošanu ar algoritmisku spēles koka analīzi. Aģenta modelēšanā var izmantot algoritmus, kas izmantoti līdzīgā kāršu spēlē Skat.

Atslēgvārdi: spēļu teorija, Zole, mākslīgais intelekts, meklēšana spēles kokā

ABSTRACT

DEVELOPING ARTIFICIAL INTELLIGENCE

FOR CARD GAME “ZOLE”

The purpose of this bachelor thesis is to analyze methods of automation and possibilities to develop an artificial intelligence player for card game Zole. Game of Zole is studied by the means of computer science and game theory. Algorithms suitable for the game and agent solutions that use them are being explored. Zole is stochastic, partial information game, which makes it sophisticated for computer analysis.

In the course of the work it is inferred that characteristic heuristics of Zole and algorithmic game tree analysis needs to be combined in order to make a good Zole playing software agent. Skat research can also be applied to modeling artificial intelligence for Zole.

Keywords: game theory, Zole, artificial intelligence, game tree search

SATURS

Apzīmējumu saraksts....4
Ievads....6
1.Kāršu spēle Zole....8
1.1.Zoles noteikumi....8
1.2.Spēles raksturojums....11
1.3.Spēles izvēļu koks....13
2.Mākslīgā intelekta risinājumi....17
2.1.Zoles risinājumi....17
2.2.Skat risinājumi....17
2.3.Risinājumu metodes....19
3.Izvēlētais risinājums....22
3.1.Izspēļu skaits un tā analīze....22
3.2.Optimizācija un binārā loģika....23
3.3.Risinājuma arhitektūra....25
Rezultāti....27
Secinājumi....30
Izmantotā literatūra un avoti....32
Pielikumi....34
1. pielikums. Nejaušu kāršu sadalījumu iespējamo izspēļu skaiti....34
2. pielikums. Gājienu aprēķināšanas ilgums – zīmīgās vērtības 100 nejaušiem sadalījumiem....35
3. pielikums. Klases Deck.cs pirmkods....36
4. pielikums. Klases Player.cs metode PlayCard....42
5. pielikums. Klases Player.cs metode SimulateGame....44

APZĪMĒJUMU SARAKSTS

Stiķis – kāršu izspēles cikls, kurā katrs spēlētājs pēc kārtas uzliek vienu kārti. Spēlētājs, kurš uzlicis stiprāko kārti ir stiķa uzvarētājs un iegūst (bet nepievieno rokai) stiķī izspēlētās kārtis.

Stiķu spēle – kāršu spēle, kurā kāršu izspēle notiek ar stiķiem. Parasti tām rezultātu nosaka pēc stiķos iegūto kāršu punktiem.

Masts – kārts īpašība. Zolē spēlētājiem stiķos jāliek vienādu mastu kārtis ar pirmā gājēja kārti (ja viņam tādi ir). Kārtis var būt stiprākas tikai par tā paša masta kārtīm.

Trumpis – īpašs kāršu masts, kura kārtis ir stiprākas par pārējo mastu kārtīm.

Zolētājs – Zoles spēlētājs, kurš izvēlējies spēlēt viens pret divu pārējo spēlētāju komandu.

Lielais – Zoles spēlētājs, kurš izvēlējies spēlēt viens pret divu pārējo spēlētāju komandu, papildus piepērkot galda kārtis.

Mazais – Zoles spēlētājs, kuram komandā ar otru mazo jāspēlē pret lielo vai zolētāju.

Galdiņš vai **piepirkuma kārtis** – divas kārtis, ko spēles sākumā neizdala un neparāda spēlētājiem. Tās piedāvā pacelt spēlētājam, kurš vēlas būt lielais.

Piepirkšana – galdiņa kāršu pievienošana rokai, pēc tam kad spēlētājs izvēlas kļūt par lielo. Pēc kāršu pacelšanas lielajam divas kārtis no rokām jāpievieno pie jau uzvarēto stiķu kārtīm.

Acis – kārts vērtība. Uzvarēto stiķu kārtīm skaita acis, lai spēles beigās noteiktu uzvarētāju.

Skat – Vācijā un pasaulē populāra Zolei līdzīga kāršu spēle.

C# – Microsoft veidota programmēšanas valoda ar ērti lietojamām objektorientētās programmēšanas īpašībām.

Heiristika – teorētiski nepamatots, bet praksē bieži labi strādājošs algoritmisks uzlabojums. Heiristiku lietošana samazina algoritma precizitāti, bet palielina tā ātrdarbību.

Aģents – datora programma, kas, balstoties uz saņemto informāciju, veic lietotāja vai citas programmas darbības. Spēļu kontekstā aģents pilda spēlētāja funkciju un veic spēlētāja darbības, zinot spēlētājam pieejamo informāciju.

Nopasēt – Zoles izvēles daļā izvēlēties nespēlēt vienam pret abu pārējo pretinieku komandu.

Izvēles daļa – autora izvēlēts nosaukums Zoles partijas sākuma daļai, pirms izspēles daļas.

Izspēles daļa – autora izvēlēts nosaukums Zoles partijas daļai, kas sākas ar pirmā stiķa izspēli un beidzas ar partijas beigām.

IEVADS

Zole ir Latvijā visizplatītākā kāršu spēle, kas ir te arī veidojusies un attīstījusies. Latvijā ar Zoles spēles popularizēšanu aktīvi nodarbojas Latvijas Zoles federācija un regulāri šo prāta spēli spēlē aptuveni 200 000 cilvēku[1]. Zole ir līdzīga Vācijā spēlētajām kāršu spēlēm *Skat* un *Doppelkopf*, no kurām tā visticamāk ir cēlusies.

Šobrīd plašu popularitāti ir guvusi iespēja spēlēt zoli tiešsaistē, ko piedāvā dažādas interneta vietnes un programmas. Kā populārākās šādas vietnes ir pieminamas *draugiem.lv*, *zolmaniem.lv*, kā arī *spokuspeles.tvnet.lv*, kuras spēlēšanai piedāvā arī *Android* lietotni. Zoles spēlēšanai domātas programmas iespējams lejupielādēt vietnēs *kreicene.lv*, *kastanis.biz*, *ansis.lv*, pēdējās divas iespējams spēlēt bezsaistē [2, 3]. Līdz ar spēles pāreju no fiziskas kāršu kavas uz digitālu formātu, rodas arī interese un iespēja aizvietot spēlētājus ar datorā veidotu mākslīgo intelektu. Tomēr publiski pieejami maz šādu Zoles risinājumu, un to spēles veiklība ir zema. Tie ir balstīti uz spēlētāju pieredzē iegūtiem pareizās spēles likumiem, nevis uz iespēju zoli analizēt ar datoru, un to spēle nelīdzinās labu cilvēku spēles gaitai. Tādējādi paliek neatrisināta problēma – kā izveidot zoli spēlējošu mākslīgo intelektu, kas spētu būt līdzvērtīgs pretinieks cilvēkam ar labām spēles zināšanām.

Problēmas izpētei autors noteica šādus mērķus:

- Izpētīt Zoles spēli, analizēt to, izmantojot spēļu teoriju.
- Izpētīt Zoles spēles modelēšanas iespējas ar datoru, analizējot līdzīgu spēļu modelēšanas metodes un teorētiskās iespējas.
- Izveidot mākslīgo intelektu, kas spēj analizēt Zoles partiju un izvēlēties piemērotākos gājienus jebkurā spēles situācijā.

Bakalaura darbs sastāv no ievada, nodaļām „Kāršu spēle Zole”, „Mākslīgā intelekta risinājumi” un „Izvēlētais risinājums”, rezultātiem, secinājumiem, izmantoto avotu un literatūras saraksta un četriem pielikumiem. Nodaļā „Kāršu spēle Zole” ir aprakstīti spēles noteikumi, kā arī analizēti Zoles darbības principi un to ietekme uz aģenta modelēšanu. Nodaļā „Mākslīgā intelekta risinājumi” apskatīta citu autoru veiktie darbi veidojot mākslīgā intelekta risinājumus stiķu spēlēm. Nodaļā „Izvēlētais risinājums” ir izklāstīts autora piedāvātais risinājums Zoles aģenta izveidei. Rezultātu nodaļa apkopo darba veidošanas gaitā iegūtos rezultātus un to nozīmi Zoles mākslīgā intelekta modelēšanas procesā.

Darba gaitā autors secina, ka, modelējot Zoles mākslīgo intelektu, var izmantot Skat spēles modelēšanā uzkrāto informāciju. Veidojot Zoles aģentu, nepieciešams apvienot dažādus spēles analīzes algoritmus, kā arī jāizvēlas optimāls kāršu kodēšanas veids datorā.

1. KĀRŠU SPĒLE ZOLE

1.1. Zoles noteikumi

Zoles spēle ir paredzēta trim spēlētājiem. Tai eksistē arī variācijas diviem, četriem vai vairāk spēlētājiem, bet tās darbā netiks apskatītas. Zolē izmanto 26 no 52 standarta (franču) kāršu komplekta kārtīm. Zole tiek spēlēta partijās, pārmaiņus vienam spēlētājam spēlējot pret pārējiem diviem. Spēles mērķis ir savākt visvairāk uzvaras punktu vairākās partijās. Gājieni notiek stiķos – katrā gājienā katram spēlētājam jāizspēlē kārts un spēlētājs ar stiprāko kārti iegūst izspēlēto kāršu punktus. Zole ir ļoti līdzīga vācu kāršu spēlei Skat, ko spēlē ar 32 kārtīm.

Citēti Latvijas Zoles asociācijas Zoles noteikumi [4]:

„Spēlē izmanto 26 kārtis. Kāravi, sākot no dūža līdz septītniekam, pārējie masti no dūža līdz devītniekam. Trumpji pēc lieluma ir šādas kārtis: visas dāmas; pēc tam kalpi un beidzot pārējie kāravi. Dāmas un kalpi šādā kārtībā: krusts, pīķis, ercs un kāravs. Kāravi pēc lieluma: dūzis, desmitnieks, kungs, devītnieks, astotnieka un septītnieks. Acis tiek skaitītas šādi: dūzis -11, desmitnieks -10, kungs – 4, dāma – 3, kalps – 2; devītnieks, astotnieks un septītnieks – 0. Pēc stiprumā nākošā kārts ir dūzis, desmitnieks, tad kungs neatkarīgi no masta.

Vietas izmeklētājs jauc kārtis, dod tās pārcelt tam spēlētājam, kurš sēd no dalītāja pa labi, un pēc tam izdala katram dalībniekam 8 kārtis: vispirms 2 kārtis sēdošajam pa kreisi no dalītāja, tad 2 kārtis nākošajam un pēc tam 2 kārtis sev vai ceturtajam spēlētājam, ja spēle notiek četratā tad vēlreiz pa divām kārtīm katram ; tad 2 (aizklātas) noliek uz galda un atlikušās 12 kārtis tāpat divas reizes pa divām katram spēlētājam. Ārpus turnīriem kārtis var tikt dalītas pa četrām galdā 2 un atkal pa četrām.

Tas, kam ir priekšroka, t.i., spēlētājs, kurš sēž no dalītāja pa kreisi, pēc savu kāršu pārbaudīšanas, paziņo, ka viņš vai nu spēlē, vai atsakās no spēles, sacīdams " Garām ". Pēc tam to pašu dara nākošais un, ja arī tas nopasē, tad beidzot trešais. To, kas ar savām kārtīm cer savākt vismaz 61 ņem galda vidū esošās divas piepirkuma kārtis un viņu sauc par lielo. Lielajam ir jāspēlē vienam pret pārējiem diviem, kurus sauc par mazajiem, kuri savos stiķos iegūtās acis skaita kopā. Lielajam pēc vidus kāršu pacelšanas 2 kārtis pēc savas izvēles ir jānorok t.i. jānoliek uz galda. Parasti cenšas nolikt to mastu, no kura ir tikai viena vai divas kārtis, mazākas par dūzi, Lai spēlē varētu šo mastu sist ar trumpi. Vēlams arī pēc iespējas nolikt kārtis ar Lielākām acīm, jo šīs 2 kārtis tiek pieskaitītas spēlē saņemtajiem stiķiem. Nav

ieteicams nolikt dūzi, ja tas ir vienīgais no tā masta, jo ir lielas izredzes ar šo dūzi ņemt stiķi. Spēlētājam, lai vinnētu, vajag iegūt vismaz 61 aci. Ja pretiniekiem kopā ir vismaz 30 acis, tad spēlētājs saņem no katra 1 punktu, ja ir mazāk par 30 acīm – 2 punktus, bet, ja nav neviena stiķa – 3 punktus.

Ja spēlētājam pēc viņa aprēķiniem ir tik labas kārtis, ka viņš var uzvarēt bez 2 kāršu piepirkšanas, viņš pieteic Zoles, kas nozīmē; ka viņš spēlē bez 2 kāršu piepirkšanas. Šīs kārtis tad tiek pieskaitītas pie pretinieku saņemtajiem stiķiem

Ja Zoles spēlētājs iegūst vismaz 61 aci, viņš saņem no katra pretinieka 5 punktus; ja 91 aci – 6 punktus; ja visus stiķus – 7 punktus.

Bet, ja spēlētājs, spēlēdams parasto spēli, neiegūst 61 aci, viņš dod katram pretiniekam 2 punktus. Ja spēlētājam ir mazāk par 31 aci, viņš dod katram – 3 punktus, ja nav neviena stiķa -4 punktus. Ja Zoles spēlētājs neiegūst 61 aci, viņš dod katram pretiniekam 6 punktus; ja ir mazāk par 31 aci – 7 punktus, bet, ja nav neviena stiķa – 8 punktus.

Gadījumā, ja neviens spēles dalībnieks nevēlas atklāt spēli un visi nopasē, tad tiek likta pule, ja spēlē četratā tad tiek liktas divas pules, ja pule jau ir tad viena. Pēc tam kārtis tiek atkal sajauktas un no jauna izdalītas, kā tas jau tika darīts pirmoreiz, tikai tagad kāršu dalītājs ir tas spēles dalībnieks, kuram iepriekšējā spēlē bija priekšroka. Nākošā spēlē kārtis dala atkal nākošais utt.

Vairākkārt nopasējot, var sakrāties vairākas pules. Nākošās spēles spēlētājs uzvarot saņem attiecīgo punktu daudzumu ne tikai no spēles pretiniekiem, bet arī papildus pa punktam no katra spēlētāja . Tas, protams, var notikt tikai tik ilgi, kamēr pierakstā ir pules. Ja ir jau vismaz viena pule, un spēlētājs spēlējot kā lielais zaudē, viņš saņem personīgo puli. To var izņemt viņš pats uzvarot spēli, bet nekādis papildus punktus nesaņemot. Ja personīgo puli izņem kāds cits, viņš saņem papildus punktus no pules īpašnieka.

Partneris (trešā roka) nedrīkst piemest savu kārti ātrāk par partneri, kam ir otrā roka. Tāpat netiek pieļauta tā saucamā kāršu vilkšana iepriekš.

Pareizi izspēlēta kārts nevar tikt apmainīta.

Uz izspēlēto kārti jāatbild vienmēr ar to pašu mastu, pie tam dāmas un kalpi ir trumpji. Ja nav rokā attiecīgās šķiras kārts, var pēc izvēles piemest kādu citu šķiru vai arī pārsist ar trumpi.

Ja mazie nevietā iziet vai nepareizu mastu uzliek, ar lielā piekrišanu var paņemt atpakaļ kārti, ja lielais nepiekrīt, spēle tiek anulēta un vainīgajam tiek piešķirta personīgā pule.

Ja turpmākā spēles gaitā vai arī tikai spēles beigās izrādās, ka kāds nepareizi rīkojies ar pieprasīto mastu, pretējā puse skaitās par uzvarētāju un vainīgais tiek sodīts ar personīgo puli. Tāpēc arī pirms spēles beigām paņemtie stiķi nedrīkst tikt sajaukti, lai katrā laikā būtu iespējams konstatēt spēles noteikumu pārkāpšanu. Pirms spēles var vēlreiz apskatīt beidzamo stiķi vai arī pieprasīt no pretējās puses to uzrādīt.

Visādas sarunas spēles laikā starp partneriem par spēles gaitu un veidu uz visstingrāko aizliegtas.”

1.2. Spēles raksturojums

Lai veidotu mākslīgo intelektu, kas spēlē zoli, jāņem vērā šādi spēļu teorijas pamatprincipi par Zoles spēli:

- **Zole ir stohastiska spēle.** Spēles iznākumā liela nozīme ir kāršu sākotnējajam sadalījumam, kas pilnībā atkarīgs no nejaušības. Pēc kāršu izdales zoli gan var uzskatīt vienkārši par determinētu, nepilnas informācijas spēli. Tomēr labam spēlētājam praktiski visu spēli jāņem vērā, ka sākumā kārtis tika izdalītas nejauši.
- **Zole ir nepilnas informācijas spēle.** Spēlētājiem nav zināmas ne pretinieku kārtis, ne galdiņa kārtis un līdz ar to nav zināms lēmumu pieņemšanas process spēlē. Līdz pat pēdējiem gājieniem turpmākās spēles prognoze jābalsta uz pieņēmumiem par pretinieku stratēģijām un kārtīm.
- **Zolei nepiemīt uzvarošās stratēģijas.** Spēlētājam nav iespējams garantēt savu uzvaru tikai ar pareizu spēlēšanu. Līdz ar to nav iespējams izveidot mākslīgo intelektu, kas vienmēr uzvarēs.
- **Zole ir nulles summas spēle.** Neatkarīgi no spēles ilguma jeb izspēlēto partiju skaita, spēlētāju kopīgi iegūtie punkti būs nulle. Nulles summas spēle ir konstantas summas spēles īpašs gadījums. [5]
- **Zolei pēc katras partijas ir definēts skaitlisks rezultāts.** Tas atvieglo spēles stratēģijas modelēšanu, jo ir skaidri zināms spēles aģenta uzdevums.
- **Zole ir sadarbības spēle.** Katrā partijā divi spēlētāji veido aliansi pret trešo spēlētāju, kurš izvēlējies būt lielais. Šiem abiem mazajiem spēlētājiem punkti tiek piešķirti vienādi un tie ir pretēji lielajam piešķirtajiem punktiem. Tas nozīmē, ka mazo spēles mērķis ir panākt, lai lielais zaudē pēc iespējas sliktāk. Mazie nedrīkst savā starpā apmainīties ar informāciju par savām kārtīm. Kaut spēlei ir trīs spēlētāji, katrā partijā ir divas pretinieku puses ar pretstatītiem mērķiem.
- **Zole ir secīga spēle.** Spēlētāju gājieni notiek pēc kārtas un spēlei piemīt laika ass. Veicot lēmumu par gājienu, spēlētājam ir zināmas visas iepriekš izspēlētās kārtis. Laikam, kas tiek patērēts izvēloties gājienu nav lielas nozīmes, jo pretinieki tikmēr nedrīkst veikt nekādas darbības. Tomēr mākslīgajam intelektam jāspēj lēmumu pieņemt pāris sekunžu laikā, jo zolē ir pieņemts veikt ātrus gājienus, nevis domāt ilgstoši kā šahā vai dambretē.

Praktiski visas augstāk uzskaitītās īpašības ir raksturīgas daudzām stiķu spēlēm, tāpēc to modelēšanā izmantotās metodes var būt derīgas arī Zoles risinājumu modelēšanai. Visbiežāk šādas spēles analizē, izmantojot spēles izvēļu koku, tāpēc to varētu izmantot arī Zoles modelēšanā.

Partijā iegūstamais rezultāts ir gadījuma lielums. Par aģenta optimizēšanas mērķi var uzskatīt šī gadījuma lieluma vidējās vērtības maksimizēšanu.

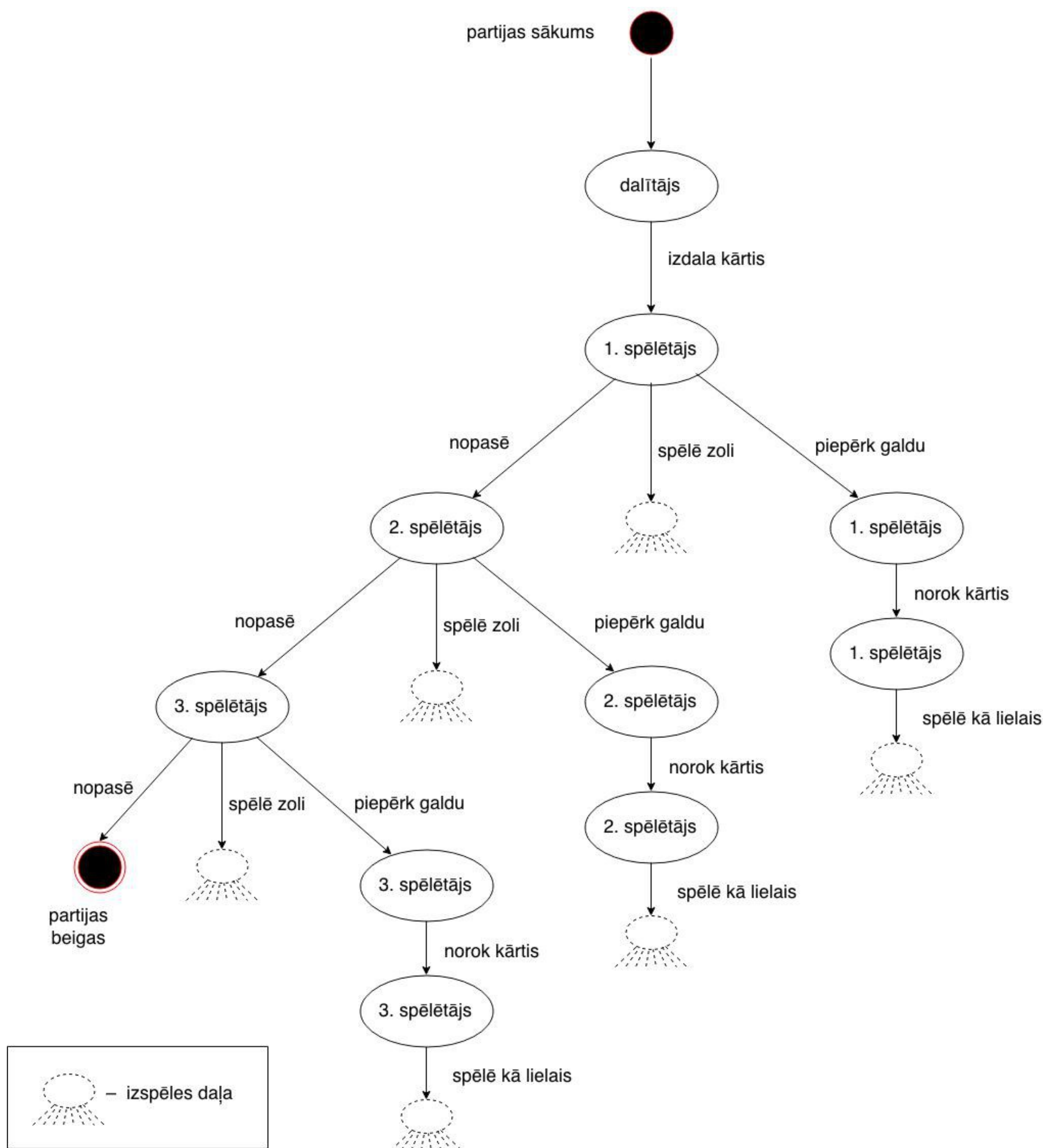
1.3. Spēles izvēļu koks

Zoles spēle sastāv no iepriekš noteikta vai nenoteikta partiju skaita, un gala rezultāts tiek iegūts summējot visās partijās iegūto punktu skaitu. Arī viena partija var būt pilnvērtīga spēle, un analizēt ir nepieciešams tikai šo Zoles minimālo formu, jo garākas spēles ir tikai tās papildus atkārtojumi. Zoles partiju var iedalīt divās daļās – *izvēles daļā* un *izspēles daļā*.

Izvēles daļa. Spēles sākumā katram spēlētājam pēc kārtas ir jāizvēlas, vai tas ir gatavs spēlēt pret abiem pārējiem. Izvēles kritērijs šķiet samērā vienkāršs: ja spēlētājs domā ka ar savām kārtīm dabūs 61 vai vairāk aci, spēlēdams pret abiem pārējiem spēlētājiem, viņam jāspēlē Zole. Ja viņš domā, ka piepērkot galda kārtis dabūs 61 vai vairāk aci, tad viņam jāspēlē kā lielajam. Citādi jāļauj izvēlēties nākamajam spēlētājam.

Taču izdarīt šo izvēli nav vienkārši. Piemēram, spēlētājam ar 55% varbūtību uzvarēt kā zolētājam jāizvērtē, vai zaudēšanas risks nav pārāk liels un vidēji labāku rezultātu nedotu spēlēšana kā lielajam ar, piemēram, 90% varbūtību uzvarēt.

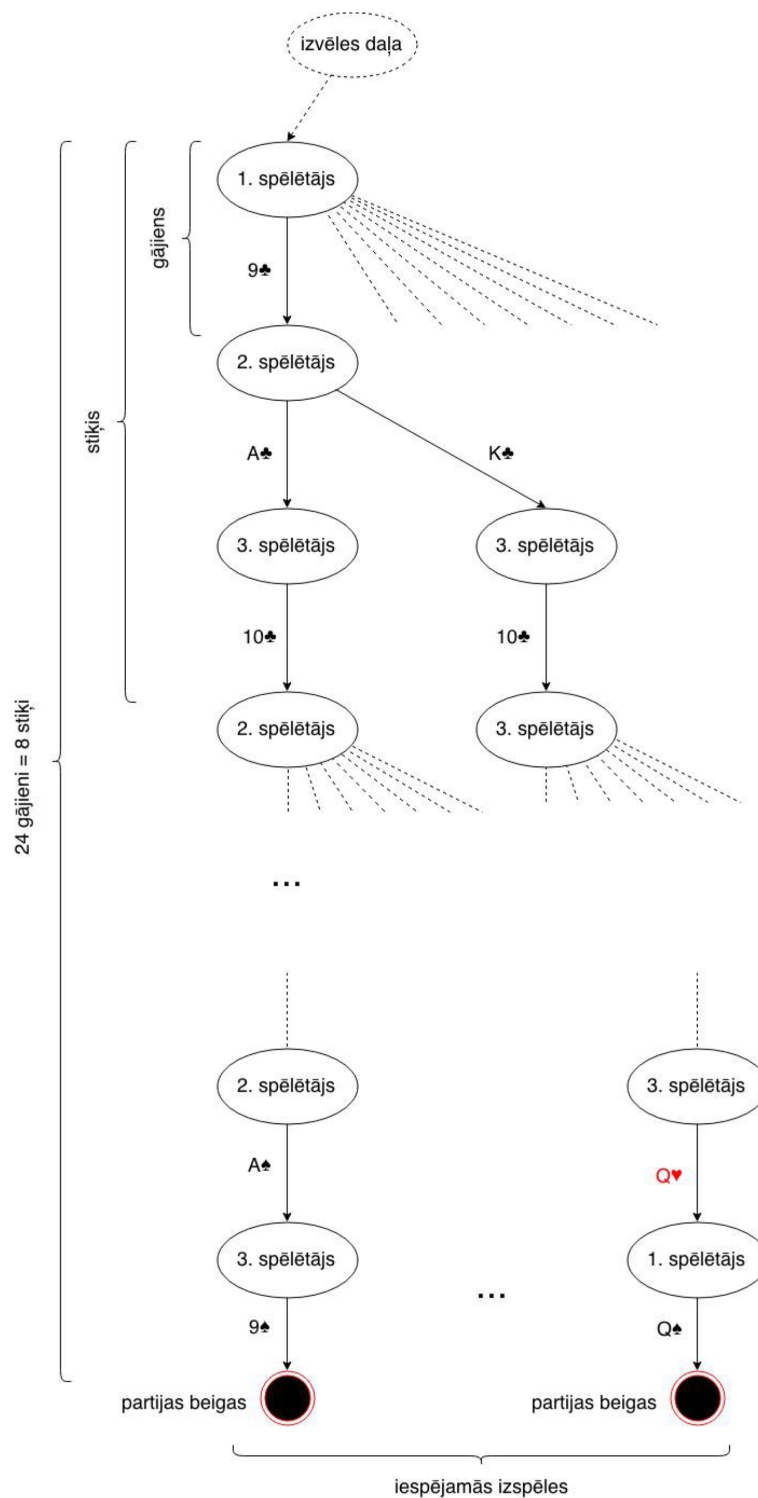
Turklāt, zinot savas iedalītās kārtis, spēlētājam jārēķinās ar $C_{18}^{2} \cdot C_{16}^{8} = 1969110$ iespējamām pretinieka un galda kāršu kombinācijām. (Zinot 8 no 26 kārtīm, paliek 18 kārtis. 2 no 18 kārtīm var būt galdā C_{18}^{2} kombinācijās. No atlikušajām 16 kārtīm 8 var būt vienam spēlētājam C_{16}^{8} kombinācijās un 8 atlikušās paliek otram.)



2.1. att. **Zoles spēles *izvēles daļas* izvēļu koks**

2.1. attēlā parādīts *izvēles daļas* spēles izvēles koka piemērs. Tajā redzama darbību secība, kādā spēlētāji izvēlas vai spēlēt vienatnē. Gadījumā ja visi spēlētāji nopasē, *izspēles daļa* nemaz nenotiek, tiek pārdalītas kārtis un uzsākta jauna partija (ierakstot spēlētājiem pules).

Izspēles daļa. Pēc *izvēles daļas* turpinoties spēlei, katra Zoles partija sastāv no 8 stiķiem. Katrs stiķis ir 3 spēlētāju gājieni, kuri kopā veido izvēles koku ar gājienu skaitu jeb augstumu 24. Izspēles daļu vienmēr uzsāk 1. spēlētājs.



2.2. att. **Zoles spēles izvēļu koka piemērs – izspēles daļa**

2.2. attēlā parādīts vienkāršots spēles koka piemērs. Tā kā Zoles noteikumi prasa spēlētājiem likt tāda paša masta kārti, kādu ir licis stiķi uzsākošais spēlētājs, izspēles koks veidojas nesabalansēts ar atšķirīga izmēra apakškokiem. Piemērā, ja kā pirmais gājiens būtu apskatīta cita kārts, nevis 9♣, tad 2. gājiena atbildes varētu būt gan vairāk, gan mazāk par 2 variantiem (A♣, K♣), veidojot gan plašākus, gan šaurākus apakškokus. Šī iemesla dēļ, līdzīgu

situāciju analīze var daudzkārt atšķirties sarežģītībā, jo jāanalizē dažādu izmēru apakškokus ar dažādiem izspēļu skaitiem. Izvēlētās kārtis arī nosaka, kurš spēlētājs uzsāks nākamo stiķi, līdz ar to spēlētāju iešanas kārtība pēc pirmā stiķa stipri atšķiras. Kā piemērā redzams, ja 2. spēlētājs sākumā izies ar A♣, viņš liks pirmo kārti nākamajā stiķī, ja viņš izies ar K♣, tad to darīs 3. spēlētājs.

Triviālais novērtējums iespējamo izspēļu skaitam pie dažādiem kāršu sadalījumiem ir šāds: $8!\leq izspēļu\ skaits\leq(8!)^3\approx6{,}6\cdot10^{13}$ jeb aptuveni $4{,}0\cdot10^4\leq izspēļu\ skaits\leq6{,}6\cdot10^{13}$. Apakšējā izspēļu skaita robeža $8!$ iegūta, pieņemot ka spēlētājiem vienmēr ir tikai 1 kārts ar ko atbildēt gājienam un brīva izvēle ir tikai uzsākot katru no 8 stiķiem. Savukārt augšējā robeža $(8!)^3$ iegūta, pieņemot ka katrā no 24 gājieniem spēlētāji var likt jebkuru kārti. To, ka reāli sadalījumi var pietuvoties noteiktajai augšējai robežai, parāda piemērs 1.1. tabulā.

1.1. tabula

Piemērs lielam iespējamo izspēļu skaitam

	kārtis	gājienu iespējas	komentārs
1. spēlētājs	J♦,J♥,J♠,J♣,Q♦,Q♥,Q♠,Q♣	$8!$	Vienmēr sāk stiķi, liek 1 no 8 trumpjiem.
2. spēlētājs	10♥,A♥,9♠,K♠,10♠,A♠,9♣,K♣	$8!$	Vienmēr var likt 1 no 8 netrumpjiem.
3. spēlētājs	7♦,8♦,9♦,K♦,10♦,A♦,10♣,A♣	$6!\cdot2!$	Sākumā liek 1 no 6 trumpjiem, pēc tam 1 no 2 netrumpjiem.
galdiņš	9♥,K♥	-	
kopā		$8!\cdot8!\cdot6!\cdot2!\approx$ $\approx2{,}3\cdot10^{12}$	

Iegūtie skaitļi norāda, ka iespējami ļoti atšķirīga izmēra izspēles koki un ka ir izspēles koki ar ļoti lielu iespējamo izspēļu skaitu. 3.1. nodaļā iegūtie rezultāti apstiprina, ka daudzi kāršu sadalījumi veido ļoti atšķirīga izmēra kokus.

2. MĀKSLĪGĀ INTELEKTA RISINĀJUMI

2.1. Zoles risinājumi

Mākslīgā intelekta risinājumi, kas spēlētu zoli, nav plaši izplatīti. Autoram izdevās atrast divus Zoles aģentus [2, 3].

Diemžēl, neizdevās izmēģināt A. A. Bērziņa publiski pieejamā Zoles spēli, kas veidota senākām operētājsistēmām un pēdējo reizi atjaunota pirms vairāk nekā 10 gadiem [3].

Autors izmēģināja A. Ikaunieka veidoto zolītes spēli [2]. Spēlei ir pieejams pirmkods, tās saskarne un mākslīgais intelekts veidoti *Delphi* programmēšanas valodā. Tai ir iespējams rediģēt datora kāršu izvēli, taču piedāvātā izvēle ir balstīta uz labas spēles pamatlikumiem, nevis padziļinātu situācijas analīzi. Spēlējot pret šo aģentu, bieži gadās, autoraprāt, šķietami neloģiski gājieni.

No tā autors secina, ka cilvēku skaits, kuri spēlē zoli izmantojot datoru, nav gana liels, lai rastos plaša interese veidot sarežģītus mākslīgā intelekta risinājumus. Vēl var secināt, ka cilvēki dod priekšroku Zoles spēlēšanai pret citiem dzīviem cilvēkiem.

2.2. Skat risinājumi

Tā kā autoram neizdevās atrast nozīmīgus pētījumus vai risinājumus, kur būtu analizēta mākslīgā intelekta veidošana priekš Zoles, tika izvēlēts apskatīt tai radniecīgās spēles Skat risinājumus. Skat ir ļoti līdzīga spēles struktūra Zolei, un Skat sastāv no Zolei ļoti līdzīgas *izspēles daļas* un vairāk atšķirīgas *izvēles daļas*. Pateicoties tam, ka Skat ir populāra spēle, ko regulāri spēlē vairāk nekā 30 miljoni spēlētāju [11], tai ir izveidoti daudzi mākslīgā intelekta risinājumi. Tajos iegūto pieredzi var izmantot par pamatu Zoles aģenta modelēšanai. Eksistē arī publisks Skat serveris (*skatgame.net/iss*), kurā savstarpēji iespējams spēlēt gan cilvēkiem, gan aģentiem, kā arī citi skat serveri. Autors padziļināti apskatīja šādu aģentu modelēšanas metodes, par kuriem bija pieejama visplašākā informācija: XSkat, Double Dummy Skat Solver, Kermit, Imperfect Information Monte Carlo un UCT aģenti.

XSkat. Populārākā atvērtā pirmkoda Skat spēles versija, kas piedāvā arī savus mākslīgā intelekta spēlētājus. Diemžēl spēlētāja iešanas loģika veidota tikai izmantojot heiristikas, kuras izmanto cilvēku spēlētāju ieteiktos labu gājienu likumus. [6]

Double Dummy Skat Solver jeb **DDSS**. Aģenta autori apvieno bridža modelēšanai izmantoto ātru perfektās informācijas analīzes risinājumu ar algoritmiem, kas pielāgo to Skat

spēlēšanai. Autori norāda, ka ja Skat spēlētu ar atklātām kārtīm kā perfektas informācijas spēli, ir iespējams gana ātri atrast labāko gājienu. Lai *izspēles daļā* apietu trūkstošās informācijas problēmu, tiek izmantota Perfect Information Monte Carlo (PIMC) meklēšanas metode. Autori norāda uz PIMC problēmām, ka, ja PIMC bāzēts aģents domā ka garantēti iegūs uzvaru, tas atdod punktus pretiniekam (tāpat kā zolē, tam ir vienalga, vai uzvarēt ar 61 aci, vai 81 aci, jo punktu rezultāts ir vienāds). Taču ar PIMC metodi iegūtie rezultāti nav gana precīzi, lai uz tiem varētu pilnībā paļauties, tāpēc, izvēloties variantu ar 61 aci, šāds aģents bieži nonāk zaudētāja lomā. Double Dummy Skat Solver šo problēmu uzveic, ar ātru algoritmu vispirms nosakot, kuras ir iespējamās uzvarošās kārtis, un pēc tam ar lēnāku algoritmu nosakot, kura no uzvarošajām kārtīm uzvarētu ar vislielāko acu skaitu. Izmantoti tiek *alpha-beta minimax* un *best-first minimax* algoritmi. Aģenta ātrdarbība tiek paaugstināta ar šādiem uzlabojumiem:

- **Gājienu sakārtošana**. Alpha-beta meklēšana darbojas krietni ātrāk, ja vispirms tiek apskatīti labākie gājieni.
- **Starpstāvokļu tabula (transposition table)**. Analizējot izspēļu koku, mēdz atkārtoties vienādas situācijas, ko nav nepieciešams atkārtoti rēķināt. Piemēram, savstarpēji apmainot vietām pirmos divus stiķus, pie nosacījuma ka gājiens ir tam pašam spēlētājam, tālākā izspēle nemainīsies un tās apakškoka aprēķinu var izmantot atkārtoti.
- **Quasi-Symmetry Reduction**. Metode, ko izmanto bridža modelēšanas algoritmos. Palielina starpstāvokļu tabulas izmantošanu, pielīdzinot un neapskatot tālāk līdzīgus stāvokļus. Nav lielas atšķirības, kuru no secīgām kārtīm izspēlē, kamēr to punktu skaits ir līdzīgs. Piemēram, ja spēlētājam rokā ir **J♥**, **J♠**, **J♣**, tad viņam izspēlējot **J♥** būs ar vienāds rezultāts kā izspēlējot **J♣** un tā apakškoks nav atkārtoti jāanalizē.
- **Adversarial Heuristics**. Autoru izveidotas Skat hēristikas metodes. Tās ļauj ātri noteikt apakškoka apakšējo robežu, *lielajam*[1] vispirms izspēlējot visas kārtis, ar kurām tas var uzvarēt stiķus un saskaitot iegūtos punktus. Līdzīgā veidā tiek iegūta arī augšējā robeža, tādējādi nodrošinot ātru darbību, nedaudz zaudējot darbības precizitāti.

1 Skat arī spēlē divi spēlētāji pret vienu vairāksolītāju, kura loma ir pielīdzināma Zoles *lielajam*.

Ar savu darbu autori apstiprina, ka kāršu izspēles daļā Monte Carlo meklēšanas metode darbojas gandrīz nevainojami un ir spēcīgs cilvēka pretinieks. Viņu izveidotā modeļa vājā vieta ir spēles pirmā daļa ar likmju likšanu, kas Zoles gadījumā atbilstu *izvēles daļai*. [7]

Sekojošā darbā tas pats autors arī atrod risinājumu *izvēles daļai*, kurā izmantojot mašīnmācīšanās metodes un spēļu paraugus, tiek uzmodelēts likmju izvēles algoritms. [8]

Imperfect Information Monte Carlo un **UCT** aģenti. Aģentu autors ir izveidojis divus risinājumus. Pirmajā risinājumā ir aprakstīta metode, kā analizēt spēles apakškokus nezaudējot nepilno informāciju. Šī risinājuma sniegums ir līdzvērtīgs Double Dummy Skat Solver. Savukārt UCT aģents gājienu meklēšanā pielieto nepilnas informācijas spēlēs līdz tam neizmantoto UCT algoritmu, beigās gan nepierādot lielus ieguvumus no šīs metodes. [9, 10]

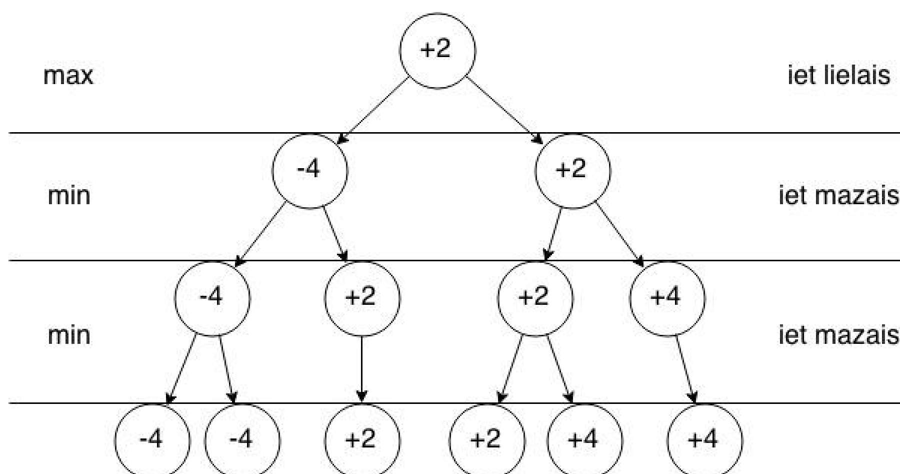
Kermit. Kermit ir viens no spēcīgākajiem Skat aģentiem, izstrādāts balstoties uz iepriekšējiem pētījumiem par Skat mākslīgajiem intelektiem. Lielākā daļa Skat aģentu spēli analizē, pieņemot, ka pretinieki savus gājienus izvēlās pēc līdzīgas domāšanas metodes kā tie. Taču faktiskās gājienu varbūtības atšķirās no paredzētās izspēles. Kermit autori ar mašīnmācīšanos no īstu spēļu turnīru datiem atrisina šo problēmu, ļaujot tam pretinieku rīcību paredzēt krietni labāk par pārējiem pieminētajiem risinājumiem. [11]

2.3. Risinājumu metodes

Šajā apakšnodaļā aprakstītas Skat risinājumos pieminētās teorēmas un algoritmi, kuri uzskatāmi par nozīmīgiem to darbības uzlabošanā un kurus vēlams izmantot arī modelējot Zoles mākslīgo intelektu.

Neša līdzsvars. Spēlēt, vadoties pēc Neša līdzsvara, nozīmē gājienu izvēli balstīt uz pieņēmumu, ka pretinieki spēlēs pēc tiem pašas izdevīgākās stratēģijas, cenšoties maksimāli samazināt spēlētāja iegūtos punktus. [12]

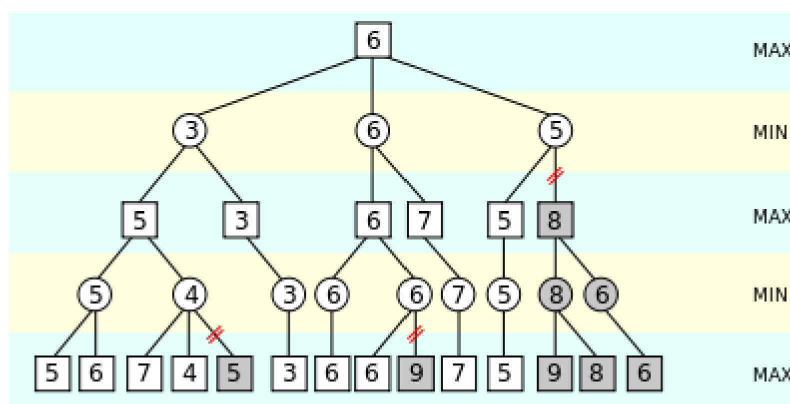
Minimax algoritms. Izmanto par pamatu Zoles *izspēles daļā* nepieciešamajam algoritmam. Minimax algoritms risina perfektas informācijas spēli, pieņemot ka pretinieks spēlēs pēc Neša līdzsvara. Minimax mēģina minimizēt spēlētāja iespējamos zaudējumus sliktākajā spēles gadījumā.



2.1. att. **Minimax algoritma darbības piemērs**

Pielāgojot Zolei minimax algoritmu, spēlētāju uzvedība veidojas kā parādīts 2.1. attēlā. Virsotnēs parādīti lielā punkti spēles beigās. Lielais savā solī vienmēr izvēlēsies rezultāta maksimumu, savukārt mazie izvēlēsies gājienus, kas lielajam dos vismazāk punktus. Atšķirībā no tradicionālā divu spēlētāju minimax izpildījuma, zolē neizpildās īpašība, ka pēc maksimuma soļa vienmēr seko minimuma solis un pēc minimuma soļa seko maksimums, kā arī minimumu un maksimumu skaits nesakrīt. [5]

Alpha-beta pruning jeb **alpha-beta meklēšana** ir minimax algoritma optimizēšana. Tā pārtrauc spēles koka apakšzara apskati brīdī, kad kļūst skaidrs, ka gājiens ir sliktāks par jau iepriekš apskatītu gājienu. Šādi netiek apskatīti atzari, kuri tāpat nevar ietekmēt rezultātu. [13]



2.2. att. **Alpha-beta pruning optimizācija [13]**

Zoles gadījumā šī optimizācija dod lielu ietekmi, jo koks ir plašs un zarošanās lielākoties notiek pie pirmajiem gājieniem.

Monte Carlo tree search jeb **MCTS** ir koka apstaigāšanas algoritms, kas katrai no izvēles kārtīm apskata nejauši izvēlētu gājienu izspēļu rezultātus. Apskatīto gājienu skaitam tuvojoties maksimumam, MCTS iegūtais rezultāts konverģē uz to, ko uzrāda pilnas apskates

minimax algoritms. Algoritma priekšrocība ir, ka to var apturēt vēlamā laika brīdī un joprojām iegūt rezultātu. Tas ļauj analizēt liela izmēra kokus, kurus pilnībā apstaigāt ar minimax algoritmu prasītu pārāk lielu laiku. [14]

Upper Confidence Bound 1 applied to trees jeb **UTC** ir Monte Carlo tree search implementācija, kas nodrošina lielāku rezultāta precizitāti, sabalansējot nejauši apskatīto rezultātu koku. UTC ir izmantots arī Skat spēlē, taču ieguvumi no tā pielietošanas ir salīdzinoši nelieli. [10]

Best-first minimax search ir koka apstaigāšanas algoritms, kas vispirms apskata vislabākā gājiena apakškokus. Algoritma ātrdarbība ir augstāka nekā alpha-beta pruning darbībai, un koka apstaigāšanas sākumā tas strādā ļoti labi. Taču spēles beigu daļā zūd gājienu precizitāte, tāpēc autori iesaka izmantot hibrīdu risinājumu, kas beigās izmanto alpha-beta pruning. Lai best-first minimax search varētu darboties, ir nepieciešama virsotnes stāvokļa novērtēšanas funkcija vēl pirms tās apakškoka apstaigāšanas. [15]

Perfect Information Monte Carlo jeb **PIMC.** Visi iepriekš minētie algoritmi domāti izmantošanai perfektas informācijas telpā. Taču pretinieku rokās esošās kārtis nav zināmas līdz to izspēlēšanas brīdim. Lai apietu šo problēmu, PIMC metode piedāvā aprēķināt spēles rezultātus nejaušiem iespējamajiem pretinieku kāršu sadalījumiem, un labāko gājienu noteikt pēc iegūto rezultātu vidējās vērtības.

Kaut algoritmam ir daudz teorētisko trūkumu, tā izmantošana praksē sasniedz labus rezultātus, un ir sevišķi spēcīgi stiķu bāzētās kāršu spēlēs. Turklāt stiķu spēlēs kā Skat un Zole pietiekoši precīzi un ātri strādā arī vienkāršā PIMC apskate (nav nepieciešams izmantot algoritma rekursīvo variantu, kas visiem apakškokiem arī apskata vidējo vērtību). [16]

Imerfect Information Monte Carlo jeb **IIMC** ir īpaša Monte Carlo metodes pielāgošana nepilnas informācijas spēlēm. Tas analizē nejauši ņemtas spēles pasaules (apakškokus), tajās nepazaudējot nepilno informāciju, kā tas notiek PIMC gadījumā. Algoritma veidotāji to ieviesa Skat spēles *Kermit* aģenta risinājumā un ar mērījumiem apstiprināja tā pārākumu pār PIMC algoritmā veidoto *Kermit*. [17]

3. IZVĒLĒTAIS RISINĀJUMS

Lai izpētītu, vai Skat spēlē izmantotie algoritmi ir piemēroti arī Zoles spēlei, tika izlemts izveidot mākslīgā intelekta risinājumu. Kā veidotā Zoles aģenta galvenie uzdevumi tika noteikti:

- Izvēlēties labāko kārti jebkurā *izspēles daļas* situācijā, ņemot vērā izspēlētās kārtis un spēlētāja lomu (lielais, mazais, zolētājs).
- Spēles sākumā izvēlēties uz kādu likmi spēlēt – vai ar iedalītajām kārtīm būt lielajam, zolēt vai laist garām.
- Piepērkot galdu, aģentam jāspēj pareizi izvēlēties kārtis, ko izņemt no rokas un pievienot iegūtajiem punktiem (pareizi norakt kārtis).
- Veikt iepriekšminētās darbības vienas sekundes laikā uz vidusmēra mūsdienu klēpjdatora. Par atskaites punktu tiek izmantots autora dators (procesors: *Intel Core i3 380M, 2.53GHz,* operatīvā atmiņa: *4GB,* operētājsistēma: *Windows 8.1*).

Par risinājuma pamatu tika noteikta Double Dummy Skat Solver aģenta funkcionalitāte un tika noteikts mērķis šī aģenta izmantotos algoritmus pielietot Zoles spēlei.

Visā risinājuma veidošanas gaitā tika izmantota programmēšanas valoda *C# 5.0* un izstrādes vide *Microsoft Visual Studio 2013 Ultimate*. Šāda izstrādes konfigurācija tika izvēlēta pamatojoties uz *C#* valodas ātrdarbību un plašajām izmantošanas iespējām, ko tā sniedz kā augsta līmeņa programmēšanas valoda. Tika ņemta vērā arī autora pieredze darbā ar līdzīgām objektorientētām valodām un vēlme apgūt tieši *C#* valodu.

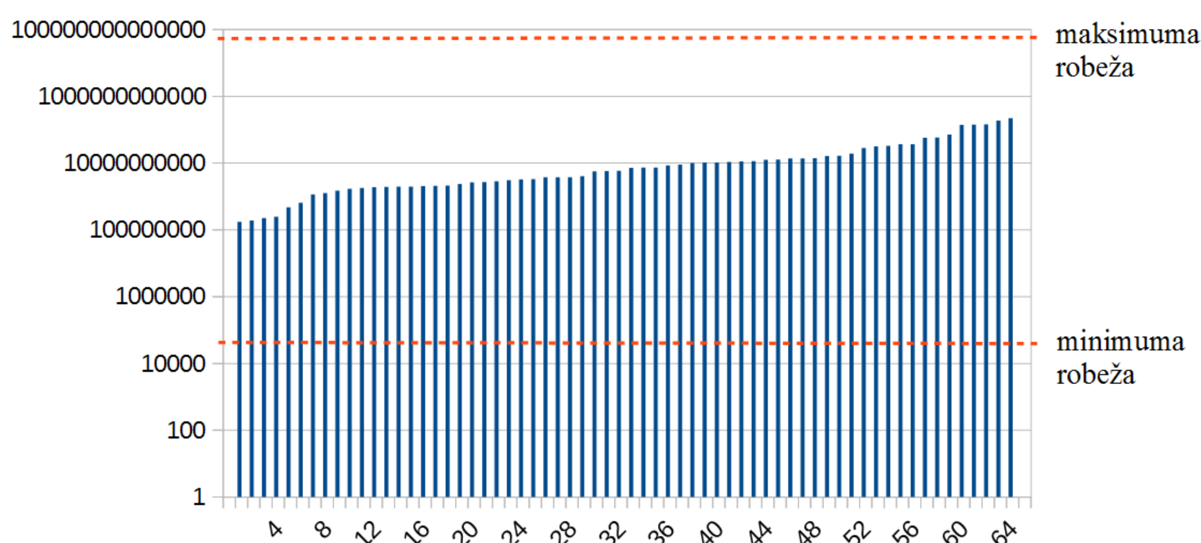
3.1. Izspēļu skaits un tā analīze

Sākotnēji tika veikta izpēte, kādi ir iespējamie izspēļu skaiti dažādiem kāršu sadalījumiem. Nejaušam kāršu sadalījumam tiek skaitīts, cik dažādos veidos atbilstoši noteikumiem var izspēlēt partiju (partijas izspēles daļu). Tika izveidota programma, kas, ievērojot spēles noteikumus, rekursīvi saskaita visu *izspēles daļas* kokā iegūstamo gala rezultātu skaitu. Tātad, nejaušam kāršu sadalījumam tiek skaitīts, cik dažādos veidos atbilstoši noteikumiem var izspēlēt partiju (partijas *izspēles daļu*). Pie 64 nejaušiem kāršu sadalījumiem tika iegūti izspēļu skaiti, kas parādīti 3.1. tabulā un 3.1. attēlā. Skaitu precīzās vērtības norādītas 1. pielikumā.

3.1. tabula

Nejaušu kāršu sadalījumu iespējamo izspēļu skaits

vidējā vērtība	23 343 725 917
mediāna	6 568 634 512
minimums	175 999 872
maksimums	223 597 089 792



3.1. att. **Nejaušu kāršu sadalījumu iespējamo izspēļu skaits**

3.1. attēlā iegūtie skaitļi parādīti augošā secībā, logaritmiskā skalā. Kā robežlīnijas attēloti 2. nodaļā aprēķinātie teorētiskā maksimuma un minimuma novērtējumi. Iegūtie rezultāti liecina, ka līdzīgu spēles situāciju analīze var nozīmēt krasi atšķirīga izmēra spēles koka analīzi, kas prasītu dažādu laiku. Tā kā augstākie iegūtie skaitļi ir mērāmi 10^{11} pakāpē, ar parasto minimax algoritmu nav iespējams izanalizēt pilno spēles koku saprātīgā laikā[2].

3.2. Optimizācija un binārā loģika

Tā kā jāanalizē ļoti liels skaits iespējamo izspēļu, autors saskārās ar ātrdarbības problēmām jau modelējot spēli. Profilējot modelēšanas algoritmu, atklājās ka visvairāk laika tiek patērēts veicot iterācijas pa kārtīm un to vērtību salīdzināšanas. Tika pieņemts lēmums atteikties no abstraktiem kāršu datu tipiem un kārtis modelēt kā bitus un to salīdzināšanai

2 Izmantojot plaši pieejamu datortehniku, nevis lieldatorus vai mākoņpakalpojumus, pāris sekundēs nebūtu iespējams veikt 10^{11} netriviālu aprēķina operāciju.

primāri izmantot binārās loģikas operācijas, kas darbojas ātrāk nekā loģiskās salīdzināšanas operācijas [18].

Zole izmanto 26 kārtis, tāpēc, priekš ērtas bināro operāciju lietošanas, kārtis var glabāt kā divnieka pakāpes, izmantojot 32 bitu *integer* tipa mainīgo. Šādi glabājot kārtis, arī spēlētāju rokas ietilpst vienā *integer* mainīgajā. Šādi kodētas kārtis ļauj ar bitu operācijām ātri veikt darbības ar vienu vai vairākām kārtīm. Piemēram, saucot spēlētāja roku par *hand* un kārti par *card*, lai pievienotu kārti rokai, *hand.add(card)* vietā jāizsauc *hand* | *card*, lai izņemtu kārti no rokas, *hand.Remove(card)* vietā jāizsauc *hand* ^ *card,* lai iegūtu rokas pēdējo (lielāko) kārti, *hand.last()* vietā jāizsauc *hand* ^ *-hand* utt.

3.2. tabula

Kāršu binārās reprezentācijas piemēri

Kārts/roka	Binārā reprezentācija	Komentārs
9♥	00000000000000000000000000000001	Mazākā kārts
K♥	00000000000000000000000000000010	Otrā mazākā kārts
...	...	...
Q♣	00000010000000000000000000000000	Lielākā kārts
A♠, 10♠, K♠, 9♠	00000000000000000000000011110000	Roka ar visiem pīķiem
K♠, 9♠, K♥, 9♥	00000000000000000000000000110011	Nejauša roka. Lai atrastu visus pīķus šajā rokā, pietiek pielietot vienu *&* operāciju pret roku ar visiem pīķiem.

Papildus pašām kārtīm, atmiņā par tām tiek glabāta arī zīmīga informācija par saistītām kārtīm. Šī informācija tiek glabāta uzmeklēšanas tabulās (*lookup tables*), kas ir kodētas kā *integer* masīvi, lai iegūtu ātru nolasīšanas ātrumu uz patērētās atmiņas rēķina. Piemēram, ir uzmeklēšanas tabulas *VALID_MOVES* un *STRONGER*, kas par katru kārti attiecīgi glabā visas kārtis, ko drīkst likt pēc kārts, un visas kārtis, kuras ir stiprākas par kārti. Uzmeklēšanas tabulas ir arī priekš kāršu kombinācijām un veselām rokām. Piemēram, tabula *VALUE* jebkurai kāršu kopai momentāni atgriež tās kāršu kopējo acu skaitu.

Šāda kāršu reprezentācija gan nedaudz apgrūtina programmēšanu, jo kārtīm nevar pielietot ērtas apstrādes bibliotēkas kā *Language-Integrated Query (LINQ)*. Darbībām ar tām jāpielieto zema abstrakcijas līmeņa programmēšanas loģiku.

```
public static List<Card> GetValidMoves(List<Card> hand, Card trickCard)
{
    List<Card> validMoves = hand.FindAll(card => card.KIND == trickCard.KIND);
    if (validMoves.Count == 0)
        validMoves = hand;
    return validMoves;
}
```

3.2. att. ***GetValidMoves*** **metode pirms optimizācijas**

```
public static int GetValidMoves(int hand, int trickCard)
{
    int validMoves = hand & Deck.VALID_MOVES[trickCard];
    if (validMoves == 0)
        validMoves = hand;
    return validMoves;
}
```

3.3. att. ***GetValidMoves*** **metode pēc optimizācijas**

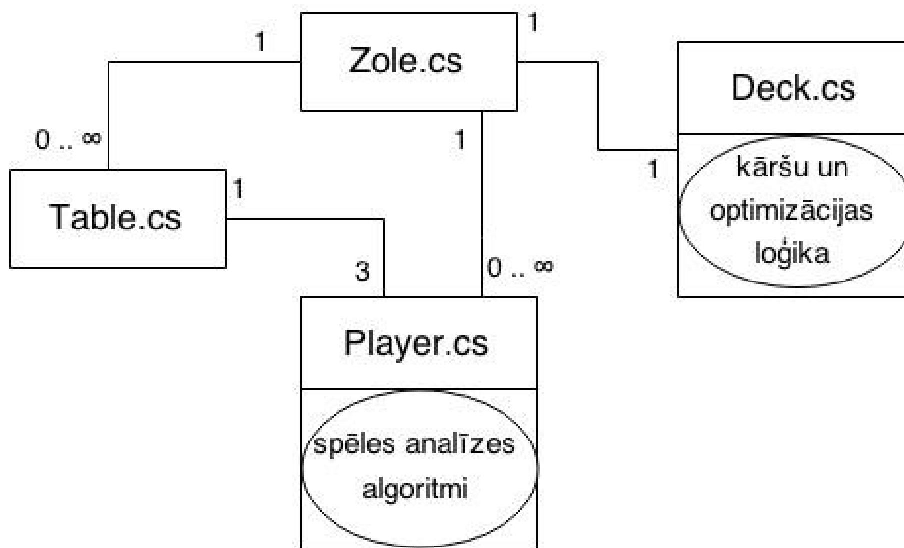
Attēlos 3.2. un 3.3. parādīts, kā tika optimizēta atļauto gājienu atrašanas metode. Abstraktie tipi vienkāršoti par *integer* tipiem un iteratīvā $O(n)$ laika metode *FindAll()* aizstāta ar momentāno $O(1)$ bitu operatoru *&*, kas pielietots uzmeklēšanas tabulai *VALID_MOVES*.

Optimizācijas metodes ir apskatāmas 3. pielikumā – klases *Deck.cs* pirmkodā. Pēc programmas optimizēšanas tās ātrdarbība, aprēķinot vienāda izmēra spēles koku, pieauga aptuveni 8 reizes.

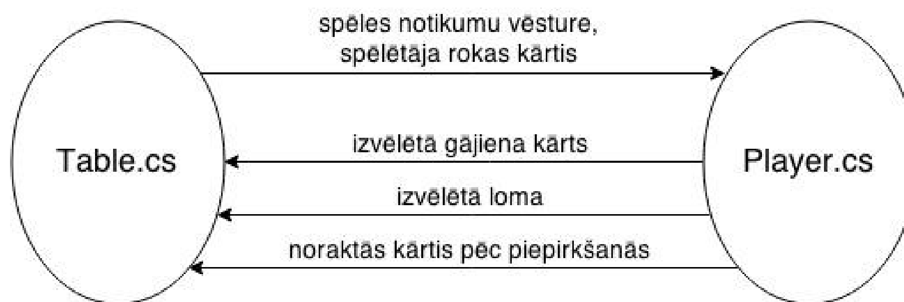
3.3. Risinājuma arhitektūra

Balstoties uz autora un 2. nodaļā minēto risinājumu pieredzi, tika izveidota arhitektūra ar 4 klasēm (vienkāršota klašu diagramma parādīta 3.4. attēlā):

- Zole.cs – koordinē kopējo programmas darbību, izveido galdiņus (*Table.cs*) un piešķir tiem spēlētājus (*Player.cs*).
- Table.cs – regulē tam piešķirtos spēlētājus, izdalot tiem kārtis un liekot pieņemt lēmumus. Kontrolē spēles noteikumu ievērošanu.
- Deck.cs – atbild par kāršu implementāciju, nosaukumiem, stiprumu un acīm. Ietver arī optimizēto funkcionalitāti, kas nepieciešama kāršu un spēles noteikumu pareizai darbībai.
- Player.cs – veic spēļu simulāciju un analīzi un no tās rezultātiem izvēlas labāko gājienu dotā spēles situācijā.



3.4. att. **Aģenta klašu diagramma, vienkāršota**



3.5. att. **Klašu Table.cs un Player.cs sadarbība**

Kā redzams 3.5. attēlā, visa spēles organizācijas loģika ir nodalīta no spēlētāja klases un ieviesta klasē *Table.cs*. Spēlētājam ir jāpilda tikai nodaļas sākumā minētie uzdevumi, saņemot patvaļīgu spēles stāvokļa informāciju no galdiņa.

Klasē *Player.cs* atrodas spēles koka analīzes metode. Analīzes algoritmam paredzēts spēles pēdējos gājienos pielietot precīzo minimax algoritmu. Savukārt spēles sākumā, kur apskatāmo izspēļu skaits ir pārāk liels, paredzēts pielietot dažādas heiristikas.

Risinājuma arhitektūras sākotnējā versijā nav iekļauta grafiskā saskarne vai saskarne ar citām programmām. Taču tās izstrāde tiek veikta, paredzot jaunas saskarnes pievienošanu un plānotu iespēju spēlēt pret dzīviem cilvēkiem vai citiem aģentiem.

REZULTĀTI

Darba rezultātā C# valodā tika implementēta Zoles noteikumu loģika un iesākts veidot spēles situāciju analizējošu aģentu. Diemžēl aģentam praktiski ieviesta tikai daļa no darbā pieminētajiem risinājumiem, kuri varētu padarīt tā spēli labāku un tam trūkst *izvēles daļas* risinājums. Aģentā ir implementēta PIMC metode, kas apakškoku analīzi veic ar alpha-beta pruning optimizētu minimax koka meklēšanu. Tātad, tiek noteikti iespējamie pretinieku kāršu sadalījumi un katrs no tiem analizēts ar koka apstaigāšanas algoritmiem.

```
file:///C:/Users/Gustavs/documents/visual studio 2013/Projects/VisasIzspelesBit...
Dealt cards:
P1: 9♠ 10♠ A♠ K♣ 10♣ A♦ J♥ Q♠
P2: 9♥ 10♥ K♠ 8♦ J♦ J♣ Q♥ Q♣
P3: K♥ A♥ 9♣ A♣ 7♦ 9♦ K♦ Q♦
table: 10♦ J♠

10♠ K♠ A♣ K♣ 8♦ 9♣ J♦ K♦ A♦ 10♥ K♥ A♠
9♥ score:       -4,24761904761905       Simulations:    3150
J♣ score:       -4,38603174603175       Simulations:    3150
Q♥ score:       -4,63746031746032       Simulations:    3150
Q♣ score:       -4,46095238095238       Simulations:    3150
Played game:
10♠ K♠ A♣ K♣ 8♦ 9♣ J♦ K♦ A♦ 10♥ K♥ A♠ Q♥ 7♦ J♥ J♣ Q♦ 9♠ A♥ 10♣ 9♥ 9♦ Q♠ Q♣
P1 score:
K♠ 10♠ A♣
37: -4
P2 score:
K♥ 10♥ A♠ 9♣ K♣ 7♦ 8♦ 9♦ K♦ A♦ J♦ J♥ Q♥ Q♠ Q♣
69: 2
P3 score:
9♥ A♥ 9♠ 10♣ J♣ Q♦
38: 2

Time elapsed: 00:00:03.1414983
```

4.1. att. **Aģenta darbība atkļūdošanas režīmā**

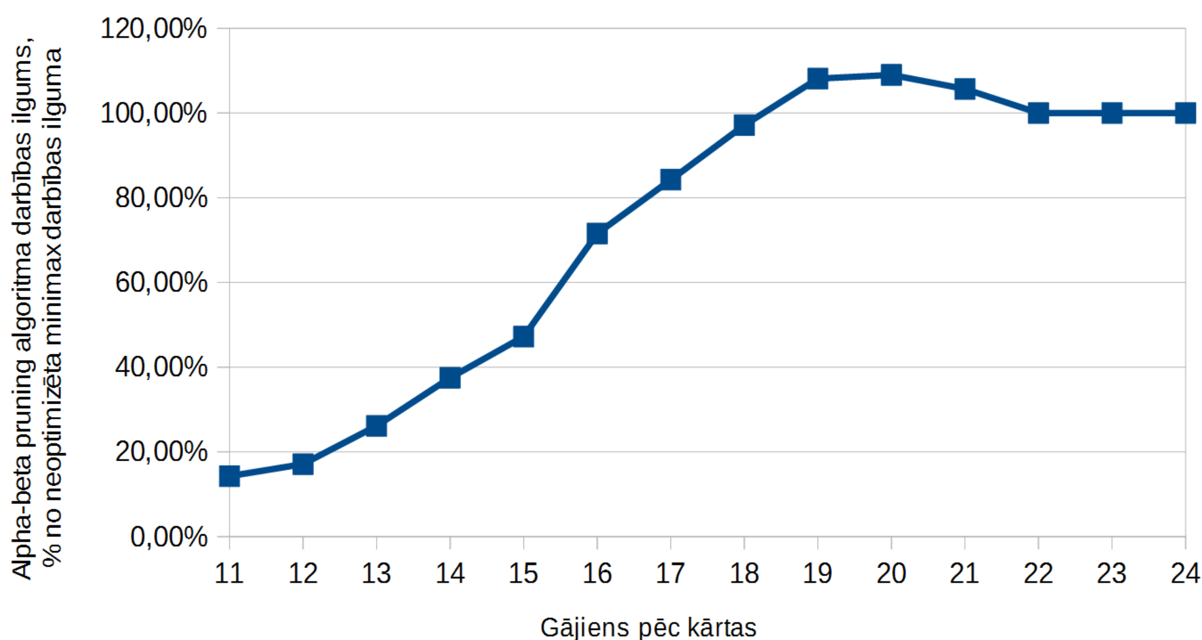
Kā redzams 4.1. attēlā, aģenta risinājuma sākotnējā versija ir *Windows* konsoles lietojumprogramma bez grafiskās saskarnes. Izveidotā programma spēj veikt pēdējo 12 gājienu analīzi un pieņemt lēmumu par labāko gājienu, balstoties uz augstāk minētajiem algoritmiem. Zoli analizējošo metožu pirmkods, kas implementēts klasē *Player.cs*, ir apskatāms 4. un 5. pielikumā.

Veicot mērījumus ar 100 nejaušiem kāršu sākuma sadalījumiem, tika izmērīts algoritmu patērētais laiks atkarībā no analizējamo gājienu skaita. Spēles sākuma daļa tika izspēlēta pēc nejaušības principa un pēc tam tika noteikts, cik sekundes nepieciešamas katra nākamā gājiena aprēķināšanai, sākot no vienpadsmitā gājiena.

Tika noteikts, ka ar neoptimizētu minimax algoritmu 11. gājiena aprēķināšana vidēji tiek patērētas 10,69 sekundes, 12. gājiena – 2,25 sekundes, 13. gājiena – 0,52 sekundes. Savukārt ar alpha-beta pruning optimizēts algoritms 11. gājiena aprēķināšanai vidēji patērē 1,53 sekundes, 12. gājiena – 0,39 sekundes, 13. gājiena – 0,14 sekundes. Abiem algoritmiem

katra nākamā gājienu aprēķināšana patērē mazāk laika par iepriekšējā gājiena aprēķināšanu. Plašāki mērījumu rezultāti ir 2. pielikumā.

Tāpat tika noteikts, ka visiem 100 sadalījumiem ar alpha-beta pruning optimizētais algoritms spēja 13. un visus tam sekojošos gājienus aprēķināt ātrāk par vienu sekundi. Kamēr vienkāršā minimax implementācija visiem sadalījumiem katru gājienu aprēķināja ātrāk par sekundi tikai sākot no 14. gājiena. Tādējādi tika noteikts, ka turpinot veidot aģentu, tas pirmajos 12 gājienos nevar izmantot tikai šīs precīzās metodes, un tam jāizmanto heiristiskās metodes, kā piemēram Monte Carlo Tree Search.



4.2. att. **Alpha-beta minimax algoritma darbības ilgums, analizējot Zoles izvēļu koku, izteikts procentos no neoptimizēta minimax algoritma darbības ilguma**

Izmantojot mērījumus ar 100 nejaušiem kāršu sākuma sadalījumiem, iespējams arī noteikt ieguvumus no alpha-beta pruning izmantošanas Zoles izvēļu koka analīzē. 4.2. attēlā parādīts, ka vislielākais ieguvums no alpha-beta pruning minimax optimizācijas ir agrākajos gājienos. Piemēram, 11. un 12. gājienā ar alpha-beta pruning optimizēts algoritms gājiena aprēķināšanai patērē mazāk nekā 20% no laika, ko tam pašam aprēķinam patērē neoptimizēta minimax implementācija.

No tā var secināt, ka šī optimizācija ir ļoti svarīga, ja Zoles spēli analizē ar minimax algoritmu. Alpha-beta pruning optimizāciju teorētiski varētu vēl paātrināt, pielietojot optimālu gājienu apskates secību, šobrīd vispirms tiek apskatītas spēlētāja vājākās kārtis.

Ņemot vērā darbā izpētīto, tika izveidots saraksts ar jauniem uzlabojumiem, kas varētu dot ieguvumus aģenta darbībā:

- Secinājumu veikšana par spēlētāju kārtīm. Ja kādā stiķī pretinieks izspēlē citu mastu nekā ir prasīts, tad var pieņemt, ka viņam rokā vairs nav prasītā masta kāršu. Realizētajā Perfect Information Monte Carlo apskatē šī Zoles likumsakarība netiek ņemta vērā, un tiek apskatītas neiespējamu gājienu varbūtības, kas noved pie sliktākiem gājieniem.
- Cilvēkiem domātu labākas spēles ieteikumu ieviešana aģenta darbībā pirmajos gājienos. Iespējams gan, ka optimizēts aģents spētu atrast labāku gājienu par ieteicamajām pirmo kāršu izspēlēm.
- Ātrdarbības uzlabošana vairāku procesora kodolu datoriem. Uz vairākiem procesora kodoliem var vienlaicīgi analizēt dažādas kārtis. Analizējot iespējamo izspēļu skaitu (3.1. nodaļa), šāda metode deva ievērojamu rēķināšanas ātruma palielināšanos.
- Ņemot vērā to, ka ieviestais risinājums daudz izmanto bitu loģiskās operācijas un vienkāršus matemātiskos aprēķinus, risinājumu varētu pielāgot rēķināšanai ar datora grafikas procesoru (*graphics processor unit*).

Ievērojamus uzlabojumus aģenta darbībā dotu arī citi 2.3. nodaļā pieminēto risinājumu ieviešana, kuri darba gaitā netika praktiski izmēģināti.

No iegūtajiem rezultātiem var secināt, ka visu Zoles spēli nevar analizēt ar to pašu algoritmu. Lai iegūtu labākos rezultātus, katrā spēles stadijā jāpielieto tai vispiemērotākās analīzes metodes. Izveidotajā risinājumā pielietotās metodes ir balstītas galvenokārt uz Skat spēles aģentu pieredzi. To patieso piemērotību veicamajam darbam būtu nepieciešams pārbaudīt īsta Zoles turnīra apstākļos.

SECINĀJUMI

Bakalaura darba mērķus var uzskatīt par izpildītiem, jo ir veikta gan Zoles spēles analīze, gan izpētītas iespējas tās spēli un spēlētāju modelēt ar datoru, gan iesākts veidot zoles aģentu, kas spēj veikt primitīvu analīzi spēles beigu daļā.

No iegūtajiem rezultātiem autors var secināt, ka:

- Zoli spēlējošu algoritmu modelēšanai var pielietot tai radniecīgajā spēlē Skat izmantotus algoritmus un zināšanas.
- Perfect information Monte Carlo metodi, kas izmanto alpha-beta pruning optimizētu minimax algoritmu, var pielietot ātrai pēdējo divpadsmit Zoles gājienu analīzei ar datoru.
- Ir gandrīz neiespējami novērtēt aģenta spēles kvalitāti bez daudzkārtējām spēlēm pret dzīviem cilvēkiem vai citiem aģentiem.
- Nav iespējams atrisināt optimālo gājienu visām iespējamajām Zoles situācijām un izmantot iegūtos rezultātus par pamatu spēles aģenta darbībai, jo spēles iespējamo situāciju skaits ir pārāk liels.
- Analizējot Zoles spēles izvēles koku ar minimax algoritmu, ļoti liela nozīme ir tā optimizācijai. Ar pamata minimax algoritmu, kas apskata pilnīgi visus izspēļu variantus, var gana ātri analizēt tikai spēles pēdējos gājienos. Spēles sākumā iegūtais izspēļu skaits būtu pārlieku liels šādai metodei, tāpēc jāpielieto algoritmi, kas samazina apskatāmo izspēļu skaitu.
- Neskaitot ļoti vienkāršas spēles, veidojot mākslīgā intelekta aģentu ir jāmeklē kompromisu starp aprēķina sarežģītību un precizitāti. Algoritmi, kuri var sniegt visdrošāko rezultātu, prasa lielus laika un/vai telpas resursus. Līdz ar to, spēcīgākie spēļu aģenti bieži ir tie, kuri vislabāk izmanto spēlei raksturīgas heiristikas, nevis tie, kuri izmanto algoritmu ar garantēti augstāko precizitāti.
- Ir nozīme, kādā veidā tiek kodētas spēļu kārtis un operācijas ar tām. Optimizējot kāršu risinājumu, var iegūt vairākkārtīgu ātrdarbības pieaugumu, kas ir tiešs ieguvums aģenta gājienu kvalitātes uzlabošanā – ātrāka darbība ļauj apskatīt vairāk izspēļu, kas savukārt sniedz plašāku informāciju lēmuma pieņemšanas procesā.

Kā turpmākas darba perspektīvas var pieminēt darbā aprakstīto risinājumu ieviešanu un aģenta pabeigšanu. Paredzēts arī izveidot grafisko saskarni, kas ļautu aģentam spēlēt pret cilvēkiem. Ieviešot saskarnes risinājumu, pavērtos iespēja novērtēt aģenta darbības stiprās un vājās puses dabiskos apstākļos kā arī iespēja pielietot mašīnmācīšanās metodes spēles uzlabošanā. Spēļu gājienu vēsturu iegūšana no esošajiem Zoles portāliem un to analīze varētu lieti noderēt tālākai aģenta uzlabošanai. Kā papildus iespēja būtu izveidot saskarni un serveri, kas ļautu Zoli spēlēt pret citiem spēles aģentiem vai cilvēkiem.

Darba gaitā autors ir apguvis spēļu teorijas pamatprincipus un to pielietošanu Zoles spēles analīzei. Iegūtās zināšanas gan ļaus pabeigt Zoles aģenta izstrādi, gan var noderēt citu mākslīgā intelekta risinājumu izstrādē, kas var arī nebūt saistīti ar kāršu spēlēm.

IZMANTOTĀ LITERATŪRA UN AVOTI

[1] Latvijas Zolītes federācija. (2014) *Mūsu nacionālā prāta spēle – zolīte*. [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: http://www.irir.lv/blogi/sports/musu-nacionala-prata-spele-zolite

[2] A. Ikaunieks (2015) *Spēlējam zolīti!* [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: http://www.kastanis.biz/Zole.html

[3] A. A. Bērziņš. (2015) *Zolītes spēle DOSā: viens dzīvs spēlētājs un trīs skaitļotājspēlētāji.* [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: http://ansis.lv/Zole

[4] Latvijas Zolītes federācija. (2015) *Zoles noteikumi.* [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: http://www.Zolei.lv/lv/Zoles-noteikumi

[5] M. Scarvalone. (2008) *Game Theory and the minimax theorem* [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: http://www.math.uchicago.edu/~may/VIGRE/VIGRE2008/REUPapers/Scarvalone.pdf

[6] G. Gerhardt. (2004) *The card game Skat for Linux, Mac OS X, Windows and Android* [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: http://www.xskat.de/xskat.html

[7] S. Kupferschmid, M. Helmert. (2006) *A Skat Player Based on Monte Carlo Simulation* [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: http://ai.cs.unibas.ch/papers/kupferschmid-helmert-cg2006.pdf

[8] T. Keller, S. Kupferschmid. (2008) *Automatic Bidding for the Game of Skat* [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: http://link.springer.com/chapter/10.1007/978-3-540-85845-4_12

[9] J. Schäfer. „Monte Carlo Simulation im Skat.” Bachelor thesis, Ottovon-Guericke-Universität Magdeburg, Fakultät für Informatik, Institut für Simulation und Graphik, 2005.

[10] J. Schäfer. „The UCT Algorithm Applied to Games with Imperfect Information.” Master’s thesis, Ottovon-Guericke-Universität Magdeburg, Fakultät für Informatik, Institut für Simulation und Graphik, 2008.

[11] M. Buro u.c. (2009) *Improving State Evaluation, Inference, and Search in Trick-Based Card Games* [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: https://skatgame.net/mburo/ps/ijcai09-skat.pdf

[12] Wikipedia, the free encyclopedia. *Nash equilibrium*. [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: http://en.wikipedia.org/wiki/Nash_equilibrium

[13] Wikipedia, the free encyclopedia. *Alpha-beta pruning*. [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

[14] Wikipedia, the free encyclopedia. *Monte Carlo tree search*. [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: http://en.wikipedia.org/wiki/Monte_Carlo_tree_search

[15] R. E. Korf, D. M. Chickering. (1993) *Best-First Minimax Search: First Results*. [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: http://aaaipress.org/Papers/Symposia/Fall/1993/FS-93-02/FS93-02-006.pdf

[16] J. Long u.c. (2010) *Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search*. [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: https://skatgame.net/mburo/ps/aaai10-mc.pdf

[17] T. Furtak, M. Buro. (2013) *Recursive Monte Carlo Search for Imperfect Information Games* [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: https://skatgame.net/mburo/ps/recmc13.pdf

[18] A. Fog. (2014) *Optimizing software in C++* [tiešsaiste] – [atsauce 28.05.2014]. Pieejams: http://www.agner.org/optimize/optimizing_cpp.pdf

PIELIKUMI

1. pielikums

Nejaušu kāršu sadalījumu iespējamo izspēļu skaiti

Npk.	Izspēļu skaits
1.	57 446 432 064
2.	36 657 684 672
3.	2 117 333 088
4.	1 783 330 592
5.	13 826 043 648
6.	1 491 903 008
7.	10 538 864 232
8.	11 483 562 912
9.	9 998 265 040
10.	1 976 742 592
11.	2 088 285 696
12.	36 899 720 576
13.	10 418 166 720
14.	3 072 988 608
15.	19 163 652 064
16.	2 725 937 536
17.	223 597 089 792
18.	144 249 136 704
19.	2 635 987 608
20.	58 921 712 640
21.	193 466 944
22.	33 173 900 928
23.	7 241 389 344
24.	7 317 166 080
25.	655 407 840
26.	13 951 030 176
27.	3 774 389 408
28.	31 848 758 592
29.	5 755 365 888
30.	1 974 903 552
31.	1 926 156 960
32.	1 688 937 968
33.	227 515 008
34.	5 895 879 680
35.	142 086 085 632
36.	16 442 507 520
37.	2 844 371 040
38.	8 548 789 248
39.	13 764 348 672
40.	175 999 872

Npk.	Izspēļu skaits
41.	3 756 311 424
42.	1 165 994 560
43.	12 834 894 384
44.	1 901 812 224
45.	4 060 816 800
46.	11 165 031 648
47.	3 287 150 784
48.	71 281 141 632
49.	189 403 067 904
50.	12 661 639 616
51.	247 059 936
52.	8 957 594 112
53.	473 162 880
54.	3 315 156 096
55.	140 906 723 328
56.	10 776 231 936
57.	28 139 495 616
58.	2 367 235 328
59.	16 537 495 808
60.	3 807 926 208
61.	5 653 196 928
62.	2 035 726 632
63.	1 280 326 108
64.	7 404 056 640

2. pielikums

Gājienu aprēķināšanas ilgums – zīmīgās vērtības 100 nejaušiem sadalījumiem

Gājiena aprēķināšinas ilgums, zīmīgās vērtības 100 mērījumiem

Gājiens pēc kārtas	Neoptimizēts minimax			
	Minimums	Maksimums	Vidējais	Mediāna
11	0,0758108	63,1847966	10,6900543	4,3142622
12	0,0133133	11,3141424	2,2539773	1,2031547
13	0,0098521	5,1150663	0,5237457	0,0542572
14	0,0008526	0,3977205	0,0815722	0,0549206
15	0,0002172	0,0597844	0,0142888	0,0086962
16	0,0001758	0,0180793	0,0051921	0,0060802
17	0,0000230	0,0030251	0,0007256	0,0005282
18	0,0000068	0,0005154	0,0001540	0,0000344
19	0,0000076	0,0001876	0,0000748	0,0000603
20	0,0000024	0,0001167	0,0000193	0,0000168
21	0,0000016	0,0000109	0,0000057	0,0000048
22	0	0	0	0
23	0	0	0	0
24	0	0	0	0

	Ar alpha-beta pruning optimizēts minimax			
	Minimums	Maksimums	Vidējais	Mediāna
11	0,0104421	10,7066580	1,5268059	0,4516843
12	0,0029663	1,9994257	0,3857110	0,2365005
13	0,0032406	0,7757780	0,1368818	0,0133305
14	0,0003521	0,1099555	0,0305893	0,0209478
15	0,0001325	0,0242811	0,0067467	0,0044576
16	0,0001377	0,0151356	0,0037155	0,0044278
17	0,0000202	0,0023455	0,0006113	0,0004391
18	0,0000068	0,0004915	0,0001496	0,0000338
19	0,0000085	0,0002172	0,0000809	0,0000855
20	0,0000024	0,0001369	0,0000210	0,0000184
21	0,0000016	0,0000158	0,0000060	0,0000052
22	0	0	0	0
23	0	0	0	0
24	0	0	0	0

Klases Deck.cs pirmkods

```
using System;

using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace VisasIzspelesBitwise
{
    public class Deck
    {
        public const int SIZE = 26;
        public const int EMPTY_CARD = 0;
        public const int MAX_CARD = 33554432;
        public const int FULL_DECK = MAX_CARD * 2 - 1;

        public const int HAND_SIZE = 8;
        public const int TABLE_SIZE = 2;

        public const double MAX_SCORE = 10000;
        public const double MIN_SCORE = -10000;

        public static readonly int[] VALID_MOVES = new int[MAX_CARD + 1];
        public static readonly int[] STRONGER = new int[MAX_CARD + 1];
        public static readonly int[] VALUE = new int[FULL_DECK + 1];
        public static readonly int[] NEXTHANDSIZE = new int[SIZE];

        private static Random rand = new Random(12);
        private static Random rand2 = new Random(23);

        public Deck()
        {
            for (int card = 1; card <= MAX_CARD; card = (card << 1))
                VALID_MOVES[card] = _VALID_MOVES(card);

            for (int card = 1; card <= MAX_CARD; card = (card << 1))
                STRONGER[card] = _STRONGER(card);

            for (int hand = 0; hand <= FULL_DECK; hand++)
                VALUE[hand] = _VALUE(hand);

            for (int move = 0; move < SIZE; move++)
                NEXTHANDSIZE[move] = _NEXTHANDSIZE(move);



            if (VALUE[FULL_DECK] != 120)
                throw new Exception("Deck value is not 120!");
        }

        public static int[] GetRandomHands()
        {
            int[] playerHands = { 0, 0, 0, 0 };
            List<int> deck = new List<int>();
            for (int i = 0; i < Deck.SIZE; i++)
                deck.Add(1 << i);
            deck = deck.OrderBy(c => (int)rand.Next()).ToList();
            for (int i = 0; i < Deck.SIZE; i++)
                playerHands[i / HAND_SIZE] |= deck.ElementAt(i);
```

```
            if ((playerHands[0] ^ playerHands[1] ^ playerHands[2] ^ playerHands[3]) != 
Deck.FULL_DECK)
                throw new Exception("Incorrect hands");
            return playerHands;
        }

        /// <summary>
        /// Returns valid moves for a lead card.
        /// </summary>
        private static int _VALID_MOVES(int card)
        {
            switch (card)
            {
                case EMPTY_CARD: return FULL_DECK;//any card
                case 1: return 15;//hearts
                case 2: return 15;
                case 4: return 15;
                case 8: return 15;
                case 16: return 240;//spades
                case 32: return 240;
                case 64: return 240;
                case 128: return 240;
                case 256: return 3840;//clubs
                case 512: return 3840;
                case 1024: return 3840;
                case 2048: return 3840;
                case 4096: return 16773120;//trumps
                case 8192: return 16773120;
                case 16384: return 16773120;
                case 32768: return 16773120;
                case 65536: return 16773120;
                case 131072: return 16773120;
                case 262144: return 16773120;
                case 524288: return 16773120;
                case 1048576: return 16773120;
                case 2097152: return 16773120;
                case 4194304: return 16773120;
                case 8388608: return 16773120;
                case 16777216: return 16773120;
                case 33554432: return 16773120;//MAX_CARD
                default: return 0;
            }
        }

        /// <summary>
        ///  Returns all cards stronger than card.
        /// </summary>
        private static int _STRONGER(int card)
        {
            switch (card)
            {
                case EMPTY_CARD: return FULL_DECK;//any card
                case 1: return 67104782;//hearts
                case 2: return 67104780;
                case 4: return 67104776;
                case 8: return 67104768;
                case 16: return 67104992;//spades
                case 32: return 67104960;
                case 64: return 67104896;
                case 128: return 67104768;
                case 256: return 67108352;//clubs
                case 512: return 67107840;
```

```
            case 1024: return 67106816;
            case 2048: return 67104768;
            case 4096: return 67100672;//trumps
            case 8192: return 67092480;
            case 16384: return 67076096;
            case 32768: return 67043328;
            case 65536: return 66977792;
            case 131072: return 66846720;
            case 262144: return 66584576;
            case 524288: return 66060288;
            case 1048576: return 65011712;
            case 2097152: return 62914560;
            case 4194304: return 58720256;
            case 8388608: return 50331648;
            case 16777216: return 33554432;
            case 33554432: return 0;//MAX_CARD
            default: return 0;
        }
    }
    /// <summary>
    /// Returns card points value for any card combination.
    /// </summary>
    private static int _VALUE(int hand)
    {
        int result = 0;
        if ((hand & 2) > 0) result += 4;
        if ((hand & 4) > 0) result += 10;
        if ((hand & 8) > 0) result += 11;
        if ((hand & 32) > 0) result += 4;
        if ((hand & 64) > 0) result += 10;
        if ((hand & 128) > 0) result += 11;
        if ((hand & 512) > 0) result += 4;
        if ((hand & 1024) > 0) result += 10;
        if ((hand & 2048) > 0) result += 11;
        if ((hand & 32768) > 0) result += 4;
        if ((hand & 65536) > 0) result += 10;
        if ((hand & 131072) > 0) result += 11;
        if ((hand & 262144) > 0) result += 2;
        if ((hand & 524288) > 0) result += 2;
        if ((hand & 1048576) > 0) result += 2;
        if ((hand & 2097152) > 0) result += 2;
        if ((hand & 4194304) > 0) result += 3;
        if ((hand & 8388608) > 0) result += 3;
        if ((hand & 16777216) > 0) result += 3;
        if ((hand & 33554432) > 0) result += 3;
        return result;
    }
    /// <summary>
    /// Returns number of cards left in next player's hand.
    /// </summary>
    private static int _NEXTHANDSIZE(int moveCount)
    {
        return HAND_SIZE - ((moveCount+1) / 3); //3 == player count
    }
    public static string SHORTNAME(int card)
    {
        switch (card)
        {
            case 1: return "9♥";//hearts
            case 2: return "K♥";
            case 4: return "10♥";
            case 8: return "A♥";
```

```
            case 16: return "9♠";//spades
            case 32: return "K♠";
            case 64: return "10♠";
            case 128: return "A♠";
            case 256: return "9♣";//clubs
            case 512: return "K♣";
            case 1024: return "10♣";
            case 2048: return "A♣";
            case 4096: return "7♦";//trumps
            case 8192: return "8♦";
            case 16384: return "9♦";
            case 32768: return "K♦";
            case 65536: return "10♦";
            case 131072: return "A♦";
            case 262144: return "J♦";
            case 524288: return "J♥";
            case 1048576: return "J♠";
            case 2097152: return "J♣";
            case 4194304: return "Q♦";
            case 8388608: return "Q♥";
            case 16777216: return "Q♠";
            case 33554432: return "Q♣";//MAX_CARD
            default: return "?";
        }
    }

    public static Player GetWinner(int[] moveHistory, int moveCount, Player
activePlayer)
    {
        if (IsStronger(moveHistory[moveCount - 2], moveHistory[moveCount - 3]))
        {
            if (IsStronger(moveHistory[moveCount - 1], moveHistory[moveCount -
2]))
                return activePlayer;
            else
                return activePlayer.Next.Next;
        }
        else if (IsStronger(moveHistory[moveCount - 1], moveHistory[moveCount -
3]))
            return activePlayer;
        else
            return activePlayer.Next;
    }

    public static int GetValidMoves(int hand, int trickCard)
    {
        int validMoves = Intersection(hand, Deck.VALID_MOVES[trickCard]);
        if (validMoves == 0)
            validMoves = hand;
        return validMoves;
    }

    public static int AllHistoryCards(int[] moveHistory, int moveCount)
    {
        int cards = 0;
        for (int i = 0; i < moveCount; i++)
            cards |= moveHistory[i];
        return cards;
    }

    /// <summary>
    /// Returns true if card1 > card2.
```

```
/// </summary>
public static bool IsStronger(int card1, int card2)
{
    return ((card1 & Deck.STRONGER[card2]) != 0);
}
public static int RemoveLowestCard(ref int hand)
{
    int card = hand & -hand;
    hand = hand ^ card;
    return card;
}
public static int GetLowestCard(int hand)
{
    return hand & -hand;
}

public static void RemoveCard(ref int hand, int card)
{
    hand = hand ^ card;
}

public static int Intersection(int hand1, int hand2)
{
    return hand1 & hand2;
}

public static int CountCards(int hand)
{
    int count = 0;
    while (hand != 0)
    {
        hand &= hand - 1;
        count++;
    }
    return count;
}

public static int PickRandomCard(int hand)
{
    int card = 0;
    int count = CountCards(hand);
    for (int i = 0; i < rand2.Next(1, count); i++)
        card = RemoveLowestCard(ref hand);
    return card;
}

public static void PrintHistory(int[] moveHistory, int moveCount)
{
    for (int i = 0; i < moveCount; i++)
        Console.Write(Deck.SHORTNAME(moveHistory[i]) + " ");
    Console.WriteLine();
}

public static void PrintHand(int hand, string message = "")
{
    if (message != "")
        Console.Write(message + ": ");
    while (hand != 0)
        Console.Write(Deck.SHORTNAME(RemoveLowestCard(ref hand)) + " ");
    Console.WriteLine();
}
```

```
        public static int GetScore(int points)
        {
            if (points == 0) return -8;
            else if (points <= 30) return -6;
            else if (points <= 60) return -4;
            else if (points < 90) return 2;
            else if (points < 120) return 4;
            else if (points == 120) return 6;
            return 0;
        }

        /// <summary>
        /// Returns all possible subsets of size "size" from given cards.
        /// </summary>
        /// <param name="cards">Set of cards</param>
        /// <param name="size">Size of subsets</param>
        /// <returns>All possible subsets of size "size" from given cards</returns>
        public static IEnumerable<int> Combinations(int cards, int size, int a = 0,
int elems = 0)
        {
            if (elems == size)
            {
                yield return a;
            }
            while (cards != 0)
            {
                int b = Deck.RemoveLowestCard(ref cards);
                foreach (int c in Combinations(cards, size, a | b, elems + 1))
                    yield return c;
            }
        }

        public static void PrintBinaryInt(int n)
        {
            char[] b = new char[32];
            int pos = 31;

            for (int i = 0; i < 32; i++)
            {
                if ((n & (1 << i)) != 0)
                {
                    b[pos] = '1';
                }
                else
                {
                    b[pos] = '0';
                }
                pos--;
            }
            Console.WriteLine(b);
        }
    }
}
```

Klases Player.cs metode PlayCard

```
public int PlayCard(int[] moveHistory, int moveCount, int[] playerTricks, int
validMoves, int trickCard, int[] playerHands) // TODO: no player hands
        {
            if (moveCount >= LAST_ROUND - 1)
                return Deck.RemoveLowestCard(ref validMoves);

            if (moveCount < START_FROM - 1)
                return Deck.PickRandomCard(validMoves);

            int[] hist = (int[])moveHistory.Clone();
            int[] hands = new int[playerHands.Length];

            // player cards won:
            int cards0 = playerTricks[0];
            int cards1 = playerTricks[1];
            int cards2 = playerTricks[2];

            double score;
            if (this.Role == PlayerRole.Lielais)
                score = Deck.MIN_SCORE;
            else
                score = Deck.MAX_SCORE;
            long gameCount = 0;
            int playedCard;
            int bestCard = Deck.GetLowestCard(validMoves);
            while (validMoves != 0)
            {
                playedCard = Deck.RemoveLowestCard(ref validMoves);

                double newScore = 0;
                int simulations = 0;
                int unknownCards = Deck.FULL_DECK ^ Deck.AllHistoryCards(moveHistory,
moveCount) ^ playerHands[this.ID];
                int unknownBurried = unknownCards;
                if (this.Role == PlayerRole.Lielais)
                    unknownBurried = playerHands[3];

                foreach (int possibleBurried in Deck.Combinations(unknownBurried,
Deck.TABLE_SIZE))
                {
                    hands[3] = possibleBurried;
                    int nextHandSize = Deck.NEXTHANDSIZE[moveCount];
                    foreach (int possibleHand in Deck.Combinations(unknownCards ^
possibleBurried, nextHandSize))
                    {
                        hands[this.ID] = playerHands[this.ID];
                        hands[this.Next.ID] = possibleHand;
                        hands[this.Next.Next.ID] = unknownCards ^ possibleBurried ^
possibleHand;

                        double alpha, beta;
                        alpha = Deck.MIN_SCORE;
                        beta = Deck.MAX_SCORE;


                        //simulates all possible games
```

```
                        newScore += this.SimulateGame(hist, moveCount, playedCard,
trickCard, hands[0], hands[1], hands[2], hands[3], cards0, cards1, cards2, alpha,
beta, ref gameCount);
                        simulations++;
                    }
                }
                newScore = newScore / simulations;
                if (((this.Role == PlayerRole.Lielais) && (score < newScore)) ||
((this.Role == PlayerRole.Mazais) && (score > newScore)))
                {
                    score = newScore;
                    bestCard = playedCard;
                }
            }
            return bestCard;
        }
```

Klases Player.cs metode SimulateGame

```
private double SimulateGame(int[] moveHistory, int moveCount, int playedCard, int
trickCard,
            int hand0, int hand1, int hand2, int burriedCards, int cards0, int cards1,
int cards2, double alpha, double beta, ref long gameCount)
        {
            int validMoves;
            if (moveCount % this.Table.playerCount == 0)
                trickCard = playedCard;
            moveHistory[moveCount++] = playedCard;
            if (this.ID == 0)
                Deck.RemoveCard(ref hand0, playedCard);
            else if (this.ID == 1)
                Deck.RemoveCard(ref hand1, playedCard);
            else// if (activePlayer.ID == 2)
                Deck.RemoveCard(ref hand2, playedCard);

            Player nextPlayer;
            if (moveCount % this.Table.playerCount == 0) // trick ends
            {
                nextPlayer = Deck.GetWinner(moveHistory, moveCount, this);
                if (nextPlayer.ID == 0)
                {
                    validMoves = hand0;
                    cards0 |= moveHistory[moveCount - 3] | moveHistory[moveCount - 2]
| moveHistory[moveCount - 1];
                }
                else if (nextPlayer.ID == 1)
                {
                    validMoves = hand1;
                    cards1 |= moveHistory[moveCount - 3] | moveHistory[moveCount - 2]
| moveHistory[moveCount - 1];
                }
                else
                {//if (nextPlayer.ID == 2)
                    validMoves = hand2;
                    cards2 |= moveHistory[moveCount - 3] | moveHistory[moveCount - 2]
| moveHistory[moveCount - 1];
                }
            }
            else
            {
                nextPlayer = this.Next;
                if (nextPlayer.ID == 0)
                    validMoves = Deck.GetValidMoves(hand0, trickCard);
                else if (nextPlayer.ID == 1)
                    validMoves = Deck.GetValidMoves(hand1, trickCard);
                else// if (nextPlayer.ID == 2)
                    validMoves = Deck.GetValidMoves(hand2, trickCard);
            }
            double score;
            if (nextPlayer.Role == PlayerRole.Lielais)
                score = Deck.MIN_SCORE;
            else
                score = Deck.MAX_SCORE;
            double newScore = score;
            while (validMoves != 0)
            {
```

```
                playedCard = Deck.RemoveLowestCard(ref validMoves);
                newScore = nextPlayer.SimulateGame(moveHistory, moveCount, playedCard,
trickCard, hand0, hand1, hand2, burriedCards, cards0, cards1, cards2, alpha, beta, ref
gameCount);
                if (nextPlayer.Role == PlayerRole.Lielais)
                {
                    score = Math.Max(score, newScore);
                    alpha = Math.Max(alpha, score);
                    if (beta <= alpha)
                        break;
                }
                else
                {
                    score = Math.Min(score, newScore);
                    beta = Math.Min(beta, score);
                    if (beta <= alpha)
                        break;
                }
            }

            // End of game
            if (moveCount == this.Table.playerCount * Deck.HAND_SIZE)
            {
                score = Deck.GetScore(Deck.VALUE[cards0 | burriedCards]);
                gameCount++;
            }
            if (score == Deck.MAX_SCORE || score == Deck.MIN_SCORE)
                throw new Exception("Bad score");
            return score;
        }
    }
}
```

Bakalaura darbs „Mākslīgā intelekta izstrāde kāršu spēlei „Zole”” izstrādāts Latvijas Universitātes Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: ________________________ Gustavs Venters

Rekomendēju/nerekomendēju darbu aizstāvēšanai

Vadītājs: Mg. dat. Kaspars Balodis_____________________________ 01.06.2015.

Recenzents: asociētais docents Dr. dat. Agris Šostaks

Darbs iesniegts Datorikas fakultātē 01.06.2015.

Dekāna pilnvarotā persona: metodiķe Ārija Sproģe _______________________

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

___.___.2015. prot. Nr. ___

Komisijas sekretārs: ____