VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
KOMPIUTERIJOS KATEDRA

Baigiamasis magistro darbas

Dirbtinio intelekto ir kompiuterinės regos algoritmų taikymas stalo futbolo žaidime

Atliko:
Donatas Kimutis parašas

Vadovas:
asist. Dr. Vytautas Valaitis

Vilnius
2018

Turinys

Santrauka 4

Summary 5

Įvadas 6

1. Susijusių darbų analizė 7
- 1.1. "Pinball" žaidimas . . . 7
- 1.2. Autonominis stalo futbolas . . . 8

2. Naudoti įrankiai ir metodai 9
- 2.1. Kompiuterinės regos bibliotekos pasirinkimas . . . 9
- 2.2. Programinės kalbos pasirinkimas . . . 9
- 2.3. Spalvos modelio pasirinkimas . . . 10

3. Kompiuterinės regos algoritmai 11
- 3.1. Paveikslėlio suliejimas . . . 11
- 3.2. Filtravimas pagal spalvą ir konvertavimas į dvejetainį vaizdą . . . 12
- 3.3. Erozija . . . 12
- 3.4. Plėstis . . . 13

4. Dirbtinio intelekto metodai ir algoritmai skirti mokymosi procesui 15
- 4.1. Markovo grandinės ir Markovo sprendimo priėmimo procesas . . . 15
  - 4.1.1. Markovo grandinė . . . 15
  - 4.1.2. Markovo procesas su atlygiu . . . 16
  - 4.1.3. Bellmano lygtis . . . 17
  - 4.1.4. Markovo sprendimo priėmimo procesas . . . 18
  - 4.1.5. Optimalios strategijos pasirinkimas . . . 20
- 4.2. Būsenos-reikšmės funkcijos įvertinimas . . . 21
  - 4.2.1. Monte-Karlo mokymasis . . . 22
  - 4.2.2. Laikinojo skirtumo (TD) mokymasis . . . 23
- 4.3. Q-mokymasis . . . 23
- 4.4. Q-mokymasis su veiksmo-vertės funkcijos aproksimacija . . . 24

5. Sukurtas kompiuterinės regos algoritmas 26

6. Sukurtas dirbtinio intelekto algoritmas 27

7. Stalo futbolo žaidimo modelis 29
- 7.1. Kamuoliuko ir žaidėjų galimi veiksmai ir būsenos . . . 29
- 7.2. Žaidėjų išlaikymas vienoje linijoje su kamuoliuku . . . 30

8. Stalo futbolo konstrukcija 32

9. Atlikti bandymai ir rezultatai 34
- 9.1. Kameros pasirinkimas . . . 34
- 9.2. HSV spalvos modelio reikšmių nustatymas . . . 34

9.3. Apšvietimas 35
9.4. Sukurto kompiuterinės regos algoritmo rezultatai 36
9.5. Sukurto mokymosi algoritmo rezultatai 38
9.5.1. Atlygio skaičiavimas 38
9.5.2. Mokymosi algoritmo parametrai ir apsibrėžtos ypatybės 39
9.5.3. Mokymosi algoritmo rezultatai 39

Išvados ir rekomendacijos **41**

Gairės **42**

Literatūros šaltiniai **43**

Priedai **44**

A. Sukurto kompiuterinės regos algoritmo veiksmų diagrama **44**

B. Sukurta stalo futbolo konstrukcija **45**

Santrauka

Šiame darbe nagrinėjome kompiuterinės regos ir dirbtinio intelekto pritaikymo galimybes stalo futbolo žaidime. Pagrindinis tikslas buvo atlikti kompiuterinės regos ir dirbtinio intelekto metodų analizę, sukurti veikiantį fizinio stalo prototipą ir programą, kuri sugebėtų sekti stalo futbolo žaidėjus ir kamuoliuką, bei priimti sprendimą naudojantis mokymosi algoritmu. Šiame darbe buvo atlikta susijusių darbų analizė, išnagrinėjome kokius įrankius ir programinę kalbą geriausia naudoti šiam darbui ir pasirinkome Python ir OpenCV. Išnagrinėjome svarbiausius kompiuterinės regos metodus (eroziją, plėstį, suliejimą) ir pristatėme savo sukurto algoritmo modelį. Atlikome išsamią mokymosi algoritmų analizę ir pasrinkome naudoti Q-mokymąsi su veiksmo-vertės funkcijos aproksimacija, ir, naudodamiesi šiuo metodu, sukūrėme ir aprašėme mokymosi algoritmą. Taip pat, sukūrėme fizinę stalo konstrukciją valdomą servo varikliukų ir aprašėme mūsų sukurtų algoritmų rezultatus. Vaizdų atpažinimui pasirinkome naudoti "Sony PlayStation 3 Eye" kamerą, su kuria kamuoliuką pavyko atpažinti 92.09% kadrų.

Summary

Artificial Intelligence and Computer Vision Algorithms in Table Soccer Game

Table soccer is a very fast paced strategic game, where there are two teams playing against each other and controlling their set of players. Ball that is in the game can reach speed up to 10 m/s which requires players to react very fast to ball position changes. This game is a very good field for artificial intelligence, computer vision and robotics research, because you have to pick a strategy to be able to score against your opponent, you have to be able to see the position of the ball and players and you have to be able to adjust your players as quickly as possible. Our main goal is to anayze computer vision and learning methods, create a working physical prototype of foosball table and create a program which is able to "understand" and follow red players, blue players, ball and make decision what to do using learning algorithm. In this paper we discussed related work, analyzed best tools and programming languages to use when creating computer vision algorithm and we decided to use Python with OpenCV, and HSV colour space instead of RGB. We analyzed the most important morphological methods (erosion, dilation, blur) in computer vision and explained how our created algorithm works. We conducted a thorough learning alghoritms analysis and decided to use Q-learning with action function approximation method to create our own learning alghoritm. We also constructed physical foosball table that is controlled with servo motors and we presented results that was achieved. For computer vision we used "Sony PlayStation 3 Eye" camera and discussed results - ball was found in 92.09% frames which is acceptable result.

Įvadas

Stalo futbolas, dar kitaip vadinamas fusbalu, užpatentuotas kaip žaidimas kurį mes žinome dabar 1923 m. Anglijoje, Haroldo Searles Thortono, tačiau atsirado šis žaidimas anksčiau - apie 1880-aisiais metais. Pirmiausia šis žaidimas buvo kaip užsiėmimas svečių ar laukiamajame kambaryje, tačiau ganėtinai greitai išpopuliarėjo kaip atskira sporto šaka, ir pasiekė savo "aukso amžių" 1978 m., kada buvo organizuojamas pasaulio stalo futbolo turnyras kurio pagrindinis prizas buvo 1 milijonas JAV dolerių. Tačiau iškart tuomet, pradėjus atsirasti kompiuterniams žaidimams, stalo futbolo populiarumas pradėjo mažėti. Šiais laikais turnyrai vis dar vyksta, tačiau pagrindinis stalo futbolo panaudojimas yra kaip laisvalaikio praleidimo būdas su draugais, taip pat šis žaidimas yra labai populiarus biuruose kaip pertraukėlių praleidimo būdas.

Stalo futbolas yra fizinis žaidimas, kurį norint žaisti reikalingi bent du žmonės, ir tai yra viena iš pagrindinių priežasčių, kodėl kompiuteriniai žaidimai tapo patrauklesniais. Pastaruoju metu buvo sukurtas dirbtinis intelektas su neuroniniais tinklais žaidžiantis arkadinius Atari žaidimus geriau nei žmogus, taip pat yra sukurti dirbtiniai intelektai kurie laimi prieš geriausius šachmatų ar pokerio žaidėjus, tačiau fiziniuose žaidimuose tai dar nėra pritaikyta, nors bandymų yra (žr. Susijusių darbų analizė).

Šiame darbe yra naudojama medžiaga iš mokslinio tiriamojo darbo (MTD), kuris buvo atliktas ta pačia tema. Panaudota medžiaga - skyriai:"Susijusių darbų analizė", "Naudoti įrankiai ir metodai", "Kompiuterinės regos algoritmai", "Sukurtas kompiuterinės regos algoritmas" bei dalis "Atlikti bandymai ir rezultatai".

Šio darbo tikslas yra sukurti kompiuterinės regos ir dirbtinio intelekto metodais paremtą stalo futbolo žaidimo modelį ir realizuoti jo prototipą. Taigi, pagrindiniai šiame darbe spręsti uždaviniai yra tokie :

1. Atlikti susijusių darbų analizę.
2. Nustatyti kompiuterinės regos metodus tinkamus stalo futbolo žaidėjų ir kamuoliuko atpažinimui.
3. Atlikti dirbtinio intelekto metodų analizę ir parinkti tinkamus taikyti stalo futbolo žaidime.
4. Sukurti žaidimo modelį remiantis kompiuterinės regos ir dirbtinio intelekto metodais, kuriame būtų apibrėžti pagrindiniai veiksniai - ar kamuoliukas yra pakankamai arti jog spirti jį, kamuoliuko judėjimo kryptį, greitį.
5. Parinkti tinkamus techninius parametrus ir sukurti fizinę stalo futbolo konstrukciją.
6. Vienai kompiuterio valdomai lazdai pritaikyti apmokymo algoritmą.
7. Eksperimentiškai išbandyti sukurtą žaidimo modelį naudojantis fiziniu stalu.

Šiame darbe buvo: apžvelgti susiję darbai, ir aptarti jų pasiekti rezultatai bei patirti sunkumai, atlikta analizė apie įrankių ir programinės kalbos pasirinkimą, atlikta analizė apie kompiuterinės regos algoritmų teoriją ir pristatytas mūsų sukurtas algoritmas. Taip pat buvo išnagrinėtos problemos, su kuriomis yra susiduriama naudojant atpažinimą pagal spalvas, bei kaip galima šias problemas spręsti, atlikta išsami dirbtinio intelekto metodų analizė, leidžianti išssirinkti geriausią metodą fiziniam stalo futbolo žaidimui, aprašyta fizinė stalo futbolo konstrukcija, aprašytas sukurtas žaidimo modelis pagal kurį atliekami pagrindiniai veiksmai, pademonstruoti pasiekti rezultatai ir numatyti ateities patobulinimai.

1. Susijusių darbų analizė

Kompiuteriniai žaidimai yra žaidžiami prieš dirbtinį intelektą, todėl noras robotizuoti fizinius žaidimus yra iškilęs jau seniai. Šiame skyriuje buvo apžvelgti tokie bandymai, ir aptarti iššūkiai, su kuriais susidūrė kūrėjai.

1.1. "Pinball" žaidimas

2011 m. Adam'as Metcalf'as parašė darbą [8] apie kompiuterinės regos ir dirbtinio intelekto panaudojimą "Pinball" žaidime."Pinball" - tai žaidimas ant plokščio stalo, kuriame yra įvairių kliūčių, o apačioje yra dvi rankenėlės, galinčios iššauti kamuoliuką į viršų. Žaidimo esmė yra surinkti kuo daugiau taškų (pataikius į sunkiau prieinamas vietas taškų skiriama daugiau) nepraradus kamuoliuko - kamuoliuką galima prarasti jeigu jis iškrenta pro rankenėles nespėjus sureguoti ir pakelti jų. Kūrėjo tikslas buvo automatizuoti "Pinball" žaidimą, todėl jis susidūrė su labai panašiomis problemomis, kurias bandysime spręsti mes - vaizdo apdorojimas kai kamuoliukas juda labai greitai, rankenėlių valdymas ir optimalaus sprendimo priėmimas. Išnagrinėkime tai detaliau.

Rankenėlių valdymui autorius panaudojo "University of Southern California" tyrėjų sukurtą valdiklį, kuris leidžia sujungti programinę įrangą su "Pinball" žaidimu. Tai ganėtinai specifiškai sukurtas įrenginys, kuris negali būti pritaikytas stalo futbolo žaidimui.

Vaizdo apdorojimui autorius pasirinko naudoti "OpenCV" [9] - atviro kodo kompiuterinės regos biblioteką. "Pinball" žaidime kamuoliukas gali judėti labai greitai, todėl vienas iš svarbiausių sprendimų - išsirinkti tinkamą kamerą darbui. Pirmiausia, autorius nusprendė naudoti "Point Gray FireFly 2" kamerą, kuri kadrus pateikdavo 640 x 480 rezoliucijos dydžiu iki 30 kadrų per sekundę. Atlikus skaičiavimus buvo nuspręsta, jog tokie parametrai yra nepakankami, kadangi per vieną kadro pokytį kamuoliukas sugebėdavo pakeisti poziciją per visą stalą. Antra naudota kamera - "Point Gray FFMV-03M2C". Ši kamera galėjo pateikti iki 122 kadrų per sekundę 320 x 240 dydžiu, o tai buvo daug geresnis rezultatas nei pirmosios kameros. Taip pat autorius pabrėžia, jog naudojantis šia kamera gauti kadrai buvo su daug mažiau "triukšmo". Norint optimizuoti vaizdo apdorojimą (ir panaikinti vėlavimą tarp kameros ir programinės įrangos) viena kompiuterio gija buvo paskirta vien tik kadrų gavimui iš kameros - tokiu būdu kompiuteris gali nelaukti kol pasibaigs skaičiavimai naudojant vieną kadrą, kad galėtų prašyti naujo. Kadrų apdorojimas pasirinktas toks - gautame kadre yra išvalomas fonas ir paliekami tik judantys pikseliai (pikseliai, kurie pakeitė poziciją nuo praėjusio kadro), tuomet šie pikseliai yra sugrupuojami į objektus, ir pagal jų dydį ir buvimo vietą yra nusprendžiama kur yra kamuoliukas. Kai jau yra žinomos kamuoliuko koordinatės, autorius paskaičiuoja greitį ir galimus veiksmus, bei atiduoda juos į sekantį programinės įrangos sluoksnį.

Autorius savo dirbtinį intelektą realizavo naudodamasis "Naujoko žaidėjo" logika - "Naujokas", tai žaidėjas kuris tiesiog bando išlaikyti kamuoliuką žaidime, negalvodamas apie tai, kokį taškų kiekį pavyks surinkti pamušus kamuoliuką į viršų. Šiai taktikai įgyvendinti, autorius tiesiog stebėjo zoną, esančią netoli rankenėlių, ir jeigu kamuoliukas patekdavo į šią zoną signalas pakelti rankenėlę buvo išsiųstas. Patobulinimas, kuris nebuvo įgyvendintas šiame darbe, tačiau trumpai aprašytas - žaidėjas stengiasi kamuoliuką pataikyti į zoną, už kurią gaus daugiausiai taškų. Tokia sąlyga įneša du papildomus uždavinius - reikia sugebėti apskaičiuoti trajektoriją kuria turi būti paleistas kamuoliukas ir reikia žinoti kuriose zonose bus surinkta daugiausiai taškų.

Iš žaidimo natūros matyti, jog "Pinball" yra paprastesnis žaidimas už stalo futbolą, kadangi yra gan daug statinių elementų, tačiau pati specifika - labai panaši. Iš šio darbo matyti, kad autorius pradžioje turėjo problemų dėl neteisingos kameros pasirinkimo, kas neleido tinkamai apdoroti žai-

dimo. Taip pat įdomus sprendimas buvo pasirinktas norint atpažinti kamuoliuką - buvo atrenkami beiskeičiantys pikseliai ir grupuojami. Toks sprendimas žinoma buvo priimtas dėl spalvų nevientisumo, tačiau parinkus išskirtinę spalvą kamuoliukui būtų galima jį atpažinti pagal spalvą naudojant HSV filtrą, kas būtų patikimiau, ir greičiau (bandant atpažinti spalvą iškyla kiti sunkumai, kurie aprašomi antrame skyriuje).

1.2. Autonominis stalo futbolas

2007 m. "Georgia Institute of Technology" keturių studentų grupė sukūrė autonominį stalo futbolo žaidimą [1]. Pagrindinis darbo tikslas buvo sukurti pigų prototipą, kurį būtų galima būtų parduoti tiek individualiems asmenims, tiek barams ar kavinėms. Autoriai aprašė kokius įrenginius naudojo, kiek jie kainavo ir kaip tai įtakojo galutinį rezultatą.

Architektūriškai, autoriai nusprendė, jog vaizdų atpažinimui naudos kamerą, kuri bus pakabinta virš stalo. Atsižvelgdami į kainos ir kokybės santykį, autoriai pasirinko kamerą "Phillips SPC900NC", kurios specifikacija teigė, jog ji gali filmuoti 90 kadrų per sekundę naudodama 1280 x 1024 pikselių rezoliuciją. Atlikus testus paaiškėjo, jog kamera filmuoja tik 30 kadrų per sekundę naudodama 320 x 240 pikselių rezoliuciją. Taip yra todėl, nes kameros tvarkyklė trigubina kiekvieną kadrą ir keturgubina pikselius rezoliucijoje - tai reiškia jog gaunama daugiau duomenų kiekio, tačiau duomenyse esančios informacijos kiekio tai nepadidina, o tai lemia jog tokie kadrai yra apdorojami ilgiau (ir po tris kartus tie patys). Žinoma, ši kamera buvo pasirinkta dėl vieno iš darbo tikslų - kainos, tačiau tai yra labai gera pamoka prieš pradedant darbą, jog išsirinkti teisingą kamerą yra labai svarbu.

Kadrų apdorojimui autoriai pasirinko naudoti Java programinę kalbą, tačiau savo darbe nepaminėjo, kokiomis bibliotekomis rėmėsi, tačiau, kadrų apdorojimas atliktas remiantis spalvų atpažinimu, todėl galima daryti prielaidą, jog buvo naudojama "OpenCV" biblioteka. Sukurtas algoritmas, pirmiausia atpažindavo stalo ribas (vartotojas turi pasirinkti spalvą, kuria yra aplink stalą nubrėžta linija formuojanti stačiakampį) - taip apibrėžiamos ribos kuriose gali būti kamuoliukas. Tuomet, vartotojas turi pasirinkti kurios spalvos žaidėjus kontroliuos (šis pasirinkimas labiau butaforinis, kadangi motorai kontroliuojantys žaidėjus yra sukonstruoti tik ant vienos pusės), ir galiausiai yra pasirenkama kamuoliuko spalva. Kai visi pasirinkimai baigti, algoritmas pradeda apdoroti ateinančius kadrus ir ieškoti kamuoliuko. Norint optimizuoti algoritmą, autoriai nusprendė pirmą paiešką kadre daryti 20 x 20 pikselių kvadrato plote, kurio centras - praeito kadro kamuoliuko pozicija - tokiu atveju reikia apdoroti mažiau informacijos, tačiau jeigu kamuoliukas šiame plote nėra randamas - jo bandoma ieškoti visame kadre.

Stalo futbolo žaidėjams valdyti autoriai panaudojo 8 vnt. servo varikliukų, iš kurių 4 skirti žaidėjus slankioti horizontaliai, o kiti 4 - pasukti žaidėją, jog šis galėtų smūgiuoti kamuoliuką. Perduoti signalą iš kompiuterio į servo varikliuką buvo naudojamas mikrovaldiklis, į kurį buvo siunčiami dviejų baitų paketai. Horizontaliam judėjimui naudoti servo varikliukai "AX-12", rotaciniam - "HS-81" - abu modeliai buvo pasirinkti dėl savo mažos kainos.

Dėl tokių sprendimų priėmimo, matomos kelios problemos. Pasirinkti pigūs įrenginiai neleido pasiekti užsibrėžtų tikslų: kameros kadrų per sekundę skaičius buvo planuotas 90, tačiau buvo - 30, kamuoliuko spyrimo greitis - 10 pėdų per sekundę, tačiau pasiektas tik 1.5! Dėl šių specifikacijų stalas negali būti naudojamas žaidime su žmogumi, nes yra tiesiog per lėtas. Taip pat, pasirinkta kadrus apdoroti naudojantis spalvų atpažinimu, tačiau darbe nebuvo paminėtas stalo apšvietimas, todėl galima daryti prielaidą, jog pasikeitus apšvietimui (debesuotą dieną) algoritmas tiesiog nustotų veikti, nes paskistų objektų atspalvis.

2. Naudoti įrankiai ir metodai

Šiame skyriuje buvo atlikta analizė apie galimą kompiuterinės regos bibliotekos ir programinės kalbos pasirinkimą.

2.1. Kompiuterinės regos bibliotekos pasirinkimas

Norint kurti kompiuterinės regos algoritmus pirmiausia reikia išsirinkti, kokią biblioteką, įrankį ar kalbą pritaikytą kompiuterinei regai naudoti būtų racionaliausia. Šiame darbe mes analizuosime gaunamų duomenų apdorojimą iš kameros realiu laiku, todėl atsirenkama buvo pagal tokius kriterijus - greitis, suderinamumas su programinėmis kalbomis ir kaina. Kompiuterinės regos bibliotekų yra sukurta nemažai: OpenCV, dlib, PCL (Point Cloud Library), SimpleCV ir kt. Plačiausiai naudojamos yra OpenCV, SimpleCV ir Matlab. Trumpai aprašysime jų privalumus ir trūkumus [11]:

1. Matlab - labai daug funkcijų turintis matematinis įrankis, kuriame rašomi skriptai vykdyti komandoms, kurie vėliau yra konvertuojami į Java programinę kalbą. Pagrindiniai šio įrankio minusai yra tie, jog jis yra pakankamai lėtas, reikalauja daug kompiuterio resursų ir yra mokamas.
2. OpenCV yra atvirojo kodo kompiuterinės regos ir dirbtinio intelekto programinės įrangos biblioteka parašyta C++ programine kalba, tačiau turinti programinės įrangos interfeisą beveik visoms pagrindinėms programavimo kalboms - C, C++, Python, Java, C# kitoms. Šioje bibliotekoje yra daugiau nei 2500 optimizuotų algoritmų, kurie yra naudojami įvairiems uždaviniams spręsti - nuo veidų atpažinimo iki judančių objektų sekimo.
3. SimpleCV - taip pat atvirojo kodo kompiuterinės regos biblioteka, kuri yra OpenCV bibliotekos apvalkalas. Pagrindinis SimpleCV bibiliotekos pliusas išplaukia iš to, jog tai yra aplankas kitai bibliotekai (OpenCV) ir jis yra supaprastintas, todėl net ir nežinant sintaksės arba kompiuterinės regos metodų implementacijos detalių galima ja naudotis. Šis pliusas yra ir pagrindinis minusas - ne visos kompiuterinės regos galimybės gali būti išnaudotos ir pasiektos, taip pat ši biblioteka yra lėtesnė nei OpenCV (nes reikalingas papildomas sluoksnis vykdant komandas).

Šiame darbe pasirinkome naudoti OpenCV kompiuterinės regos bibilioteką, nes ji atitinka mūsų keliamus reikalavimus - yra optimizuota (greitis), nemokama (kaina), gali būti naudojama su dauguma programavimo kalbų (suderinamumas).

2.2. Programinės kalbos pasirinkimas

Šiame darbe koncentruojamasi į veikiančio prototipo sukūrimą, kuriuo remiantis būtų galima tęsti darbą toliau ir jį tobulinti. Atlikus analizę, kuria programavimo kalba yra lengviausia (lengviausia - paprasta rašyti algoritmus nesigilinant į kalbos sintaksę) pradėti kurti kompiuterinės regos algoritmus, dažniausiai minima yra Python kalba, todėl šiame darbe naudosime būtent ją. Taip pat verta paminėti, jog svarbūs faktoriai kalbos pasirinkimui buvo ir šie: yra labai daug mokomosios medžiagos kaip programuojant Python kalba naudotis OpenCV bibilioteka, atlikti skaičiavimus galima naudoti kitas Python bibliotekas (Numpy, Scipy) kurios yra optimizuotos darbui su duomenimis.

Analizės metu paaiškėjo ir neigiama Python kalbos pasirinkimo pusė - sukurtas kompiuterinės regos algoritmas bus lėtesnis, nei leidžia OpenCV galimybės, o greitis yra labai svarbus faktorius

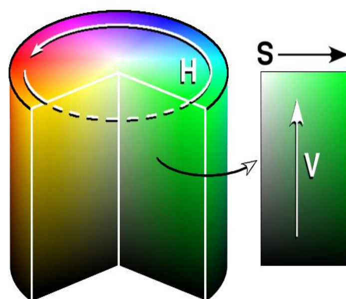
kuriant aplikaciją apdoroti duomenis realiu laiku. Norint sukurti labiausiai optimizuotą algoritmą, geriausia naudoti C++ programinę kalbą, nes būtent šia kalba yra parašyta OpenCV biblioteka, todėl išvengiama funkcijų ir metodų interpretavimo veiksmo. Atsižvelgus į šį faktą, sekančiame darbe bus perrašytas programinis kodas su C++ kalba, ir palyginami gaunami rezultatai - jeigu algoritmas iš tiesų veiks apčiuopiamai greičiau nei naudojantis Python programine kalba, tolimesni darbai bus atliekami naudojantis C++.

2.3. Spalvos modelio pasirinkimas

Dažniausiai naudojamas ir plačiausiai žinomas spalvos modelis yra RGB, kur R - raudona, G - žalia, B - mėlyna. Naudojantis šiuo spalvos modeliu kiekvieną kitą spalvą galime išreikšti adityviai - tai reiškia sudėdami raudoną, mėlyną ir žalią spalvas ir varijuodami jų kiekiu. Kiekis yra apibrėžiamas kiekvienai spalvai nuo 0 iki 255 ir išreiškiamas tripletu (RGB kodu), pavyzdžiui norėdami išreikšti geltoną spalvą galėtume užrašyti tokį RGB kodą - (255,255,0). Pagrindinis kompiuterinės regos ir mūsų darbo uždavinys yra atpažinti ir išskirti objektus erdvėje, patikimiausias būdas tai padaryti yra išskirti objektus pagal spalvas. Nors RGB spalvos modelis ir yra dažniausiai naudojamas, tačiau šiam uždaviniui labiau tinkamas spalvos modelis yra HSV, kur H - spalvos atspalvis, išreikštas laipsniais nuo 0 iki 360, S - sodrumas arba spalvos įsisotinimas, išreikštas procentais nuo 0 iki 100, V - spalvos šviesumas, išreikštas procentais nuo 0 iki 100 (žr. 1 pav.). Norint išreikšti tą pačią geltoną spalvą HSV spalvos kodu, galėtume užrašyti taip (60,100,100). Šis spalvos modelis yra pranašesnis už RGB dėl keletos priežasčių:

- HSV spalvos modelis yra intuityvus žmogui, kadangi spalva yra išreiškiama parinkus jos atspalvį, sodrumą ir šviesumą, tai yra būtent tai ką mūsų akis ir mato, kur RGB tą pačią spalvą išreikštų trijų spalvų sudėjimu, ką įsivaizduoti yra sudėtingiau;
- bandant atpažinti objektą kompiuterinėje regoje, dažnai susiduriama su atspalvių problema - prie skirtingo apšvietimo labai pakinta spalvos sodrumas (nors spalva išlieka ta pati). Šios problemos sprendimą labai palengvina HSV spalvos modelio natūra - žinant kokią spalvą norime išskirti (tai yra žinant H reikšmę), galime nurodyti spalvos pilkumo/ryškumo rėžius (S ir V reikšmes) - tokiu būdu sugebėsime išskirti pasirinktą spalvą net ir nežymiai pasikeitus apšvietimui.

Verta paminėti, jog naudojant OpenCV biblioteką HSV spalvos modelio rėžių reikšmės turi skirtingas reikšmes (pavyzdžiui H kinta nuo 0 iki 179), tačiau tai yra šios bibliotekos specifika ir šis faktas nedaro įtakos spalvos modelio veikimui.



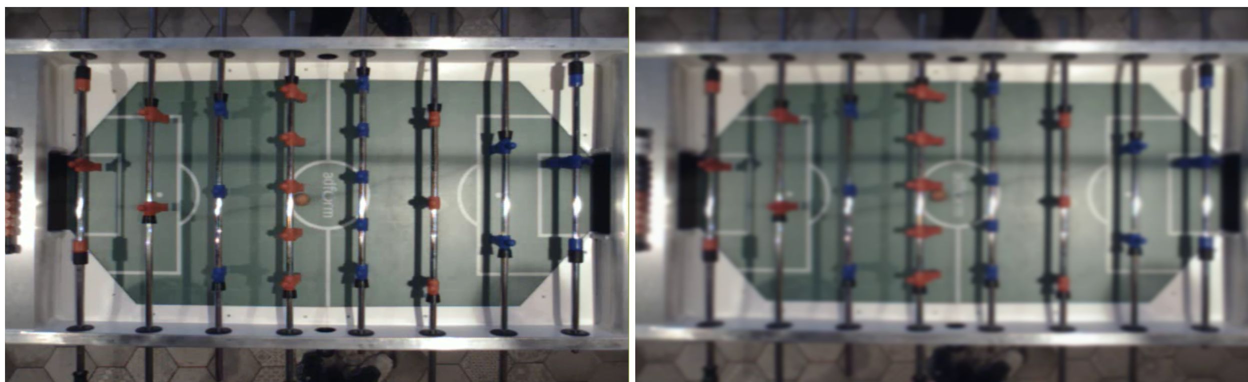
1 pav. HSV spalvos modelio vizualizacija [6]

3. Kompiuterinės regos algoritmai

Norint žaisti stalo futbolo žaidimą, reikia sugebėti matyti kamuoliuką, savo ir priešininko valdomus žaidėjus, todėl buvo aprašyti naudoti kompiuterinės regos metodai. Kompiuterinės regos algoritmuose yra naudojami matematinės morfologijos [**?**] metodai.

3.1. Paveikslėlio suliejimas

Prieš pradedant išskyrinėti objektus pagal spalvas, rekomenduojama atlikti paveikslėlio suliejimą (angl. blur). Tai yra daroma dėl keletos priežasčių: iš originalaus paveikslėlio (žr. 2a pav.) pašalina nereikalingą mikro triukšmą, paveikslėlyje sumažėja detalumo, dėl to tampa lengviau išskirti spalvas, taip pat sušvelnėja atspindžių (nuo apšvietimo) daroma įtaka (žr. 2b pav.).



(a) Originalus paveiksliukas

(b) Sulietas paveiksliukas

2 pav. Paveiksliuko suliejimo pavyzdys

Yra bent jau keletas metodų [2] kuriuos galima naudoti kompiuterinėje regoje norint sulieti paveikslėlį:

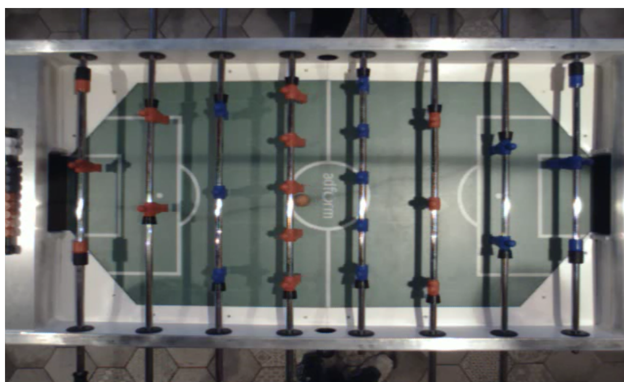
- pikselių reikšmių suliejimas vidurkinant reikšmes - tai yra pats primityviausias būdas, kurį galima apsirašyti pačiam programuotojui. Šis būdas gan gerai pašalina triukšmą iš paveikslėlio, tačiau yra prarandama nemažai reikšmingos informacijos;
- Gausinis suliejimas - daug geresnis suliejimo būdas nei pikselių vidurkinimas, kadangi triukšmas pašalinamas taip pat gerai, o naudingos informacijos beveik neprarandama (šiek tiek prarandama aplink kontūrus). Verta paminėti tai, jog Gausinis suliejimas yra atliekamas sprendžiant tiesines lygtis, todėl jis yra greičiausias iš trijų;
- medianinis suliejimas - gaunamų rezultatų atžvilgiu artimas Gausiniam suliejimo metodui, ir yra geresnis už jį tuo, jog nėra prarandama informacijos paveikslėlio kraštuose, tačiau, atliekant skaičiavimus tenka spręsti netiesines lygtis, todėl medianinis suliejimas yra lėtesnis nei Gausinis suliejimas.

Šiame darbe pasirinkome naudoti Gausinį suliejimo būdą dėl pagrindinės jo savybės - greičio. Kadangi kuriamas prototipas turės apdoroti duomenis realiu laiku, o informacijos praradimas aplink paveikslėlio kontūrus nėra esminis, todėl šis suliejimo metodas puikiai atitinka mūsų poreikius.

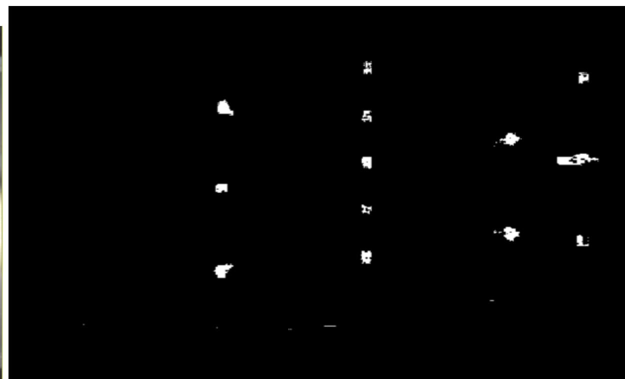
Suliejimo metodo naudojimas yra naudingas ne visais atvejais, pavyzdžiui jis gali netikti tuomet, jeigu sprendžiamas uždavinys yra surasti visas skirtingas spalvas paveikslėlyje, arba jeigu norima surasti labai mažas detales, kurios galbūt užima vos vieną pikselį - tokiu atveju pritaikius paveikslėlio suliejimą galimai bus prarasta dalis informacijos.

3.2. Filtravimas pagal spalvą ir konvertavimas į dvejetainį vaizdą

Filtravimas pagal spalvą yra esminis žingsnis norint sugebėti sekti išsirinkto objekto poziciją. Nors tai yra esminis žingsnis, tačiau jis visai paprastas naudojant kompiuterinės regos metodą "inRange" - išsirinkus objekto spalvos kodą pavyzdžiui HSV spalvos modelyje, užtenka į metodą paduoti norimą apdoroti paveikslėlį (žr. 3a pav.) ir spalvos rėžius (viršutinį ir apatinį, kad krentant šešėliui arba pasikeitus šviesos stiprumui visvien sugebėtume išskirti norimą objektą). Šis metodas grąžina paveikslėlį sukonvertuotą į dvejetainį pavidalą, kur pikseliai, kuriuose spalva sutapo su ieškoma yra žymimi 1, o kurie nesutapo - 0, tokiu būdu gauname juodai baltą paveikslėlį, kuriame baltas plotas yra mūsų išskirtas objektas pagal spalvą (žr. 3b pav.))



(a) Originalus paveiksliukas



(b) Dvejetainis paveiksliukas su išskirtais objektais pagal spalvą

3 pav. Originalaus ir filtruoto pagal pasirinktą spalvą ir konvertuoto į dvejetainį pavidalą paveiksliuko pavyzdys

3.3. Erozija

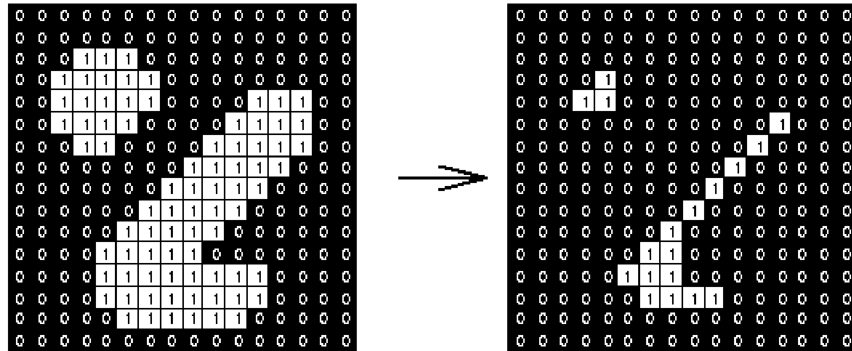
Erozija - vienas iš morfologijos metodų, naudojamas pašalinti triukšmą iš turimo paveikslėlio dvejetainiu formatu. Tai yra labai svarbus metodas, norint pasiekti aukšto tikslumo rezultatus išskiriant objektą pagal spalvą. Dažniausiai, pritaikius filtrą pagal spalvą, gautame dvejetainiame paveikslėlyje lieka atsitiktinių baltų pikselių - taip yra dėl atrinkimo pagal spalvas metodo natūros (klaidingai teigiamų rezultatų). Pritaikius erozijos metodą šią problemą galima lengvai pašalinti.

Erozijos metodas veikia taip: tarkime, jog turime originalų dvejetainį paveiksliuką (kuris iš tiesų yra tiesiog pikselių matrica, kurią pažymėkime A raide) iš kurio norime pašalinti triukšmą, tada turime sekančią matricą,kurios dydį parenkame patys (nuo to priklauso metodo veikimas ,kuo didesnį dydį parinksime, tuo stipresnį poveikį erozijos metodas turės originaliam paveikslėliui) ir literatūroje ši matrica vadinama struktūriniu elementu, arba trumpiau - branduoliu. Apsibrėžkime branduolio matricą kaip 3x3 dydžio ir pažymėkime ją B raide, ir pabandykime pritaikyti erozijos metodą, kuris gali būti apibrėžtas tokia formulę (formulė paimta iš [10] šaltinio):

$$A_{naujas}(x, y) = \min_{(x',y'):B(x',y')\neq 0} A(x + x', y + y') \tag{3.1}$$

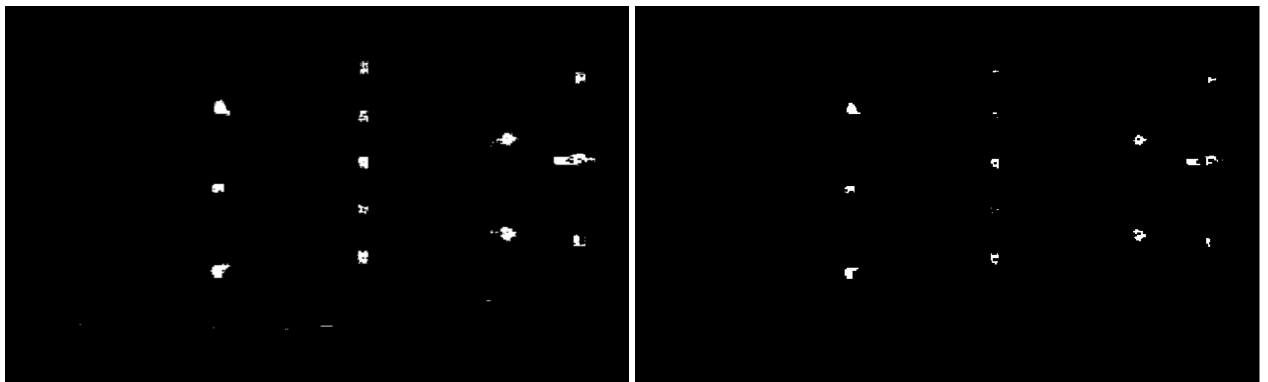
kur x,y - pikselio koordinatė paveikslėlyje, x',y' - branduolio matricos dydžiai. Šią formulę (3.1) galime paaiškinti kitais žodžiais: pažymėkime pereinamąją matricą raide C, o jos dydis bus toks pat kaip mūsų branduolio matricos - 3x3. Skaičiuojame naują matricos A nario A_{ij} reikšmę taip: iš matricos A "išimame" naują matricą C taip, jog narys A_{ij} būtų centrinis matricos C narys (jeigu

matrica 3x3 dydžio, tuomet A_{ij} turi būti pozicijuojamas kaip A_{11}). Kai jau turime matricą C, tikriname, ar visi šios matricos nariai yra lygūs 1, jeigu taip, tuomet naujasis narys tampa 1, jeigu ne - 0. Jeigu skaičiuojamas narys A_{ij} yra kampinis (tai yra neturi kaimynų iš kairės ar dešinės pusės), tuomet šio nario reikšmė tampa 0 (žr. 4 pav.)



4 pav. Erozijos metodo veikimo pavyzdys [5] su matricomis, kai branduolio matrica parinkta dydžio 3x3

Galime pažiūrėti, kaip erozijos metodas paveikia stalo futbolo žaidėjų filtruojamus žaidėjus. Galime matyti nedidelį baltų pikslelių plotą paveikslėlio apačioje, prieš pritaikant erozijos metodą (žr. 5a pav.), o pritaikius erozijos metodą šis baltas plotas dingsta (žr. 5b pav.). Tačiau, galime pastebėti, jog pastebimai sumažėjo ir norimų išskirti objektų balti ploteliai.



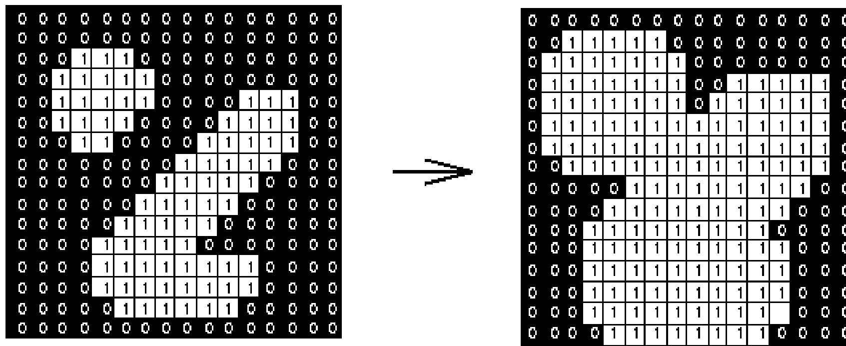
(a) Mėlynos spalvos filtro dvejetainis vaizdas prieš pritaikant erozijos metodą

(b) Mėlynos spalvos filtro dvejetainis vaizdas pritaikius erozijos metodą

5 pav. Erozijos metodo veikimo pavyzdys

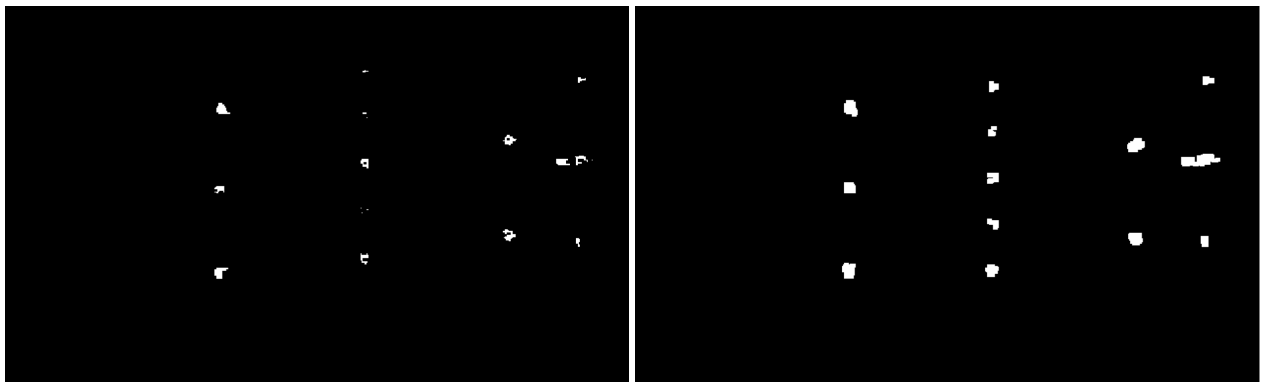
3.4. Plėstis

Erozijos metodo aprašymą pabaigėme tuo, jog iš pavyzdžio pamatėme, kad sumažėjo mūsų išskiriamų objektų balti ploteliai. Šioje vietoje galima (ir daugumoje literatūros šaltinių - siūloma) panaudoti kitą morfologijos metodą - plėstį. Pasiruošimas naujų matricos narių skaičiavimui plėsties metodu yra lygiai toks pat kaip erozijos metode, tačiau kai turime matricą C, pritaikome atvirkščią logiką - jeigu bent vienas matricos narys yra lygus 1, tai naujai skaičiuojamas matricos A narys prilyginamas 1, jeigu visi nariai lygūs nuliui, tuomet paliekama reikšmė 0. (žr. 6 pav.)



6 pav. Plėsties metodo veikimo pavyzdys [3] su matricomis, kai branduolio matrica parinkta dydžio 3x3.

Jeigu erozija išvalo triukšmą pašalindama nereikalingus baltus pikselius iš dvejetainio paveikslėlio (žr. 7a pav.), tai plėstis daro atvirškčią dalyką - praplečia likusių po erozijos baltų pikselių plotą, taip iš esmės atstatydama originalų sekamo objekto plotą (žr. 7b pav.).



(a) Mėlynos spalvos filtro dvejetainis vaizdas pritaikius erozijos metodą

(b) Mėlynos spalvos filtro dvejetainis vaizdas pritaikius plėsties metodą

7 pav. Plėsties metodo veikimo pavyzdys

Plėstis yra apibrėžiama tokia formule:

$$A_{naujas}(x,y) = \max_{(x',y'):B(x',y')\neq 0} A(x+x', y+y') \tag{3.2}$$

kur x,y - pikselio koordinatė paveikslėlyje, x',y' - branduolio matricos dydžiai.

4. Dirbtinio intelekto metodai ir algoritmai skirti mokymosi procesui

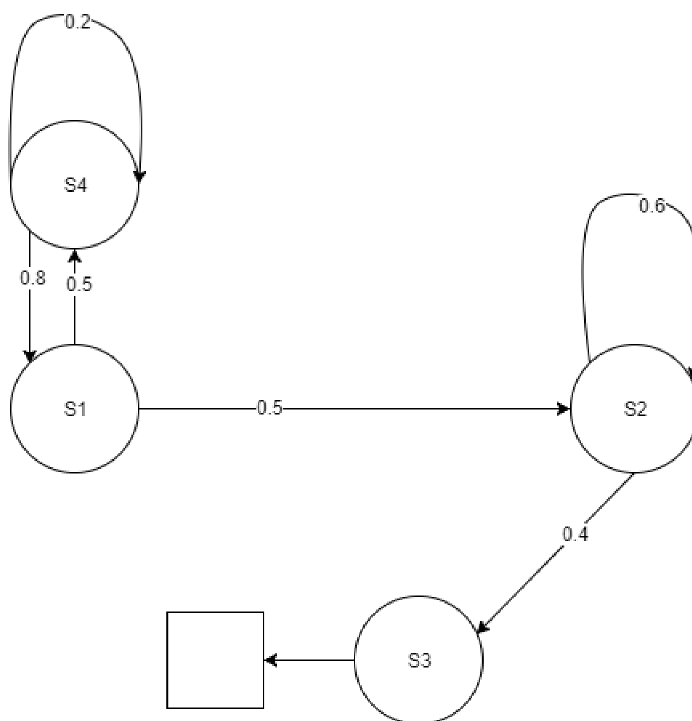
Šiame skyriuje buvo išnagrinėtos pagrindines dirbtinio intelekto rūšys ir pasirinkta mums priimtiniausia stalo futbolo žaidimo pritaikyme.

4.1. Markovo grandinės ir Markovo sprendimo priėmimo procesas

Dirbtinio intelekto sprendžiamas uždavinys yra sugebėjimas priimti sprendimus kaip reikia elgtis duotoje aplinkoje. Markovo sprendimo priėmimo procesas (angl. Markov Decision process - toliau MDP) būtent tai ir apibrėžia - duotoje aplinkoje, kurioje turime baigtinį žinomų būsenų skaičių galime atlikti veiksmus su tikimybe ir taip naviguoti turimoje aplinkoje. MDP formaliai apibrėžia aplinką mokymosi papildymu (angl. Reinforcement learning - toliau RL) metodui kurį pristatysime vėliau.

4.1.1. Markovo grandinė

Tam, kad galėtume kalbėti apie MDP reikia suprasti ir išnagrinėti kas yra Markovo grandinės [14].



8 pav. Markovo grandinės su keturiomis būsenomis ir tikimybėmis pereiti į jas pavyzdys.

Pažvelkime į 8 Pav. - jame matome pavaizduotą Markovo grandinę, kurioje yra keturios galimos būsenos $S1, S2, S3, S4$ ir kiekviena iš šių būsenų turi tikimybę su kuria pereina į sekančia būseną, arba lieka toje pačioje. Vienos būsenos tikimybės pereiti į kitą būseną privalo sumoje būti lygios 1. Taigi, galime paimti pavyzdį : sakykime jog esame būsenoje S1 ir yra 50 proc. tikimybė jog pereisime į būseną $S4$ arba $S2$. Jeigu atsidūrėme būsenoje $S2$ tai yra 60 proc. tikimybė jog joje ir pasiliksime, bet yra 40 proc. tikimybė jog pereisime į būseną $S3$ ir baigsime savo darbą

(stačiakampis mūsų schemoje žymi proceso pabaigą). Markovo grandinė yra apibrėžiama kaip atsitiktinis procesas t.y grandinėje esančių būsenų atsitiktinė seka, kuri tenkina Markovo sąlygą

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, ..., S_t], \text{kur } \mathbb{P} \text{ - tikimybių matrica} \tag{4.1}$$

Taigi, formaliai galime užrašyti, jog Markovo grandinė yra apibrėžiama šiais dydžiais $\langle S, P \rangle$, kur

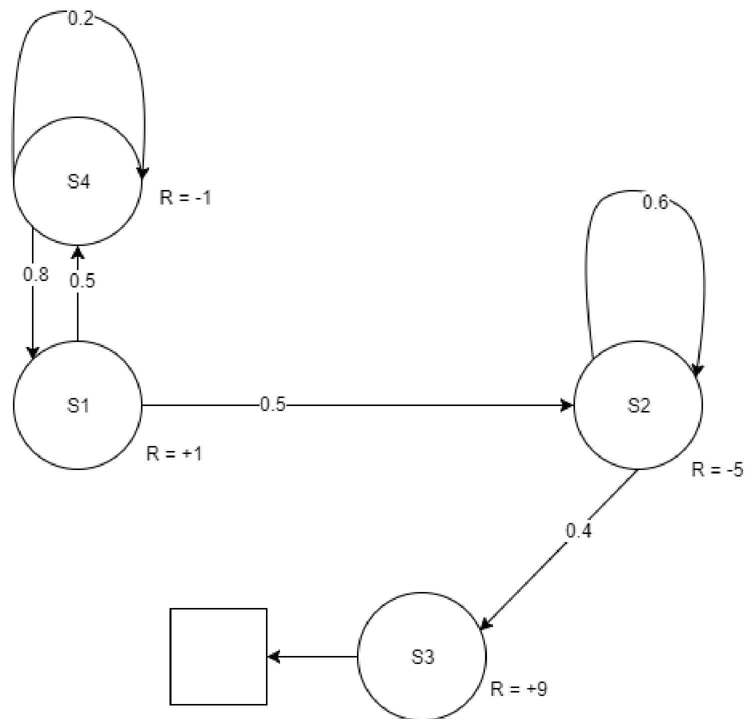
S yra baigtinė aibė galimų būsenų,

P yra perėjimo į kitas būsenas tikimybių matrica,

$P_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s]$, kur s' yra sekanti būsena, o t - laiko būsena duotuoju momentu.

4.1.2. Markovo procesas su atlygiu

Bet Markovo grandinė apibrėžia tiesiog atsitiktines sekas, todėl žinoti tik tai neužtenka norint išspręsti dirbtinio intelekto algoritmo sukūrimo uždavinį, todėl galime įsivesti dar vieną sąvoką - atlygis (angl. reward). Taigi, dabar galime apsibrėžti, jog perėjimas į kiekvieną iš mūsų sistemoje esančių būsenų turi tam tikrą atlygį (atlygis gali būti tiek teigiamas dydis, tiek neigiamas), ir atlygį žymėsime raide R - tokios sitemos pavyzdį galime matyti 9 Pav.



9 pav. Markovo proceso su atlygiu pavyzdys.

Žinant atlygius galime apsibrėžti mūsų sistemos grąžos funkciją (RL mokymosi tikslas - maksimizuoti šios funkcijos reikšmę) :

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \text{kur } \gamma \in [0; 1] \tag{4.2}$$

Šioje lygtyje γ žymi nuolaidą mūsų grąžos funkcijai. Jeigu parinksime γ artimą 0, tuomet mums labai svarbus atlygis, kurį gauname iš karto ir mums nesvarbu kad viso proceso metu toliau

gausime mažai taškų, jeigu γ parenkame artimą 1 - tuomet mes žiūrime į ilgalaikį atlygį gaunamą viso proceso metu. Sekanti labai svarbi funkcija yra vertės funkcija (angl. value function). Vertės-būsenos funkcija apibrėžia kiekvienos būsenos vertingumą ilgame laikotarpyje - tai reiškia, ji įvertina kokį atlygį mes gausime jeigu pradėsime tyrinėti sistemą nuo pasirinktos būsenos. Šią funkciją žymėsime:

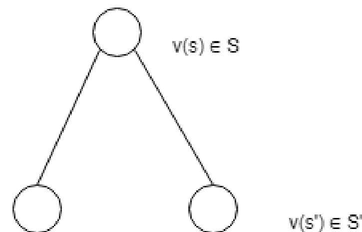
$$v(s) = \mathbb{E}[G_t|S_t = s], \text{ kur } \mathbb{E} \text{ - atsitiktinio dydžio vidurkis (dar vadinamas matematine viltimi).} \tag{4.3}$$

4.1.3. Bellmano lygtis

Bellmano (pavadinta mokslininko Ričardo Bellmano vardu) lygtis yra fundamentali norint suprasti ir teisingai spręsti RL keliamą uždavinį. Bellmano lygtis [7] teigia, jog vertės funkciją 4.3 galime išskaidyti į dvi dalis - iš karto gaunamą atlygį perėjus į būseną ir atlygį kurį gausime užbaigdami procesą nuo tos būsenos į kurią perėjome. Markovo procesui su atlygiu Bellmano lygtį galime apibrėžti taip:

$$\begin{aligned} v(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + ...)|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1})|S_t = s] \end{aligned}$$

Norint geriau suprasti Bellmano lygties naudą galime išnagrinėti realaus gyvenimo pavyzdį: įsivaizduokime jog stovime prie dviejų durų ir žinome, jog jeigu eisime pro pirmąsias duris gausime +2 taškus, jeigu eisime per antrąsias duris gausime +5 taškus. Tačiau šiuo metu mes nežinome koks kelias mūsų laukia atidarius duris, ir gali būti jog antrosios durys, kurios suteikia didesnį atlygį gaunamą iš karto mus veda į pražūtį. Bellmano lygtis mums leidžia atsidaryti abejas duris ir įvertinti koks atlygis mūsų laukia už jų ir tik tuomet priimti sprendimą, kurį kelią pasirinkti. Bellmano lygtis būsenos-vertės funkcijai gali būti pavaizduota štai tokiu grafu:



10 pav. Bellmano lygties Markovo atlygio procesui grafas. Juodi skrituliai žymi veiksmus, balti apskritimai - būsenas

Taigi, matome jog būsenos-vertės funkciją tikrai galime užrašyti tokiu pavidalu:

$$v(s) = R_s + \gamma \sum_{s' \in S} P_{ss'} v(s'), \text{ kur } s' \text{ yra sekanti būsena.}$$

4.1.4. Markovo sprendimo priėmimo procesas

Iki šiol apibrėžėme aplinką, kurioje turime būsenas, tikimybes pereiti į kitas būsenas ir atlygius, tačiau mes patys niekaip negalime įtakoti kur atsidursime t.y mes negalėjome priimti sprendimo kurią būseną norėtume pasirinkti. Šį kintamąjį įveda Markovo sprendimo priėmimo procesas (MDP), kuris iš esmės yra Markovo atlygio procesas su galimybe atlikti sprendimą, kokį veiksmą atlikti. MDP sistemą galime apibrėžti taip: tai yra procesas priklausantis nuo šių dydžių:

$\langle S, A, P, R, \gamma \rangle$, kur:

S yra baigtinė aibė galimų būsenų,

A yra baigtinė galimų veiksmų aibė,

P yra perėjimo į kitas būsenas tikimybių matrica,

$P^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$, kur s' yra sekanti būsena,

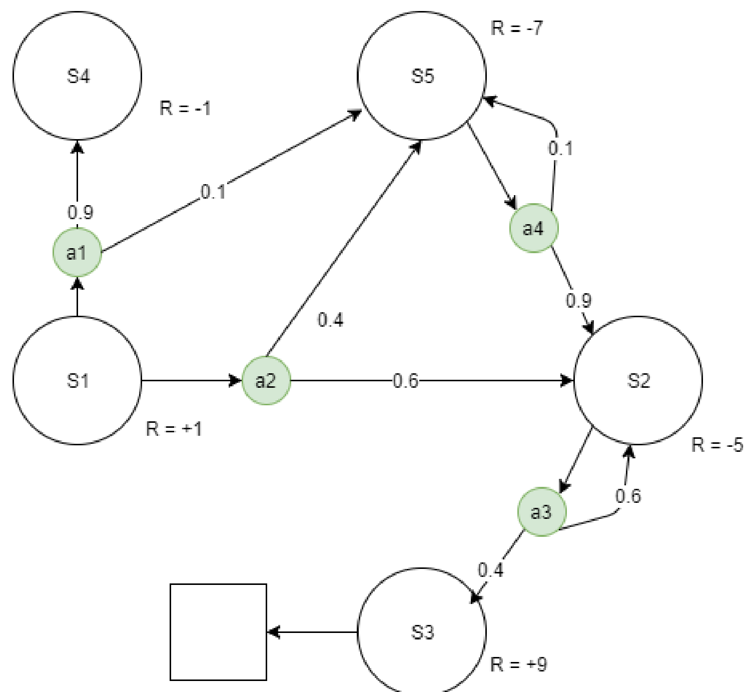
a - konkretus veiksmas iš visos veiksmų aibės A, o t - laiko būsena duotuoju momentu,

R yra atlygio funkcija,

$R^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$,

γ - nuolaidos faktorius, kur $\gamma \in [0; 1]$

Tokios sistemos pavyzdį galime matyti 11 pav. - turime būsenas, atlygius ir veiksmus (žali skrituliai), tokiu būdu perėjimo į kitą būseną tikimybė priklauso nuo veiksmo kurį pasirinksime.



11 pav. Markovo sprendimo priėmimo proceso (MDP) pavyzdys.

Taigi, turėdami tokią sistemą galime priiminėti sprendimus į kurias būsenas norime patekti. Pasiskirstymas tarp mūsų priimamų sprendimų/veiksmų yra perėjimo į tam tikrą būseną po jų yra vadinama - strategija, kurią žymėsime taip:

$$\pi(a|s) = \mathbb{P}[A_t = a | S_t = s] \tag{4.4}$$

Verta pabrėžti, jog turėdami tam tikrą strategiją mes galime pilnai apibrėžti mūsų programos, kuri veikia duotoje aplinkoje, elgseną. Taip pat, MDP strategijos atsižvelgia tik į ateities veiksmus, o tai kas buvo praeityje tampa nebesvarbu. Labai svarbu suprasti kodėl taip yra - pasirinkę tam tikrą strategiją mes norime pasiekti geriausią rezultatą (gauti daugiausiai įmanomos naudos iš mūsų sistemos) bet kuriuo duotu laiko momentu, todėl mums nėra svarbu tai kas vyko prieš tai, nes tai jau yra praeitis, ir tai neįtakoja mūsų esamo sprendimo, nes mums svarbu pasirinkti veiksmą, kuris mus perkels į būseną iš kurios gausime daugiausia naudos esamuoju laiko momentu. Dabar galime grįžti šiek tiek atgal, kur apibrėžėme būsenos-vertės funkciją 4.3 Markovo atlygio procese. Šioje lygtyje trūksta sprendimo priėmimo reikšmės, kuri yra pagrindinis faktorius mūsų procese. Pagrindinis dalykas kuris pasikeičia yra tai, jog mes nebeturime vienos būsenos-vertės funkcijos įverčio, nes įvertis priklauso nuo to, kokius veiksmus pasirinksime atlikti - nuo mūsų pasirinktos strategijos. Taigi, galime naujai apibrėžti būsenos-vertės funkciją, kuri priklauso ir nuo veiksmų taip:

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s] \tag{4.5}$$

Tačiau apibrėžti vien tik būsenos-reikšmės funkcijos mums neužtenka, kadangi dabar mes jau galime pasirinkti kokį veiksmą norime atlikti, todėl norint išsirinkti teisingą strategiją, mums reikia žinoti ir kokią galimą naudą duos pasirinktas veiksmas. Šią vertę mes vadinsime veiksmo-vertės funkcija, ir žymėsime taip:

$$q_\pi(s, a') = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{4.6}$$

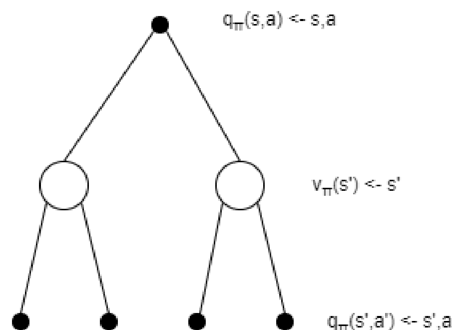
Šiai veiksmo-vertės funkcijai galime pritaikyti Bellmano lygtį ir išskaidyti į dvi dalis: kokią naudą gausime pasirinkę konkretų veiksmą iš karto ir kiek naudos gausime atsidūrę naujoje būsenoje po veiksmo iki proceso pabaigos. Tai galime užrašyti taip:

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_t, A_t)|S_t = s, A_t = a]$$

Ir analogiškai galime išskaidyti būsenos-vertės būseną su veiksmų priklausomybe:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$$

Šias išraiškas spręsdami savo MDP galime ir kombinuoti, tai reiškia jog galime spręsti Bellmano lygtį su dviem žingsniais - pažiūrėjimu į veiksmo naudą ir pažiūrėjimu į būsenos naudą. Tai galime pavaizduoti tokius grafu:



12 pav. Bellmano lygties grafas kai kombinuojame būsenos-vertės ir veiksmo-vertės funkcijas. Juodi skrituliai žymi veiksmus, balti apskritimai - būsenas

Šiame grafe matome, jog jeigu esame pirmoje žingsnyje, tai pagal pasirinktą strategiją su tam tikra tikimybe (tikimybė priklauso nuo strategijos pasirinkimo - apie tai plačiau pakalbėsime 4.1.5 poskyryje) galime atlikti veiksmą ir atsidurti vienoje iš dviejų galimų būsenų s'. Iš kiekvienos iš šių būsenų, yra galimi dar du veiksmai. Suskaičiavus būsenų ir veiksmų vertės funkcijos įverčių vidurkius pagal pasirinktą strategiją mes sužinome, kaip naudinga mums yra būti pirmajame žingsnyje esamuoju laiku. Visa tai gali būti užrašyta tokia formule:

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s',a')$$

Naudojant tokį skaičiavimą, galime rasti visas q_π reikšmes ir išspręsti savo turimą MDP. Tačiau iki dabar darėme prielaidą, jog turime kažkokią strategiją pagal kurią priimame veiksmus, bet kaip nuspręsti ir surasti geriausią strategiją turimoje sistemoje?

4.1.5. Optimalios strategijos pasirinkimas

Taigi, jeigu turime suskaičiavę savo MDP būsenos-vertės funkcijos reikšmes visoms būsenoms prie atsitiktinės strategijos, tai optimalią strategiją rasime jeigu pasirinksime maksimalią būsenos-reikšmės funkcijos įvertį kiekvienam laiko momentui:

$$v_*(s) = \max_\pi v_\pi(s),$$

Analogiškai galime rasti optimalią strategiją kokius veiksmus norime pasirinkti, naudodami šią formulę:

$$q_*(s,a) = \max_\pi q_\pi(s,a).$$

Jeigu mes žinome visus savo sistemoje egzistuojančius q_* - mūsų MDP yra išspręstas ir mes galime pasirinkti geriausią strategiją. Bet šios reikšmės vis dar neatsako į klausimą, kuri strategija yra geriausia. Norint išsirinkti geriausią strategiją reikia apsibrėžti kaip mes galime palyginti strategijas ir taip nuspręsti kuri yra geresnė. Taigi, apibrėžiame dalinį rikiavimą:

$$\pi \geq \pi' \text{ , jeigu } v_\pi(s) \geq v_{\pi'}(s) \text{ , } \forall s$$

Taigi, jeigu mūsų pasirinktos strategijos būsenos-reikšmės funkcijos įvertis yra didesnis arba lygus kiekvienoje būsenoje - galime teigti jog ši strategija yra geresnė (formaliai - ne blogesnė strategija, todėl ir vadiname daliniu rikiavimu. Kiekvienoje MDP sistemoje mes turime bent jau vieną tokią strategiją, kuri yra geresnė arba vienodai gera nei kitos). Norėdami iteratyviai suskaičiuoti geriausią strategiją, galime pasinaudoti šia paprasta išraiška:

$$\pi_*(s,a) = \begin{cases} 1 & \text{jeigu } a = \arg\max_{a \in A} q_*(s,a) \\ 0 & \text{kitais atvejais} \end{cases}$$

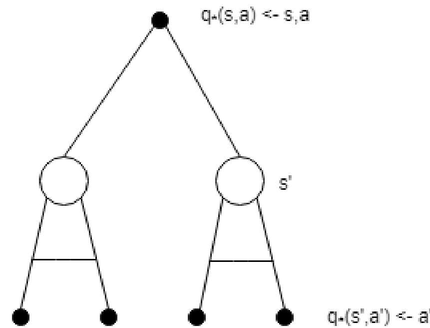
Galiausiai vėl galime pasinaudoti Bellmano lygtimi apskaičiuoti būsenos-vertės funkcijos reikšmę, tačiau dabar vietoj to, jog skaičiuotume visų galimų veiksmų atlygio vidurkį, mes tiesiog pasirenkame tą, kuris mums atneš daugiausiai naudos (atlygio) - optimalią būseną:

$$v_*(s) = \max_a q_*(s,a)$$

Tačiau jeigu norime paskaičiuoti optimalią veiksmo-reikšmės funkcijos reikšmę mes vistiek turime skaičiuoti galimų būsenų verčių vidurkį. Taip yra todėl, jog kai mes jau esame veiksme, mes negalime kontroliuoti kas įvyks po to - todėl jeigu turime tris skirtingas būsenas kuriose galime atsidurti mums reikia atsižvelgti į jas visas. Optimalaus veiksmo-vertės funkcija yra užrašoma taip:

$$q_*(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')$$

Norėdami suskaičiuoti patikimas q_* reikšmes, mes turime taikyti kombinuotą dviejų žingsnių Belmano lygties panaudojimą - pradedant nuo kažkurio veiksmo įvertiname kiekvienos galimos būsenos ir po jos sekančių veiksmų maksimumo vidurkius. Tai galime atvaizduoti žemiau esančiu grafu.



13 pav. Bellmano dviejų žingsnių optimalumo grafas veiksmo-vertės funkcijos reikšmei gauti. Juodi skrituliai žymi veiksmus, balti apskritimai - būsenas

Taigi, iš šio grafo ir matome - pirmiausia esame tam tikrame veiksme ir negalime pasirinkti kurioje būsenoje atsidursime, todėl turime skaičiuoti tų būsenų verčių vidurkį, tačiau pačios būsenos vertę mes galime suskaičuoti rinkdamiesi maksimalią sekančio veiksmo vertę (tai jog pasirinksime tik vieną reikšmę žymi sujungtos linijos grafe). Tokiu būdu gauname rekursyvią lygtį q atžvulgiu:

$$q_*(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q_*(s',a')$$

Tokią lygtį jau galime išspręsti. Norint pritaikyti šį metodą stalo futbolo žaidime, mums reikėtų žinoti visas galimas būsenos-vertės $v_*(s)$ ir veiksmo-vertės $q_*(s,a)$ reikšmes, o atsižvelgiant į tai, jog vien skirtingų būsenų gali būti $(640 * 480) * 60^8$ - tai labai sulėtintų mūsų programos veikimą, ir neleistų apdoroti vaizdo realiu laiku. Sekančiame poskyryje nagrinėsime, kaip galime pašalinti šią problemą.

4.2. Būsenos-reikšmės funkcijos įvertinimas

Pastarajame skyrelyje išnagrinėjome, jog jeigu norėtume tiesiog išspręsti MDP tai turėtume suskaičiuoti visas q_* reikšmes kiekienai būsenos ir veiksmo porai, tam kad galėtume išsirinkti geriausią strategiją π_*, bet kaip ir minėjome, stalo futbolo žaidime tai užtruktų labai ilgai ir paieška tokioje matricoje taptų labai ilgai trunkančia ir netinkančia aplikacijai veikiančiai realiu laiku. Vienas iš MDP sprendimo patobulinimų galėtų būti būsenos-vertės ir veiksmo-vertės funkcijų reikšmių apytikslis įvertinimas vietoje to jog skaičuotume tikrąsias vertes. Du pagrindiniai metodai naudojami šiems įvertinimams atlikti yra Monte-Karlo ir laikinojo skirtumo (angl. temporal difference learning, toliau - TD) mokymosi metodai. Išnagrinėsime juos abu ir nuspręsime, kuris metodas mums labiau tinka stalo futbolo žaidimo atveju.

4.2.1. Monte-Karlo mokymasis

Apibrėžkime vieną sąvoką : epizodas - tai baigtinė būsenų ir veiksmų seka mūsų turimoje MDP sistemoje. Jeigu turime MDP su trimis būsenomis ir dviem veiksmais, tai epizodo pavyzdys galėtų būti : $S_1, a_1, S_2, a_2, pabaiga$. Monte-Karlo mokymosi algoritmas pasižymi šiomis savybėmis:

- nagrinėjama MDP sistema privalo būti baigtinė, t.y turėti būseną kurioje procesas baigiasi, ir ta būsena privalo būti pasiekiama
- mokymosi algoritmas veikia tik su pilnais baigtiniais epizodais
- naudoja pačią paprasčiausią idėją vertės funkcijoms skaičiuoti - vertės funkcija yra lygi atlygio grąžos G_t funkcijos reikšmei
- nereikia žinoti MDP modelio veikimo savybių, kad galėtume naudoti šį metodą

Monte-Karlo mokymosi algoritmas skaidosi dar į du : pirmo apsilankymo ir kiekvieno apsilankymo mokymąsi. Mes koncentruosimės į pastarąjį, kadangi jis yra naudojamas plačiausiai. Kiekvieno apsilankymo metodas veikia taip:

- norėdami įvertinti būsenos-reikšmės funkcijos vertę, atliekame daug savo MDP epizodų, ir skaičiuojame kiek kartų apsilankėme savo stebimoje būsenoje. Skaitliuką galime pažymėti $N(S)$
- skaičiuojame grąžos funkciją nuo savo steibimos būsenos S kiekvieną kartą kai joje apsilankome ir sumuojame į kintamąjį $S(s) = S(s) + G_t$
- atlikus visus epizodus, skaičiuojame grąžos funkcijos vidurkį, ir jį priskiriame savo būsenos-vertės funkcijai $v(s) = \frac{S(s)}{N(S)}$
- kai apsilankymų skaičius stebimoje būsenoje yra pakankamai didelis, tai būsenos-vertės funkcija konverguoja į optimalią savo reikšmę. Kai $N(S) \to \infty$, tai $v(s) \to v_\pi(s)$

Taigi, jeigu naudosime Monte-Karlo mokymosi algoritmą ir savo turimoje MDP atliksime pakankamai daug epizodų net ir nežinodami kaip aplinka veikia, mes galime surasti ir įvertinti būsenos-reikšmės funkciją. Tačiau naudodami kiekvieno apsilankymo Monte-Karlo metodą, norėdami rasti savo įverčius turėtume pirma atlikti daug epizodų (kuriuos atliekant mes vis dar nesimokom, o tik kaupiam informaciją) ir tik tuomet galėtume atnaujinti savo vertės-funkcijos reikšmę, ir kartoti visą procesą su daug epizodų iš naujo kad gautume naują (tikėtinai tikslesnį) įvertį. Galima tai padaryti efektyviau naudojant kitą Monte-Karlo metodą - žingsninį mokymasi. Šis metodas leidžia atnaujinti būsenos-vertės funkcjos reikšmę po kiekvieno epizodo, todėl mokymasis tampa efektyvesnis, nes sekančiame epizode jau naudojame naujausią savo būsenos-vertės funkcijos įvertį. Tai galime išreikšti šia formule:

$$V(S_t) = V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t)) \tag{4.7}$$

Pažvelkime atidžiau į 4.7 funkciją : mūsų naujas būsenos-vertės funkcijos įvertis yra lygus prieš tai buvusiam įverčiui $V(S_t)$ pridėjus gautą paklaidą - $G_t - V(S_t)$, kur G_t yra mūsų atlikto epizodo grąžos funkcija, ir ši paklaida yra padalinama iš apsilankymo skaičiaus, norint išvengti didelių nuokrypių, ir taip pat nurodyti kaip imliai norime mokintis iš naujų epizodų. Bet šiuo metu imlumo koeficiento mes negalime kontroliuoti - kuo daugiau epizodų atliksime, tuo mažiau dėmesio kreipsime į gautą paklaidą. Jeigu mūsų MDP nėra stacionari (kaip stalo futbole), tai mums yra svarbu turėti galimybę nurodyti, norime mokintis imliai ar ne iš naujų epizodų, todėl 4.7 galime perrašyti taip:

$$V(S_t) = V(S_t) + \alpha(G_t - V(S_t)), \text{ kur } \alpha[0;1] \tag{4.8}$$

Šioje lygtyje α yra mokymosi imlumo koeficientas, kurį mes patys galime nustatyti pagal savo poreikius (jeigu α = 1 - labai imlus mokymasis, α = 0 - visiškai nebesimokome iš naujų epizodų). Monte-Karlo mokymosi algoritmas yra tinkamas apibrėžtoms nedidelėms MDP sistemoms, kuriose yra galimybė simuliuoti elgseną, ir epizodai yra sąlyginai trumpi. Jeigu turime didesnę sistemą tai šis metodas tampa per lėtai progresuojančiu (reikia atlikti be galo daug epizodų norint pamatyti naudą), todėl stalo futbolo projektui šis algoritmas nėra tinkamas.

4.2.2. Laikinojo skirtumo (TD) mokymasis

TD mokymosi algoritmas yra labai panašus į Monte-Karlo [15], tačiau jis turi pranašumą jog nereikia atlikti pilnų epizodų, kad galėtume pateikti savo būsenos-vertės funkcijos įvertį. Naudojant TD mes galime daryti prielaidas iš dalinių epizodų t.y atliekame kelis ėjimus savo MDP ir atnaujiname vertės funkcijos reikšmę naudodami spėjimą, kuo baigsis mūsų MDP. Tai yra pagrindinė TD mokymosi algoritmo idėja ir savybė. Užrašykime tai formule:

$$V(S_t) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1} - V(S_t)) \tag{4.9}$$

Taigi, šioje formulėje 4.9 mes atnaujiname savo būsenos-vertės funkciją naudodamiesi apytikslia sekančio žingsnio vertės funkcija $(R_{t+1} + V(S_{t+1}))$, kur $V(S_{t+1})$ yra mūsų kol kas turimas geriausias sekančios būsenos įvertis. Taigi, TD algoritmas veikia su tam tikra paklaida. Dabar galime iškelti klausimą - jeigu TD algoritmas veikia su tam tikra paklaida, o Monte-Karlo neturi paklaidos, tai kodėl geriau naudoti TD? Atsakymas:

- atliekant kiekvieną epizodą, yra pereinama per tam tikrą būsenų ir veiksmų seką. Jeigu iš veiksmas mus gali perkelti į keletą būsenų, tai reiškia, jog mes negalime to kontroliuoti - tai kontroliuojama aplinkos, todėl naudojantis Monte-Karlo metodu ir skaičiuojant įverčius iš pilno epizodo, tame įvertyje atsiranda daug daugiau triukšmo, ir pati būsenos-vertės funkcija turi daug didesnį pasiskirstymą. Tuo tarpu TD mokymosi algoritmą įtakoja tik vieno veiksmo triukšmas, todėl nors jis ir yra su paklaida, bet konverguoja į tikrąją optimalią būsenos-vertės funkciją daug greičiau
- iš pastarojo punkto akivaizdu, jog TD daug efektyvesnis metodas
- TD algoritmas visuomet mums suranda tikrąjį optimalų būsenos-reikšmės funkcijos įvertį $V(S) \rightarrow V_\pi(S)$

Kadangi TD yra efektyvus metodas skirtas naudoti ne stacionarioje aplinkoje, jis yra tinkamas stalo futbolo žaidimo MDP sprendimui.

4.3. Q-mokymasis

Prieš tai buvusiame skyrelyje, kalbėjome kaip naudojant TD mokymosi algoritmą įvertinti būsenos-reikšmės funkcijos reikšmes, bet tai reiškia, jog mes vis dar esame priklausomi nuo modelio, nes turime sužinoti kaip vyksta perėjimas tarp būsenų t.y turime žinoti perėjimo matricą $\mathbb{P}$, kad surasti strategiją:

$$\pi'(s) = \arg\max_{a \in A} R_s^a + P_{ss'}^a V(S')$$

Šią problemą galime išspręsti paprastai - vietoj to, jog geriausios strategijos ieškotume naudodami būsenos-vertės funkciją, galime naudoti optimalius veiksmo-vertės funkcijos įverčius $Q(s, a)$

[17]. Tokiu būdu tampame visiškai nepriklausomi nuo MDP modelio ir geriausią strategiją galime rinktis rinkdamiesi geriausią galimą veiksmą:

$$\pi'(s) = \arg\max_{a \in A} Q(s, a)$$

Savo eksperimentų pradžioje, galime priskirti pradines reikšmes $Q(s, a)$ matricai, ir su naujais patyrimais iš kiekvieno epizodo tikslinti Q reikšmes ir taip surasti optimalią strategiją. Bet atliekant tokį strategijos suradimą,galime susidurti su godumo problema: tarkime atliekame keletą epizodų per kuriuos randame tokias $Q(s, a)$ reikšmes, jog gauname teigiamą sistemos grąžą, todėl naujame epizode mes jau renkamės Q reikšmes kurios turi teigiamus įverčius ir siekiame vėl gauti teigiamą. Tai reiškia, jog mes galime neaplankyti visų galimų būsenų ir veiksmų porų, kurie galbūt mūsų sistemoje duotų geresnį rezultatą. Šią problemą sprendžia epsilon-godumo metodas, kuris teigia:

- pasirenkame tam tikrą ε reikšmę
- su tikimybe $1 - \varepsilon$ pasirenkame godų veiksmą - geriausią iš iki šiol suskaičiuotų Q reikšmių
- su tikimybe ε pasirenkame atsitiktinį veiksmą ne iš mūsų turimų Q reikšmių
- naudojant šią strategiją neišvengiamai bus aplankytos visos būsenų veiksmų poros

Naudodami šį metodą galime užtikrinti, kad mes visuomet tobulinsime savo pasirinktą strategiją (Q reikšmes), ir po pakankamo epizodų skaičiaus ji taps artima optimaliai. Svarbu paminėti, jog kai tik pradedame atlikinėti veiksmus savo MDP sistemoje, tai ε reikšmė turėtų būti pakankamai didelė (pvz. $\varepsilon = 0.5$), tam jog mūsų mokymasis būdų godus, ir aplankytume kaip įmanoma daugiau būsenų, tačiau ilgainiui ε reikia mažinti, nes jau esame apskaičiavę nemažai Q reikšmių ir galimai radę arti optimalią strategiją. Taigi, turėdami visą šią informaciją, galime užrašyti Q reikšmių skaičiavimo formulę:

$$Q(s, a) = Q(s, a) + \alpha(R + \gamma(Q(S', A') - Q(S, A))) \tag{4.10}$$

4.10 Q reikšmių skaičiavimo išraiška yra pagrindinė naudojama mokymosi su papildymu (RL) algoritmuose. Taigi, jeigu norėtume naudoti Q-mokymosi algoritmą stalo futbolo žaidime, mūsų pseudo algoritmas galėtų atrodyti taip:

1. Suteikiame pradines reikšmes $Q(s, a), \forall s \in S, \forall a \in A$ pasirinktinai, pavyzdžiui 0.
2. Atliekame veiksmą iš būsenos s (taikydami epsilon-godumo metodą, kad būtinai aplankytume visas būsenas).
3. Kartojame šį žingsnį tol, kol $Q(s, a)$ pradeda konverguoti
   - gauname atlygį atlikus veiksmą ir atsiduriame naujoje būsenoje s';
   - atnaujiname savo $Q(s, a)$ reikšmės įvertį su naujai gauta reikšme iš dalinio epizodo naudojant TD algoritmą.
4. Baigiame procesą.

Tokį algoritmą jau galėtume taikyti stalo futbolo žaidime, bet mums vistiek reikėtų laikyti atmintyje visas $Q(s, a)$ reikšmes, kurių šiame žaidime būtų labai daug ($(640 * 480) * 60^8$), todėl pats algoritmas norint atnaujinti Q reikšmes, ir surasti Q reikšmes visvien veiktų per lėtai realiu laiku bandant apdoroti kiekvieną kadrą.

4.4. Q-mokymasis su veiksmo-vertės funkcijos aproksimacija

Pažvelkime į stalo futbolo žaidimo specifiką iš arčiau: jeigu kamuoliukas yra ant stalo pozicijoje (110;90) ir (111;90) tai situacija labai panaši (galbūt net identiška, šias pozicijas gauname

atlikdami kompiuterinės regos algoritmus kurie gali turėti minimalių neatitikimų su tai kas yra realybėje). Taip pat, mūsų kompiuterio valdomi žaidėjai y ašimi gali slinkti lazdą tik kas 3px (taip yra dėl servo varikliukų tikslumo ir mūsų pasirinktos konstrukcijos). Taigi, būtų pravartu sugebėti generalizuoti panašias situacijas esančias stalo futbolo žaidime. Tai galime padaryti taikydami Q-mokymąsi su veiksmo-vertės funkcijos aproksimacija [13] [16], ir išreikšti Q reikšmes tiesine funkcija, kaip kombinaciją ypatybių priskirtų būsenų ir veiksmų poroms su svoriais. Būsenos veiksmo poroje atliekamą veiksmą vadinsime ypatybe (angl. feature) ir žymėsime raide f. Naudodami Q-mokymąsi su veiksmo-vertės funkcijos aproksimacija, Q reikšmes galime apibrėžti taip:

$$Q(s,a) = \omega_1 f_1(s,a) + \omega_2 f_2(s,a) + ... + \omega_n f_n(s,a) \text{ , kur } \omega \text{ - ypatybės svoris} \tag{4.11}$$

Tokiu atveju, mums nereikia žinoti visų Q reikšmių kombinacijų tarp veiksmų ir būsenų, o užtenka žinoti kiekvienos apsibrėžtos ypatybės svorį. Stalo futbolo žaidime ypatybės galėtų būti:

- spirti kamuoliuką į priekį, jeigu jo pozicija (būsena) yra prieš mūsų valdomą žaidėją;
- blokuoti kamuoliuką, jeigu kamuoliukas yra spiriamas varžovo;
- išlaikyti mūsų valdomus žaidėjus vienoje linijoje x ašyje su kamuoliuku

Pradines ω reikšmes galime pasirinkti laisvai, kadangi jos nuolat bus atnaujinamos pagal gautus rezultatus, todėl mes savo algoritme pradine ω reikšme laikysime 1.0 . Norėdami atnaujinti svorio reikšmę, turime atlikti veiksmą a, po kurio atsiduriame sekančioje būsenoje s' ir jau žinome kokį atlygį gavome būsenoje s pasirinkę veiksmą a. Tai galime užrašyti tokia formule:

$$\omega_n = \omega_n + \alpha f_n(s,a)((R(a,s') + \gamma \max_{a'} Q(s',a')) - Q(s,a)) \tag{4.12}$$

Šiame darbe pritaikysime šį metodą apmokinimo procesui ir patikrinsime, ar jį naudojant mūsų Q ir ω reikšmės pradeda konverguoti.

5. Sukurtas kompiuterinės regos algoritmas

Naudojantis kompiuterinės regos metodais, aprašytais 3 skyriuje, buvo sukurtas algoritmas įgyvendinantis vieną iš šio darbo tikslų, kuris yra pavaizduotas veiksmų diagrama (žr.A Priedą). Trumpai aprašysime ir paaiškinsime veiksmų diagramą:

- 1 žingsnis - gauti kadrą iš kameros. Jeigu nepavyko, bandyti gauti sekantį;
- 2-5 žingsniai - turimą kadrą pradžioje apdorojame 3.1 skyrelyje aprašytais metodais, tam kad turėtume nuo triukšmo išvalytą ir į dvejetainį paveikslėlį sukonvertuotą kadrą. Taip pat matome, jog antrajame žingsnyje apkarpome gautą kadrą, tam, kad turėtume tik stalo vaizdą (plačiau apie tai 9.5 poskyryje). Kadro filtravimas yra išskaidytas į tris dalis, kur kiekvienoje iš jų filtruojame pagal skirtingą spalvą, kad galėtume atskirti, kurie objektai yra priešininko žaidėjai, kurie kompiuterio valdomi žaidėjai ir kur yra kamuoliukas, todėl baigus filtravimą turime tris dvejetainius paveikslėlius, kuriuose yra išskirti atitinkamos spalvos objektai;
- 6 žingsnis - sugrupuojame rastų objektų pikselius į kontūrus - tokiu būdu sužinome kokį plotą užima mūsų sekamas objektas ir galime apskaičiuoti: žaidėjų kraštinių pozicijas (aplink žaidėjus brėžiame stačiakampius), kamuoliuko centro poziciją (aplink kamuoliuką brėžiame apskritimą);
- 7 - 8 žingsniai - kamuoliuko ir priešininko valdomų žaidėjų poziciją užfiksuojame ir paliekame galioti tokią iki sekančio kadro;
- 9 žingsnis - patikriname, ar aplink kompiuterio valdomą žaidėją yra kamuoliukas (tai darome skaičiuodami vienmačius atstumus tarp x ir y koordinačių), tam jog galėtume perduoti šiuos duomenis į 11 žingsnį;
- 10 žingsnis - užfiksuojame kompiuterio valdomų žaidėjų padėtį;
- 11 žingsnis - atliekame papildomus skaičiavimus ir naudojame sukurtą dirbtinio intelekto algoritmą nuspręsti, kokį veiksmą reikėtų priimti (tai plačiau aprašyta 6 skyriuje).

6. Sukurtas dirbtinio intelekto algoritmas

Vienas iš šio darbo uždavinių yra sukurti apmokymo algoritmą vienai kompiuterio valdomai lazdai, todėl šiame skyriuje aprašysime mūsų sukurtą dirbtinio intelekto algoritmą naudojantis Q-mokymu su veiksmo-vertės funkcijos aproksimacija. Sukurtą algoritmą atvaizduosime pseudo-algoritmu, esančiu žemiau ir kiekvieną jame naudojamą funkciją paaiškinsime plačiau.

1 algoritmas. Q-mokymasis su veiksmo-vertės funkcijos aproksimacija

Įvestis: $\varepsilon = 0.7$
Įvestis: $\omega = [1.0, 1.0]$
  while gauname naują kadrą **do**
    $\omega = atnaujintiSvorius()$
    if $random(0, 1) > \varepsilon$ **then**
      $veiksmas = atsitiktinisVeiksmas()$
    else
      $veiksmas = geriausiasVeiksmas()$
    end if
  end while

Kaip matome, šį algoritmą atliekame kiekvienam gautam kadrui iš kameros, ir kartojame tol, kol gauname duomenis. Kiekvienoje iteracijoje, norime naudotis naujausiomis reikšmėmis, todėl pirmasis veiksmas kurį atliekame savo algoritme - iškviečiame funkciją kuri atnaujina ypatybių svorius ω. Šioje funkcijoje suskaičiuojame visas Q reikšmes naudodamiesi 4.11 formule, išsirenkame geriausią Q reikšmę ir atnaujiname savo ypdatybės svorio reikšmes naudodami formulę 4.12. Toliau, taikome epsilon-godumo metodą, kuris yra aprašytas 4.3 poskyryje, tam jog užtikrintume, kad visuomet bandome ir naujas būsenas kuriose dar nesame buvę, o ne tik geriausias iki šiol aplankytas. Jeigu atliekame atsitiktinį veiksmą, tuomet tiesiog atsitiktinai iš visų galimų servo varikliuko pozicijų išsirenkame vieną, suskaičiuojame šios pozicijos Q reikšmę, ir atliekame šį atsitiktinį veiksmą. Jeigu norime atlikti geriausią veiksmą, turime atlikti daugiau veiksmų, todėl šią funkciją patogiau pavaizduoti pseudo-algoritmu esančiu žemiau.

2 algoritmas. Geriausio veiksmo apskaičiavimas

Įvestis: $maxQ = -100000$
  $geriausiosServoPozicijos = []$
  for all $servoLaipsnis = servoLaipsniai(43, 100)$ **do**
    $ypatybiuReiksmes = gautiYpatybiuReiksmes()$
    $qReiksme = ypatybiuReiksmes * \omega$
    if $qReiksme \geq maxQ$ **then**
      if $qReiksme > maxQ$ **then**
        $geriausiosServoPozicijos = []$
      end if
      $maxQ = qReiksme$
      $geriausiosServoPozicijos+ = servoLaipsnis$
    end if
  end for
  $geriausiasVeiksmas = random(geriausiosServoPozicijos)$

Šiame algoritme skaičiuojame Q reikšmes kiekvienai galimai servo pozicijai, ir išsirenkame geriausią, o jeigu yra kelios pozicijos su vienodomis Q reikšmėmis - atsirenkame visas tokias, ir apskaičiuodami geriausią veiksmą išsirenkame atsitiktinę iš geriausių. Taip darome tam, kad mokymasis būtų efektyvesnis - jeigu visuomet pasirinktumėme paskutinę servo poziciją su geriausia Q reikšme, tai neištyrinėtume kitų pozicijų (tyrinėtume kitas pozicijas tik atsitiktinio veiksmo metu), kurios galimai po daugiau atliktų veiksmų iš tikrųjų yra geresnės.

Paskutinė neišnagrinėta funkcija - $gautiYpatybiuSvorius()$. Šioje funkcijoje mes tikriname mūsų valdomų žaidėjų poziciją su kamuoliuko pozicija, judėjimo kryptimi ir greičiu, tam kad apskaičiuoti, koks veiksmas yra tikėtinas sekančiame kadre.

Algoritme naudoti parametrai, ypatybės ir gauti rezultatai yra aprašyti 9.5 poskyryje.

7. Stalo futbolo žaidimo modelis

Norint sugebėti apsibrėžti teisingą žaidimo modelį, reikia išanalizuoti kokios yra galimos būsenos mūsų žaidime, ir kokius veiksmus galime atlikti.

7.1. Kamuoliuko ir žaidėjų galimi veiksmai ir būsenos

Stalo futbolo žaidimas yra labai dinamiškas ir gali turėti labai daug būsenų, todėl šiame poskyryje nagrinėsime tik mūsų kuriamam algoritmui svarbias būsenas. Taigi, šiame žaidime turime kamuoliuką ir aštuonias lazdas prie kurių yra pritvirtinti žaidėjai. Kamuoliuko savybės yra tokios: jis gali judėti tam tikru greičiu, turi kryptį, kurią keičia atsitrenkęs į stalo futbolo sienelę arba pastumtas kurio nors iš žaidėjų. Taip pat galime apibrėžti, jog kamuoliukas gali turėti tokias būsenas:

1. Kamuoliukas yra kompiuterio valdomų žaidėjų zonoje:
   (a) kompiuterio valdomi žaidėjai gali pasiekti kamuoliuką ir jis yra iš dešinės;
   (b) kompiuterio valdomi žaidėjai gali pasiekti kamuoliuką ir jis yra iš kairės;
   (c) kompiuterio valdomi žaidėjai negali pasiekti kamuoliuko;
2. Kamuoliukas yra priešininko valdomų žaidėjų zonoje:
   (a) priešininko valdomi žaidėjai gali pasiekti kamuoliuką ir jis yra iš dešinės;
   (b) priešininko valdomi žaidėjai negali pasiekti kamuoliuko ir jis yra iš kairės;
3. Kamuoliuko nėra ant stalo.
4. Kamuoliukas yra vartuose (ši būsena yra skirtinga nuo 3, nes tai yra išskaičiuota būsena kai buvo pelnytas įvartis).

Šios būsenos mums bus svarbios vėliau, modeliuojant savo mokymosi algoritmo ypatybes, kadangi nuo šių būsenų priklausys kokie veiksmai turėtų būti pasirinkti (pvz. jeigu kamuoliukas yra kompiuterio valdomų žaidėjų zonoje, ir žaidėjai gali pasiekti kamuoliuką, yra labai svarbu žinoti iš kurios pusės kamuoliukas yra, nes vienu atveju norėsime atlikti smūgiavimą vartų link, o kitu atveju norėsime neužtverti kelio kamuoliukui jog jis galėtų atriedėti link kompiuterio valdomų žaidėjų).

Norėdami daryti įtaką kamuoliuko būsenoms, turime sugebėti valdyti stalo futbolo žaidėjus. Kaip ir minėjome yra aštuonios lazdos su žaidėjais, iš kurių keturios yra valdomos kompiuterio, likusios keturios valdomos priešininko. Kiekviena lazda gali suktis aplink savo ašį, tačiau žaidimo taisyklės apriboja, jog lazdą galime apsukti vieną kartą aplink savo ašį, kitaip bus pažeistos taisyklės ir kamuoliukas atiteks priešininkui. Mūsų modelyje, apsiribosime lazdos sukiojimu 180° laipsnių, kadangi to užtenka norint atlikti pagrindinius veiksmus. Taip pat lazda su žaidėjais gali būti stumdoma vertikaliai (y ašimi), tam jog sugebėtų blokuoti arba susitabdyti kamuoliuką. Taigi, galime išskirti galimas lazdų būsenas:

1. Pasirinktos žaidėjų ašys gali būti nuleisti statmenai stalui.
2. Pasirinktos žaidėjų ašys gali būti pakeltos (taip kad sudarytų 0° laipsnių kampą su stalo paviršiumi).
3. Pasirinktos žaidėjų ašys gali būti sukamos aplink savo ašį atliekant smūgiavimą į kamuoliuką.
4. Pasirinktos žaidėjų ašys gali būti slenkamos verikaliai.

Kai jau turime apsibrėžę galimas lazdų su žaidėjais būsenas, galime apibrėžti ir kokius veiksmus galime atlikti:

1. Kamuoliuko blokavimas, kai varžovas spiria (naudojant pirmą žaidėjų būseną).
2. Kamuoliuko sustabdymas ir blokavimas kai jis nėra prieš mūsų žaidėjus (naudojant ketvirtą žaidėjų būseną).

3. Kamuoliuko spyrimas (naudojant trečią žaidėjų būseną).
4. Kamuoliuko praleidimas riedėti tolyn (naudojant antrą žaidėjų būseną).

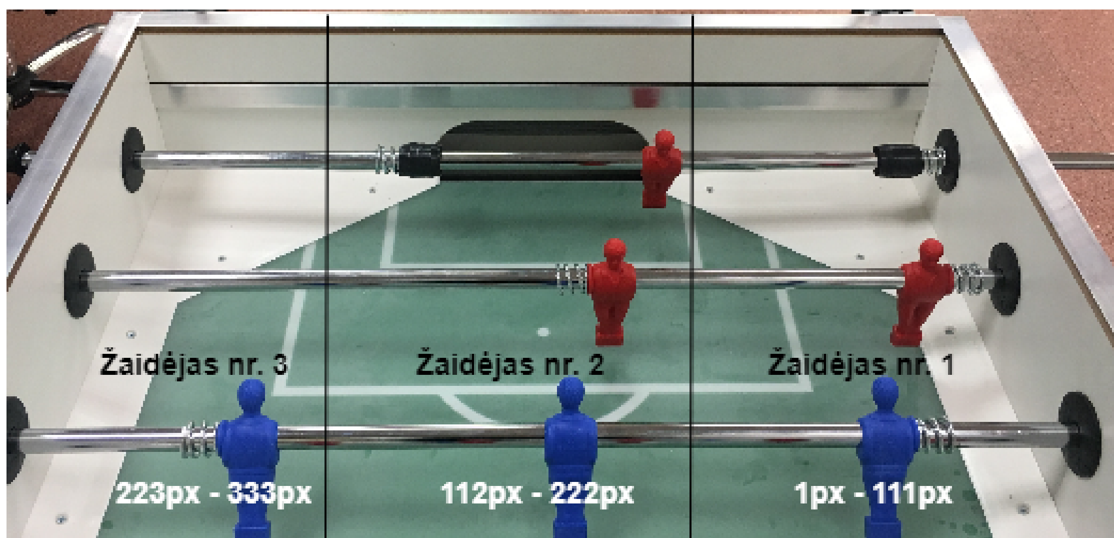
Taigi, kombinuojant aprašytas kamuoliuko būsenas, žaidėjų būsenas ir žaidėjų galimus atlikti veiksmus, galime savo programoje aprašyti algoritmą kuris apibrėš mūsų strategiją kiekvienu laiko momentu (apie tai plačiau skaityti 6 skyriuje).

7.2. Žaidėjų išlaikymas vienoje linijoje su kamuoliuku

Stalo futbolo žaidime labai svarbu mokėti spirti kamuoliuką, bet jeigu norėdami spirti kamuoliuką mes tik laukiame kol jis atriedės iki mūsų - tai yra labai neefektyvu, ir bet kuris žaidėjas sugebėtų nugalėti mūsų algoritmą. Norint žaisti prieš realius oponentus, reikia sugebėti sekti kamuoliuko poziciją su žaidėjais t.y. išlaikyti žaidėjus vienoje linijoje su kamuoliuku. Norint tai padaryti, reikia atkreipti dėmesį į keletą aspektų:

- kamuoliukas gali laisvai judėti y ašimi stalo paviršiumi;
- ant kiekvienos stalo futbolo lazdos yra daugiau nei vienas žaidėjas (išskyrus vartininko lazdas);
- lazdos yra stumdomos y ašimi naudojant servo varikliukus, kurie yra valdomi nurodant reikiamą pasukimo laipsnį.

Išnagrinėkime puolėjų lazdos (žr. 14 pav. mėlynus žaidėjus) valdymą. Ant puolėjų lazdos yra trys žaidėjai nutolę vienas nuo kito per 110px.



14 pav. Kompiuterio valdomi puolėjai ir zonos, išreikštos pikselių intervalais, kuriose tie žaidėjai gali pasiekti kamuoliuką

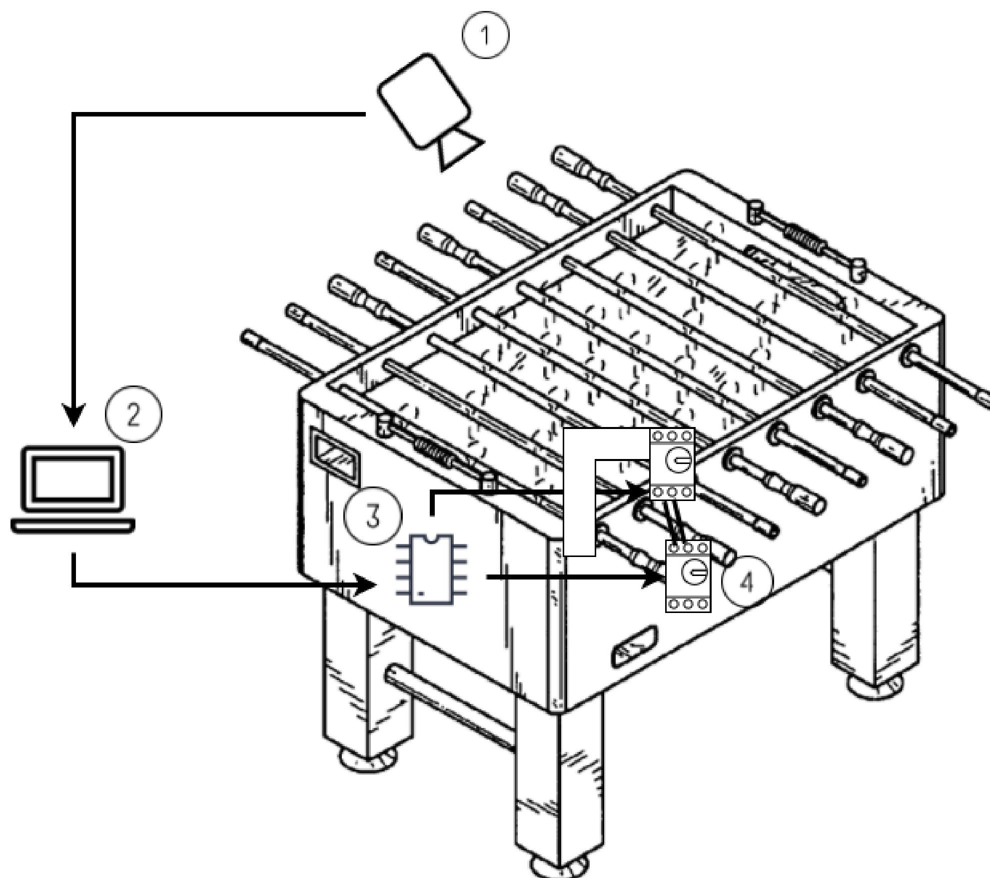
Taigi, norint išlaikyti žaidėjus vienoje linoje su kamuoliuku, reikia žinoti kokiu kampu reikia pasukti servo varikliuką kai kamuoliukas yra bet kurioje y ašies pozicijoje. Eksperimentiškai pamatavus, mūsų kamera mato kamuoliuko y pozicijas nuo 1px iki 333px. Norint nustumti žaidėjus, jog kraštinis žaidėjas būtų vienoje linijoje su kamuoliuku, kai kamuoliukas yra pozicijoje 1px y ašyje, servo varikliuką reikia pasukti 100° laipsnių, o jeigu kamuoliukas yra pozicijoje 333px - 43°. Tačiau, jeigu kamuoliukas yra pozicijoje 150px y ašyje, mes nežinome kokiu laispniu reikėtų pasukti servo varikliuką. Kadangi puolėjų lazdoje, yra trys žaidėjai, nutolę vienas nuo kito tokiu pačiu atstumu, todėl norėdami visose pozicijose išlaikyti žaidėjus vienoje linijoje su kamuoliuku, privalome paskirstyti y ašį į tris dalis:

1. Šioje dalyje naudosime žaidėją nr. 1 (žr. 14 pav.), kad išlaikyti vieną liniją su kamuoliuku. Šis žaidėjas y ašyje padengia pozicijas nuo 1px iki 111px. Servo varikliukas yra sukiojamas atitinkamai: jeigu kamuoliuko pozicija 1px, tai servo pasukamas $100°$ laipsnių, o jeigu kamuoliukas yra pozicijoje 111px - $43°$. Kai tik kamuoliuko pozicija tampa 112px, žengiame į antrą dalį.
2. Šioje dalyje naudosime žaidėją nr. 2 (žr. 14 pav.), kad išlaikytii vieną liniją su kamuoliuku. Šis žaidėjas y ašyje padengia pozicijas nuo 112px iki 222px.Servo varikliukas yra sukiojamas atitinkamai: jeigu kamuoliuko pozicija 112px, tai servo pasukamas $100°$ laipsnių, o jeigu kamuoliukas yra pozicijoje 222px - $43°$. Kai tik kamuoliuko pozicija tampa 223px, žengiame į trečią dalį.
3. Šioje dalyje naudosime žaidėją nr. 3 (žr. 14 pav.), kad išlaikyti vieną liniją su kamuoliuku. Šis žaidėjas y ašyje padengia pozicijas nuo 223px iki 333px.Servo varikliukas yra sukiojamas atitinkamai: jeigu kamuoliuko pozicija 223px, tai servo pasukamas $100°$ laipsnių, o jeigu kamuoliukas yra pozicijoje 333px - $43°$.

Pritaikius tokį algoritmą, galime žaidėjus išlaikyti vienoje linijoje su kamuoliuku. Atliekant bandymus, susidūrėme su viena problema - jeigu kamuoliuko pozicija kinta tokia seka y ašyje: 111px, 112px, 111px, 112px... Tuomet servo varikliukai turi būti persukami nuo $43°$ laipsnių iki $100°$ laipsnių ir atvirkščiai. Jeigu servo varikliukai veikia pilnu greičiu, esant tokiai situacijai labai greitai sudyla dalis, kuri yra pritvirtinta prie servo varikliuko ir lazdos kurią jis stumdo - jeigu šį dalis sudyla, servo varikliukas prasisuka ir lazda stovi vietoje. Šiame darbe pritaikėme paprastą sprendimą - Python programoje suskaičiuojame kokį pokytį turės padaryti servo varikliukas nuo prieš tai buvusio kampo, ir jeigu tas pokytis yra didesnis nei $55°$ laipsniai, tuomet į mikrokontrolerį yra siunčiamas įvykdymo laiko parametras 12 milisekundžių. Šis parametras nurodo, per kiek laiko servo varikliukas turi atlikti nurodytą veiksmą - tokiu būdu yra reguliuojamas servo varikliukų greitis. Visais kitais atvejais, į servo varikliuką paduodame įvykdymo laiką 1 milisekundę - tai reiškia jog norime įvykdyti paduotą komandą maksimaliu greičiu.

8. Stalo futbolo konstrukcija

Šiame skyriuje aprašysime fizinę stalo futbolo sistemos konstrukciją.



15 pav. Stalo futbolo konstrukcijos modelis. Stalo kontūras [4]

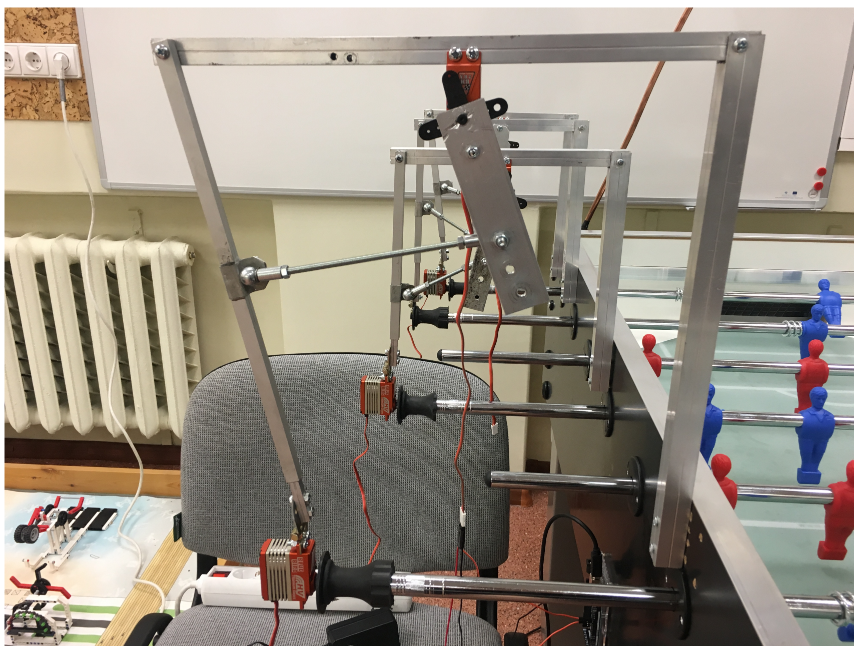
Paveikslėlyje galime matyti pagrindinius keturis elementus, iš kurių buvo sudaryta mūsų fizinio stalo konstrukcija (pats stalas nėra įtraukiamas į elementų sąrašą) :

1. Virš stalo pritvirtinta kamera, kuri yra pakabinta ant dviejų lankų besiremenčių į stalo konstrukciją. Kamera siunčia vaizdus į kompiuterį per USB jungtį.
2. Kompiuteris, kuriame veikia mūsų sukurta programa priimanti vaizdus iš kameros. Programoje yra apdorojamas kiekvienas iš vaizdo kameros gautas kadras, atliekami skaičiavimai mūsų sukurtam žaidimo modeliui ir naudojantis nuosekliąja jungtimi (angl. Serial Port) siunčiamas signalas į mikrokontrolerį.
3. Mikrokontroleris - šiame projekte pasirinkome naudoti "Arduino Mega 2560". Į mikrokontrolerį yra įkelta programa, kuri priima baitų paketą [12], kuriame yra nurodyta kokiu laipsniu ir greičiu pasukti kiekvieną servo varikliuką ir tuomet siunčia signalą į varikliuką.
4. Servo varikliukai - šiam projektui pasirinkta naudoti BMS-28A modeliai. Paveikslėlyje (žr. 15 pav.) matome, jog vienai lazdai kontroliuoti reikalingi du servo varikliukai - vienas atsakingas už rotacinį judėjimą, kitas už šoninį. Už šoninį judėjimą atsakingas varikliukas yra pakabintas ant metalinės konstrukcijos, kuri yra sumontuota ant stalo ir tiesiogiai sujungtas su varikliuku atsakingu už rotacinį judėjimą.

B priede galima matyti, kaip ši sistema atodo realybėje.

Taip pat 15 paveikslėlyje galime matyti jog servo varikliukai prie stalo pritvirtinti naudojant mūsų sukurtą konstrukciją. Kuriant šią konstrukciją, buvo susidurta su keletu problemų - pati

konstrukcija labai nestabili, servo varikliukui nepakanka galios pastumti lazdą dėl konstrukcijos trinties, konstrukcija turi būti keičianti ilgį. Kiekviena iš šių problemų buvo išspręsta mechaniškai, pritaikant fizikos žinias, tačiau šiame darbe apie tai nesiplėsime. 16 pav. galime matyti galutinę servo varikliukus laikančios konstrukcijos versiją.



16 pav. Servo varikliukus laikanti ir lazdas slankiojanti bei sukiojanti konstrukcijos dalis

9. Atlikti bandymai ir rezultatai

Šiame skyriuje aprašysime atliktus bandymus ir gautus rezultatus. Taip pat aptarsime, kokią kamerą pasirinkome ir su kokiais iššūkiais buvo susidurta.

9.1. Kameros pasirinkimas

Dirbant su kompiuterinės regos algoritmais svarbiausios naudojamos kameros charakteristikos yra filmuojamos rezoliucijos dydis ir kadrų per sekundę (toliau - KPS) kiekis. Atliekant bandymus šiame darbe buvo išbandytos dvi kameros - telefono "Iphone 6s" ir "Sony PlayStation 3 Eye". Telefono kamera buvo naudojama tyrimo pradžioje ir pasižymėjo šiomis savybėmis:

- sugebėjo filmuoti iki 240 KPS su 1280x720 rezoliucija - parinkus tokius nustatymus video kokybė buvo labai gera, tačiau apdoroti tokį video yra labai daug kompiuterio resursų reikalaujantis darbas, todėl ši charakteristika tapo minusu;
- nedidelės konfigūravimo galimybės. "Prasčiausios" kokybės video galima pasirinkti 30 KPS su 1280x720 rezoliucija.

Dėl šių priežasčių, "Iphone 6s" kamera negalėjo būti pritaikyta šiame darbe.

Kita išbandyta kamera - "Sony Playstation 3 Eye" kamera, kuri pasižymėjo tokiomis savybėmis:

- gali filmuoti iki 120 KPS su 640x480 rezoliucija;
- galima lengvai keisti kadrų per sekundę skaičių ir rezoliucijos dydį;
- video yra sąlyginai prastos kokybės, todėl užima nedaug vietos ir nereikalauja daug kompiuterio resursų norint juos apdoroti;
- žema kaina.

Šiame darbe naudoti pasirinkome pastarąją kamerą, nes ji visiškai atitiko mūsų keltus reikalavimus.

9.2. HSV spalvos modelio reikšmių nustatymas

Bandymams naudotas standartinis stalo futbolo stalas, tai reiškia jog žaidėjų spalvos - raudona ir mėlyna, kamuoliuko spalva - oranžinė. Žinant spalvas, kurias norėsime išskirti iš kadro, sekantis žingsnis yra parinkti jų reikšmių rėžius HSV spalvos modelyje. Spalvų rėžių parinkimui buvo panaudota Python kalbos biblioteka "imutils" ir joje esantis skriptas "range-detector.py" - šis skriptas leidžia įkelti paveiksliuką ir keičiant atspalvį, sodrumą ir šviesumą stebėti kaip tai įtakoja originalų paveiksliuką, kai pasiekiamas tenkinantis rezultatas, HSV reikšmės užfiksuojamos. Įdomu tai, kad konkrečiai gautų reikšmių mes nenaudojame, o prie kiekvienos iš jų pridedame (taip gauname viršutinį spalvos rėžį) ir atimame (taip gauname apatinį spalvos rėžį) tam tikrą konstantą Δ. Dažniausiai literatūroje rekomenduojama $\Delta = 10$. Atlikus bandymus, savo kuriamam programos prototipui pasirinkome naudoti šias spalvos modelio reikšmes:

1 lentelė. HSV spalvų parinkimas. Raidės po spalvomis: v - viršutinis rėžis, a - apatinis rėžis

Spalva	H_v	H_a	S_v	S_a	V_v	V_a
Mėlyna	130	110	255	50	255	50
Oranžinė	17	9	255	80	255	86
Raudona	7	0	255	70	255	50
Raudona	179	170	255	70	255	50

Verta paminėti keletą dalykų:

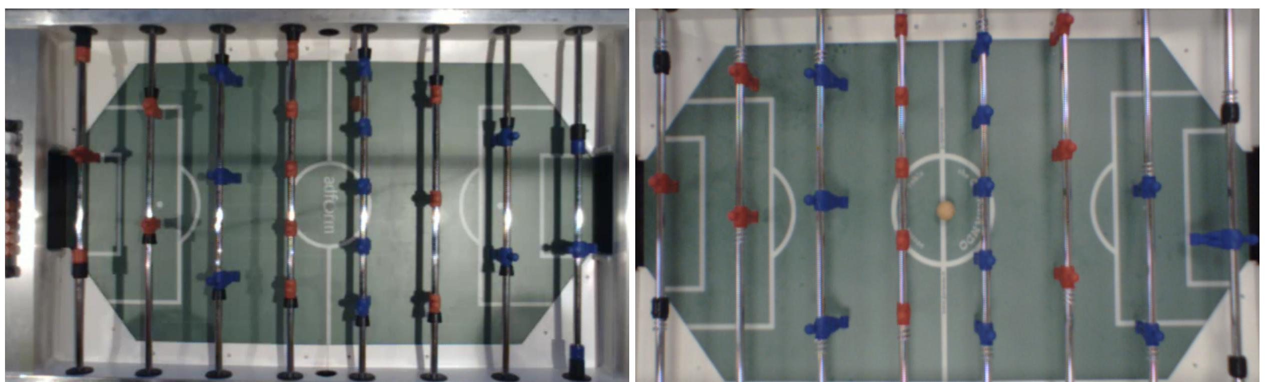
- konstanta Δ negali būti nustatoma vienareikšmiškai, nes yra spalvų, kurios turi ilgesnį atspalvio rėžį (pavyzdžiui mėlynos spalvos - 20);
- raudonos spalvos išskyrimui reikia kombinuoti du rėžius. Tai paaiškinti galima taip - HSV spalvos skalė yra atvaizduojame kaip cilindras (žr. 1 pav.), o cilindro pagrindas - apskritimas. Raudonos spalvos H reikšmė šiame apskritime yra susitelkusi aplink 360°ribą, todėl reikia tikrinti ne tik aplink 360°ribą, bet ir aplink 0°ribą (OpenCV biblioteka H reikšmes apibrėžia nuo 0 iki 179 atitinkamai).

9.3. Apšvietimas

Atliekant bandymus buvo pastebėta, jog pakitus stalo apšvietimui sekti žaidėjus ir kamuoliuką tampa neįmanoma, kol neparenkamos naujos HSV spalvos modelio reikšmės. Tai yra viena iš pagrindinių problemų bandant sekti judančius objektus pagal spalvas, nes pasikeitus apšvietimui, H reikšmė pakinta kardinaliai (netelpa į nustatytą rėžį). Šią problemą galima spręsti dvejopai:

- pritaikant statinį apšvietimą stalui, kad nekistų apšvietimas ir tuo pačiu spalvos rėžiai;
- pritaikant automatinį kalibravimą prieš kiekvieną žaidimą.

Šiame darbe naudojome pirmąjį problemos sprendimo būdą, tačiau buvo pastebėta dar viena su apšvietimu susijusi problema - šešėliai. Jeigu buvo naudojamas apšvietimas iš stalo viršaus, tuomet žaidėjus laikantys laikikliai sukuria juostos formos šešėlį (žr. 17a pav.) - kai žaidėjas atsiduria šioje juostoje, jį susekti tampa neįmanoma, kadangi jis įgyja labai tamsų atspalvį.



(a) Kamuoliukas praranda savo spalvą patekęs į šešėlį (b) Stalo vaizdas pritaikius apšvietimą iš šonų

17 pav. Stalo vaizdas su šešėliais ir be jų

Šešėlių problemą šiame darbe išsprendėme pakeisdami stalo poziciją šviestuvų atžvilgiu - stalas buvo pastatytas tarp dviejų šviestuvų, todėl šviesa krito ne tiesiai stalo link, o iš šonų, tokiu

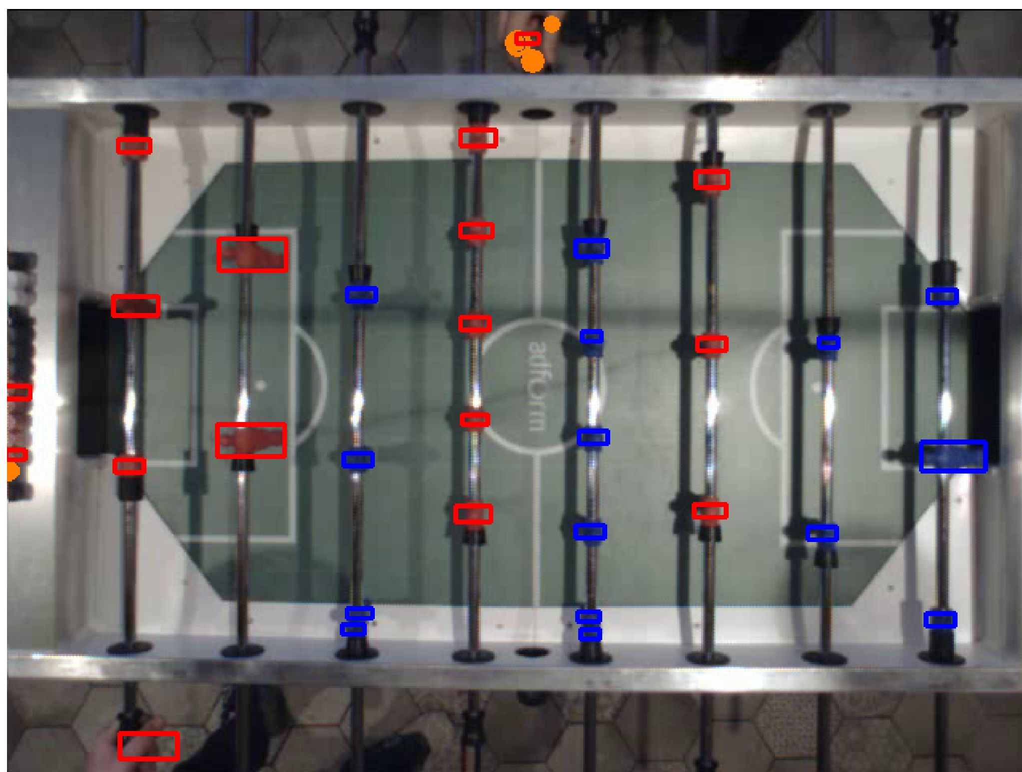
būdu jeigu šešėliai ir susidaro, tai labai neryškūs ir neįtakojantys mūsų algoritmo veikimo (žr. 17b pav.).

9.4. Sukurto kompiuterinės regos algoritmo rezultatai

Atliekant kompiuterinės regos algoritmo (aprašyto 6 skyriuje) bandymus, buvo dirbama su nufilmuotais žaidimo video (tam, kad nereikėtų nuolat būti prie stalo). Išbandžius keletą variantų, buvo pasirinkta dirbti su 640x480 rezoliucijos video, pateiktais 75 KPS. Norint patikrinti, ar sukurtas algoritmas veikia teisingai, reikia įsivesti tam tikras metrikas. Šiame darbe vienas iš pagrindinių iškeltų uždavinių - surasti ir atskirti kompiuterio valdomus, priešininko valdomus žaidėjus ir kamuoliuką, todėl nusprendėme skaičiuoti kadrus, kuriuose buvo rastas kamuoliukas ir žaidėjai ir šį skaičių lyginti su visu kadrų skaičiumi. Tokiu būdu, turėsime ieškomų objektų radimo procentinę išraišką ir teigsime, jog jeigu ši procentinė išraiška yra apie 90% - gautas rezultatas patenkinamas.

Prieš pradedant nagrinėti gautus rezultatus, pagrįsime, kodėl reikia apkarpyti kadrą, kaip nurodyta mūsų algoritmo antrame žingsnyje (žr. **??** pav.):

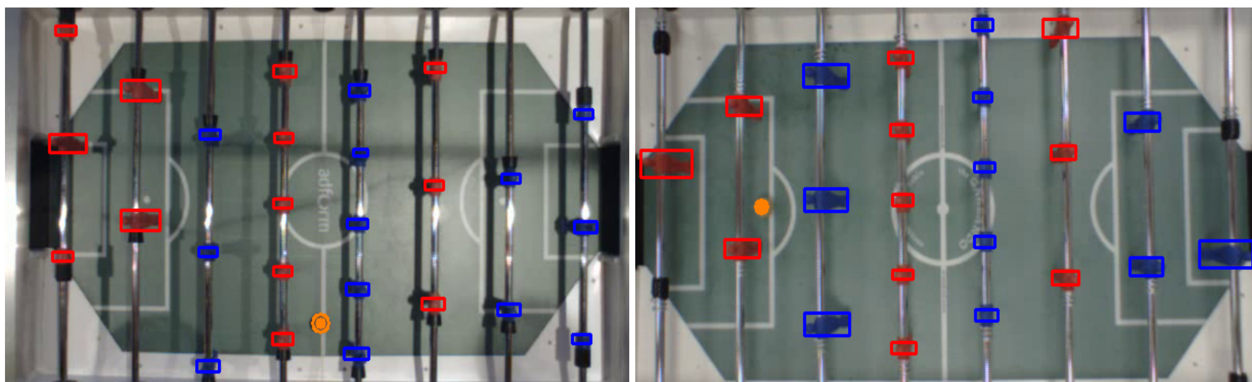
1. Apkarpius nagrinėjamą paveiksliuką, reikia apdoroti mažiau duomenų, kas leidžia algoritmui apdoroti iki 4 KPS daugiau.
2. Apkarpius nagrinėjamą paveiksliuką, gaunami tikslesni rezultatai, nes dingsta visi pašaliniai objektai, kurie gali būti interpretuojami kaip žaidėjai ar kamuoliukas (žr. 18 pav.).



18 pav. Neapkarpytas paveikslėlis. Matome, jog ranka yra traktuojama kaip kamuoliukas arba raudonos spalvos žaidėjas, dėl panašios spalvos

Dabar pavaizduosime kaip atrodo apkarpytas kadras su surastais žaidėjais ir kamuoliuku (žr. 19a pav.). Šiame paveiksliuke matome - raudona stačiakampio formos figūra apibrėžtus priešininko valdomus žaidėjus, mėlyna stačiakampio formos figūra apibrėžtus kompiuterio valdomus žaidėjus, ir oranžine spalva pažymėtą rastą kamuoliuką. Stačiakampiai apie sekamus žaidėjus braižomi di-

namiškai, atsižvelgiant į dvejetainiame kadre rasto kontūro dydį, tam, jog būtų lengviau suvokiama kaip veikia filtravimas.



(a) Apsaugos nuo smūgių interpretuojamos kaip žaidėjai

(b) Apsaugos nuo smūgių nėra interpretuojamos kaip žaidėjai

19 pav. Stalo vaizdas su atpažintais žaidėjais ir kamuoliuku

Šiame paveikslėlyje (žr. 19a pav.) taip pat matome, jog kairiajame ir dešiniajame stalo laikikliuose yra išskirta po tris žaidėjus, nors ten yra tik vienas žaidėjas - taip yra todėl, jog šie laikikliai turi apsaugas nuo smūgio į stalo šoną, kurios yra tokios pačios spalvos, kaip ieškomi žaidėjai. Šią problemą išsprendėme pačiu paprasčiausiu būdu - apsaugas nuo smūgio į šoną apvyniojome juodos spalvos juosta - juoda spalva mūsų algoritmui yra neutrali, tokiu būdu apsaugos nėra interpretuojamos kaip žaidėjai (žr. 19b pav.), ir gauname tikslią stalo futbolo žaidėjų interpretaciją.

Naudodami įsivestą metriką, galime patikrinti kaip kokybiškai veikia mūsų sukurtas algoritmas. Testas atliktas su 50 sekundžių ilgio video, todėl iš viso buvo apdoroti 3768 kadrai, kuriuose:

- kamuoliukas buvo rastas 3360 kadruose, t.y 89.17% - patenkinamas rezultatas;
- visi priešininko valdomi žaidėjai buvo rasti 3637 kadruose, t.y 96.52% - labai geras rezultatas;
- visi kompiuterio valdomi žaidėjai buvo rasti 3687 kadruose, t.y 97.85% - labai geras rezultatas.

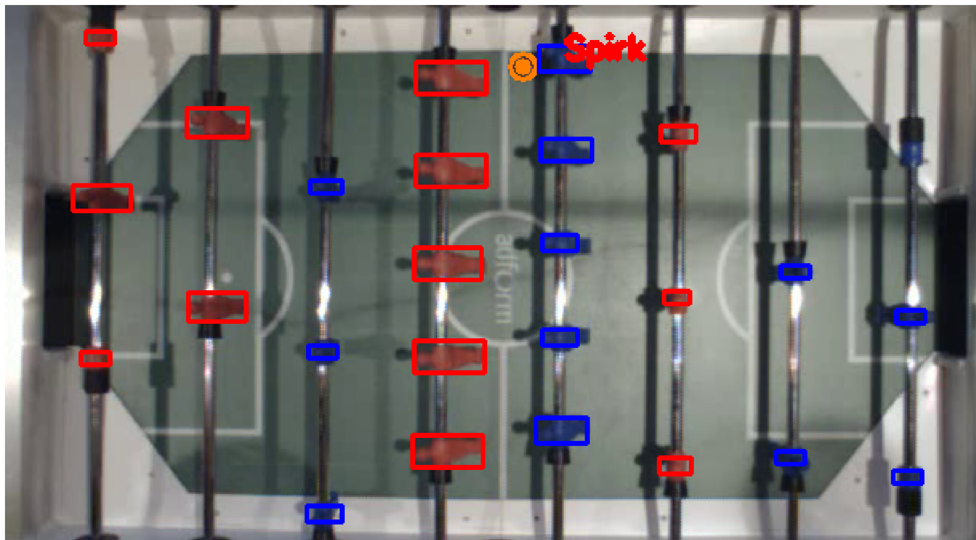
Reikia pastebėti, jog nagrinėtame video kamuoliuko nebuvo aikštelėje apie 1.5 sekundes arba 110 kadrų (po įvarčio iki video pabaigos). Todėl, perskaičiavus procentinę išraišką gauname, jog tuo metu kai kamuoliukas buvo aikštelėje jis buvo rastas 92.09% kartų. Tai yra labai geras rezultatas, kurį dar galima tobulinti algoritmiškai, tačiau šiame darbe laikysime, jog galime pasitikėti kamuoliuko radimo funkcionalumu.

Naudojantis kompiuterinės regos algoritmu, tai pat buvo įgyvendintas labai svarbus funkcionalumas norint sugebėti žaisti stalo futbolą - apskaičiavimas, kada kamuoliukas yra pozicijoje kurioje kompiuterio valdomas žaidėjas gali jį pasiekti ir smūgiuoti vartų link. Įsivesti metriką šio funkcionalumo kokybei patikrinti - sudėtinga, kadangi sprendimas yra priimamas dinamiškai, atsižvelgiant į kamuoliuko ir žaidėjų poziciją. Žaidžiant stalo futbolą, kamuoliuką reikia smūgiuoti tuo metu, kai jis yra prie tavo valdomo žaidėjo, tačiau mes norime smūgiuoti kamuoliuką, kad jis skrietų tiesia trajektorija, o ne kampu, todėl pirmiausia patikriname ar kamuoliukas vertikaliai nuo žaidėjo yra arčiau nei 15 pikselių (atliekant bandymus, pastebėta jog esant šiam atstumui kamuoliukas yra beveik statmenai prieš žaidėją). Taip pat, bandymų metu, buvo pastebėta, jog žaidėjas gali pasiekti kamuoliuką, jeigu jis nuo žaidėjo kontūro centro yra nutolęs mažiau nei 30 pikselių. Galime apsibrėžti atstumo tarp žaidėjo ir kamuoliuko įvertį, kurį žymėsime raide D, ir minima-

lią atstumo ribą tarp kamuoliuko ir žaidėjo, kai žaidėjas gali pasiekti kamuoliuką - E. Atlikus bandymus, apskaičiuotas įvertis buvo:

$$E = \sqrt{30^2 + 15^2} \approx 33.54 \tag{9.1}$$

Šis apskaičiuotas įvertis buvo sumažintas iki $E = 30$ norint užtikrinti, jog klaidingai teigiamų rezultatų būtų kaip įmanoma mažiau. Taigi, jeigu kamuoliukas nutolęs nuo žaidėjo vertikaliai mažiau nei 15 taškų, horizontaliai mažiau nei 30 taškų ir atstumas tarp jų yra mažesnis nei 30 taškų - yra priimamas sprendimas spirti kamuoliuką ($D < E$). Taip atrodo pavyzdys (žr. 20 pav.), kuriame buvo priimtas sprendimas spirti kamuoliuką.



20 pav. Paveikslėlis, kuriame priimtas sprendimas spirti kamuoliuką (pažymėta raudonu užrašu "Spirk")

9.5. Sukurto mokymosi algoritmo rezultatai

Šiame poskyryje aprašysime kokius parametrus, atlygius ir ypatybes naudojome savo sukurtame algoritme ir išnagrinėsime gatus rezultatus.

9.5.1. Atlygio skaičiavimas

Vienas iš pagrindinių aspektų, norint sukurti mokymosi algoritmą, yra apsibrėžti kokius atlygius gauname už atliktus veiksmus. Atlygius apsibrėšime ir teigiamus, ir neigiamus, tam jog mūsų mokymosi procesas būtų efektyvesnis. Teigiami atlygiai ir veiksmai:

- žaidėjų pozicijos išlaikymas vienoje linijoje su kamuoliuku : +1
- kamuoliuko spyrimas link vartų : +2
- kompiuterio valdomų žaidėjų pelnytas įvartis : +10

Neigiami atlygiai ir veiksmai:

- žaidėjų pozicijos neišlaikymas vienoje linijoje su kamuoliuku : -0.005
- leidimas kamuoliukui prariedėti link mūsų ginamų vartų : -2
- priešininko valdomų žaidėjų pelnytas įvartis : -10

Atlygius už atliktus veiksmus parenkame taip, kad už mažesnės svarbos ir dažnai nutinkančius veiksmus gautume mažesnį absoliutųjį atlygį, o už svarbius įvykius - didesnį. Atlygių įverčius parinkome bandymų būdu, stebėdami kaip veikia sukurtas algoritmas prie įvairių būsenų, ir kurios būsenos pasikartoja dažniausiai ir turi didžiausią (ir mažiausią) įtaką žaidimo eigai.

9.5.2. Mokymosi algoritmo parametrai ir apsibrėžtos ypatybės

Mūsų sukurtas mokymosi algoritmas realizuoja 4.11 ir 4.12 formules, todėl svarbiausi parametrai kuriuos turėjome apsibrėžti - ypatybių svoriai ω, nuolaida grąžos funkcijai γ, mokymosi imlumo koeficientas α, epsilon-godumo metodo patemetras ε ir pradinės ypatybių f_n reikšmės. Kadangi, f_n ir ω reikšmės yra nuolat kintančios, todėl jų pradinės reikšmės yra nelabai svarbios, tačiau jos apibrėžiamos intuityviai - kadangi pradžioje visi ypatybių svoriai yra vienodi, tai sakome kad $\omega = [1.0, 1.0]$, o ypatybių reikšmės $f_n = 0, \forall n$. Grąžos funkcijos nuolaidos parametrą $\gamma = 0.8$, todėl jog mums svarbu ne tik iš karto gauti atlygiai, bet ir tolimesnė įvykių seka. Mokymosi pradžioje, imlumo koeficientas α galėtų būti parinktas $\alpha = 1$, jeigu visuomet apskaičiuotume geriausią galimą veiksmą. Bet mes norime ištyrinėti aplinką kartais pasirinkdami ir atsitiktinius veiksmus naudodami epsilon-godumo metodą, todėl jeigu kiekvieną kartą atnaujintume savo Q reikšmę, įvykus atsitiktiniui veiksmui jos gali išsikraipyti. Todėl, šiame darbe pasirinkome naudoti $\alpha = 0.8$ ir $\varepsilon = 0.7$.

Mokymosi algoritmo realizavimui apsibrėžėme dvi ypatybes:

1. Ypatybė - apskaičiuoja, ar sekančiame kadre tikėtina jog galėsime spirti kamuoliuką. Ši ypatybė patikrina kiekvieno žaidėjo poziciją x ir y ašyje kamuoliuko atžvilgiu, bei patikrina, kokiu greičiu ir kuria kryptimi juda kamuoliukas.
2. Ypatybė - apskaičiuoja, ar sekančiame kadre tikėtina, jog žaidėjai bus vienoje linijoje su kamuoliuku.

Ypatybės gali įgyti tokias reikšmes:

$$f_{1,2} = \begin{cases} 1 & \text{jeigu apskaičiuojama, jog ypatybė yra tikėtina} \\ 0 & \text{kitais atvejais} \end{cases}$$

9.5.3. Mokymosi algoritmo rezultatai

Stalo futbolo žaidimas turi labai daug galimų būsenų ir veiksmų porų, todėl norint tinkamai ištestuoti sukurtą mokymosi algoritmą, reikėtų sužaisti labai daug partijų, atlikti skirtingus veiksmus, ir stebėti, ar ypatybių svorių reikšmės pradeda konverguoti. Fiziškai sužaisti tiek partijų - labai sudėtinga, todėl nusprendėme patikrinti savo mokymosi algoritmo veikimą pritaikant jį jau nufilmuotiems žaidimams, kai kompiuteris kontroliuoja puolėjų lazdą, o žmogus kontroliuoja priešininkų gynėjų lazdas. Bandymai buvo atliekami su 33 sekundžių video, kuriame kamuoliukas nuolat yra kompiuterio valdomų puolėjų zonoje (tam, kad testuotume prasmingus duomenis). Pirmą kartą paleidus programą su šiuo video, mokymosi algoritmo parametrai ω turi pradines reikšmes, todėl mokymasis yra tik pradedamas. Pasibaigus vaizdo įrašui, ω reikšmės yra išsaugomos tekstiniame faile, iš kurio antrą kartą paleidžiant programą su tuo pačiu video jos yra nuskaitomos ir naudojamos apsimokyme. Tokiu būdu įvykdžius programą keletą kartų, ω reikšmės turėtų pradėti konverguoti.

Žemiau, pateikiame lentelę, kurioje matome, kaip kito ω reikšmės vykdant programą su tuo pačiu video keletą kartų, ir taip pat keičiant ε ir α reikšmes (nes ilgainiui norime pradėti rinktis tik geriausius veiksmus).

2 lentelė. ω reikšmių kitimas vykdant sukurtą programą su tuo pačiu video daug kartų keičiant tik mokymosi algoritmo parametrus ε ir α.

Bandymų kiekis	ε	α	ω_1	ω_2
0	0.7	0.8	1.0	1.0
5	0.7	0.8	0.007887189202672	0.507454332248433
10	0.7	0.8	0.002869207057006	0.419908438833459
15	0.9	0.99	0.033660089859367	1.288677635621754
20	0.9	0.99	0.091194439735896	0.835491886451383
25	0.9	0.99	0.084329758431414	0.947904483245996
30	0.9	0.99	0.036485777867897	0.964544305045052
35	1.0	0.99	0.016016133299776	1.151961150488660
40	1.0	0.99	0.015755945750362	1.151961150491756
45	1.0	0.99	0.015745900400230	1.151961150491876

Galime matyti, jog atliekant bandymus su $\varepsilon = 0.7$ ir $\alpha = 0.8$ gaunamos ypatybių svorių reikšmės neartėja prie jokio įverčio - tai yra tikėtinas rezultatas, kadangi mes tik pradedame apsimokymo procesą ir 30% kartų pasirenkame atsitiktinį veiksmą. Parinkus $\varepsilon = 0.9$ ir $\alpha = 0.99$ matome, jog reikšmės vis dar yra nepastovios, tačiau pradeda skirtumas tarp jų šiek tiek mažėja. Parinkus $\varepsilon = 1.0$ ir $\alpha = 0.99$, mes visuomet norime rinktis geriausią veiksmą pagal darbartines apmokymo žinias - dėl to galime ir matyti, jog svorių reikšmės pradeda konverguoti: $\omega_1 \rightarrow 0.0157$, $\omega_2 \rightarrow 1.1519$. Teoriškai - galime teigti jog sukurtas mokymosi algoritmas veikia teisingai, tačiau praktiškai - reikėtų atlikti daug daugiau bandymų, kurie turėtų būti atlikti ne su statiniu vaizdo įrašu, o su nuolat kintančiu žaidimu. Tai pasiekti galima keliais būdais - žaisti labai daug partijų prieš mūsų sukurtą algoritmą realiu laiku, sukurti stalo futbolo simuliaciją, kurią galėtume sujungti su mūsų sukurtu algoritmu ir naudoti gautus duomenis apmokymui arba sukonstruoti servo varikliukus prie abiejų stalo futbolo žaidėjų pusės ir leisti žaisti algoritmui prieš save.

Išvados ir rekomendacijos

Šiame darbe buvo atlikta išsami analizė kompiuterinės regos ir mokymosi metoduose tam, kad išsirinkti tinkamiausius stalo futbolo žaidimui. Žaidėjų ir kamuoliuko atpažinimui pasirinkome naudoti OpenCV biblioteką, HSV spalvos modelį ir pritaikėme matematinės morfologijos metodus, kad pašalinti triukšmą. Taip pat buvo sukonstruota visa stalo futbolo konstrukcija - prie stalo pritvirtinta kamera, kuri siunčia kadrus į kompiuterį, kuriame veikia mūsų sukurta programa kuri apdorojusi gautą kadrą siunčia baitų paketą į mikrovaldiklį, kuris yra pritvirtintas prie stalo ir prijungtas prie servo varikliukų. Mikrovaldiklis apdoroja gautus duomenis ir siunčia signalus į prijungtus servo varikliukus, kurie yra pritvirtinti prie stalo specialia konstrukcija (žr. 16 pav.), kuri leidžia slinkti ir sukinėti lazdą. Buvo sukurtas žaidimo modelis analizuojant gaunamus duomenis iš kameros, kuris apibrėžia kamuoliuko judėjimo kryptį, greitį ir poziciją - žinant šiuos parametrus galime apibrėžti situacijas kuriose reikia spirti kamuoliuką arba keisti žaidėjų poziciją, kad išlaikyti juos vienoje linijoje su kamuoliuku. Buvo pritaikytas mokymosi algoritmas kompiuterio valdomai puolėjų lazdai, kuris sugeba priimti kokį sprendimą reikėtų atlikti esamoje būsenoje.

Atlikus šį darbą, galime pateikti tokias išvadas ir rekomendacijas:

1. Sukurtas algoritmas remiantis kompiuterinės regos ir dirbtinio intelekto metodais sugebantis surasti kamuoliuko ir žaidėjų padėtį.
2. Sukurtas algoritmas sugeba apdoroti apie 20 KPS - tai patenkinamas rezultatas, tačiau norint jog stalas sugebėtų žaisti prieš labiau patyrusius žaidėjus, reikėtų sugebėti apdoroti daugiau kadrų per sekundę.
3. Taikant objektų atpažinimo pagal spalvas metodą iškyla problemų dėl apšvietimo - pasikeitus šviesos kiekiui, pakinta ir bandomų išskirti objektų atspalvis ir jie tampa "nematomais" mūsų algoritmui.
4. Iš atliktos analizės galime matyti, jog mums tinkamiausias mokymosi algoritmas stalo futbolo žaidime - naudojant Q-mokymąsi su veiksmo-vertės funkcijos aproksimacija.
5. Taikant objektų atpažinimo pagal spalvas metodą patogu naudoti HSV spalvos modelį, nes galima nurodyti spalvos atspalvio rėžius - taip padidėja tikimybė rasti ieškomą objektą net ir šiek tiek pasikeitus apšvietimui.
6. Filmuojant stalą iš viršaus atsiranda tikimybė, jog kamuoliukas tam tikrais laiko tarpais bus uždengtas žaidėjo (ir taip bus nematomas algoritme). Ši problema kyla dėl filmavimo iš viršaus natūros, tačiau ją galima išspręsti algoritminiu būdu - jeigu kamuoliuko apdorojamame kadre nėra, o jo paskutinė pozicija buvo šalia žaidėjo - galima daryti prielaidą jog kamuoliuko pozicija yra tokia pati kaip žaidėjo. Šią prielaidą reikėtų ištestuoti ateities darbuose.
7. Reikėtų patikrinti, ar ieškant žaidėjų kadre užtektų surasti po vieną žaidėją iš kiekvieno laikiklio, o kitus išskaičuoti jeigu reikia, nes žaidėjų pozicija yra statinė. Tokiu būdu galimai pavyktų apdoroti daugiau kadrų per sekundę.
8. Reikėtų patikrinti, ar pozicionuojant kamerą virš stalo taip, jog nereikėtų apkarpyti kadro kiekvieną kartą jį gavus iš kameros galėtume apdoroti daugiau kadrų per sekundę.

Gairės

1. Optimizuoti šiuo metu sukurtą algoritmą, pašalinant bet kokias nereikalingas operacijas ir išlygiagretinant programinį kodą kaip įmanoma labiau.
2. Norint tinkamai patikrinti mokymosi algoritmą, reikėtų sukurti stalo futbolo simuliacijos programą, prie kurios būtų galima prijungti mūsų sukurtą programą, arba sukonstruoti servo varikliukus prie abiejų stalo futbolo valdomų komandų žaidėjų, ir leisti žaisti kompiuteriui prieš save. Tačiau, tokiu būdu vis tiek reikėtų modifikuoti stalą taip, jog įmušus įvartį kamuoliukas grįžtų į žaidimo aikštelę.
3. Sukurti mokymosi algoritmą pritaikytą situacijai, kai kompiuteris valdo visų žaidėjų lazdas.
4. Įsigyti geresnę kamerą, kuri yra pritaikyta greitai judantiems objektams stebėti, pavyzdžiui "POINT GREY FLEA3".

Literatūros šaltiniai

[1] Michael Aeberhard, Shane Connelly, Evan Tarr, and Nardis Walker. Autonominis stalo futbolas. 2007.

[2] Paveikslėlio suliejimo būdai, 2012. http://docs.opencv.org/2.4/doc/tutorials/imgproc/gausian_median_blur_bilateral_filter/gausian_median_blur_bilateral_filter.html.

[3] Matematinės morfologijos metodas plėstis, 2003. http://homepages.inf.ed.ac.uk/rbf/HIPR2/dilate.htm.

[4] T.M. Doherty and W.C. Boettcher. Under lit foosball table, November 30 2004. US Patent D499,149.

[5] Matematinės morfologijos metodas erozija, 2003. http://homepages.inf.ed.ac.uk/rbf/HIPR2/erode.htm.

[6] HSV spalvos modelio vizualizacija, 2017. https://researchgate.net/publication/277898686_Automated_Opal_Grading_by_Imaging_and_Statistical_Learningl.

[7] Mario Martin. Bellmano lygtis ir jos reikšmių apskaičiavimas, 2011. http://www.cs.upc.edu/~mmartin/Ag4-4x.pdf.

[8] Adam Metcalf. Pinball žaidimas: dideliu greičiu judančio kamuoliuko stebėjimas ir sekimas realiu laiku. 2011.

[9] OpenCV oficialus puslapis, 2017. http://opencv.org/l.

[10] OpenCV paveikslėlių filtravimo dokumentacija, 2017. http://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html.

[11] OpenCV ir Matlab palyginimas, 2012. http://blog.fixational.com/post/19177752599/opencv-vs-matlab.

[12] Servo varikliukų valdymas naudojant arduino mikrovaldiklį, 2016. https://playground.arduino.cc/Learning/SingleServoExample.

[13] David Silver. Būsenos-vertės ir veiksmo-vertės funkcijų aproksimacija, 2017. http://www0.cs.ucl.ac.uk/sta /d.silver/web/Teaching_files/FA.pdf.

[14] David Silver. Markovo grandinės, 2017. http://www0.cs.ucl.ac.uk/sta /d.silver/web/Teaching_files/MDP.pdf.

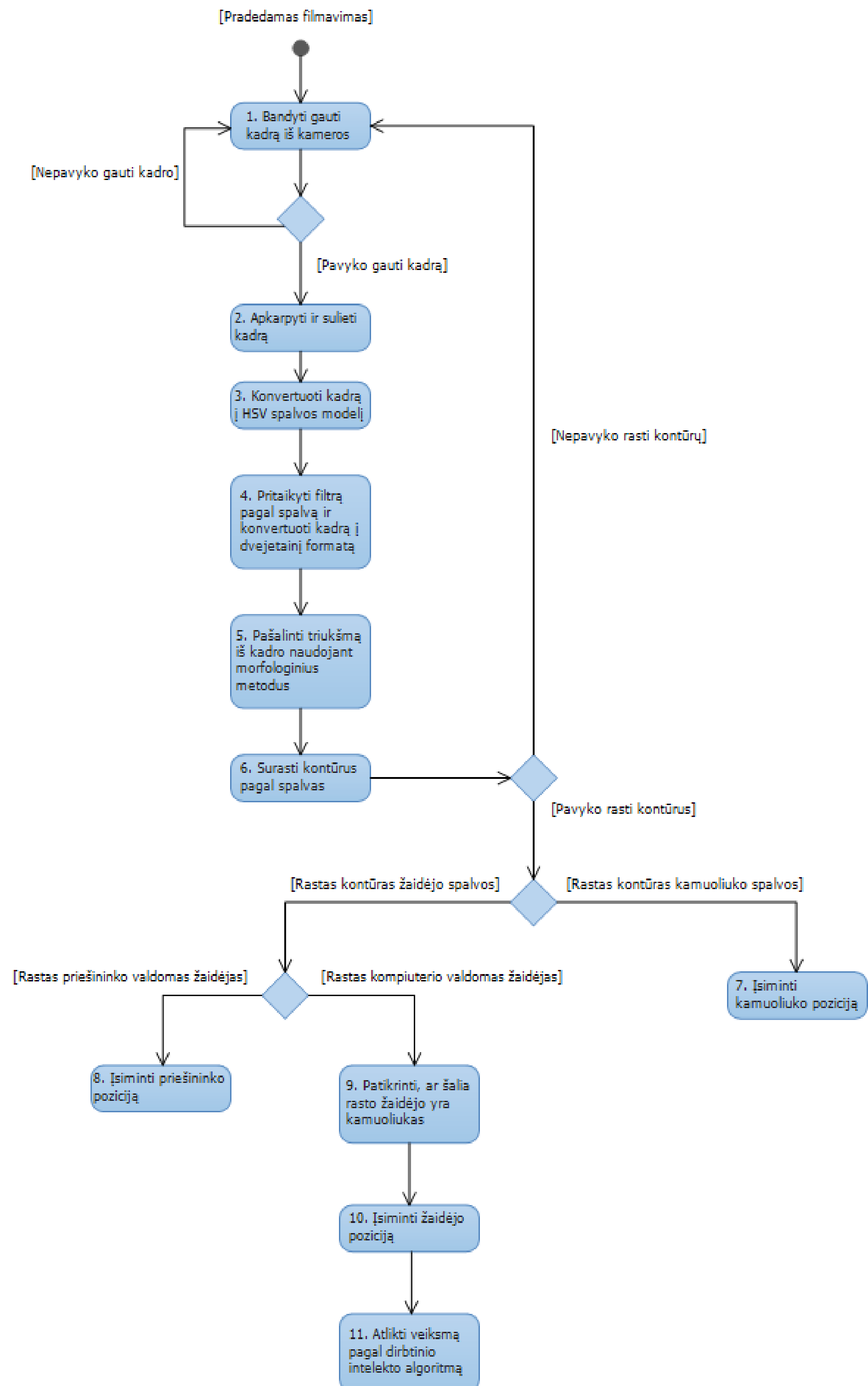
[15] David Silver. Monte-karlo ir laikinojo skirtumo metodai, 2017. http://www0.cs.ucl.ac.uk/sta /d.silver/web/Teaching_files/MC-TD.pdf.

[16] Richard S Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding, 1996.

[17] Christopher JCH Watkins and Peter Dayan. Q-learning, 1992.

Priedai

A. Sukurto kompiuterinės regos algoritmo veiksmų diagrama



B. Sukurta stalo futbolo konstrukcija