Planiranje robotskog djelovanja primjenom principa "pojačanog učenja"

Polančec, Mateo

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Mateo Polančec

Zagreb, 2018.

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Mentori:

Prof. dr. sc. Bojan Jerbić, dipl. ing.
Dr. sc. Marko Švaco, dipl. ing.

Student:

Mateo Polančec

Zagreb, 2018.

Izjavljujem da sam ovaj rad izradio samostalno koristeći znanja stečena tijekom studija i navedenu literaturu.

Zahvaljujem se profesoru dr.sc.Bojanu Jerbiću te kolegama dr.sc. Marku Švaci i mag.ing.meh. Filipu Šuligoju na vrijednim informacijama i konstruktivnim kritikama.

Mateo Polančec

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE
Središnje povjerenstvo za završne i diplomske ispite
Povjerenstvo za diplomske ispite studija strojarstva za smjerove:
procesno-energetski, konstrukcijski, brodostrojarski i inženjersko modeliranje i računalne simulacije

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum	Prilog
Klasa:	
Ur. broj:	

DIPLOMSKI ZADATAK

Student: **Mateo Polančec** Mat. br.: 0035190494

Naslov rada na hrvatskom jeziku: **Planiranje robotskog djelovanja primjenom principa "pojačanog učenja"**

Naslov rada na engleskom jeziku: **Robot Task Planning by Applying the Principle of "Reinforcement Learning"**

Opis zadatka:

Robotska sposobnost interpretacije prostornih struktura iznimno je važna za razumijevanje radnog prostora i postizanje kognitivnih sposobnosti za rad u ljudskoj okolini. Prostorna struktura podrazumijeva raspored objekata u prostoru. Rad je potrebno ograničiti na ravninske (2D) probleme, odnosno prepoznavanje rasporeda objekata u ravnini. U radu je potrebno razviti upravljački algoritam planiranja robotskog djelovanja za rekonstrukciju poznatih prostornih struktura iz slobodno raspoređenih objekata u ravnini. Pri tome je potrebno primijeniti principe metode "pojačanog učenja" (eng. "Reinforcement Learning"). Razvijeni algoritam verificirati na robotskoj opremi u Laboratoriju za projektiranje izradbenih i montažnih sustava.

U radu navesti korištenu literaturu i eventualno dobivenu pomoć.

Zadatak zadan:	Datum predaje rada:	Predviđeni datum obrane:
16. studenog 2017.	18. siječnja 2018.	24., 25. i 26. siječnja 2018.

Zadatak zadao: Predsjednica Povjerenstva:

Prof. dr. sc. Bojan Jerbić Prof. dr. sc. Tanja Jurčević Lulić

SADRŽAJ

POPIS SLIKA

POPIS OZNAKA

Oznaka	Jedinica	Opis
V	-	vrijednosna funkcija stanja
α	-	korak iteracije
R	-	skalarna funkcija nagrade
γ	-	faktor propadanja
S	-	funkcija stanja
δ	-	vrijednost privremenih razlika
Q	-	Q funkcija
A	-	funkcija akcija
π	-	strategija (politika)
ε	-	faktor vjerojatnosti odabira slučajne akcije
G	-	skalarna funkcija nagrade definirana za n koraka unaprijed
$\boldsymbol{\theta}$	-	vektor parametara funkcije aproksimacije
$\mathbf{s}$	-	vektor stanja
v	-	vrijednost vrijednosne funkcije stanja za određeno stanje
$\bar{v}$	-	aproksimirana vrijednost vrijednosne funkcije stanja za određeno stanje
ϕ	-	bazna funkcija stanja
α, β, γ	-	Eulerovi kutevi
$\mathbf{R}$	-	matrica rotacije
$\vec{u}\ \vec{v}\ \vec{w}$	-	tri trodimenzionalna vektora kojima je definirana matrica rotacije $\mathbf{R}$
$\mathbf{R}_x, \mathbf{R}_y, \mathbf{R}_z$	-	matrice rotacije oko osi x,y i z
$\mathbf{T}$	-	matrica homogene transformacije
$\mathbf{e}$	-	jedinični vektor koji ukazuje na smjer osi vrtnje
$\mathbf{R}_\mathbf{E}$	-	matrica transformacije nastala na temelju transformacije iz Eulerovih kuteva
$\mathbf{R}_{\mathbf{A-A}}$	-	matrica transformacije nastala na osnovi angle-axis zapisa
$\mathbf{X}$	-	trodimenzionalni vektor koji se sastoji od *X, Y i Z* koordinata vrha alata
ς	-	maksimalna greška kalibracije alata
$\mathbf{t}$	-	trodimenzionalni vektor koordinata u baznom koordinatnom sustavu robota
$\bar{q}$	-	aproksimirana vrijednost Q funkcije za određeno stanje
a	-	vrijednost pojedine akcije

SAŽETAK

Proces učenja koji proizlazi kao odgovor na vizualnu spoznaju okoline polazna je odrednica brojnih istraživanja iz područja robotike te umjetne inteligencije. Proces planiranja djelovanja autonomnog robota nad neuređenim skupom objekata obrađen je u ovom radu koristeći principe pojačanog učenja. Korištene su Metode Privremenih Razlika uz primjenu linearnih baznih funkcija za aproksimaciju vrijednosne funkcije stanja zbog prevelikog broja diskretnih stanja u kojim se sustav može naći. Cilj je pronaći optimalan slijed akcija kojima agent (robot) premješta predmete dok ne postigne unaprijed definirano ciljno stanje. Algoritam je podijeljen u dva dijela. U prvom dijelu cilj je naučiti parametre kako bi mogli pravilno aproksimirati Q funkciju, dok se u drugom dijelu algoritma iskorištavaju dobiveni parametri za definiranje slijeda akcija koje se šalju UR robotu pomoću TCP protokola. Pojačano učenje pokazalo se prikladnim za navedeni problem te su rezultati prikazani na slikama (26) i (27). Pošto je u radu korišten dvodimenzionalni pristup problemu u vidu budućeg rada postoji mogućnost modificiranja algoritma za kreiranje 3D prostornih struktura.

Ključne riječi: Robotika, Pojačano učenje, Autonomni roboti

SUMMARY

Abstract

The learning process that arises in response to visual perception of the environment is the starting point for numerous research in the field of applied and cognitive robotics. In this research we propose a reinforcement learning based action planning for an autonomous robot in an unstructured environment. We have developed an algorithm based on temporary difference methods using linear basic functions for the approximation of the value state function because of the vast number of discrete states that the autonomous robot can encounter. The aim is to find the optimal sequence of actions that the agent (robot) needs to take in order to move objects in a 2D environment until they reach the predefined target state. The algorithm is divided into two parts. In the first part, the goal is to learn the parameters in order to properly approximate the Q function. In the second part of the algorithm the obtained parameters are used to define the sequence of actions sent to the UR robot using the TCP protocol. Our algorithm which is based on the SARSA algorithm from the reinforcement learning framework has given good results. The algorithm has been validated in an experimental laboratory scenario. In future work we plan to address a more complex of the assembly of 3D space structures.

Keywords: Robotics, Reinforcement learning, Autonomous robot

1. Uvod

Prema [1] , razvoj tehnike danas ponajviše karakterizira razvoj autonomnih tehničkih sustava koji su sposobni tumačiti djelovanje okoline te učiti na temelju iskustva. Autonoman robot je uređaj s motoričkim sposobnostima i senzorima za prikupljanje povratnih informacija iz promjenjive okoline. Prikupljene informacije obrađuje upravljački program odgovoran za rad izvršnih elemenata ostvarujući aktivnu interakciju s okolinom u realnom vremenu. U određenoj mjeri to je samodostatan uređaj koji može djelovati u realnom svijetu bez izravnog ljudskog utjecaja. Proces učenja koji proizlazi kao odgovor na vizualnu spoznaju okoline polazna je odrednica brojnih istraživanja iz područja robotike te umjetne inteligencije. Proces planiranja djelovanja autonomnog robota nad neuređenim skupom objekata obrađen je u radovima [1] i [2]. Suštinska razlika u ova dva rada odnosi se metodologiju učenja robota. U [2] se proces učenja odvija kroz imitaciju učitelja dok se u [1] precipira prostorni raspored objekata te se pomoću genetskog algoritma uz unaprijed zadan niz akcija pokušava dobiti optimalan redoslijed. Metodologija rada korištena u ovom radu koristi principe pojačanog učenja (*eng. Reinforcement learning*) na temelju kojih agent (robot) uči na temelju iskustava u interaciji s okolinom. Prilikom interakcije s okolinom robot uz odabir prave akcije dobiva pozitivnu te u obrnutom slučaju negativnu nagradu. Prikupljanjem nagrada kroz niz epizoda biva sposoban vrednovati stanja u kojima se nalazi te odabrati optimalnu akciju. Također, brojna istraživanja provedena su u području mobilne robotike koristeći principe pojačanog učenja [3], [4] u kojima su razvijeni modeli autonomnog odlučivanja. Odgovore na ključna pitanja percipiranja okoline, kreiranja funkcije nagrade, odbira skupine akcija te stanja u kojima se robot može naći određena su u ovisnosti o specifičnosti zadatka. Sposobnost modeliranja sustava uvelike ovisi o inžinjeru te u konačnici pridonosi rješenju zadatka. Jedan od načina unaprijeđenja modela pojačanog učenja odnosi se na nadogradnju već postojeći algoritama sa iskustvenim funkcijama ograničenja te nekim od modela strojnog učenja kao što su neruonske mreže poradi klasificiranja stanja te aproksimacije vrijednosne funkcije stanja kod problema kontinuiranih te općenito prevelikog broja diskretnih stanja u kojima se agent može naći. Generalno postoji nekoliko principa rješavanja gore navedenih problema. Prvi se odnosi na direktno traženje optimalne strategije odnosno niza akcija koje vode do rješenja te optimizacija iste aproksimacijom vrijednosne funkcije stanja koja su detaljno objašnjena u [5] te hijerarhijska podijela cjelokupnog zadatka na niz manjih podzadataka unutar kojih se traže

optimalne lokalne politike [6], [7], [8]. Ovo istraživanje nastavlja se na [1] u kojem je potrebno odrediti optimalno djelovanje UR robota zasnovano na tumačenju prostornih struktura. Prostorne strukture podijeljene su na nekoliko osnovnih oblika (kvadar, valjak, prizma). Svakoj od navedenih struktura definirano je početno stanje te pomoću Mađarske metode pridodijeljeno konačno stanje. Pomoću pojačanog učenja agent (robot) uči na osnovi iskustava u interakciji s okolinom te biva sposoban isplanirati djelovanje (odrediti niz akcija koje vode do željenog cilja), odnosno odrediti optimalnu strategiju (*eng. Optimal policy*). Problem prevelikog broja diskretnih stanja u kojima se agent može naći rješen je pomoću linearnih baznih funkcija, koristeći Fourierove redove, prema [9].

2. Pojačano učenje u kontekstu robotike

Planiranje robotskog djelovanja u kontekstu robotike svodi se na percipiranje okoline u kojoj se agent nalazi te pronalaženja optimalnog slijeda akcija koje vode ka izvršenju zadatka. Osnovna razlika u odnosu na učenje s učiteljem odnosi se na nedostatak informacije o potrebnom nizu akcija koje agent mora izvršiti kako bi došao do konačnog stanja, no međutim u nekim situacija prema [5] i [10] postoji mogućnost kombinacije te dvije vrste učenja. Pojačano učenje u interakciji s okolinom kroz odabir mogućih akcija prima skalarnu vrijednost nagrade te na taj način kroz niz epizoda agent biva sposoban sam donosit odluke. U osnovi dijeli se na direktno traženje optimalne strategije (*eng. Policy search methods*) unutar koje se inicijalno definira strategija (politika, odnosno slijed akcija) koju je potrebno optimizirati te na traženje vrijednosne funkcije stanja (*eng. Value function-function of states*). Mnoge metode direktnog traženja optimalne strategije svode se na unaprijed definiranu strategiju π koju je potrebno optimizirati u vidu parametara koji su joj pridodijeljeni računanjem gradijenta $\Delta\boldsymbol{\theta}_i$. Cilj je povećati ukupnu povratnu nagradu kroz proces iterativnog ažuriranja parametara prema jednadžbi (1)

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + \Delta\boldsymbol{\theta}_\mathbf{i} \tag{1}$$

Prednosti u odnosu na modele koji koriste vrijednosne funkcije stanja su: brža konvergencija poradi manjeg broja parametara koje je potrebno izračunati u slučaju jednostavnijih problema te mogućnost vrlo jednostavnog integriranja ekspertnog znanja. No međutim u slučaju velikog broja diskretni stanja prema [11] potrebno je koristiti modele koji koriste aproksimacijske funkcije za računanje vrijednosne funkcije stanja. Te u slučaju korištenja modela koji direktno traže optimalnu strategiju također je važno napomenuti da je jedino moguće naći lokalno, a ne i globalno optimalnu strategiju [12]. Problem rješavanja zadataka na temelju direktnog traženja optimalne strategije svodi se na pravilnu inicijalizaciju iste što generalno znači da mora zadovoljiti brojne parametre. Dakle, strategija, odnosno politika mora biti opisana glatkim kontinuiranim trajektorijama bez naglih skokova, mora uzeti u obzir sva ograničenja (maksimalni moment motora, dosezi robotske ruke, prepreke itd.), odabirom akcija ne smije doći do velikih skokova u funkciji kojom je opisana, mogućnost parametrizacije strategije bez prvotnog znanja o krajnjem rezultatu, mogućnost nadogradnje unaprijed poznatim znanjem, strategija mora biti invarijantna reprezentacija zadatka, treba imati svojstvo regularizacije te mnoge druge [13]. Za opis strategija najčešće se koriste kubni splajnovi te polinomi višeg

reda. No međutim, problem odabira predhodno navedenih funkcija očituje se u nemogućnosti nošenja sa perturbacijama odnosno neočekivanim promjenama okoline. Zbog navedenog problema razvijeni su brojni drugi opisi strategija primjerice: „Gaussian Mixture Model (GMM)“ [14] te „Dynamic Movement Primitives (DMP)“ [15]. Kreiranjem kompleksnije parametrizacije (uzimanjem u obzir velikog broj parametara sustava) strategije postoji mogućnost zaglavljivanja u lokalnom minimumu te spora konvergencija osobito u slučajevima velikog prostora mogućih stanja, dok kod jednostavnije parametrizacije konvergencija je brža te se u većini slučajeva pronalazi sub-optimalno rješenje. Odabir kompleksnosti parametrizacije bazira se na pokušaju i pogrešci te je samo traženje strategije koja može dovoljno dobro opisati dani problem dugotrajan proces. Neki od primjera korištenja algoritama direktnog traženja optimalne strategije navedeni su na slici (1).

Slika 1 Lijeva slika: humanooidni robot iCub sa 53 stupnja slobode, cilj: pogoditi sredinu mete; desna slika: prikaz humanoidnog robota COMAN, kretanje uz minimizaciju utroška energije [13]

Metode direktnog traženja optimalne strategije pokazale su se učinkovite u kombinaciji sa učiteljem. Dakle, inicijalizacija parametara strategije dobiva se na temelju imitacije učitelja te se daljnjim postupkom optimizacije dolazi do optimalnih parametara sustava. Na slici (2) prikazan je postupak okretanja palačinke za 180 stupnjeva pomoću robotske ruke.

Slika 2 Prikaz momentom kontroliranog Barrett WAM robota naučenog okrenuti palačinku za 180 stupnjeva u zraku i ponovno je uhvatiti [13]

Algoritmi koji računaju vrijednosnu funkciju stanja u osnovi dijele se na modele kod kojih je potrebno odrediti funkciju vjerojatnosti tranzicije *T(s' , a, s) (eng. model-based algorithams),* kao što su Dinamičko Programiranje te na modele kod kojih to nije potrebno kao što su Monte Carlo te Metode Privremenih Razlika (*eng. model-free algorithams*). Prilikom korištenja Monte Carlo metode agent prima osrednjenu nagradu te ažurira vrijednosnu funkciju na kraju epizode što uzrokuje visoku varijancu. Nasuprot tome Metode Privremenih Razlika *(eng. Temporal difference learning*) ažuriraju vrijednosnu funkciju u svakom koraku iteracije (*eng. Bootstraping*) što je ujedno velika prednost u odnosu na Monte Carlo metodu [9].

Pojačano učenje također predstavlja most između klasične teorije optimalnog upravljanja, adaptivnog upravljanja te prirodom inspiriranih tehnika učenja nastalih proučavanjem ponašanja životinja [16]. Inžinjeri s područja optimalnog upravljanja uvidjeli su potrebu za naprednijim modelima potrebnim za opisivanje kompleksnih dinamičkih sustava zbog toga što klasične metode optimalnog upravljanja rade „off-line“ odnosno ne mogu uzeti u obzir dinamičke promjene okoline. U skladu s tim razvili su se modeli adaptivnog dinamičkog programiranja (ADP) i neurodinamičkog programiranja (NDP). Kod neurodinamičkog programiranja kao funkcije aproksimacije koriste se umjetne neuronske mreže što predstavlja rješenje za problem prevelikog broja diskretnih stanja i akcija (*eng. course of dimensionality*). Napredniji modeli pojačanog učenja ranije spomenuti nude mogućnost rješavanja problema bez potrebe za definiranjem modela cjelokupne dinamike sustava što predstavlja veliku prednost u odnosu na dinamičko programiranje. No međutim jedna od glavnih motivacija primjene pojačanog učenja kod problema optimalnog upravljanja zasniva se na činjenici da kod implementacije nije potrebno nikakvo znanje o parametrima robota već samo poznavanje mogućih stanja i mjerenje kontrolnog signala. Modeli pojačanog učenja također se koriste kod

nemogućnosti modeliranja pojedinih fiziklanih pojava kao što su trenje u zglobovima robotske ruke, pojava vibracija visokih frekvencija nastalih zbog elastičnosti u zglobovima i mnogih drugih pojava. Naime, u tom slučaju nije potrebno matematički modelirati te pojave već ih naučiti [17]. Također sve veći broj istraživanja zalazi u područje medicinske robotike u kojoj se osim područjem optimalnog upravljanja te interakcije čovjeka i robota [18] zalazi u područje računalne vizije, prema [19]. Poradi gore navedenih razloga u ovom radu biti će korištena jedna od Metoda Privremenih Razlika koja slijedi unaprijed definiranu strategiju pod nazivom SARSA uz aproksimaciju vrijednosne funkcije stanja pomoću linearnih baznih funkcija.

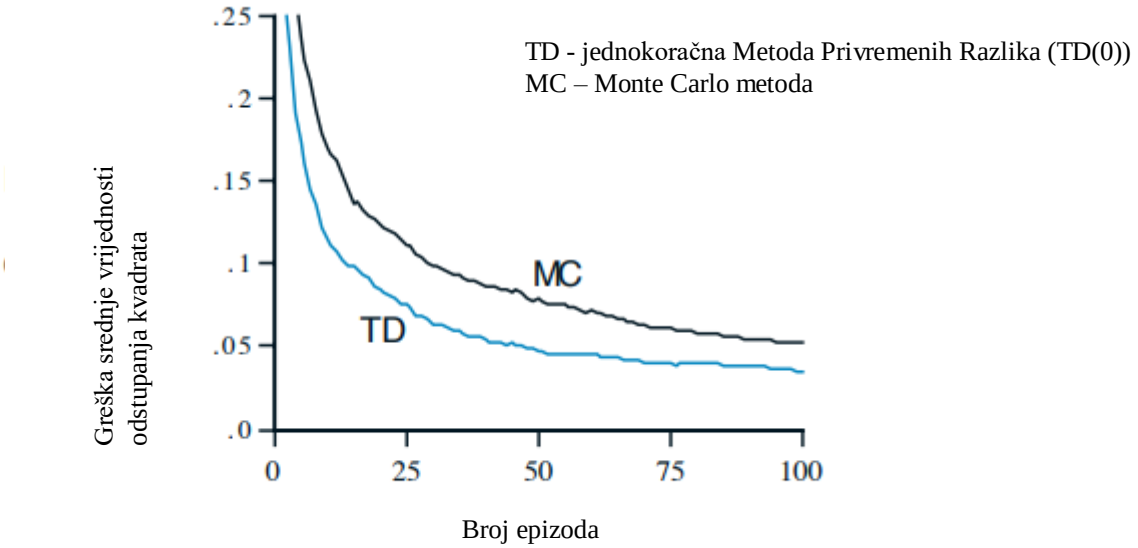
3. Teorijaska podloga

U narednom poglavlju biti će detaljno objašnjene sve metode korištene prilikom izrade zadatka.

3.1. Pojačano učenje

3.1.1. Metode Privremenih Razlika (eng. Temporal difference learning)

Glavne karakteristike Metode Privremenih Razlika odnose se na mogućnost ažuriranja vrijednosne funkcije stanja u svakom koraku iteracije te nije potrebno definirati model okoline u vidu funkcije vjerojatnosti tranzicije *T(s' , a, s)*(*eng. Joint probability distribution*). Prema [9], iako to nije moguće matematički dokazati Metode Privremenih Razlika konvergiraju u praksi brže od Monte Carlo metode te je dokazana njihova mogućnost konvergencije u tabličnoj formi te prilikom korištenja aproksimacijskih linearnih baznih funkcija.



Slika 3 Usporedba konvergencije uprosječene ukupne greške između TD(0) i Monte Carlo metode uz konstantan α [9]

Prema jednadžbi (2) dana je funkcija vrijednosti stanja za osnovnu jednokoračnu Metodu Privremenih Razlika (TD(0)).

$$V(S_t) = V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{2}$$

$V(S_t)$ – vrijednosna funkcija stanja

R_{t+1} – skalarna vrijednost nagrade za odabranu akciju

γ – faktor propadanja, određuje utjecaj buduće nagrade (*eng. discount rate*)

α – korak iteracije

Vrijednost privremenih razlika (*eng. temporal-difference error*) definirana je jednadžbom (3).

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{3}$$

U praksi zapravo je potrebno definirati funkciju vrijednosti prelaska iz jednog u drugo stanje (*eng. Value function of state-action pairs*), Q funkcija koja se definira prema jednadžbi (4):

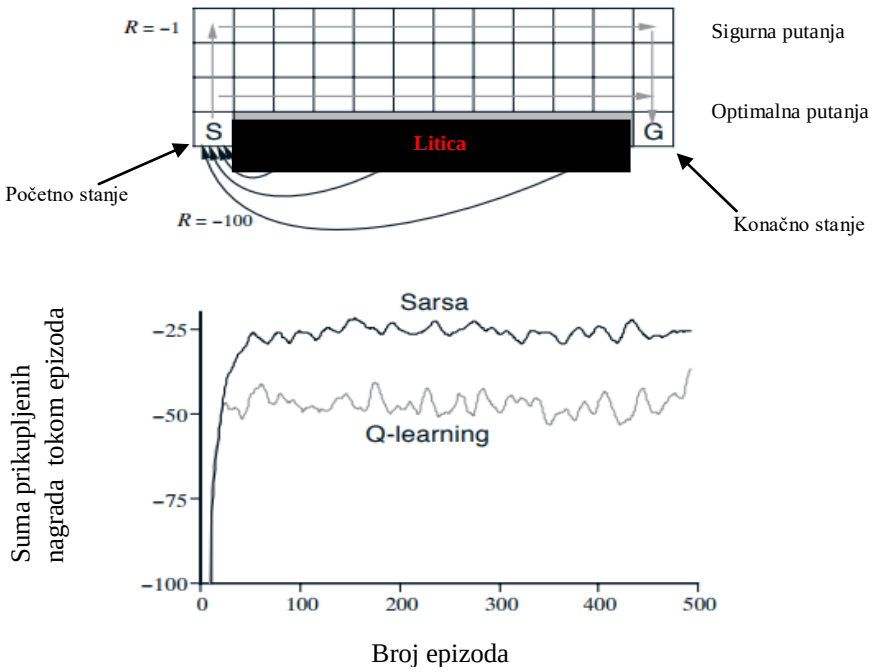
$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q(S_t, A_t) - Q(S_t, A_t)\right] \tag{4}$$

U konačnici se niz akcija odnosno strategija odabire prema izrazu (5).

$$\pi(s) = \arg\max_a Q(s, a) \tag{5}$$

3.1.2. Usporedba SARSA i Q-learning algoritama

Prema definiranju slijeda akcija razlikuju se On-policy (metode koje ažuriraju vrijednosne funkcije stanja na osnovi unaprijed zadane strategije) te Off-policy (metode koje za ažuriranje stanja uzimaju najbolju moguću akciju u danoj iteraciji odnosno akciju za koju je iz danog stanja najveća vrijednost Q funkcije). Glavni predstavnik On-policy metoda je SARSA algoritam dok je glavni predstavnik Off-policy metoda Q-learning algoritam. Na osnovi dijagrama na slici (4) te prema zaključima iz literature [9] uz ε-greedy odabir akcija SARSA se pokazala pouzdanijim izborom. Naime, ε-greedy odabir akcija temelji se na ε vjerojatnosti odabira slučajne akcije te $(1-\varepsilon)$ vjerojatnosti odabira najbolje akcije iz trenutnog stanja. U slučaju da je $\varepsilon \approx 0$ oba algoritma daju podjednake rezultate.



Slika 4 Usporedba SARSA i Q-learning algoritama na primjeru zadatka hodanja uz liticu [9]

Primjer hodanja uz liticu modeliran je kao „*eng. gridworld*“ zadatak. Moguće akcije su gore, dolje lijevo i desno. Svaka akcija donosi skalarnu vrijednost nagradu -1 osim u slučaju pada s litice kada nagrada iznosi -100. Na temelju dijagrama sa slike (4) može se uočiti promjena razlika sume nagrada oba algoritma tijekom epizoda. Kod Q-learning algoritma može se uočiti veća negativna vrijednost nagrade tijekom epizoda što zapravo znači veću vjerojatnost pada s litice odnosno manju pouzdanost algoritma u danom primjeru. Osnovna razlika gore navedenih algoritama očituje se u definiranju Q-funkcije prema jednadžbama (6) i (7). Jednadžba (6) prikazuje izraz za Q funkciju SARSA algoritma dok jednadžba (7) prikazuje izraz za Q funkciju Q-learning algoritma.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q(S_t, A_t) - Q(S_t, A_t)\right] \tag{6}$$

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{7}$$

Kao što je navedeno na početku ovog podpoglavlja glavna razlika svodi se na način definiranja odabira akcija te ažuriranja Q funkcije.

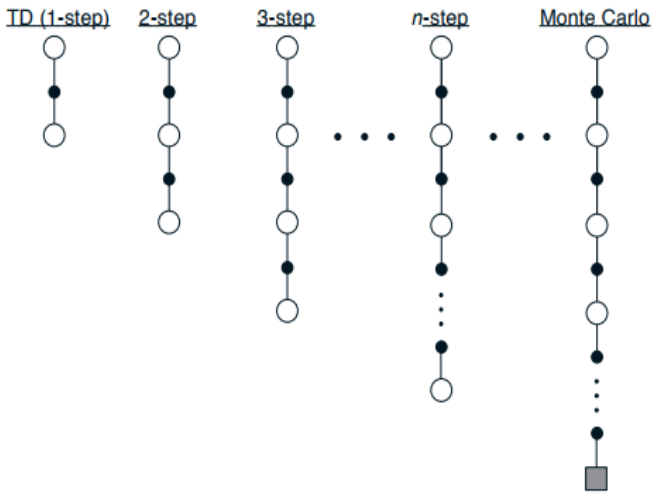
3.1.3. n-koračne Metode Privremenih Razlika

Sponu između Monte Carlo i TD(0) metode čine upravo n-koračne metode privremenih razlika (*eng. n-step Temporal difference methods*) TD(n) koje uzimaju u obzir nekoliko koraka unaprijed te im je nagrada definirana jednadžbom (8)

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-t-1} R_T \tag{8}$$

u kojoj T predstavlja zadnji vremenski korak epizode, dok γ predstavlja faktor propadanja.

Na slici (5) prikazan je spektar TD(n) metoda. Raspon završava sa Monte Carlo metodom jer upravo ako se uzme u obzir suma nagrada svih koraka do kraja jedne epizode dobiva se preklapanje TD(T) sa Monte Carlo metodom.



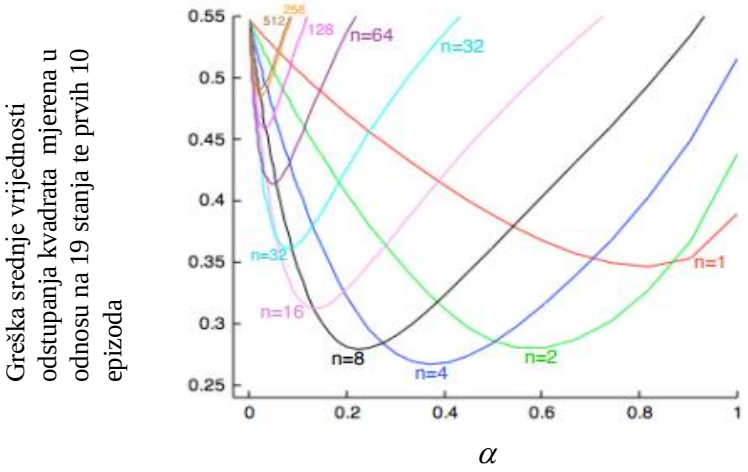
Slika 5 Prikaz povratne nagrade spektra TD(n) metoda

Osnovne relacije koje se koriste kod TD(n) metoda prikazane su jednadžbama (9) i (10):

$$G_t^n = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}), \quad n \geq 1, 0 \leq t < T - n \tag{9}$$

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha \left[G_t^n - Q_{t+n-1}(S_t, A_t) \right], \quad 0 \leq t \leq T \tag{10}$$

Na osnovi primjera iz literature [9] pod nazivom „*eng. Random walk example*“ testirani su brojni TD(n) algoritmi te prikazana usrednjena ukupna greška kroz deset epizoda u ovisnosti o parametru α te se pokazalo da algoritmi metode privremenih razlika u rasponu od 2 - 8 koraka daju najbolje rezultate prema slici (6).



Slika 6 Usporedba TD(n) algoritama u ovisnosti o veličini koraka iteracije

3.2. Funkcije aproksimacije

U slučaju kontinuiranih ili velikog broja diskretnih stanja i akcija potrebno je iskoristiti aproksimacijske funkcije (*eng. curse of dimensionality*). Koriste se dva tipa aproksimacijskih funkcija. Nelinearne, primjerice umjetne neuronske mreže te linearne bazne funkcije koje u osnovi su linearne u parametrima, ali nelinearne u komponentama vektora stanja. Najčešće se koriste za aproksimaciju vrijednosnih funkcija stanja, iako mogu biti primjenjene i za direktnu aproksimaciju strategija (politika). Aproksimacijske funkcije znatno reduciraju potrebno vrijeme proračuna, jer nije potrebno uzeti u obzir sve kombinacije stanja i akcija već na osnovi određenog broja epizoda treniraju se parametri vektora $\boldsymbol{\theta}$ za kasniju predikciju optimalne strategije. U pojačanom učenju se najčešće susrećemo sa nestacionarnim problemima te na bazi toga potrebno je odabrati aproksimacijske funkcije koje se mogu nositi sa tim izazovom. U slučaju nelinearnih aproksimacijskih funkcija (umjetne neuronske mreže, stabla podijele itd.) osim što su mnogo kompleksnije postoji velika vjerojatnost zaglavljivanja u lokalnom minimumu. Iz navedenih razloga se najčešće koriste linearne aproksimacijske funkcije [9]. U dolje navedenim jednadžbama opisan je postupak korištenja aproksimacijskih funkcija kod On-policy TD(0) algoritma. Vektor parametara $\boldsymbol{\theta}$ koji je potrebno izračunati definiran je kao $\boldsymbol{\theta} = \left[\theta_0, \theta_1, \theta_2 .... \theta_n\right]^T$. Za izračun promjene vrijednosti parametara koristi se stohastički gradijentni spust (SGD) te su osnovne relacije navedene jednadžbama (11), (12),(13) i (14).

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{1}{2}\alpha\nabla\left[v_\pi(\mathbf{s}_t) - \bar{v}(\mathbf{s}_t, \boldsymbol{\theta}_t) \right]^2 \tag{11}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\left[v_\pi(\mathbf{s}_t) - \bar{v}(\mathbf{s}_t, \boldsymbol{\theta}_t) \right]\nabla \bar{v}(\mathbf{s}_t, \boldsymbol{\theta}_t) \tag{12}$$

$$\bar{v}(\mathbf{s}, \boldsymbol{\theta}) = \boldsymbol{\theta}^T \phi(\mathbf{s}) = \sum_{i=1}^{n} \boldsymbol{\theta}_i \phi_i(\mathbf{s}) \tag{13}$$

Te u konačnici dobivamo izraz:

$$\boldsymbol{\theta}_{t+n} = \boldsymbol{\theta}_{t+n-1} + \alpha\left[G_t^n - \bar{v}(\mathbf{s}_t, \boldsymbol{\theta}_{t+n-1}) \right]\nabla \bar{v}(\mathbf{s}_t, \boldsymbol{\theta}_{t+n-1}) \tag{14}$$

Funkcija nagrade G_t^n se računa prema izrazu (9).

$v_\pi(\mathbf{s}_t)$ – stvarna vrijednost vrijednosne funkcije stanja

$\bar{v}(\mathbf{s}, \boldsymbol{\theta})$ – aproksimirana vrijednost vrijednosne funkcije stanja

G_t^n – skalarna funkcija nagrade $\left(eng.cumulative\ discounted\ reward\right)$

$\phi(\mathbf{s})$ – bazna funkcija stanja

3.2.1. Bazne funkcije stanja

Za više dimenzionalne prostore stanja funkcije aproksimacije primjenjene u pojačanom učenju imaju mnogo toga zajedničkog sa regresijskim metodama čiji je cilj definirati funkciju nad danim skupom podataka. Bazne funkcije stanja mogu biti definirane kao polinomi n-tog reda Fourierovi redovi te razne druge. U narednim podpoglavljima biti će navede i objašnjene polinomne bazne funkcije te Fourierovi redovi.

3.2.1.1. Polinomne bazne funkcije

Kod polinomnih baznih funkcija na osnovi vektora stanja primjerice u dvodimenzionalnom prostoru stanja $\mathbf{s} = (s_1, s_2)^T$ postoji mnogo načina prikaza vektora stanja. Primjerice vekor stanja može biti predstavljen kao polinom 2. reda na način $\mathbf{s} = (1, s_1, s_2, s_1 s_2, s_1^2, s_2^2)$. Jedan od načina definiranja reda polinoma potrebnog za pravilno aproksimiranje vrijednosne funkcije stanja je testiranjem istih te provjerom njihove točnosti nad danim skupom podataka.

Polinomi višeg reda uzimaju u obzir veći stupanj interakcije između pojedinih komponenata vektora stanja, stoga za kompleksnije funkcije potrebno je uzeti u obzir polinome većeg stupnja imajući u vidu činjenicu da broj komponenata vektora stanja tada eksponecijalno raste. Važno je napomenuti iako se koriste polinomi višeg reda nad vektorom stanja vrijednosna funkcija stanja ostaje linearna u odnosu na parametre $\boldsymbol{\theta}$.

3.2.1.2. Fourierovi redovi kao bazne funkcije

Furierov red predstavlja težinsku sumu periodičnih funkcija (kosinusa i sinusa). Fourierova serija ili općenitije Fourierova transformacija vrlo često se koristi u primjenjenoj znanosti zbog toga što ako je poznata funkcija koju je potrebno aproksimirati vrlo je jednostavno odrediti težinske faktore baznih funkcija. Nadalje, sa dovoljnim brojem baznih funkcija moguće je vrlo prezicno aproksimirati bilo koju funkciju. U domeni pojačanog učenja funkcije koje je potrebno aproksimirati su nepoznate te su stoga Fourierove serije vrlo pogodan model baznih funkcija te je koristeći ih moguće postići vrlo visoku točnost prema [9]. Jednadžba (15) predstavlja općeniti izraz za Fourierovu seriju.

$$\bar{f} = \frac{a_0}{2} + \sum_{k=1}^{n}\left[a_k \cos\left(k\frac{2\pi}{T}x \right) + b_k \sin\left(k\frac{2\pi}{T}x \right)\right] \tag{15}$$

Prema [20], prilikom modeliranja problema u domeni pojačanog učenja pošto je funkcija koju je potrebno aproksimirati nepoznata nemoguće je egzaktno odrediti parametre a,b Fourierove serije, ali ih možemo tretirati kao parametre i testirati njihov utjecaj na konvergenciju rezultata. Zbog jednostavnosti uzima se da je T=2. Nadalje dobiva se izraz

$$\phi_i(x) = \begin{cases} 1 & i = 0 \\ \cos\left(\frac{i+1}{2}\pi x\right) & \mathrm{i} \succ 0, \mathrm{i} \to \mathrm{odd} \\ \sin\left(\frac{i}{2}\pi x\right) & \mathrm{i} \prec 0, \mathrm{i} \to even \end{cases} \tag{16}$$

Zbog toga što vrijednosne funkcije stanja nisu ni parne ni neparne ni periodične možemo ih definirati na intervalu [-1, 1] uz uvjet da ulazne komponente vektora stanja budu između [0, 1]. To rezultira prema [20] periodičnom funkcijom na intervalu [-1, 1]. Nadalje na temelju tog zaključka definira se bazna funkcija prema jednadžbi (17).

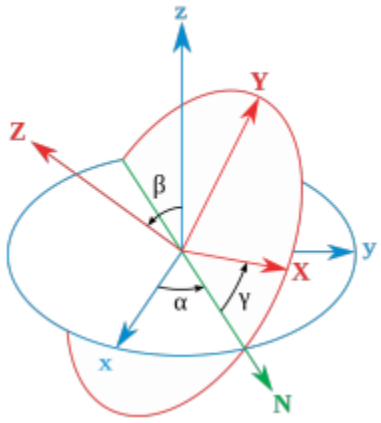
$$\phi_i(x) = \cos(i\pi x) \tag{17}$$

Zaključno, važno je primjetiti da povećanje reda Fourierovih baznih funkcija korespondira sa većom frekvencijom komponenata vrijednosne funkcije stanja.

3.3. Eulerovi kutevi

Orijentacija jednog koordinatnog sustava u odnosu na drugi određena je sa tri kuta α, β, γ koje nazivamo Eulerovi kutovi. Značenje pojedinih kutova ovisi o redoslijedu rotacija te je prije primjene potrebno proučiti o kojoj konvenciji se radi. Redoslijeda rotacija, tj. konvencija ima 12 i podijeljene su u dvije skupine:

1. Eulerovi kutovi (*z-x-z*, *x-y-x*, *y-z-y*, *z-y-z*, *x-z-x*, *y-x-y*)
2. Tait–Bryanovi kutovi (*x-y-z*, *y-z-x*, *z-x-y*, *x-z-y*, *z-y-x*, *y-x-z*)



Slika 7 Eulerovi kutovi po konvenciji z-x-z [21]

Slika (7) prikazuje redoslijed rotacija po konvenciji $z - x' - z''$ u unutarnjim rotacijama ili $z - x - z$ u vanjskim rotacijama. Pri tome kut α predstavlja rotaciju oko z-osi, kut β oko *N*-osi (ili x'-osi), a kut γ oko *Z*-osi (ili z''-osi). Danas se najčešće koristi konvencija $z - y' - x''$ ili $z - y - x$ gdje se kutovi φ, θ i ψ nazivaju valjanje, poniranje i skretanje (*eng. yaw, pitch, roll*).

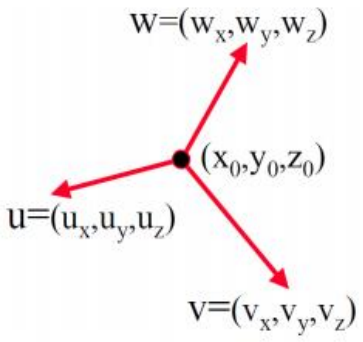
3.4. Matrica Rotacije

Matrica rotacije **R** odnosno DCM (*eng. Direction Cosine Matrix*) je 3 × 3 matrica koja opisuje orijentaciju nekog koordinatnog sustava u prostoru te je njen opći izraz dan jednadžbom (18).

$$\mathbf{R} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \tag{18}$$

Ukoliko je koordinatni sustav zadan sa tri trodimenzionalna vektora $\vec{u}, \vec{v}, \vec{w}$ kao na slici (8) tada orijentacija tog koordinatnog sustava glasi:

$$\mathbf{R} = \begin{bmatrix} \vec{u} & \vec{v} & \vec{w} \end{bmatrix} = \begin{bmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{bmatrix} \tag{19}$$



Slika 8 Koordinatni sustav opisan sa tri vektora [21]

3.4.1. Transformacija iz Eulerovih kutova u matricu rotacije

Množenjem triju matrica $\mathbf{R}_x, \mathbf{R}_y$ i $\mathbf{R}_z$ (svaka od njih sadrži iznos rotacije oko određene osi) izvedena je prema *x-y-z* konvenciji matrica transformacije $\mathbf{R}_\mathbf{E}$.

$$\mathbf{R}_x(\phi) = \text{Roll}(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \tag{20}$$

$$\mathbf{R}_y(\theta) = \text{Pitch}(\theta) = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \tag{21}$$

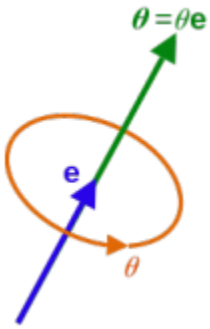
$$\mathbf{R}_z(\psi) = \mathrm{Yaw}(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{22}$$

$$\mathbf{R}_\mathbf{E} = \begin{bmatrix} \cos\theta\cos\psi & \cos\phi\sin\psi + \sin\phi\sin\theta\cos\psi & \sin\phi\sin\psi - \cos\phi\sin\theta\cos\psi \\ -\cos\theta\sin\psi & \cos\phi\cos\psi - \sin\phi\sin\theta\sin\psi & \sin\phi\cos\psi + \cos\phi\sin\theta\sin\psi \\ \sin\theta & -\sin\phi\cos\theta & \cos\phi\cos\theta \end{bmatrix} \tag{23}$$

3.5. Axis–angle zapis rotacije

Axis–angle prikaz rotacije čini trodimenzionalni vektor **θ** koji se sastoji od jediničnog vektora **e** koji ukazuje na smjer osi vrtnje i kuta **θ** koji određuje iznos rotacije oko te osi. Definicija vektora $\boldsymbol{\theta}$ dana je jednadžbom (24).

$$\boldsymbol{\theta} = \theta\mathbf{e} = \begin{bmatrix} R_x & R_y & R_z \end{bmatrix} \tag{24}$$



Slika 9 Grafički prikaz vektora θ [21]

3.5.1. Transformacija iz Axis–angle zapisa u matricu rotacije

Transformacija u matricu rotacije vrši se prema jednadžbi (25). Vektor **θ** te parametari $e_{x,}e_y$ i e_z koji određuju vektor **e** dani su izrazima (27), (28), (29) te (30).

$$\mathbf{R}_{\mathbf{A-A}} = \begin{bmatrix} Ce_x^2 + \cos\theta & Ce_xe_y - e_z\sin\theta & Ce_xe_z + e_y\sin\theta \\ Ce_ye_x + e_z\sin\theta & Ce_y^2 + \cos\theta & Ce_ye_z - e_x\sin\theta \\ Ce_ze_x - e_y\sin\theta & Ce_ze_y + e_x\sin\theta & Ce_z^2 + \cos\theta \end{bmatrix} \tag{25}$$

Konstanta C dana je izrazom (25).

$$C = 1 - \cos\theta \tag{26}$$

$$\theta = \sqrt{R_x^2 + R_y^2 + R_z^2} \tag{27}$$

$$e_x = \frac{R_x}{\theta} \tag{28}$$

$$e_y = \frac{R_y}{\theta} \tag{29}$$

$$e_z = \frac{R_z}{\theta} \tag{30}$$

3.5.2. Transformacija iz matrice rotacije u Axis–angle zapis

Transformacija iz matrice rotacije **R** u Axis–angle zapis vrši se prema sljedećim formulama:

$$\theta = \left(\frac{1}{2}\left[R_{11} + R_{22} + R_{33} - 1 \right] \right) \tag{31}$$

$$e_x = \frac{R_{32} - R_{23}}{2\sin\theta} \tag{32}$$

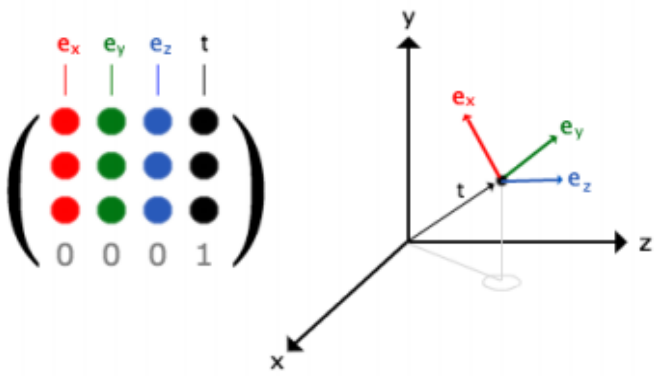
$$e_y = \frac{R_{13} - R_{31}}{2\sin\theta} \tag{33}$$

$$e_z = \frac{R_{21} - R_{12}}{2\sin\theta} \tag{34}$$

3.6. Matrica homogene transformacije

Pozicija i orijentacija jednog koordinatnog sustava u odnosu na drugi definirana je matricom homogene transformacije. Matrica homogene transformacije je 4 × 4 matrica koja se sastoji od matrice rotacije (DCM matrice) i vektora translacije prema jednadžbi (35).

$$\mathbf{T} = \begin{bmatrix} a & b & c & t_x \\ d & e & f & t_y \\ g & h & i & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0 & 1 \end{bmatrix} \tag{35}$$



Slika 10 Slikovito objašnjenje matrice homogene transformacije [21]

3.6.1. Transformacija iz jednog u drugi koordinatni sustav

Ukoliko u prostoru postoje dva koordinatna sustava O_1 i O_2, a njihov položaj i orijentacija je definirana u nekom trećem, baznom, koordinatnom sustavu O tada vrijede jednakosti:

$$\overset{O}{O_1}\mathbf{T} \cdot \overset{O_1}{O2}\mathbf{T} = \overset{O}{O_2}\mathbf{T} \tag{36}$$

$$\overset{O_1}{O2}\mathbf{T} = \overset{O}{O_1}\mathbf{T}^{-1} \cdot \overset{O}{O_2}\mathbf{T} \tag{37}$$

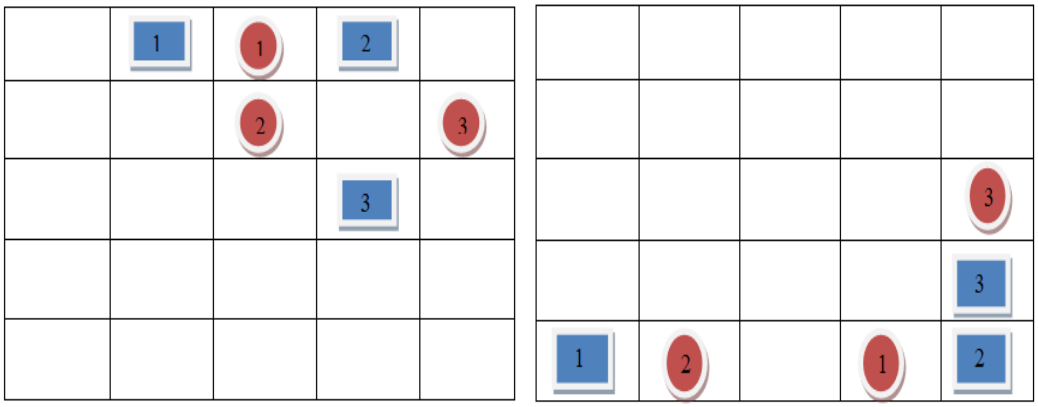
3.7. Kuhn-Munkresov algoritam

Jedan od postojećih algoritama koji se koristi u srodnim problemima kombinatorne optimizacije je Kuhn-Munkresov algoritam (poznat pod nazivom Mađarska metoda). Kuhn-Munkresov algoritam upotrebljava se za određivanje optimalnog načina dodjeljivanja n elemenata na n mjesta i to tako da svaki element bude pridružen samo jednome mjestu. Svako pridruživanje ima svoj trošak ili vrijeme potrebno za obavljanje određene radnje koja je vezana za pridruživanje (npr. obavljanje posla na stroju). Potrebno je pronaći rješenje koje minimizira trošak izvođenja cijelog skupa radnji tj. aktivnosti. Modelom teorije grafova problem se definira pronalaženjem najmanjeg troška savršenog sparivanja težinskog bipartitnog grafa. Problem pridruživanja poseban je slučaj problema linearnog programiranja,

te se može rješavati pomoću algoritama koji rješavaju te općenite slučajeve, no zbog specifičnosti ovog problema, postoje efikasniji algoritmi koji ga rješavaju. Mađarska metoda jedan je od najpoznatijih algoritama kombinatorne optimizacije za rješavanje linearnog problema pridruživanja u polinomnom vremenu [1].

3.7.1. Primjer korištenja Kuhn-Munkresovog algoritma

U slučaju da se u prostoru stanja nalazi više identičnih predmeta (primjerice valjaka ili kocaka) bez određenog algoritma nemoguće je odrediti koje točno ciljno stanje pripisati kojem predmetu. Primjerice u prostoru stanja nalaze se dvije kocke na svojim trenutnim pozicijama. Kao ciljna stanja odabrana su dva željena. Postavlja se pitanje koju točno kocku postaviti u koje od ta dva ciljna stanja. U tom slučaju koristi se Mađarska metoda koja svakoj kocki odnosno svakom predmetu zasebno pripisuje određeno ciljno stanje na osnovi funkcije gubitka koja se u tom slučaju minimizira. Funkcija gubitka koju je potrebno minimizirati odnosi se na broj koraka odaljenosti svakog pojedinog objekt od mogućih ciljnih stanja. Ulazna matrica za Mađarsku metodu može se kreirati na dva načina. Unutar prvog načina neovisno o broju istih te različitih predmeta kreira se jedna matrica. Unutar te matrice udaljenosti početnog i ciljnog stanja za iste predmete definirane su brojem koraka od početnog do ciljnog stanja dok su udaljenosti za početno stanje jedne vrste predmeta do ciljnog stanja druge vrste predmeta uzete neke velike vrijednosti (primjerice 999). Velike vrijednosti uzete su poradi sigurnosti kako se ne bi dogodilo da predmet jedne vrste (primjerice kocka) dođe na mjesto druge vrste predmeta (primjerice valjka). Unutar drugog načina zadaje se broj matrica ovisno o broju različitih predmeta. Dakle, za jednu vrstu predmeta zadaje se jedna matrica unutar koje se postavljaju početna i ciljna stanja te se svako početno stanje pridodjeljuje jednom ciljnom stanju. U sklopu ovog zadatka izabran je drugi način zadavanja zbog tehničkih razloga (načina dinamičkog programiranja te upisivanja vrijednosti u matrice te izvlačenja podataka iz samih matrica pri završetku proračuna). Primjer kreiranja matrica za Mađarsku metodu u slučaju potrebe određivanja ciljnih stanja za 3 valjka te 3 kocke. Stanja su definira od lijevog gornjeg polj do desnog donjeg počevši sa nultim poljem te završno sa poljem 24. Udaljenosti su izražene na osnovi četiri osnovna pomaka: gore, dolje, lijevo i desno te svaki pomak iznosi jediničnu vrijednost.



Slika 11 Prikaz početnog i ciljnog stanja predmeta kao ulaz u Kuhn Munkresov algoritam

Na slikama (12), (13) i (14) bit će prikazana ulazna stanja, definicija ulazne matrice te rješenje Kuhn Munkresovog algoritma.

kocke	1	2	3
početno	1	3	13
ciljno	20	24	19

valjci	1	2	3
početno	2	7	9
ciljno	23	21	14

Slika 12 Prikaz početnih i ciljnih stanja predmeta

Početno /krajnje	p1	p2	p3
c1	5	7	5
c2	7	5	3
c3	6	4	2

Početno /krajnje	p1	p2	p3
c1	5	4	4
c2	5	4	6
c3	4	3	1

Slika 13 Ulazne matrice u Kuhn-Munkresov algoritam

kocke	1	2	3
početno	1	13	3
ciljno	20	24	19

valjci	1	2	3
početno	7	2	9
ciljno	23	21	14

Slika 14 Prikaz početnih i ciljnih stanja svih predmeta dobivenih kao rješenje Kuhn-Munkresovog algoritma

3.8. TCP protokol

TCP (*eng. Transmission Control Protocol*) je jedan od osnovnih protokola unutar IP grupe protokola. Korištenjem TCP protokola aplikacija na nekom od hostova umreženog u računalnu mrežu kreira virtualnu konekciju prema drugom hostu, te putem te ostvarene konekcije zatim prenosi podatke. Stoga ovaj protokol spada u grupu tzv. spojnih protokola, za razliku od bespojnih protokola kakav je primjerice UDP (*eng. User Datagram Protocol*). TCP garantira pouzdanu isporuku podataka u kontroliranom redoslijedu od pošiljatelja prema primatelju. Osim toga, TCP pruža i mogućnost višestrukih istovremenih konekcija prema jednoj aplikaciji na jednom hostu od strane više klijenata. TCP osigurava sigurnu, nedupliciranu poruku udaljenom korisniku. [22] TCP upotrebljava određen raspon portova kojima razdjeljuje primjenske programe na strani pošiljatelja i primatelja. Svaka strana TCP konekcije ima dodijeljenu 16-bitnu oznaku za obje strane aplikacije (slanje, primanje). Portovi su u osnovi podijeljeni u 3 kategorije:

1. poznati portovi (0 – 1023)
2. registrirani portovi (1024 – 49150)
3. dinamički/privatni portovi (49151 – 65535)

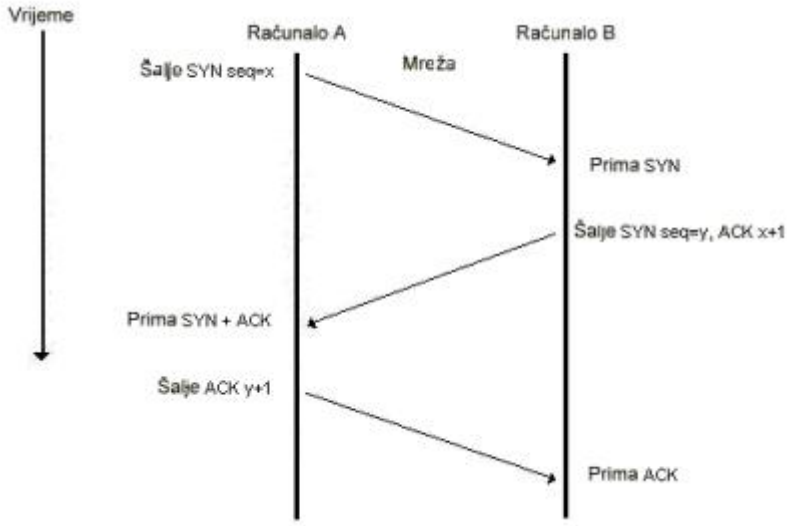
3.8.1. Uspostavljanje veze

Proces koji se izvodi na jednom računalu prilikom uspostavljanja veze s procesom na nekom drugom računalu naziva se uspostavljanje veze (*eng. Connection Establishment*). Računalo koje traži uspostavu veze naziva se klijent (*eng. client*), a drugo računalo se naziva poslužitelj (*eng. server*). Klijentski proces informira klijentski TCP da želi uspostaviti vezu s poslužiteljem. Klijentsko računalo tada šalje poslužitelju prvi specijalni segment. Poslužitelj odgovara drugim specijalnim TCP segmentom i konačno klijent odgovara trećim specijalnim segmentom. Ova procedura se naziva "*three-way handshake*". U nastavku slijedi objašnjenje spomenutih segmenata:

1. Klijent prvi šalje specijalni TCP segment poslužitelju. Taj specijalni segment ne sadrži podatke aplikacijske razine. U zaglavlju segmenta jedan je od bitova zastavica – tzv. SYN bit, postavljen na 1. Iz tog razloga, taj specijalni segment zove se SYN segment. Nadalje, klijent odabire inicijalni redni broj (client_isn) i stavlja ga u polje za redni broj inicijalnog TCP SYN segmenta.

2. Pod pretpostavkom da IP datagram koji sadrži TCP SYN segment stigne do poslužitelja, on izdvaja TCP SYN segment iz datagrama, alocira TCP spremnik i varijable i šalje segment kojim odobrava uspostavu veze klijentu. Taj segment odobravanja veze također ne sadrži podatke aplikacijske razine, ali sadrži tri važne informacije u zaglavlju segmenta. Prvo, SYN bit je postavljen na 1. Drugo, Acknowledgment polje zaglavlja TCP segmenta se namješta na isn+1. Na kraju, poslužitelj odabire svoj inicijalni redni broj (server_isn) i stavlja vrijednost u polje zaglavlja TCP segmenta.
3. Kada klijent primi segment odobravanja veze, također alocira spremnik i varijable u vezi. Klijent tada šalje poslužitelju još jedan segment koji potvrđuje da je dobio segment odobravanja veze. To radi tako da stavi vrijednost server_isn+1 u acknowledgement polje zaglavlja. SYN bit postavlja se u 0 budući da je veza uspostavljena.

Kada se sva tri koraka obave, klijent i poslužitelj jedan drugome mogu slati segmente koji sadrže podatke. U svakom budućem segmentu SYN će biti postavljen na 0. Slika (15) prikazuje „*three-way handshake*" proceduru. [23]



Slika 15 „Three-way handshake" [23]

3.9. Kalibracija alata

Kalibracija vrha alata ili pivotiranje alata je tehnika koja se u robotici koristi kako bi se precizno odredile koordinate vrha alata. Potrebno je dovesti vrh alata robota u istu točku u prostoru sa minimalno tri različite konfiguracije robota. Za svaku točku spremaju se njezine koordinate u baznom koordinatnom sustavu robota u trodimenzionalni vektor $\mathbf{t_i}$ i njezina orijentacija u DCM matricu $\mathbf{R_i}$. Jednadžba koja treba biti zadovoljena glasi [24]:

$$\begin{bmatrix} \mathbf{R_2 - R_1} \\ \mathbf{R_3 - R_2} \\ \vdots \\ \mathbf{R_n - R_{n-1}} \end{bmatrix} \mathbf{X_t} = \begin{bmatrix} \mathbf{t_1 - t_2} \\ \mathbf{t_2 - t_3} \\ \vdots \\ \mathbf{t_{n-1} - t_n} \end{bmatrix} \Rightarrow \mathbf{AX_t = B} \tag{38}$$

iz koje slijedi da je:

$$\mathbf{X_t} = \mathbf{(A^T \times A)^{-1} A^T \times B} \tag{39}$$

$\mathbf{X_t}$ je trodimenzionalni vektor koji se sastoji od *X, Y i Z* koordinata vrha alata. Maksimalna greška kalibracije ς može se izračunati prema sljedećoj formuli:

$$\varsigma = \left\| \begin{bmatrix} \mathbf{R_2 - R_1} \\ \mathbf{R_3 - R_2} \\ \vdots \\ \mathbf{R_n - R_{n-1}} \end{bmatrix} \mathbf{X_t} - \begin{bmatrix} \mathbf{t_1 - t_2} \\ \mathbf{t_2 - t_3} \\ \vdots \\ \mathbf{t_{n-1} - t_n} \end{bmatrix} \right\| = \sqrt{\sum_{i=0}^{n-1} \left| \mathbf{(R_{i+1} - R_i)X_t - t_i + t_{i+1}} \right|^2} \tag{40}$$

4. Prikaz korištenih algoritama i primjenjenih funkcija

Za izradu ovog zadatka korišten je programski jezik Python 3 te paketi Numpy i OpenCv 3.0.

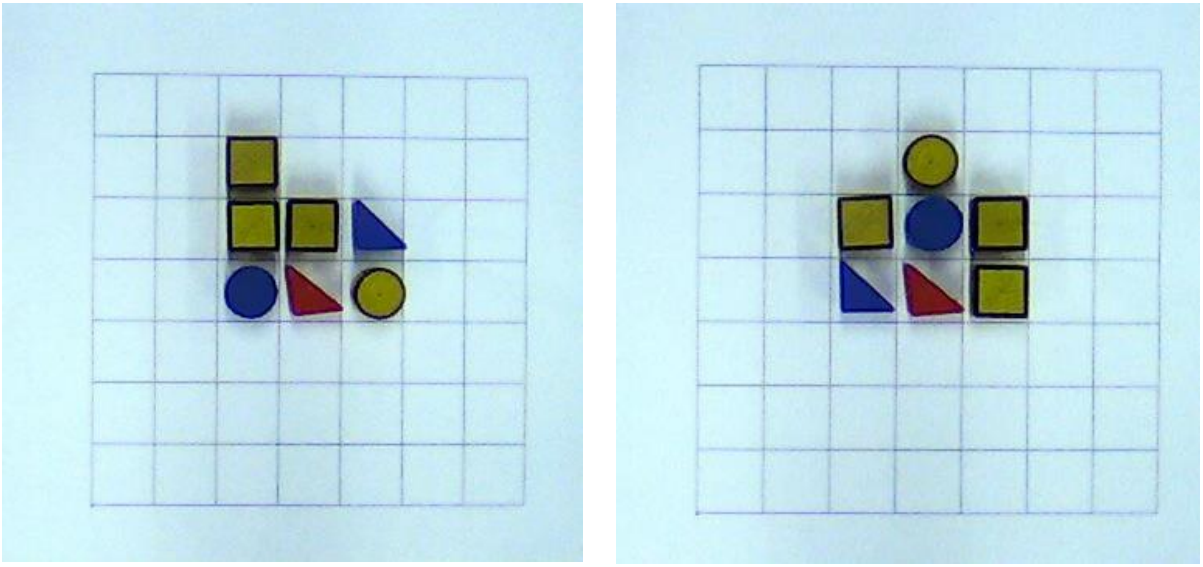
Cilj je kreirati unaprijed zadanu geometrijsku strukturu baziranu na nekoliko osnovnih elemenata (valjak, kocka, prizma) uz najmanji broj akcija robota, odnosno koraka algoritma.

Prva važna postavka problema odnosi se na definiranje radne okoline. Dakle, uvedeno je pojednostavljenje u vidu dvodimenzionalnog slučaja, što znači da prilikom definiranja stanja treća dimenzija neće biti uzeta u obzir. To u konkretnom slučaju znači da se neće u obzir uzimati mogućnost kreiranja trodimenzionalnih geometrijskih struktura odnosno da neće postojati mogućnost pozicioniranja jednog na drugi geometrijski objekt. Problematika modeliranja problema u domeni pojačanog učenja svodi se na definiranje vektora stanja i akcija, dodijeljivanja konačnih stanja predmetima na osnovi definiranih početnih te odabira prikladnih algoritama za konkretni slučaj. Cilj zadatka bio je odrediti niz akcija koje robot mora izvesti kako bi u danoj okolini odnosno unaprijed definiranom prostoru stanja premjestio predmete sa njihovih početnih pozicija u konačne na osnovi definirane funkcije nagrade uz najmanji broj koraka. Glavni kriterij uspješnosti izvršenja zadatka sastoji se u prikladno definiranom vektoru stanja te skalarnoj funkciji nagrade koju agent koji pretražuje prostor dobiva za svaku izvedenu akciju. Na temelju niza isprobanih akcija te prikupljenih nagrada računa se Q funkcija. U danom zadatku okolina se sastoji od 49 međusobno povezanih polja u obliku šahovnice 7 x 7 sa predmetima postavljenim u odgovarajućim poljima prema slici (16). Pošto se u samoj okolini može nalaziti više predmeta stanje je definirano na temelju svih, a ne pojedinačnog predmeta. Povećanjem broja predmeta broj mogućih kombinacija stanja eksponencijalno raste te su stoga primjenjene linearne bazne funkcije na temelju Fourierovih redova pomoću koji se aproksimira Q funkcija. Sve potrebne funkcije biti će detaljnije objašnjene u narednim podpoglavljima.

4.1. Vizijski sustav

Za vizijski sustav koristi se web kamera koja je pozicionirana iznad radnog područja. Kamera se koristi za uzimanje dva uzorka slika. Na jednoj od slika prikazani su predmeti u početnom stanju te na drugoj u konačnom, slika (16). Kako bi se odredilo na kojoj poziciji se koji predmet na osnovi slike nalazi sliku je potrebno podijeli prema dimenzijama šahovnice odnosno na 7 x 7 matricu. Svako polje sa slike opisano je područjima piksela u x i y smjeru.

Glavni cilj vizijskog sustava je odrediti poziciju svakog predmeta na slici. Postupak određivanja pozicije predmeta na slici temelji se na određivanju kontura te centra svake pojedine konture. Nakon određivanja centra svake pojedine konture traži se polje na slici odnosno pozicija kojoj centar te konture pripada. Kako bi to bilo moguće odrediti koriste se funkcije *imread()* (za učitavanje slike), *findConturs()* (za pronalazak kontura) te *moments()* (na osnovi koje se određuje centar konture) iz paketa OpenCv 3.0.



Slika 16 Prikaz početog i konačnog stanja predmeta u radnoj okolini gledano iz perspektive kamere

U slučaju da se predmet cijelom svojom površinom baze ne nalazi u polju potrebno je odrediti udaljenost u pikselima od centra traženog polja. Definiranjem odstupanja centra predmeta od centra polja kojem svojim većinskim udijelom pripada, u pikselim, jednostavnom formulom prema jednadžbama (41) i (42) pretvara se u milimetre (kalibracija kamere).

$$x_{mm} = \frac{h}{n * d} \tag{41}$$

x_{mm} – koordinata x izražena u milimetrima
h – koordinata x izražena u pikselima
n – ukupan broj podijela polja u x smjeru
d – širina jednog polja izraženja u milimetrima

$$y_{mm} = \frac{w}{m * b} \tag{42}$$

y_{mm} – koordinata y izražena u milimetrima
w – koordinata y izražena u pikselima
m – ukupan broj podijela polja u y smjeru
b – širina jednog polja izraženja u milimetrima

Početne i krajnje pozicije predmeta koriste se kao ulazni parametri za definiranje vektora stanja.

4.2. Algoritam

U ovom podpoglavlju biti će objašnjene sve funcije korištene prilikom izrade algoritma. Sam algoritam podijeljen je u dvije faze. Prva faza odnosi se na treniranje parametara funkcije aproksimacije, dok se druga faza odnosi na predviđanje najboljih akcija na temelju dobivenih parametara.

4.2.1. Definiranje stanja

Vektor stanja definira se na temelju sume euklidskih udaljenosti trenutnih pozicija predmeta od ciljnih prema jednadžbi (43).

$$\mathbf{s} = \sum_{n=1}^{N} \sqrt{\left[(\mathbf{x}_n^f - \mathbf{x}_n^c)^2 \right]}, \qquad n = 1.....N \tag{43}$$

$\mathbf{x}$ – vektor kordinata položaja predmeta u prostoru, (x_1, x_2)
$\mathbf{s}$ – vektor stanja

N predstavlja ukupan broj predmeta dok indeksi f i c predstavljaju konačnu i trenutnu poziciju predmeta. Prema objašnjenjima iz poglavlja 3.2.1.2. korišeni su Fourierovi redovi za definiranje komponenata vektora stanja prema jednadžbama (44) i (45):

$$\hat{q}(\mathbf{s}, \boldsymbol{\theta}) = \boldsymbol{\theta}^{\mathrm{T}} \phi(\mathbf{s}) = \sum_{i=1}^{n} \boldsymbol{\theta}_{\mathrm{i}} \phi_i(\mathbf{s}) \tag{44}$$

$$\phi(\mathbf{s}) = a_0 \cos(i\pi\mathbf{s}), \qquad i = 1....K \tag{45}$$

K predstavlja ukupan broj baznih funkcija. Iako se u navedenoj literaturi u poglavlju 3.2.1.2. navodi da je moguće uzeti $a_0 = 1$ u ovom konkretnom zadatku to nije bio slučaj te se nije mogla postići konvergencija rezultata uzimanjem vrijednosti parametara $a_0 \succ 0.4$.

4.2.2. Definiranje funkcije nagrade

Za svaku akciju koju poduzme agent odnosno za svako premještanje predmeta sa jedne na drugu poziciju agentu se dodijeljuje nagrada na osnovi svih predmeta u radnom prostoru prema skalarnoj funkciji definiranoj jednadžbom (46).

$$R = \sum_{i=1}^{N} \begin{cases} 1, \text{ ako se predmet nalazi u ciljnoj poziciji} \\ 0, \text{ ako se predmet ne nalazi ni u vlastitom niti ciljnom položaju bilo kojeg od preostalih predmeta} \\ -1, \text{ ako se predmet nalazi u ciljnoj poziciji nekog drugog predmeta} \end{cases}$$

(46)

4.2.3. Definiranje akcija

Mijenjanjem položaja pojedinog predmeta definirana je moguća akcija u danom koraku iteracije. Korištena je $\varepsilon - greedy$ metoda odabira akcija koja sa ε vjerojatnošću odabire slučajnu akciju dok sa $(1-\varepsilon)$ vjerojatnošću odabire akciju za koju agent dobiva najveću moguću nagradu. Dakle, ovdje se radi o *eng. exploration-explotation* dilemi. U slučaju odabira velike vrijednosti ε parametra mogućnost istraživanja se povećava odnosno agent pretražuje veći mogući prostor stanja dok se u obrnutom slučaju slijedi strategija čiji slijed akcija daje najveću nagradu agentu. Potrebno je ovisno o konkretnom slučaju naći kompromis te se u većini literature iz područja pojačanog učenja koriste vrijednosti parametra ε između [0.1, 0.2]. U svakom slučaju potrebno je provesti testiranje i istražiti utjecaj navedenog parametra na konkretnom slučaju. Nadalje, navodi se algoritam za definiranje akcija.

Definiranje akcija

Definiranje parametra ε

Definiranje mogućih akcija na osnovi trenutnih pozicija predmeta

Izbor nasumičnog broja r u intervalu [0, 1]

Ako je $r > \varepsilon$

- Za svaku moguću akciju izračunati skalarnu vrijednost nagrade na osnovi novog vektora stanja uz odabranu akciju

- U slučaju da više mogućih akcija daje agentu jednaku nagradu – za svaki predmet izračunati euklidsku udaljenost od nove pozicije do njegove ciljne pozicije te odabrati akciju koja daje najmanju vrijednost sume euklidskih udaljenosti svih predmeta
- U slučaju da samo jedna akcija daje najveću nagradu agentu nju odabrati

Ako je $r < \varepsilon$

Odabrati slučajnu akciju

4.2.4. Prva faza algoritma – treniranje parametara

Za testiranje konvergencije parametara kreirana su četiri algoritma. Prvi algoritam odnosi se On-policy jednokoračnu TD(0) metodu, dok se drugi odnosi na Off-policy jednokoračnu (eng. one-step) TD(0) metodu. Naredna dva su nastala na osnovi navedenih sa razlikom da su korištena dva koraka unaprijed za definiranje nagrade opisane prema jednadžbi (8). U nastavku će biti definirana dvokoračna TD(1) On-policy metoda te će za ostale metode u odnosu na ovu biti navedene razlike.

Treniranje parametara

Inicijalizacija vektora $\boldsymbol{\theta}$ (vektor slučajnih vrijednosti na intervalu [0, 1])

Definiranje parametara $\alpha,\ \gamma$

For petlja *n* epizoda:

Izaberi početne pozicije predmeta i akciju te odredi $\phi(\mathbf{s}_1)$

While petlja dok se ne zadovolji uvijet trenutno stanje == ciljno stanje:

Na osnovi izabranog stanja izabrati novu akciju i odrediti $a_1 \rightarrow \mathbf{s}_2,\ a_2,\quad \phi(\mathbf{s}_2)$

Na osnovi drugog izabranog stanja izabrati novu akciju i odrediti $a_2 \rightarrow \mathbf{s}_3,\ a_3,\quad \phi(\mathbf{s}_3)$

$$q_{1i}(\mathbf{s}_{1i}, a_{1i}) = \boldsymbol{\theta}_i \phi(\mathbf{s}_{1i})^T$$

$$q_{3i}(\mathbf{s}_{3i}, a_{3i}) = \boldsymbol{\theta}_i \phi(\mathbf{s}_{3i})^T$$

$$\delta_i = r_{12} + \gamma r_{23} + \gamma^2 q_{3i} - q_{1i}$$

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + \alpha \delta_i \phi(\mathbf{s}_{1i})$$

$$\mathbf{s}_{1i+1} = \mathbf{s}_{2i}$$

$$\phi(\mathbf{s}_{1i+1}) = \phi(\mathbf{s}_{2i+1})$$

Osnovna razlika u odnosu na jednokoračne metode očituje se na način da se prilikom treniranja parametara uzima u obzir jedan korak više za definiranje nagrade. Dok se osnovna razlika u odnosu na Off-policy metode očituje u odabiru posljednje akcije na način da se odabire ona akcija koja zadovoljava uvijet prikazan jednadžbom (47).

$$a \rightarrow \max_a q(\mathbf{s}_{i+1}, a) \tag{47}$$

4.2.5. Druga faza algoritma – predviđanje budućih akcija

U drugoj fazi na osnovi izračunatih parametara cilj je predvidjeti odnosno odrediti akcije na temelju trenutnih stanja u kojima se sustav nalazi. U končanici predviđanjem akcija dolazi se do krajnjeg odnosno ciljnog stanja.

Predviđanje budućih akcija

Uzeti vektor parametara $\boldsymbol{\theta}$ iz prve faze algoritma

Odredi početno stanje $\mathbf{s}_c$

While petlja dok se ne postigne ciljno stanje:

For petlja za svaki pojedini predmet:

For petlja za svaku moguću akciju:

Izračunati $q(\mathbf{s}, a) = \boldsymbol{\theta}\phi(\mathbf{s})^T$

Odrediti akciju na temelju uvijeta $a \rightarrow \max_a q(\mathbf{s}_{i+1}, a)$

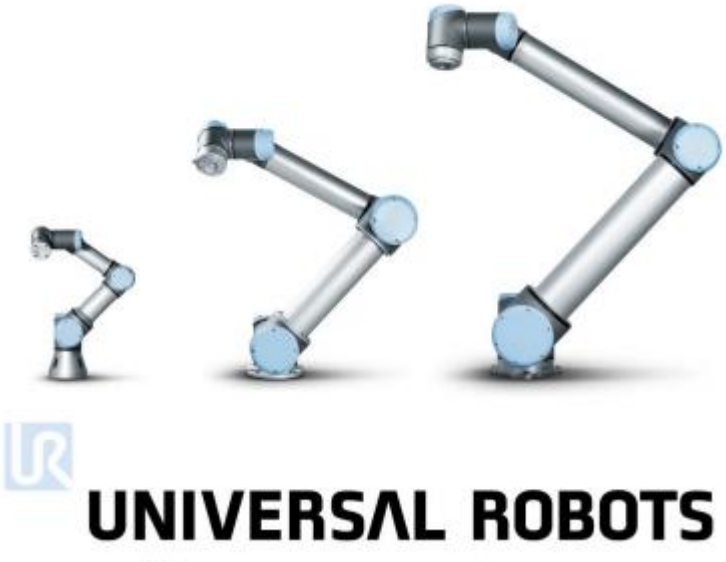
Odrediti novu poziciji predmeta za koju je zadovoljen uvijet predhodnog koraka

Odrediti novo stanje prema $\mathbf{s} = \sum_{n=1}^{N} \sqrt{\left[(\mathbf{x}_n^f - \mathbf{x}_n^c)^2 \right]}, \quad \phi(\mathbf{s})$

U konačnici se sustav dovodi iz nekog unaprijed zadanog do konačnog stanja, odnosno predmeti se određenim nizom akcija prateći optimalnu strategiju dovode iz početnih u ciljne pozicije.

5. Universal Robots

Tvrtku Universal Robots osnovali su Esben Østergaard, Kasper Støy i Kristian Kassow 2005. godine sa sjedištem u Odenseu u Danskoj. Njihova tri glavna proizvoda su roboti UR3, UR5 i UR10 slika (20). Brojke u imenu označavaju nosivost svakog robota u kilogramima. Svo troje navedenih, vrlo su lagani 6–osni roboti čije su težine redom 11kg, 18kg i 28kg. Svi zglobovi mogu se rotirati $\pm 360^0$ brzinom do $180^0 s^{-1}$, a zadnji zglob robota UR3 ima beskonačnu rotaciju. Točnost ponavljanja iznosi im $\pm 0.1mm$. UR3, UR5 i UR10, slika (17) su "Kolaborativni roboti" što znači da mogu raditi odmah uz osoblje bez sigurnosnih dodataka.



Slika 17 UR3, UR5 i UR10 [21]

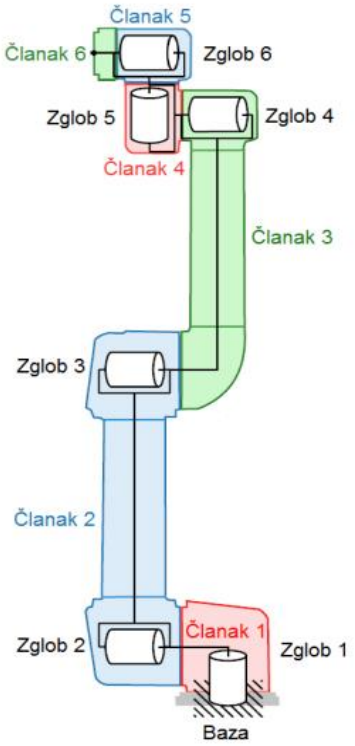
5.1. UR5 robot

Prilikom izrade ovog diplomskog rada korišten je UR5 robot, iako su kodovi navedeni u dodatku primjenjivi na sva tri robota. Njegove karakteristike navedene su na slici (18).

Težina:	18,4kg
Nosivost:	5kg
Doseg:	850mm
Radni opseg zglobova:	±360°
Brzina svih zglobova:	180°/s
Brzina alata:	1m/s
Ponovljivost:	±0, 1mm
Broj stupnjeva slobode gibanja:	6
Veličina upravljačke jedinice:	475mm × 423mm × 268mm
Komunikacija:	TCP/IP 100 Mbit & Modbus TCP
Programiranje:	Grafičko korisničko sučelje PolyScope
Potrošnja energije:	≈ 200W
Materijali:	Aluminij i Polipropilen (PP)
Temperaturno radno područje:	0°C do 50°C
Napajanje:	100 – 240 VAC, 50 – 60Hz

Slika 18 UR5 Tehničke specifikacije

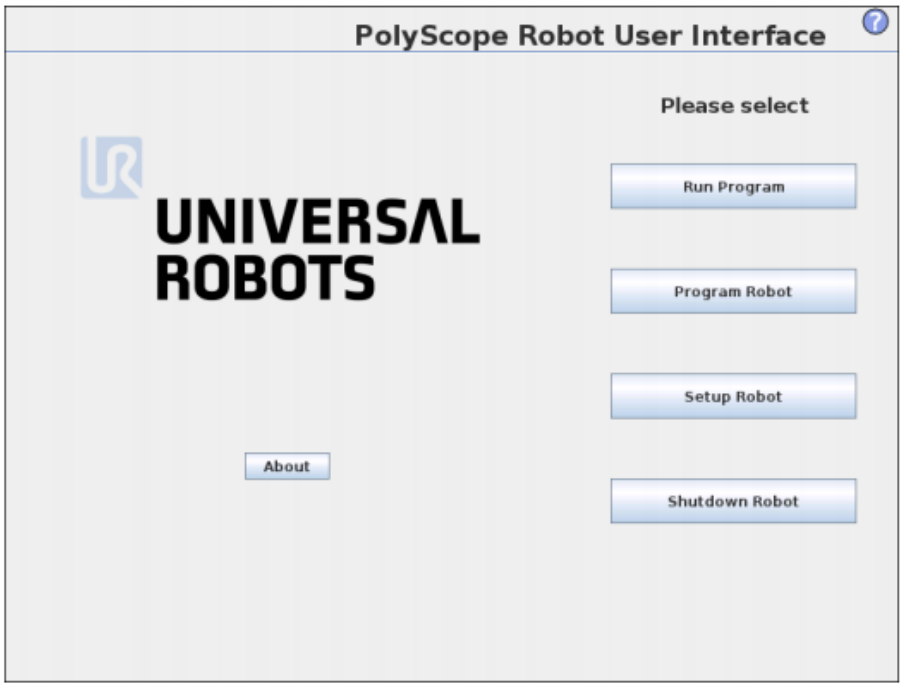
Slika (19) prikazuje sve zglobove i segmente navedenog robota.



Slika 19 Zglobovi i segmenti robota UR5 [25]

5.1.1. Programiranje robota

UR robote je moguće programirati putem privjeska za učenje, skripte ili putem C–API sučelja. Privjesak za učenje se sastoji od 12–inčnog ekrana osjetljivog na dodir, sigurnosne tipke i jednog USB porta u koji je moguće uštekati tipkovnicu, miš ili USB stick. Korisničko sučelje u kojem se robot programira naziva se PolyScope i prikazano je na slici (20). Sučelje je vrlo jednostavno i intuitivno. Za razliku od ostalih industrijskih robota gdje je potrebno proći obuku za rad s robotom, kod UR robota to nije slučaj. Korisnik ovdje može u malom vremenu savladati osnove i izraditi neke jednostavne programe. Programiranje robota preko skripte je zahtjevnije i potrebno je znanje iz programiranja i poznavanje robotovog programskog jezika URScript. U ovom radu robot je programiran preko skripte tako da se naredbe ili cijeli program na njega šalje putem TCP protokola.



Slika 20 PolyScope korisničko sučelje

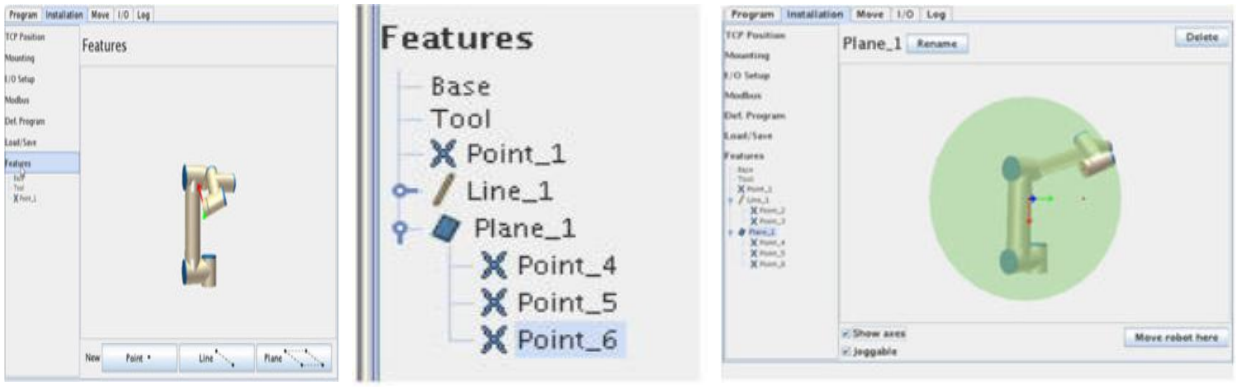
5.1.2. Komunikacija s robotom

Za komunikaciju s robotom korišten je sekundarni server (port 30002). Sekundarni server služi za programiranje i dobivanje informacija o stanju robota brzinom 10Hz. Sekundarni server služi samo za slanje stanja robota. Robotom je moguće upravljati preko tog servera tako da mu se šalju naredbe u obliku skripte. To može biti jedna naredba ili cijeli program. Pojedinačne naredbe koje se pošalju robotu na taj port on izršava trenutno bez obzira je li u

toku rada ili miruje. Ukoliko se robotu šalje cijeli program, on ga izvršava nakon što je primio zadnji red programa, tj. naredbu end. Bitno je napomenuti da svaka naredba mora završavati sa „\n" što ustvari predstavlja novi red, tj. enter. [21]

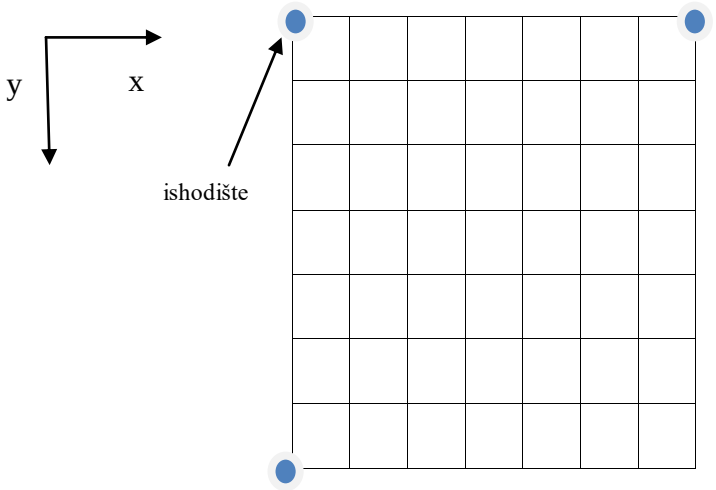
5.1.3. Definiranje ravnine radne okoline

Prilikom slanja pozicija vrha alata (hvataljke) UR robotu potrebno je definirati matricu transformacije iz ravnine radne okoline u baznu ravninu robota. Ravnina radne okoline definira se sa tri točke u prostoru do koje je potrebno dovesti vrh hvataljke uz predhodno definiranu kalibraciju alata, slika(22).



Slika 21 Definiranje ravnine radne okoline [26]

Prilikom odabira tri točke (prva-ishodište, druga-y_smjer, treća-x_smjer) definira se ishodište te smjerovi *y* i *x* osi ravine. Na temelju kreirane ravnine potrebno je očitati parametre translacije i rotacije u odnosu na baznu ravninu robota te kreirati matricu transformacije objašnjenu u poglavljima 3.5 i 3.6. Prilikom slanja naredbi robotu odnosno koordinata hvataljke, koordinate se šalju u baznoj ravnini robota.



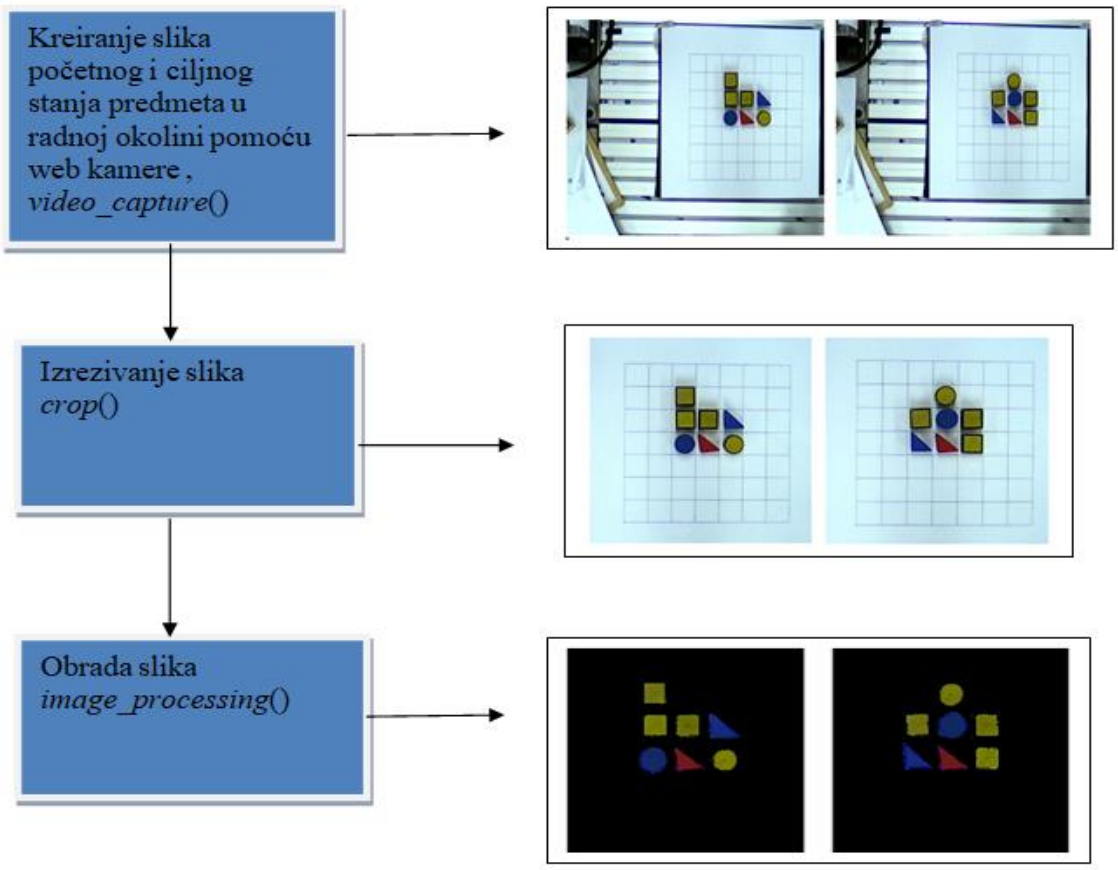
Slika 22 Prikaz odabira točaka za kreiranje ravnine radne okoline

6. Upravljački program

Pod pojmom upravljački program podrazumijeva se niz funkcija koje se pozivaju iz glavne (*eng. main*) funkcije te određenim redoslijedom izvršavaju naredbe potrebne kako bi robot obavio svoju zadaću. U nastavku će biti spomenute sve funkcije i algoritmi korišteni prilikom izrade ovog zadatka uz prikaz programskih kodova spomenutih u dodatku A.

6.1. Kreiranje i obrada slika

Prije definiranje glavne strukture upravljačkog algoritma potrebno je definirati funkcije za kreiranje slika pomoću web kamere, izrezivanje (*eng. crop*) dijela slike koji predstavlja radnu okolinu te obrade slika, dodaci A.1., A.2., A.3.. Kako bi se u upravljačkom algoritmu mogle pronaći samo konture predmeta potrebno je obraditi slike na način da cijela radna površina bude crne boje kako se nebi mogla raspoznati sahovnica na bijeloj podlozi već samo predmeti obojeni odgovarajućim bojama prema slici (23).



Slika 23 Kreiranje i obrada slika

6.2. Struktura upravljačkog algoritma

Unutar strukture glavnog algoritma definiranju se trenutne i konačne pozicije svih predmeta te se pomoću Kuhn-Munkresovog algoritma za određene predmete u početnim pozicijam pridodjeljuju konačne pozicije. Nadalje, definira se prva faza algoritma pojačanog učenja (treniranje parametara) kojim se nakon konvergencije dobivaju vrijednosti vektora $\boldsymbol{\theta}$. Dobiveni vektor $\boldsymbol{\theta}$ služi kao ulaz u drugi dio algoritma (predviđnje budućih akcija) .

Na osnovi drugog dijela algoritma dobiva se niz akcija koje se pomoću TCP protokola šalju robotu.

6.2.1. Pronalazak kontura predmeta (frame_filter.py)

Unutar *frame_filter.py* python datoteke (dodatak A.4.) definiran je razred *FrameFilter* sa svim potrebnim metodama za određivanje trenutnih i konačnih pozicija svakog pojedinog predmeta. Slike kreirane funkcijom *image_processing()* služe kao ulazni parametri *FrameFilter* konstruktora. Pošto je nad objektima slika nužno provesti nekoliko metoda korišten je koncep objektno orjentiranog programiranja. Nadalje bit će navedena objašnjenja pojedinih metoda unutar razreda *FrameFilter*.

- *load_picture()* – učitavanje slike
- *binary_image()* – prevaranje slike u binary format
- *find_conturs()* – pronalazak kontura
- *detect()* – definiranje oblika predmeta na osnovi pronađenih kontura
- *colision()* – definiranje odstupanja centra predmeta od centra polja kojem pripada u mm
- *process()* – glavna metoda koja poziva sve ostale metode te vraća rječnik (eng. dictionary) sa parovima - vrsta predmeta:pozicije (*eng. key:value*) te odstupanjima centra pojedinog predmeta od centra polja kojem pripada

6.2.2. Kuhn-Munkres algoritam (munkres.py)

Na osnovi Kuhn-Munkresovog algoritma svakom pojedinom predmetu pridodijeljena je ciljna pozicija. Kreiran je razred *MunkresClass*. Kao ulazni parametar konstruktora uzima se rječnik dobiven iz predhodnog koraka te se nakon poziva *process()* metode kao rezultat dobiva također rječnik unutar kojeg se nalaze parovi kao i u ulaznom rječniku s tom razlikom da su

promjenjene ciljne pozicije pojedinim predmetima. Euklidska udaljenost između početne i mogućih ciljnjih pozicija pojedinog predmeta uzima se kao ulazni parametar.

6.2.3. Treniranje parametara (SARSA1.py)

Za treniranje parametara korištena su četiri različita algoritma objašnjena u poglavlju 4.2. Kao što će biti objašnjeno u narednim poglavljima SARSA1 algoritam se pokazao najprikladniji za ovaj konkretan slučaj te je njegova implementacija do u detalje objašnjena u poglavlju 4.2.4. aprogramski kodovi su dani u dodatku A.6.

6.2.4. Predviđanje budući akcija (predict.py)

Za predviđanje budućih akcija prikazana je implementacija u dodatku A.7. te detaljno objašnjenje algoritma nalazi se u poglavlju 4.2.5.

6.2.5. Slanje naredbi UR robotu (robot_script.py)

Za slanje naredbi robotu (dodatak A.8.) kreiran je razred *RobotCommands* koji sadrži metode *transformation()* i *move_robot()*. Pomoću prve navedene metode transformiraju se koordinate iz ravnine radne okoline u baznu ravninu robota, dok se pomoću druge navedene metode direktno šalju naredbe robotu. Korištene su dvije vrste naredbi. Prva se odnosi na slanje pozicije pneumatske hvataljke robota u baznoj ravnini robota pomoću naredbe *movel*, dok se druga odnosi na aktivaciju (set_digital_out(0,True)) iste prilikom uzimanja predmeta odnosno deaktivaciju (set_digital_out(0,False)) prilikom ispuštanja predmeta. Unutar metode *move_robot()* definirano je osam akcija koje robot izvodi te one glase:

1. Dolazak do predmeta na visini tri puta od visine samog predmeta
2. Spuštanje po predmet na visinu samog predmeta
3. Uzimanje predmeta
4. Podizanje predmeta na visinu tri puta od visine samog predmeta
5. Prijenos predmeta do novog položaja
6. Spuštanje predmeta na visinu samog predmeta
7. Ispuštanje predmeta
8. Podizanje na visinu tri puta od visine samog predmeta

6.2.6. Popratne funkcije (utils.py)

Unutar datoteke utils.py nalaze se funkcije za definiranje novog stanja *feature()*, akcije *chooseAction()*, nagrade *reward()*, računanje euklidske udaljenosti *distance()*, za izračun vrijednosti privremenih razlika *delta()* kod SARSA0 te Q(0) algoritma te *delta_two_steps()* za izračun vrijednosti privremenih razlika kod SARSA1 te Q(1) algoritma (dodatak A.9.).

6.2.7. Glavna funkcija upravljačkog algoritma (main.py)

U datoteci main.py nalazi se funkcija koja je odgovorna za pravilan redoslijed izvođenja svih ranije navedenih funkcija i algoritama. Unutar nje se nalazi redoslijed poziva istih te su navedene sve konstante potrebne za izvođenje programa.

7. Rezultati

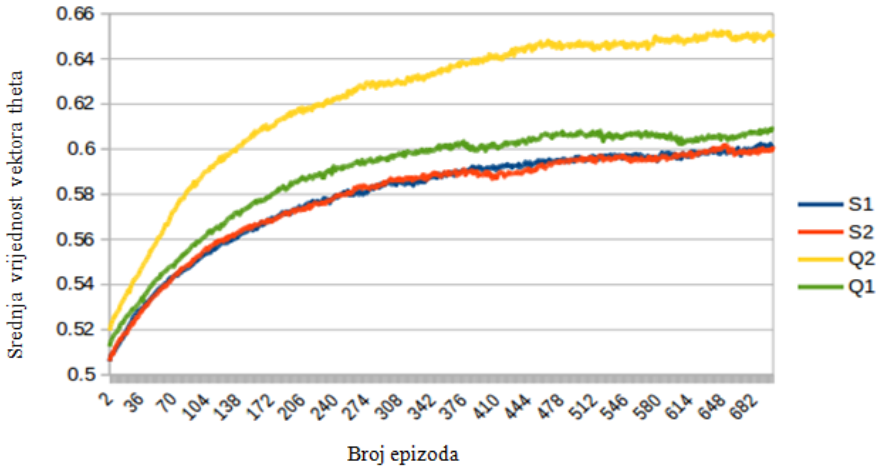
U ovom poglavlju biti će prikazana usporedba algoritama za slučaj opisan u poglavlju 7.2.2. te će biti prikazan niz akcija za dva praktična primjera na kojima su algoritmi testirani.

7.1. Usporedba algoritama

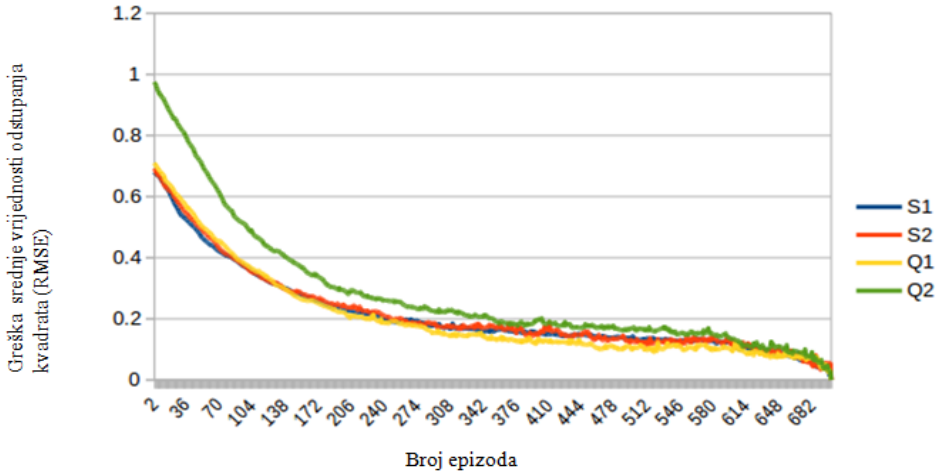
α se koristi pri izračunu vrijednosti parametara za svaku iteraciju prema $\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + \alpha G_i \phi(\mathbf{s}_{1i})$. Vrijednosti parametra α kreće se u rasponu [0, 1]. Parametar γ kreće se također u intervalu [0, 1]. Na temelju izvršenih eksperimenata odabrane su vrijednosti parametara $\alpha = 0.4$ i $\gamma = 0.6$. Problem kod provjere konvergencije u pojačanom učenju za razliku od učenja sa učiteljem (*eng. supervised learning*) očituje se u nedostatku ciljne varijable na temelju koje možemo prema određenoj metrici odrediti točnost pojedinog algoritma stoga je potrebno napraviti kompromis odnosno uzeti određene pretpostavke u obzir. Dakle, za provjeru točnosti koristit će se greška srednje vrijednosti odstupanja kvadrata (RMSE) između konačne i trenutne vrijednosti vektora parametara na način da se za konačnu vrijednost vektora parametara uzima vrijednost nakon koje je zadovolje kriterij konvergencije. Uspoređivat će se sljedeći algoritmi:

S1 – On-policy TD(0) metoda, (SARSA0)
S2 – On-policy TD(1) metoda, (SARSA1)
Q1 – Off-policy TD(0) metoda, (Q-learning(0))
Q2 – Off-policy TD(1) metoda,(Q-learning(1))

Na slikama (24) i (25) prikazana je usporedba sva četiri navedena algoritma te se na osnovi dijagrama prikazanih na tim slikama može zaključiti da algoritmi S1 i S2 imaju najbržu konvergenciju uz najmanje oscilacije srednje vrijednosti vektora $\boldsymbol{\theta}$.



Slika 24 Usporedba promjene srednje vrijednosti vektora theta za sve navedene algoritme



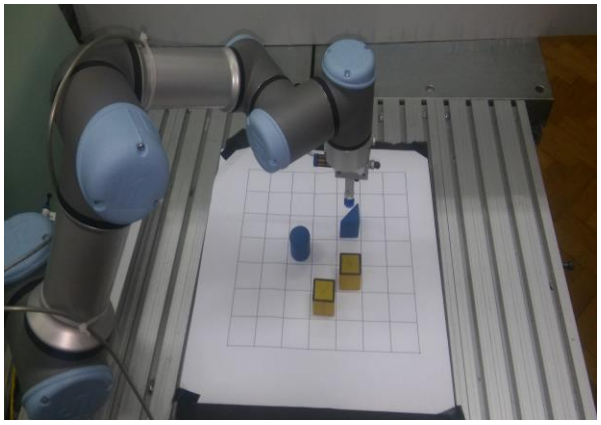
Slika 25 Usporedba promjene greške srednje vrijednosti odstupanja kvadrata za sve navedene algoritme

Najmanje oscilacije u rješenjima govore o pouzdanosti navedenih algoritama, a to se u konačnici očituje u drugoj fazi algoritma odnosno u predviđanju optimalnog slijeda akcija odnosno optimalne strategije. Naime, algoritmi Q1 i Q2 su za sve navedene promjene parametara u konačnici rezultirali najmanje jednom akcijom više od algoritama S1 i S2. Pošto je cilj zadatka u najmanjem mogućem broju koraka doći od početnog do konačnog stanja u tom slučaju algoritmi S1 te S2 su se pokazali podjednako korisni odnosno daju jednaka rješenja niza akcija za pojedine slučajeve te će jedan od njih biti prikazan u sljedećem podpoglavlju.

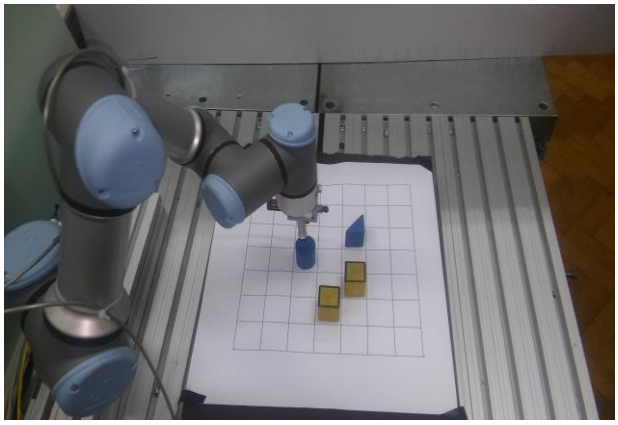
7.2. Prikaz djelovanja robota na temelju dva primjera

U ovom podpoglavlju biti će prikazan niz slika koje u cjelini prikazuju niz akcija koje robot izvodi za svaki pojedini primjer uz primjenu S2 algoritma. Prva slika prikazuje početno, dok zadnja konačno stanje predmeta u radnoj okolini.

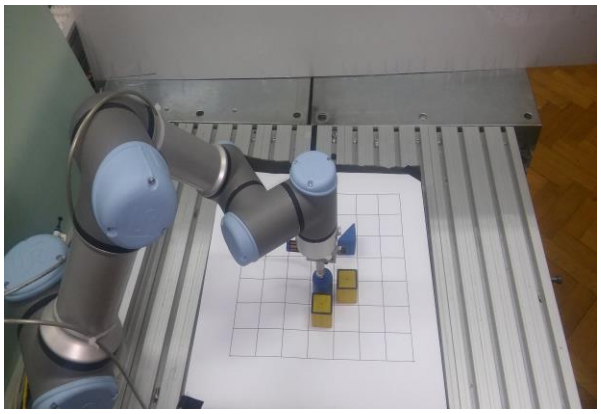
7.2.1. Primjer 1



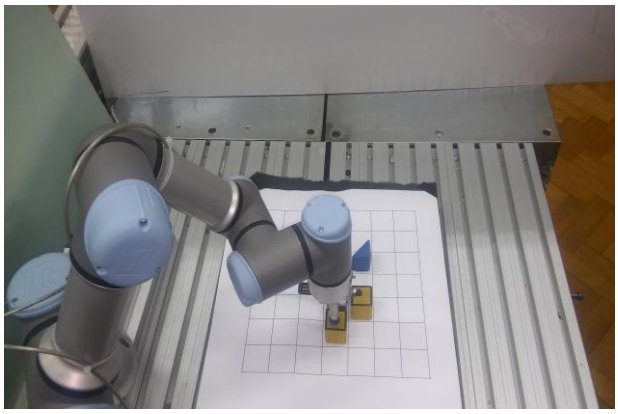
a)



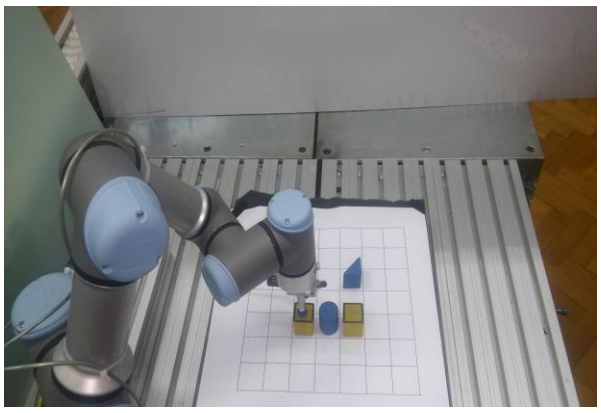
b)



c)

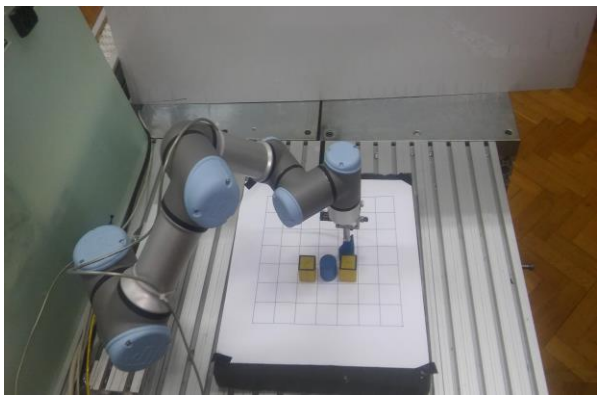


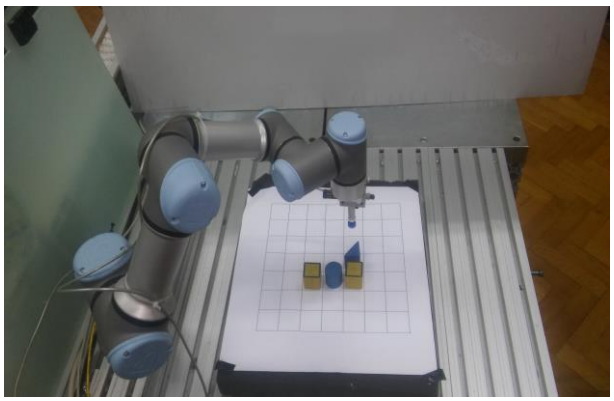
d)



e)

f)

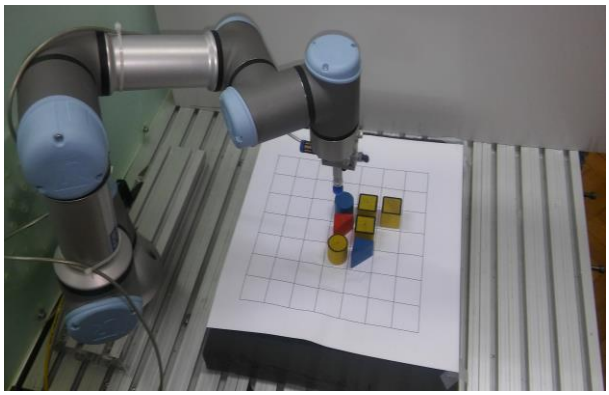


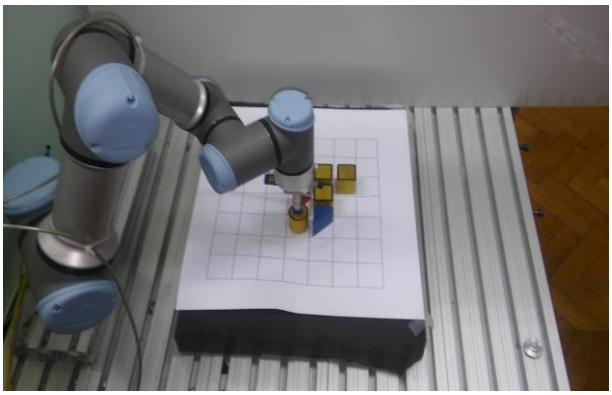


g) h)

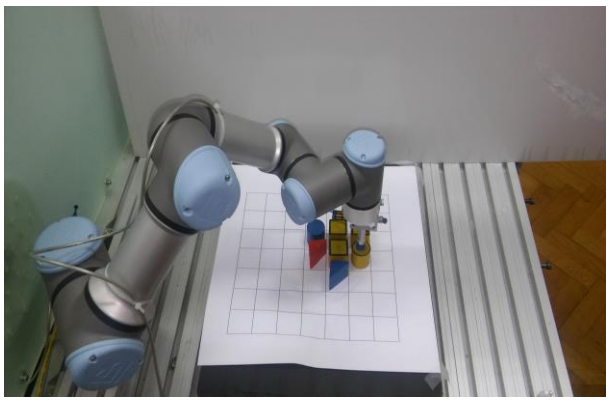
Slika 26 Slika a) prikaz predmeta u početnom stanju, slike od b) do g) prikaz akcija koje izvodi robot, slika h) prikaz predmeta u konačnom stanju

7.2.2. Primjer 2

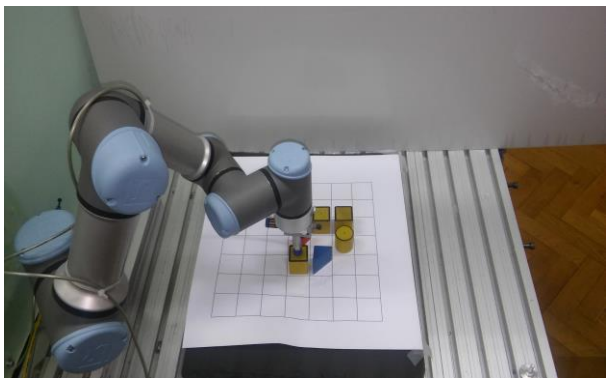




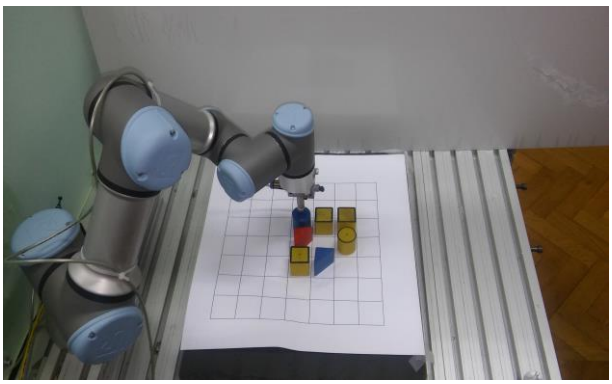
a) b)



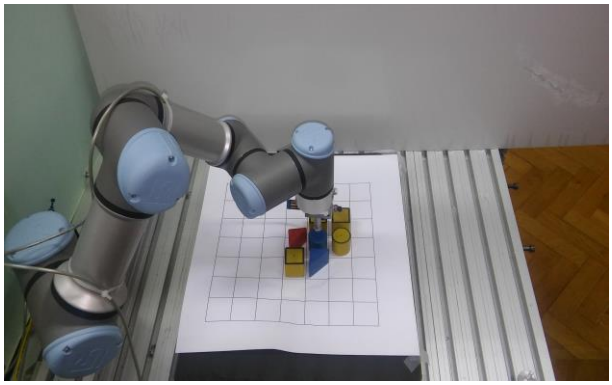
c) d)



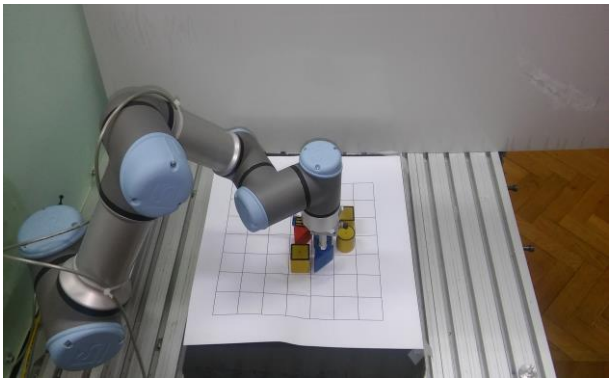
e)



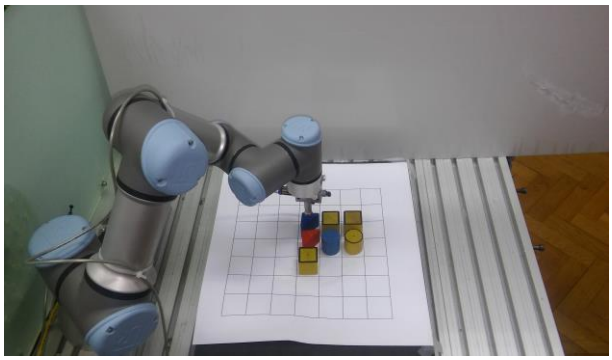
g)



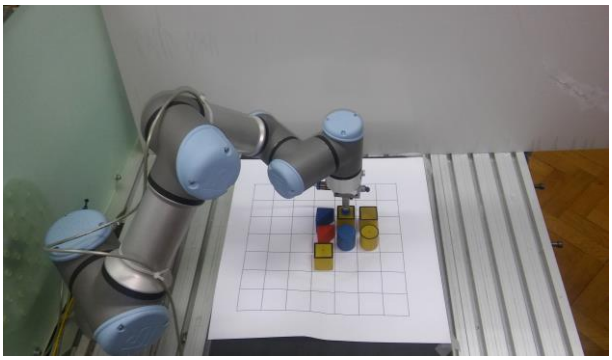
h)



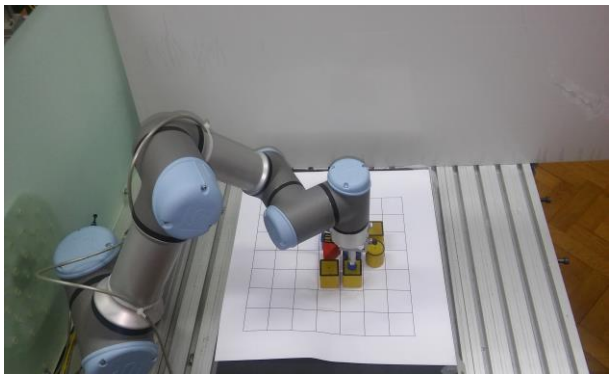
i)



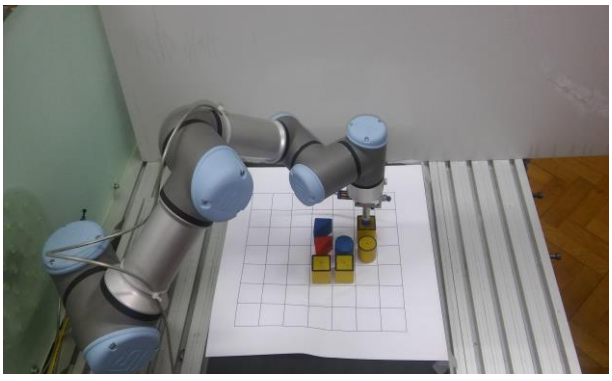
j)



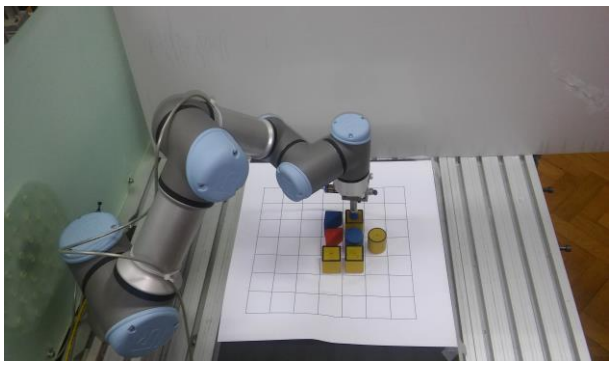
k)



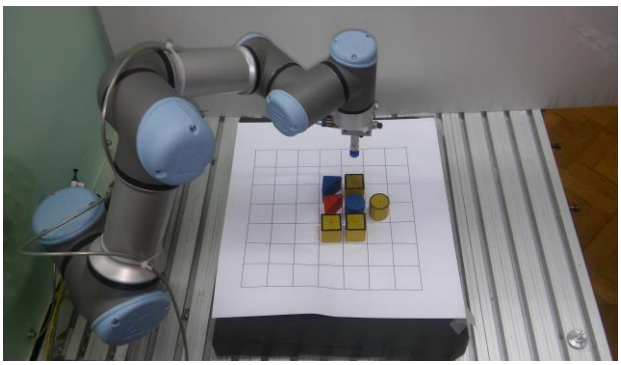
l)



m)



n)



o)

Slika 27 Slika a) prikaz predmeta u početnom stanju, slike od b) do n) prikaz akcija koje izvodi robot, slika o) prikaz predmeta u konačnom stanju

Na danom primjeru korištene su tri različite vrste objekata: valjak, prizma te kocka. Prvo se određuje koliko pojedinih objekata pripada kojoj klasi te se pomoću Mađarske metode pridodijeljuju određene pozicije svakom od njih iz unaprijed određenog skupa konačnih pozicija. Na temelju odabranih početnih i konačnih pozicija svakog predmeta na osnovi dobivene strategije izvršava se manipulacija nad objektima. Točnost u prvom redu ovisi o funkciji nagrade, o načinu definiranja stanja, vrsti algoritma te definiranju i odabiru akcija.

Na osnovi slika (26) i (27) može se zaključiti da je navedeni zadatak u potpunosti obavljen odnosno da je robot slijedio strategiju defniranu algoritmom S2 uz vrijednosti navedenih parametara iz predhodnog poglavlja.

8. Zaključak

Na temelju ovog istraživanja pokazalo se da pojačano učenje daje pouzdane rezultate u području planiranja robotskog djelovanja. Imajući u vidu problem prevelikog broja diskretnih stanja i akcija koriste se aproksimacijske funkcije. U ovom konkretnom zadatku korištene su linearne bazne funkcije. Kao što je objašnjeno u poglavlju 3.2. postoje brojne mogućnosti odabira baznih funkcija no međutim u ovom slučaju korišteni su Fourerovi redovi jer su se prema literaturi [20] pokazali kao odličan izbor. Uz pravilno definiranje stanja te funkcije nagrade agent je naučen izvesti niz akcija s ciljem postizanja prvilnog rasporeda geometrijskih objekata u radnoj okolini uz najmanji mogući broj koraka odabirući pritom akcije koje minimiziraju udaljenosti koje robot mora prijeći. Zadnja tvrdnja potkrepljena je činjenicom da agent odabire akcije na način objašnjen u poglavlju 4.2.3. Na temelju eksperimenata odabrane su optimalne vrijednosti parametara $\alpha = 0.4$ te $\gamma = 0.6$. Također pokazalo se da algoritmi koji koriste On-policy metode daju kvalitetnija rješenja u odnosu na algoritme koji koriste Off-policy metode. To se očituje u predviđanju budućih akcija na način da algoritmi temeljeni na On-policy metodama u manjem broju koraka dolaze do unaprijed definiranog ciljnog stanja što je bio glavni kriterij odabira algoritma.

9. Budući rad

Na temelju kreiranih algoritama za rješavanje problema planiranja robotskog djelovanja moguća su proširenja u vidu definiranja stanja predmeta koji mogu zauzimati više od jednog polja. U tom slučaju potrebno je uzeti u obzir i orijentacije predmeta te površine koje pojedini predmeti mogu zauzimati kako bi se mogao pravilno definirati vektor stanja koji u tom slučaju osim odstupanja u vidu translacije sadrži i komponente kutnog odstupanja. Ako se uzme u obzir on-line učenje primjerice, funkcija nagrade mogla bi se bazirati na sličnosti slika konačnog te trenutnog stanja predmeta nakon svake akcije koju agent poduzme. U tom slučaju bilo bi potrebno nadograditi postojeći algoritam sa algoritmima učenja sa učiteljem primjerice Umjetnim neuronskim mrežama. Također u vidu proširenja zadatka postoji mogućnost nadogradnje algoritma za 3D slučaj u kojem bi postojala mogućnost kreiranja trodimenzionalnih struktura. U tom slučaju javlja se potreba za kreiranjem boljih rješenja u vidu vizijskog sustava koji bi mogao raspoznavati 3D strukture. Također problem se može modelirati direktnim traženjem strategije uz adaptivnu mogućnost parametrizacije u toku učenja, odnosno njenom inicijalizacijom i optimizacijom. U konačnici postoji mnogo mogućnosti daljnjeg razvoja algoritma u kombinaciji sa drugim algoritmima iz područja umjetne inteligencije.

LITERATURA

[1] Švaco M.: Planiranje robotskog djelovanja zasnovano na tumačenju prostornih struktura, Doktroski rad, 2015.

[2] Ekvall S., Kragić D.: Robot Learning from Demonstration: A Task-level Planning Approach, Article

[3] Minoru Asada, Shoichi Noda, Sukoya Tawaratsumida, and Koh Hosoda: Vision Based Reinforcement Learning for Purposive Behavior Acquisition, Article, 1995.

[4] John Demiris and Gillian Hayes: Imitative Learning Mechanismsin Robots and Humans

[5] Jens Kober J., Andrew Bagnell, Jan Peters: Reinforcement Learning in Robotics: A Survey, Article, 2013.

[6] Bram Bakker, Schmidhuber J.: Hierarchical Reinforcement Learning Based on Subgoal Discovery and Subpolicy Specialization

[7] Eric Brochu, Vlad M. Cora and Nando de Freitas,: A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning

[8] Jun Morimoto, Kenji Doya: Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning, 2001.

[9] Richard S. Sutton and Andrew G. Barto: Reinforcement Learning: An Introduction, 2016.

[10] Cang Ye, Nelson H. C. Yung, Danwei Wang, Member: A Fuzzy Controller With Supervised Learning Assisted Reinforcement Learning Algorithm for Obstacle Avoidance, 2003.

[11] Andrew Ng: Reinforcement Learning and Control, CS229 Lecture notes, Part XIII

[12] Marc Peter Deisenroth, Gerhard Neumann and Jan Peters: A Survey on Policy Search for Robotics, 2013.

[13] Petar Kormushev, Sylvain Calinon, Darwin G. Caldwell: Reinforcement Learning in Robotics: Applications and Real-World Challenges,2013. Article

[14] Guenter, F.; Hersch, M.; Calinon, S.; Billard, A. Reinforcement learning for imitating constrained reaching movements. Adv. Robot. 2007

[15] Kober, J.; Peters, J. Policy search for motor primitives in robotics. In Advances in Neural Information Processing Systems; MIT Press: Cambridge, MA, USA, 2009; Volume 21, pp. 849–856.

[16] Said G. Khan, Guido Herrmann, Frank L. Lewis, Tony Pipe, Chris Melhuish: Reinforcement learning and optimal adaptive control: An overview and implementation examples, 2012.

[17] Dan Zhang, Bin Wei: On the Development of Learning Control for Robotic Manipulators, 2017.

[18] Zhijiang Du, Wei Wang, Zhiyuan Yan, Wei Dong and Weidong Wang: Variable Admittance Control Based on Fuzzy Reinforcement Learning for Minimally Invasive Surgery Manipulator, Article, 2017.

[19] Filip Šuligoj, Marko Švaco, Bojan Jerbić : Automated Marker Localization in the Planning Phase of Robotic Neurosurgery, 2017.

[20] George Konidaris, Scott Kuindersma, Roderic Grupen, Andrew Barto: Autonomous Skill Acquisition on a Mobile Manipulator

[21] Nikola Kirić: Primjena 3D vizijskog sustava za praćenje objekata robotom u realnom vremenu, Diplomski rad, 2015.

[22] TCP, https://hr.wikipedia.org/wiki/TCP , 9.1.2018.

[23] TCP protokol, http://mreze.layer-x.com/s040100-0.html , 9.1.2018.

[24] Yin, S., Ren, Y., Zhu, J., Yang, S., Ye, S.: A Vision-Based Self-Calibration Method for Robotic Visual Inspection Systems, Sensors, stranice 16571–16572, 2013.

[25] Kufieta, K.: Force Estimation in Robotic Manipulators: Modeling, Simulation and Experiments, NTNU Norwegian University of Science and Technology, 2014.

[26] http://www.zacobria.com/universal-robots-zacobria-forum-hints-tips-how-to/features-screen/ , 9.1.2018.

PRILOZI

A. Programski kod

B. CD-R disc

A. Programski kod

A.1. video_capture.py

```
import cv2
import time

stanje = "pocetno" # konacno

def video_capture(stanje):
    cam = cv2.VideoCapture(1)
    cv2.namedWindow("test")
    img_counter = 0
    time.sleep(3)
    for i in range(1):
        ret, frame = cam.read()
        cv2.imshow("test", frame)
        img_name = "{}_stanje.png".format(stanje)
        cv2.imwrite(img_name, frame)
        img_counter += 1
    cam.release()
    cv2.destroyAllWindows()
```

A.2. crop.py

```
import argparse
import cv2


stanje = "pocetno" # konacno

def crop(stanje):
    refPt = []
    cropping = False
    def click_and_crop(event, x, y, flags, param):
        global refPt, cropping
        if event == cv2.EVENT_LBUTTONDOWN:
            refPt = [(x, y)]
            cropping = True
        elif event == cv2.EVENT_LBUTTONUP:
            refPt.append((x, y))
            cropping = False
            cv2.rectangle(image, refPt[0], refPt[1], (0, 255, 0), 2)
            cv2.imshow("image", image)
    ap = argparse.ArgumentParser()
    ap.add_argument("-i", "--image", required=True, help="Path to the image")
    args = vars(ap.parse_args())
    image = cv2.imread(args["image"])
    clone = image.copy()
    cv2.namedWindow("image")
    cv2.setMouseCallback("image", click_and_crop)
    while True:
        cv2.imshow("image", image)
        key = cv2.waitKey(1) & 0xFF
        if key == ord("r"):
            image = clone.copy()
        elif key == ord("c"):
            break
    if len(refPt) == 2:
        roi = clone[refPt[0][1]:refPt[1][1], refPt[0][0]:refPt[1][0]]
        cv2.imshow("ROI", roi)
        img_name = "roi_{}.png".format(stanje)
        cv2.imwrite(img_name, roi)
        cv2.waitKey(0)
    cv2.destroyAllWindows()
```

A.3. image_processing.py

```
import cv2
import numpy as np

stanje = "pocetno" # konacno

def callback():
    pass

def image_processing():
    cv2.namedWindow('image')
    ilowH = 0
    ihighH = 179
    ilowS = 0
    ihighS = 255
    ilowV = 0
    ihighV = 255
    cv2.createTrackbar('lowH','image',ilowH,179,callback)
    cv2.createTrackbar('highH','image',ihighH,179,callback)
    cv2.createTrackbar('lowS','image',ilowS,255,callback)
    cv2.createTrackbar('highS','image',ihighS,255,callback)
    cv2.createTrackbar('lowV','image',ilowV,255,callback)
    cv2.createTrackbar('highV','image',ihighV,255,callback)
    while(True):
        frame = cv2.imread("roi_{}.png".format(stanje))
        ilowH = cv2.getTrackbarPos('lowH', 'image')
        ihighH = cv2.getTrackbarPos('highH', 'image')
        ilowS = cv2.getTrackbarPos('lowS', 'image')
        ihighS = cv2.getTrackbarPos('highS', 'image')
        ilowV = cv2.getTrackbarPos('lowV', 'image')
        ihighV = cv2.getTrackbarPos('highV', 'image')
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        lower_hsv = np.array([ilowH, ilowS, ilowV])
        higher_hsv = np.array([ihighH, ihighS, ihighV])
        mask = cv2.inRange(hsv, lower_hsv, higher_hsv)
        frame = cv2.bitwise_and(frame, frame, mask=mask)
        cv2.imshow('image', frame)
        k = cv2.waitKey(1000) & 0xFF
        if k == 27:
            cv2.destroyAllWindows()
        elif k == ord('s'):
            cv2.imwrite("obradena_slika_{}.png".format(stanje), frame)
            cv2.destroyAllWindows()
```

A.4. frame_filter.py

```
import sys
sys.path.pop(3)
import cv2
import numpy as np


class FrameFilter:

    def __init__(self, t, fileName, n):
        self.t = t
        self.fileName = fileName
        self.n = n

    def load_picture(self):
        img = cv2.imread(self.fileName)
        height = np.size(img, 0)
        width = np.size(img, 1)
        height_vector = np.linspace(0, height, num=self.n + 1)
        width_vector = np.linspace(0, width, num=self.n + 1)
        return height_vector, width_vector, img

    def states(self):
        s = []
        for i in range(self.n):
            for j in range(self.n):
                b = []
                b.append(i)
                b.append(j)
                s.append(b)
        return s

    def binary_image(self, img):
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        blur = cv2.GaussianBlur(gray, (5, 5), 0)
        (self.t, binary) = cv2.threshold(blur, self.t, 255, cv2.THRESH_BINARY)
        return binary

    def find_conturs(self, binary):
        (_, contours, _) = cv2.findContours(binary, cv2.RETR_EXTERNAL,
                                            cv2.CHAIN_APPROX_SIMPLE)
        centroid = dict()
        for z, j in enumerate(contours):
            h = []
            M = cv2.moments(j)
            cx = int(M['m10'] / M['m00'])
            cy = int(M['m01'] / M['m00'])
            h.append(cx)
            h.append(cy)
            centroid[z] = h
        return contours, centroid

    def numerical_order_of_states(self, centroid, height_vector, width_vector):
        coordinates = dict()
        for i in centroid:
            vec_x = int
            vec_y = int
            x = centroid[i][0]
            y = centroid[i][1]
            for j in range(0, self.n):
                if y < width_vector[j + 1] and y > width_vector[j]:
                    vec_y = j
                if x < height_vector[j + 1] and x > height_vector[j]:
                    vec_x = j
            coordinates[i] = [vec_y, vec_x]
```

```
        return coordinates

    def real_states(self, coordinates, states):
        real_states = []
        for ii in range(len(coordinates)):
            for ij in range(len(states)):
                if coordinates[ii][0] == states[ij][0] and coordinates[ii][1] ==
states[ij][1]:
                    real_states.append(ij)
        return real_states

    def detect(self, c):
        peri = cv2.arcLength(c, True)
        approx = cv2.approxPolyDP(c, 0.04 * peri, True)
        if len(approx) == 3:
            shape = "triangle"
        elif len(approx) == 4:
            (x, y, w, h) = cv2.boundingRect(approx)
            ar = w / float(h)
            shape = "square" if ar >= 0.90 and ar <= 1.1 else "rectangle"
        else:
            shape = "circle"
        return shape

    def shapes(self, contours, real_states):
        current_state = dict()
        shapes = []
        count_s = 0
        count_r = 0
        count_c = 0
        count_t = 0
        for index, c in enumerate(contours):
            shape = self.detect(c)
            shapes.append(shape)
            if "square" in (shapes):
                current_state[str(shape) + str(count_s)] = real_states[index]
                count_s += 1
            elif "rectangle" in (shapes):
                current_state[str(shape) + str(count_r)] = real_states[index]
                count_r += 1
            elif "circle" in (shapes):
                current_state[str(shape) + str(count_c)] = real_states[index]
                count_c += 1
            else:
                current_state[str(shape) + str(count_t)] = real_states[index]
                count_t += 1
        return current_state

    def munkres_states(self, current_state):
        munkres = dict()
        circle = []
        square = []
        rectangle = []
        triangle = []
        for key, value in current_state.items():
            if key[0] == "c":
                circle.append(value)
            if key[0] == "s":
                square.append(value)
            if key[0] == "r":
                rectangle.append(value)
            if key[0] == "t":
                triangle.append(value)
        items = [circle, square, rectangle, triangle]
        sorted_items = sorted(items, key=lambda x: len(x), reverse=True)
```

```
        for i in range(len(sorted_items)):
            if len(sorted_items[i]) > 0:

                munkres[i] = sorted_items[i]
        return munkres

    def colision(self, centroid, real_states, height_vector, width_vector):
        diffe = dict()
        for i in range(len(real_states)):
            for j in range(self.n):
                for k in range(self.n):
                   if real_states[i] == ((j*self.n) + k):
                       center_x = (height_vector[k+1] - height_vector[k])/2 +
height_vector[k]
                       center_y = (width_vector[j+1] - width_vector[j])/2 +
width_vector[j]
                       cx = centroid[i][0]
                       cy = centroid[i][1]
                       diff_x = cx - center_x
                       diff_y = cy - center_y
                       diffe[real_states[i]] = [diff_x, diff_y]
        x_mm = (height_vector[-1] / self.n) / 4
        y_mm = (width_vector[-1] / self.n) / 4
        diff_mm = dict()
        for key, pixel in diffe.items():
            x_m = pixel[0]/x_mm
            y_m = pixel[1]/y_mm
            diff_mm[key] = [x_m, y_m]
        return diffe, diff_mm

    def process(self):
        height_vector, width_vector, img = self.load_picture()
        states_vector = self.states()
        binary = self.binary_image(img)
        contours, centroid = self.find_conturs(binary)
        coordinates = self.numerical_order_of_states(centroid, height_vector,
width_vector)
        real_states = self.real_states(coordinates, states_vector)
        current_state = self.shapes(contours, real_states)
        munkres_dict = self.munkres_states(current_state)
        diffe, diff_mm = self.colision(centroid, real_states, height_vector,
width_vector)
        return munkres_dict, diff_mm
```

A.5. munkres.py

```
import sys
sys.path.pop(3)
import numpy as np
from munkres import Munkres, print_matrix
import math


class MunkresClass:

    def __init__(self, numberOfStates, **kwargs):
        self.numberOfStates = numberOfStates
        self.kwargs = kwargs

    def creating_Munkres_Matrix(self):
        length = []
        matrix = []
        distanceMatrix = []
        total = []
        numberOfCellsInOneRow = int(math.sqrt(self.numberOfStates))
        for l in self.kwargs["current"]:
            length.append(len(self.kwargs["current"][l]))
        for i in range(len(length)):
            matrix.append(np.ones((length[i], length[i])))
        for array in range(len(matrix)):
            for b in range(numberOfCellsInOneRow):
                for g in range(numberOfCellsInOneRow):
                    distanceMatrix.append((float(b),float(g)))
        for current_position in self.kwargs["current"]:
            diff_all = []
            for j in self.kwargs["final"][current_position]:
                diff = []
                for z in self.kwargs["current"][current_position]:
                    f = distanceMatrix[j]
                    p = distanceMatrix[z]
                    diff.append(abs(f[0] - p[0]) + abs(f[1] - p[1]))
                diff_all.append(diff)
            total.append(diff_all)
        total = np.array(total)
        out = dict()
        for total_row in range(len(total)):
            m = Munkres()
            indexes = m.compute(total[total_row])
            out[total_row] = indexes
        position_dict = dict()
        current_dict = dict()
        final_dict= dict()
        current_new = []
        final_new = []
        for o in out:
            for j in out[o]:
                current_new.append(self.kwargs["current"][o][j[1]])
                final_new.append(self.kwargs["final"][o][j[0]])
        for k in range(len(current_new)):
            current_dict[k] = current_new[k]
            final_dict[k] = final_new[k]
        position_dict["current"] = current_dict
        position_dict["final"] = final_dict
        return position_dict

    def process(self):
        states = self.creating_Munkres_Matrix()
        return states
```

A.6. SARSA1.py

```
import numpy as np
import random
import xlsxwriter
from copy import deepcopy

from utils import feature, distance, distMatrix, reward, chooseAction, delta,
choose_Q_action, rmse, delta_two_steps


def SARSA2(d, numberOfStates, nepisodes, numOfROws, alpha, delta_factor,ep):
    current = d["current"]
    final = d["final"]
    len_features = feature(current, final, numOfROws)
    len_features = len(len_features)
    feature_matrix = np.zeros(((numberOfStates*len_features*len_features+1),
numberOfStates*len(current)))
    feature_matrix[-1, :] = float(1)
    theta = np.random.rand(1, (numberOfStates * len_features * len_features) +
1)[0]
    theta = np.asarray(theta)
    even_numbers = np.arange(0, len(current)*numberOfStates + numberOfStates,
numberOfStates)
    mean_s_err = []
    parameters = []
    for i in range(nepisodes):
        state = []
        while len(state) < len(current):
            integer = random.randint(0, numberOfStates - 1)
            if integer not in state:
                state.append(integer)
        f1, action1 = chooseAction(state, numberOfStates, current, final,
len_features, numOfROws, ep)
        while (True):
            state_f1 = deepcopy(state)
            state_f1[f1] = action1
            features1 = feature(state_f1, final, numOfROws)
            row_index = even_numbers[f1]
            row = (len_features * row_index) + (action1 * len_features)
            column = row_index + (action1)
            feature_matrix[row:row+len_features, column] = features1
            q = np.dot(theta,(feature_matrix[:, column].T))
            f2, action2 = chooseAction(state_f1, numberOfStates, current, final,
len_features, numOfROws, ep)
            state_f2 = deepcopy(state_f1)
            state_f2[f2] = action2
            f3, action3 = chooseAction(state_f2, numberOfStates, current, final,
len_features, numOfROws, ep)
            state_f3 = deepcopy(state_f2)
            state_f3[f3] = action3
            features3 = feature(state_f3, final, numOfROws)
            row_index3 = even_numbers[f3]
            row3 = (len_features * row_index3) + (action3 * len_features)
            column3 = row_index3 + (action3)
            feature_matrix[row3:row3 + len_features, column3] = features3
            q3 = np.dot(theta, (feature_matrix[:, column3].T))
            deltaTheta, nagrada_a = delta_two_steps(state_f2, state_f3, final, q,
q3, delta_factor)
            theta = theta + alpha*deltaTheta*feature_matrix[:, column]
            f1 = f2
            action1 = action2
            state = state_f1
```

```
            if state_f1 == final:
                theta_end_episode = deepcopy(theta)
                parameters.append(theta_end_episode)
                break
    mean_theta = []
    for parameter in parameters:
        rms_error = rmse(theta, parameter)
        mean_s_err.append(rms_error)
        mean_theta.append(parameter)
    mean_s_err = np.array([mean_s_err])
    mean_theta = np.array([mean_theta])
    workbook = xlsxwriter.Workbook('SARSA2_theta.xlsx')
    worksheet = workbook.add_worksheet()
    row = 0
    for col, data in enumerate(mean_theta):
        worksheet.write_column(row, col, data)
    workbook.close()

    workbook = xlsxwriter.Workbook('SARSA2_rmse.xlsx')
    worksheet = workbook.add_worksheet()
    row = 0
    for col, data in enumerate(mean_s_err):
        worksheet.write_column(row, col, data)
    workbook.close()
    return theta
```

A.7. predict.py

```
import numpy as np

from utils import feature
from SARSA2 import SARSA2


def predict(current, final, numOfStates, numOfActions, numOfRows, numOfFeatures, d,
nepisodes, ep):
    theta = SARSA2(d, numOfStates, nepisodes, numOfRows, alpha, delta_factor, ep)
    all_states = []
    all_features = []
    all_actions = []
    q = np.zeros((1, numOfActions*len(current)))[0]
    feature_matrix = np.zeros(((numOfActions * numOfFeatures * numOfFeatures + 1),
numOfActions * len(current)))
    even_numbers = np.arange(0, len(current) * numOfActions + numOfActions,
numOfActions)
    state = current
    while(True):
        features_list = []
        action_list = []
        for f in range(len(final)):
            for a in range(0, numOfActions):
                if a not in state:
                    features_list.append(f)
                    action_list.append(a)
                    newState = deepcopy(state)
                    newState[f] = a
                    features = feature(newState, final, numOfRows)
                    row_index2 = even_numbers[f]
                    row2 = (numOfFeatures * row_index2) + (a * numOfFeatures)
                    column2 = row_index2 + a
                    feature_matrix[row2:row2 + numOfFeatures, column2] = features
                    q[even_numbers[f]+a]= np.dot(theta,
(feature_matrix[:,even_numbers [f]+a].T))
                else:
                    features_list.append(f)
                    action_list.append(a)
                    row_index2 = even_numbers[f]
                    row2 = (numOfFeatures * row_index2) + (a * numOfFeatures)
                    column2 = row_index2 + a
                    feature_matrix[row2:row2 + numOfFeatures, column2] = [0] *
numOfFeatures
                    q[even_numbers[f]+a] = np.dot(theta,
(feature_matrix[:,even_numbers[f]+a].T))
        feat_act = np.argmax(q)
        feature_to_move = features_list[feat_act]
        action = action_list[feat_act]
        state[feature_to_move] = action
        all_states.append(state)
        all_actions.append(action)
        all_features.append(feature_to_move)
        if state == final
            break
    return all_features, all_actions, all_states
```

```
def robot_commands(all_states, all_features, all_actions, current):
    commands = []
    all_states.insert(0, current)
    for i in range(len(all_features)):
        move = all_states[i][all_features[i]]
        commands.append((move, all_actions[i]))
    return commands
```

A.8. robot_script.py

```
import socket
import time
import numpy as np


class RobotCommands:

    def __init__(self, commands, states_coordinates, tx, ty, tz, Rx, Ry, Rz, z_up,
z_down, HOST, PORT):
        self.commands = commands
        self.states_coordinates =  states_coordinates
        self.tx = tx
        self.ty = ty
        self.tz = tz
        self.Rx = Rx
        self.Ry = Ry
        self.Rz = Rz
        self.z_up = z_up
        self.z_down = z_down
        self.HOST = HOST
        self.PORT = PORT

    def transformation(self):
        theta = np.sqrt(self.Rx ** 2 + self.Ry ** 2 + self.Rz ** 2)
        ex = self.Rx / theta
        ey = self.Ry / theta
        ez = self.Rz / theta
        c = 1 - np.cos(theta)
        R = np.array(
            [[c * (ex ** 2) + np.cos(theta), c * (ex * ey) - ez * np.sin(theta), c
* ex * ez + ey * np.sin(theta)],
             [c * (ex * ey) + ez * np.sin(theta), c * (ey ** 2) + np.cos(theta), c
* ey * ez - ex * np.sin(theta)],
             [c * ex * ez - ey * np.sin(theta), c * (ez * ey) + ex * np.sin(theta),
c * (ez ** 2) + np.cos(theta)]])
        t = np.array([[self.tx, self.ty, self.tz]])
        T = np.concatenate((R, t.T), axis=1)
        add_vec = np.array([[0,0,0,1]])
        T_all =  np.concatenate((T, add_vec), axis=0)
        print(T_all)
        return T_all

    def adjust(self, move_coordinates):
        move_coordinates = move_coordinates.tolist()
        move_coordinates.pop(3)
        move_coordinates.append(self.Rx)
        move_coordinates.append(self.Ry)
        move_coordinates.append(self.Rz)
        return move_coordinates


    def move_robot(self):
        # spajanje s robotom
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((self.HOST, self.PORT))
        for i in self.commands:
                state_current = i[0]
                state_next = i[1]
                x_current = -(self.states_coordinates[state_current][0])
                y_current = -(self.states_coordinates[state_current][1])
                x_next = -(self.states_coordinates[state_next][0])
                y_next = -(self.states_coordinates[state_next][1])
```

```
                # dolazak do predmeta
                pose = np.array([-x_current, -y_current, -self.z_up, 1])
                move_coordinates = np.dot(self.transformation(), pose.T)
                move_coordinates = self.adjust(move_coordinates)
                s.send(("movel(" + "p" + str(move_coordinates) + ", a= 0.09,
v=0.05)" + "\n").encode("ascii"))
                data = s.recv(1024)
                print("Received", repr(data))
                time.sleep(3)
                #  spuštanje po predmet
                pose = np.array([-x_current, -y_current, -self.z_down, 1])
                move_coordinates = np.dot(self.transformation(), pose.T)
                move_coordinates = self.adjust(move_coordinates)
                s.send(("movel(" + "p" + str(move_coordinates) + ", a= 0.09,
v=0.05)" + "\n").encode("ascii"))
                data = s.recv(1024)
                print("Received", repr(data))
                time.sleep(3)
                s.send(("set_digital_out(0,True)" + "\n").encode("ascii"))
                time.sleep(2)
                # dizanje s predmetom
                pose = np.array([-x_current, -y_current, -self.z_up, 1])
                move_coordinates = np.dot(self.transformation(), pose.T)
                move_coordinates = self.adjust(move_coordinates)
                s.send(("movel(" + "p" + str(move_coordinates) + ", a= 0.09,
v=0.05)" + "\n").encode("ascii"))
                data = s.recv(1024)
                print("Received", repr(data))
                time.sleep(3)
                # odlazak do ciljne pozicije s predmetom
                pose = np.array([-x_next, -y_next, -self.z_up, 1])
                move_coordinates = np.dot(self.transformation(), pose.T)
                move_coordinates = self.adjust(move_coordinates)
                s.send(("movel(" + "p" + str(move_coordinates) + ", a= 0.09,
v=0.05)" + "\n").encode("ascii"))
                data = s.recv(1024)
                print("Received", repr(data))
                time.sleep(3)
                # spustanje predmeta
                pose = np.array([-x_next, -y_next, -self.z_down, 1])
                move_coordinates = np.dot(self.transformation(), pose.T)
                move_coordinates = self.adjust(move_coordinates)
                s.send(("movel(" + "p" + str(move_coordinates) + ", a= 0.09,
v=0.05)" + "\n").encode("ascii"))
                data = s.recv(1024)
                print("Received", repr(data))
                time.sleep(3)
                s.send(("set_digital_out(0,False)" + "\n").encode("ascii"))
                time.sleep(2)
                # dizanje nakon ostavljanja predmeta
                pose = np.array([-x_next, -y_next, -self.z_up, 1])
                move_coordinates = np.dot(self.transformation(), pose.T)
                move_coordinates = self.adjust(move_coordinates)
                s.send(("movel(" + "p" + str(move_coordinates) + ", a= 0.09,
v=0.05)" + "\n").encode("ascii"))
                data = s.recv(1024)
                print("Received", repr(data))
                time.sleep(3)
        s.close()
```

A.9. utils.py

```
import numpy as np
import math
import random
from copy import deepcopy


def feature(current_state, final_state, numOfRows):
    distanceMatrix = distMatrix(numOfRows)
    features = []
    features_fourier = []
    for s in range(len(current_state)):
        dist = distance(current_state[s], final_state[s], distanceMatrix)
        features.append(dist)
    for s in features:
        features_fourier.append(0.2*math.cos(2*math.pi*s))
    return features_fourier


def distance(state1, state2, distanceMatrix):
    return math.sqrt((distanceMatrix[state1][0] - distanceMatrix[state2][0]) ** 2 +
(distanceMatrix[state1][1] - distanceMatrix[state2][1]) ** 2)


def distMatrix(numberOfCellsInOneRow):
    distanceMatrix = []
    for b in range(numberOfCellsInOneRow):
        for g in range(numberOfCellsInOneRow):
            distanceMatrix.append((float(b),float(g)))
    return distanceMatrix


def reward(state1, finalstate):
    count = 0
    for i in range(len(state1)):
        if state1[i] == finalstate[i]:
            count += 1
        for j in range(len(state1)):
            if j != i:
                if state1[j] == finalstate[i]:
                    count -= 1
    return float(count)


def delta(state1, finalstate, q, qnext, gamma):
    r = float(reward(state1, finalstate))
    temporalDifferencing = r + (gamma * qnext) - q
    reward_all = r + (gamma * qnext)
    return temporalDifferencing, reward_all


def delta_two_steps(state1, state2, finalstate, q, qnext, gamma):
    r12 = float(reward(state1, finalstate))
    r23 = float(reward(state2, finalstate))
    temporalDifferencing = r12 + (gamma*r23) + ((gamma)**2) * qnext) - q
    reward_all = r12 + (gamma*r23) + (gamma**2) * qnext)
    return temporalDifferencing, reward_all


def rmse(predictions, targets):
    return np.sqrt(((predictions - targets) ** 2).mean())
```

```
def chooseAction(state, numberOfStates, current, final, len_features, numOfRows,
ep):
    distanceMatrix = distMatrix(numOfRows)
    random_number = random.random()
    avaliable_features = list(range(0, len(current)))
    actions_list = list(range(0, numberOfStates))
    avaliable_actions_list = []
    total_reward = []
    state_numbers = np.arange(0, len(current) * numberOfStates + numberOfStates,
numberOfStates)
    feature_g = int()
    action = int()
    for ac in actions_list:
        if ac not in state:
            avaliable_actions_list.append(ac)
    for s in range(len(current)):
        for a in actions_list:
            newState = deepcopy(state)
            newState[s] = a
            collected_reward = reward(newState, final)
            total_reward.append(collected_reward)
    for ik in range(0, len(actions_list)):
        if (ik not in avaliable_actions_list):
            for ip in state_numbers:
                if (ip > state_numbers[-2]):
                    break
                else:
                    total_reward[ik + ip] = float(-1000)
    if (random_number > ep):
        action_feature = [index for index, i in enumerate(total_reward) if i ==
max(total_reward)]
        if (len(action_feature) > 1):
            all_states = dict()
            feature_action = dict()
            for act in action_feature:
                coeff = act/numberOfStates
                for ih in range(0, len(state_numbers)):
                    newState = deepcopy(state)
                    news = []
                    if (coeff == float(state_numbers[ih])):
                        action_1 = 0
                        feature_g_1 = int(coeff)
                        newState[feature_g_1] = action_1
                        for sh in range(len(final)):
                            dist = distance(newState[sh], final[sh],
distanceMatrix)
                            news.append(dist)
                        all_states[act] = sum(news)
                        feature_action[act] = [feature_g_1, action_1]
                        break
                    if (act < state_numbers[ih]):
                        action_1 = act - state_numbers[ih - 1]
                        feature_g_1 = int(ih - 1)
                        newState = deepcopy(state)
                        newState[feature_g_1] = action_1
                        for sh in range(len(final)):
                            dist = distance(newState[sh], final[sh],
distanceMatrix)
                            news.append(dist)
                        all_states[act] = sum(news)
                        feature_action[act] = [feature_g_1, action_1]
                        break
            vallues = []
            keeys = []
```

```
            for key, value in all_states.items():
                vallues.append(value)
                keeys.append(key)
            vallues = np.array(vallues)
            keeys = np.array(keeys)
            arg = np.argmin(vallues)
            action_feature = keeys[arg]
        else:
            action_feature = action_feature[0]
        division = (action_feature) / (numberOfStates)
        for it in range(0,len(state_numbers)):
            if (division == float(state_numbers[it])):
                action = 0
                feature_g = int(division)
                break
            if (action_feature < state_numbers[it]):
                action = action_feature - state_numbers[it - 1]
                feature_g = it - 1
                break
    else:
        action = random.choice(avaliable_actions_list)
        feature_g = random.choice(avaliable_features)
    return int(feature_g), int(action)
```

A.10. main.py

```
from copy import deepcopy

from frame_filter import FrameFilter
from creating_Munkres_Matrix import MunkresClass
from predict import predict, robot_commands
from camera_calibration import camera_calibration
from robot_script import RobotCommands

# primjer ulaznih parametara
t = 28
file1 = "obradena_slika_p.png"
file2 = "obradena_slika_f.png"
n = 7
numOfStates = 49
numOfActions = 49
numOfRows = 7
numOfFeatures = 7
nepisodes = 700
alpha= 0.4
delta_factor = 0.6
ep = 0.2
z_up = 0.004
z_down = 0.012
HOST = "192.168.123.55"
PORT = 30002
tx = 0.44203213
ty = -0.34710908
tz = 0.002881421
Rx = -0.57763785
Ry = 3.073138
Rz = -0.0076691783

def main():
    ff_final = FrameFilter(t, file2, n)
    final, diff_mm_f = ff_final.process()
    ff_current = FrameFilter(t, file1, n)
    current, diff_mm_p = ff_current.process()
    d = {"current": current, "final": final}
    states = MunkresClass(numOfStates, **d)
    states = states.process()
    d_current = []
    d_final = []
    for key, value in states["current"].items():
        d_current.append(value)
    for key, value in states["final"].items():
        d_final.append(value)
    d_start = deepcopy(d_current)
    states_dict = {"current": d_current, "final": d_final}
    all_features, all_actions, all_states = predict(d_current, d_final,
numOfStates, numOfActions, numOfRows, numOfFeatures,
                                states_dict, nepisodes, alpha, delta_factor, ep)
    rco = robot_commands(all_states, all_features, all_actions, d_start)
    xy_matrix = camera_calibration()
    robot = RobotCommands(rco, xy_matrix, tx, ty, tz, Rx, Ry, Rz, diff_mm_p, z_up,
z_down, HOST, PORT)
    robot.move_robot()
    return all_features, all_actions,  all_states

all_features, all_actions, all_states = main()
```