Primjena umjetne inteligencije za optimizaciju putanje mobilnog robota

Bosiljevac, Grgur

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka, Faculty of Engineering / Sveučilište u Rijeci, Tehnički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:190:673608

SVEUČILIŠTE U RIJECI

TEHNIČKI FAKULTET

Diplomski sveučilišni studij elektrotehnike

Diplomski rad

PRIMJENA UMJETNE INTELIGENCIJE ZA OPTIMIZACIJU PUTANJE MOBILNOG ROBOTA

Rijeka, svibanj 2024.

Grgur Bosiljevac

0069079614

SVEUČILIŠTE U RIJECI

TEHNIČKI FAKULTET

Diplomski sveučilišni studij elektrotehnike

Diplomski rad

PRIMJENA UMJETNE INTELIGENCIJE ZA OPTIMIZACIJU PUTANJE MOBILNOG ROBOTA

Mentor: prof. dr. sc. Zlatan Car

Rijeka, svibanj 2024.

Grgur Bosiljevac

0069079614

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
POVJERENSTVO ZA DIPLOMSKE ISPITE

Rijeka, 21. ožujka 2022.

Zavod: **Zavod za automatiku i elektroniku**
Predmet: **Osnove robotike**
Grana: **2.03.06 automatizacija i robotika**

ZADATAK ZA DIPLOMSKI RAD

Pristupnik: **Grgur Bosiljevac (0069079614)**
Studij: Diplomski sveučilišni studij elektrotehnike
Modul: Automatika

Zadatak: **Optimizacija putanje mobilnog robota primjenom umjetne inteligencije / Mobile robot path optimization using artificial intelligence**

Opis zadatka:

Napraviti pregled postojećih istraživanja i primjene algoritama umjetne inteligencije korištenih u optimizaciji putanje mobilnog robota. Napraviti virtualni model mobilnog robota u prostoru sa nepomičnim preprekama. Implementirati različite algoritme umjetne inteligencije za njegovo gibanje od početne pozicije do cilja izbjegavajući prepreke. Ispitati mogućnost korištenja različitih senzora kao dodatnih informacija s ciljem poboljšanja optimizacije putanje mobilnog robota.

Rad mora biti napisan prema Uputama za pisanje diplomskih / završnih radova koje su objavljene na mrežnim stranicama studija.

Grgur Bosiljevac

Zadatak uručen pristupniku: 21. ožujka 2022.

Mentor:

Prof. dr. sc. Zlatan Car

Predsjednik povjerenstva za diplomski ispit:

Prof. dr. sc. Viktor Sučić

Izjava o samostalnoj izradi rada

Ja, Grgur Bosiljevac, rođen 24. 04. 1998. u Karlovcu izjavljujem da sam diplomski rad pod naslovom "Primjena umjetne inteligencije za optimizaciju putanje mobilnog robota" samostalno napisao i da sam njegov jedini autor. Rad sam izradio pod mentorstvom prof. dr. sc. Zlatana Cara i komentara dr. sc. Nikole Anđelića. Svi izvori korišteni u pisanju ovog rada navedeni su u poglavlju Literatura.

U Rijeci, svibanj 2024.

Grgur Bosiljevac

Ime i Prezime

Zahvala

Zahvaljujem se svima koji su mi pružili podršku tokom studija i tokom pisanja diplomskog rada. Posebno se zahvaljujem mentoru prof. dr. sc. Zlatanu Caru i asistentu doc. dr. sc. Nikoli Anđeliću na pomoći i konzultacijama kod izrade diplomskog rada.

Sadržaj

1. UVOD

Cilj ovog diplomskog rada je optimizirati putanju mobilnog robota u CoppeliaSim programskom paketu implementacijom umjetne inteligencije, kao što je navedeno u zadatku diplomskog rada. Ideja je izraditi scenu unutar CoppeliaSim programskog paketa koja se sastoji od modela mobilnog robota te polja sa preprekama odnosno labirinta. Potrebno je da se mobilni robot kreće kroz taj labirint na način da se nastoji kretati sredinom između dvaju zidova te da izvršava akcije skretanja onda kada je to potrebno. Nakon toga plan je implementirati umjetnu inteligenciju za određivanje optimalnog, odnosno najkraćeg mogućeg puta od početne do ciljne točke labirinta. Nakon što je taj put određen, ideja je rekreirati taj put u CoppeliaSim-u.

Za realizaciju ovog problema, potrebno je istražiti mogućnosti koje nudi programski paket CoppeliaSim, izradom ili korištenjem već izrađenog modela mobilnog robota sa svojim elementima poput zglobova, senzora, te napraviti modifikaciju na istom ako je to potrebno. Zatim je potrebno istražiti metode umjetne inteligencije radi obavljanja zadaći koje su navedene u zadatku diplomskog rada.

CoppeliaSim, koji je prije bio poznat kao V-REP, široko je korišten programski paket, generalno kod istraživanja robotike. Razvijen je od strane Coppelia Robotics te je dostupan besplatno za akademske svrhe [1]. Simulacijski programski paket nudi široku funkcionalnost koja se može jednostavno integrirati i kombinirati putem ugrađenog skriptiranja i opsežnog API-a (Application Programming Interface) temeljenom na programskim sučeljima poput LUA (koji je i korišten u ovom projektu), Python, C/C++, Java itd. Program može savladati inverznu odnosno unaprijednu kinematiku bilo kojeg tipa mehanizma sa vizualizacijom radnog prostora, simulacijom senzorike, proračuna udaljenosti i detekcije sudara.

2. ROBOTIKA

Robotika se može zamisliti kao svijet kreacije uređaja koji oponašaju živa bića sposobna obavljati zadatke i ponašati se kao da su svjesni svijeta oko njih. Ova je zamisao na ljudskome umu još od samog vremena kada su mogli graditi stvari [23]. Roboti su izrazito napredni elementi današnje industrije. Sposobni su obavljati razne zadatke i operacije, precizni su, i ne zahtjevaju opće mjere zaštite i udobnosti koje ljudi trebaju u radnim okruženjima [23]. Međutim, za funkcionalan rad robota potreban je značajan ulog resursa i ljudskog napora.

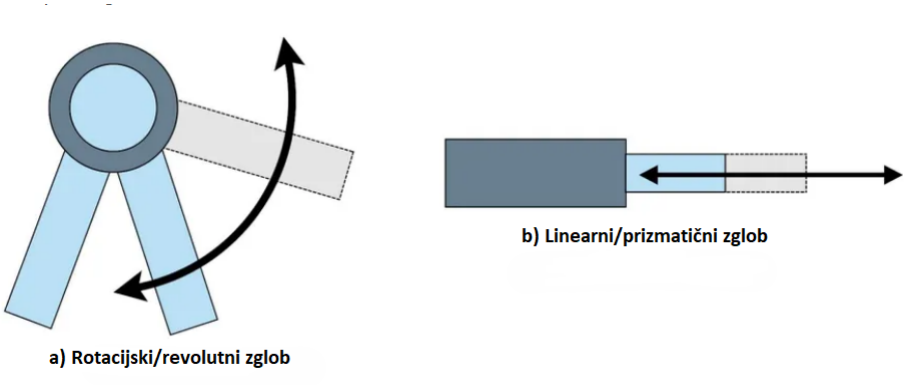
Iznimno bitno pitanje koje na početku ove teme treba postaviti je što je to uopće robot. Kada se usporede konvencionalni robotski manipulator sa, npr., dizalicom pričvršćenom na vuču, mogu se primjetiti izrazite sličnosti. Oba elementa se sastoje od brojnih zglobova koji su serijski povezani s člancima, omogućavaju svakom segmentu da se pokreće pomoću aktuatora. U oba sustava, krajnji djelovatelj (eng. End-effector) može se kretati kroz prostor i doseći bilo koju željenu lokaciju unutar radnog prostora sustava. Međutim, unatoč njihovoj sličnosti, jedan se naziva robotom, a drugi manipulatorom. Ključna razlika leži u njihovim mehanizmima kontrole: dizalice i manipulatore obično upravlja čovjek, pri čemu operatori upravljaju aktuatorima. Nasuprot tome, robotski manipulator upravlja neko računalo ili mikroprocesor koji izvršava programirane zadatke. Ta razlika određuje kategoriju uređaja u pitanju. U principu, robote upravlja računalo, te su nakon programiranja autonomni od ljudske kontrole. Cilj robotike je stvoriti uređaj (robot) sposoban izvoditi raznolike zadatke, opskrbljujući ga fleksibilnošću bez daljnjeg projektiranja [23].

2.1. Robotski elementi

Bitan robotski element je zglob. Zglob robota je mehanizam koji dozvoljava razmjerni pokret između dijelova robotske ruke [26]. Namjena zglobova je da omogući robotu pozicioniranje krajnjeg djelovatelja. Dvije glavne vrste zglobova su revolutni (rotacijski) i prizmatični (linearni) zglobovi.

Revolutni zglobovi su jedni od najzastupljenijih zglobova u robotici. Omogućuju rotaciju oko jedne osi. Dizajn ovakvog zgloba uglavnom uključuje neku vrstu motora koji omogućuje rotacijsku silu, i set zupčanika koji prebacuje tu silu na zglob [25]. Prizmatični zglobovi su još jedan tip

zglobova koji omogućuju kretanje u ravnoj liniji. Dizajn prizmatičnog zgloba uključuje linearni aktuator koji generira linearnu silu u pravcu. Prikaz robotskih zglobova nalazi se na Slici 2.1.



Slika 2.1. *Prikaz robotskih zglobova [25]*

Svrha zglobova je ze pokretanje i pozicioniraje krajneg djelovatelja (eng. End Effector). Krajnji djelovatelj omogućuje robotu izvođenje posebnih radnji [3]. Postoji podjela na dvije osnovne vrste: hvataljke i alate. Hvataljke se obično upotrebljavaju za dohvat i držanje objekta i njegovo premještanje na drugo mjesto, a neke vrste hvataljki obuhvaćaju mehaničke, magnetske i prianjajuće hvataljke, vakuumske "šalice", kuke i slično. Od alata obično se radi o nekom alatu kojim robot zamjenjuje radnika, a to može biti alat za bradavičasto ili lučno varenje, bušenje itd.

Kod mobilnih robota, o kojima će se reći nešto u sljedećem poglavlju, izrazitu važnost imaju senzori. Ako se robot mora kretati prema zidu, i stati ispred njega, npr., 50 cm od njega, robotu je potreban način za detekciju tog zida. Senzor je komponenta koja odmjerava neke karakteristike radnog okruženja [24]. Senzori su podijeljeni na senzore položaja, senzore dometa, senzore brzine i senzori neposredne blizine (udaljenosti).

Senzori položaja služe za informaciju o položaju zglobova, koje se vraćaju do nadzornog sustava koji ima zadaću da utvrdi točnost položaja. Senzori dometa služe za informaciju o udaljenosti od referentne točke do neke bitne točke, koja se određuje preko kamera ili ultrazvučnih odašiljača i prijemnika. Senzori brzine služe za određivanje brzine kojom se manipulator kreće. Istosmjerni brzinomjer je jedan od najčešćih uređaja za povrat informacije brzine. Konačno, senzori neposredne blizine se upotrebljavaju za određivanje udaljenosti robota od drugih objekata u radnom okruženju [26]. Senzori udaljenosti su uglavnom aktivni senzori; odašiljaju signal i primaju njegovu refleksiju od objekta. Jedan način za određivanje udaljenosti je mjerenje vremena koje signalu treba da dosegne objekt i da se reflektira od njega u prijemnik [24]. Ovi senzori se dijele na infracrvene,

ultrazvučne, laserske i senzore dodira.

2.2. Podjela robota

Tri najzastupljenije vrste podjela na koje se roboti dijele su podjela robota po okruženju, podjela u odnosu na konfiguraciju te podjela robota prema nadzornom sustavu.

Ako se govori o podjeli robota prema načinu kretanja, oni se mogu podijeliti na stacionarne i mobilne robote [24]. Ove dvije vrste robota imaju skroz drukčije radno okruženje, pa isto tako i različite radne sposobnosti. Stacionarnim robotima se uglavnom smatraju robotski manipulatori koji rade u dobro definiranom radnom okruženju prilagođenom za robote. Tu spadaju industrijski roboti koji obavljaju određen repetitivni zadatak poput bojanja dijelova u autoindustrijskim postrojenjima. Nasuprot ovim robotima, mobilni roboti se kreću i obavljaju zadatke u radnim okruženjima koja nisu specifično dizajnirana za rad s robotima [24]. Moraju se adekvatno ponašati u situacijama koje nisu poznate unaprijed i koje se potencijalno mijenjaju kroz vrijeme [4].

Druga podjela je podjela robota u odnosu na konfiguraciju. Vrste robota prema podjeli na konfiguraciju su kartezijski, cilindrični, sferični, artikulirani te SCARA sklop. Glavna razlika između ovih konfiguracija je broj i vrsta zglobova koje posjeduju.

Zadnja podjela robota je prema nadzornom sustavu. Prema ovoj podjeli robote kategoriziramo na Point-to-point (PTP), continuous-path control (CP) i controlled-path robote. Međusobno se razlikuju po slobodi kretanja po stazi, kompleksnosti staze po kojoj se mogu kretati te na koliko točaka na toj stazi mogu stati.

2.3. Mobilna robotika

Mobilni roboti su roboti koji se mogu autonomno kretati od jednog mjesta do drugog, to jest, bez pomoći vanjskog ljudskog operatora [4]. Za razliku od industrijskih robota koji su stacionarni, mogu se slobodno kretati u radnom okruženju kako bi obavljali zadatke. Ova mobilnost čini ih prikladnim za široku primjenu u strukturiranim i nestrukturiranim okruženjima [4]. Mobilni roboti dijele se na:

- Zemaljske mobilne robote - mobilni roboti sa kotačima (eng. wheeled mobile robots - WMRs) i mobilni roboti sa nogama (eng. legged mobile robots - LMRs),

- Zračne mobilne robote - bespilotne letjelice (eng. unmanned aerial vehicles - UAVs) i
- Vodene mobilne robote - autonomna podvodna vozila (eng. aoutnomous underwater vehicles - AUVs).

Kod zemaljskih mobilnih robota, mobilni roboti sa nogama prikladni su za zadatke u nekonvencionalnim okruženjima, stepenicama, okruženjima sa mnoštvom ruševina itd. Mobilni roboti sa kotačima su prikladniji za tipične primjene sa malom mehaničkom složenosti [4]. U ovom su radu relevantniji mobilni roboti sa kotačima, te će se o njima nešto više reći u nastavku.

Ono što je bitno za raspoznati kod mobilnih robota sa kotačima je koju vrstu kotača koristi, te koji pogon. O njima ovisi manevarska sposobnost, stabilnost i kontrola mobilnog robota. Kod mobilnih robota sa tri ili više kotača ravnoteža je automatski postignuta, no kod njih je potrebna uporaba sustava suspenzije koji osigurava da svi kotači imaju kontakt sa zemljom na grubom terenu.

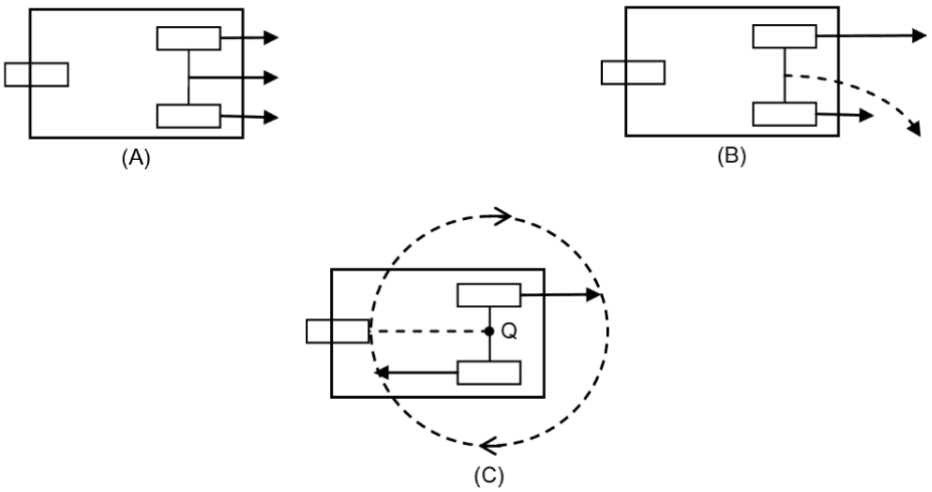
Vrste kotača korištenih u mobilnim robotima s kotačima su konvencionalni i specijalni kotači. U konvencionalne kotače spadaju:

- Pogonjeni fiksni kotači (eng. powered fixed wheels) - pogonjeni motorima montiranim na fiksnim pozicijama vozila. Njihova os rotacije ima fiksan smjer u odnosu na koordinatni okvir platforme,
- Pomoćni kotači (eng. castor wheels) - mogu rotirati slobodno oko osi okomite na os njihove rotacije. Nisu pogonjeni,
- Pogonjeni okretni kotači (eng. powered steering wheels) - sadrže motor za njihovu rotaciju pomoću kojeg mogu skretati po osi okomitoj od osi rotacije. Također sadrže motor za njihov pogon.

Specijalni kotači su dizajnirani da imaju aktivnu vuču u jednom smjeru i pasivno kretanje u drugom [4], što im daje veći stupanj manevarske sposobnosti u pretrpanim okruženjima. Vrste specijalnih kotača su univerzalni, Mecanum te kugla. Konvencionalni kotači imaju veću nosivost i veću toleranciju na nepravilnosti zemlje u usporedbi sa specijalnim kotačima.

Postoje razne vrste pogona mobilnih robota na kotačima. Dijele se na diferencijalni pogon, tricikl, omnidirekcionalni, sinkro pogon, te Ackerman i Skid skretanje. Za potrebe ovog rada, zbog njegove jednostavnosti i primjenjivosti koristit će se diferencijalni pogon.

Diferencijalni pogon je jako jednostavan mehanizam vožnje često korišten u praksi, pogotovo za manje mobilne robote [27]. Roboti sa ovakvim pogonom u pravilu imaju dva pogonjena fiksna kotača kao glavne kotače, koji su montirani na lijevoj i desnoj strani robota, kao i jedan ili više pomoćnih kotača za potporu vozila kako bi se spriječilo naginjanje. Jednostavnost ovog pogona proizlazi iz činjenice da nije potrebna rotacija osi ovih kotača, pošto se skretanje izvodi razlikom brzina lijevog i desnog kotača. Rotiraju li se oba glavna kotača istom brzinom, robot se kreće ravno. U slučaju okretanja bilo kojeg kotača brže od drugog, robot skreće u smjeru onog sporijeg. U slučaju da se jedan kotač pogoni u jednom, a drugi u drugom smjeru, robot se okreće oko sredine dvaju pogonskih kotača [4]. Vizualni prikaz objašnjenog pogona nalazi se na Slici 2.2.



Slika 2.2. *Prikaz diferencijalnog pogona - mogućnosti kretanja [4]*

Trenutačno središte zakrivljenosti (eng. Instantaneous center of curvature - ICC) mobilnog robota leži na sjecištu svih osi pogonskih kotača. ICC je središte kruga sa polumjerom R ovisnim o brzini dvaju kotača. Taj polumjer ovisi o odnosu:

$$\frac{v_l - v_r}{2a} = \frac{v_r}{R - a}, \tag{2.1}$$

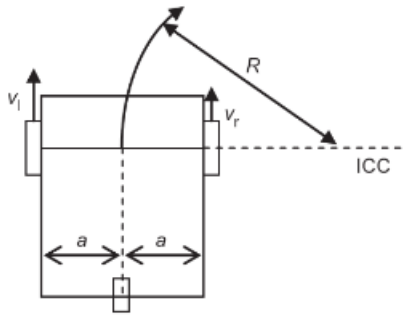
stoga:

$$R = a\frac{v_l + v_r}{v_l - v_r}, v_l \geq v_r, \tag{2.2}$$

gdje su:

- a - polovica razmaka dvaju kotača,
- v_l - brzina lijevog kotača i
- v_r - brzina desnog kotača.

Kada je $v_l = v_r$, tada je $R = \infty$ (ravno kretanje), a kada $v_l = -v_r$, tada je $R = 0$ (rotacijsko kretanje). Ova relacija prikazana je na Slici 2.3.



Slika 2.3. *Izračun ICC-a mobilnog robota [4]*

2.4. Rješavanje problema labirinta

Labirint je složeni put koji se sastoji od različitih grana prolaza gdje je cilj rješavača doći do odredišta pronalaženjem rute unutar određenog vremena [5]. Ako je za neki slučaj potrebno pronaći optimalan put prijeđen u najkraćem mogućem vremenu, vitalnu ulogu igra primjena umjetne inteligencije (skraćeno UI). Teorija grafa također je efikasan alat za dizajniranje tehnike rješavanja labirinta. Neke od metoda teorije grafa su:

1. Dubinska pretraga (eng. Depth First Search - DFS) - u DFS-u, počevši od korijena grafa istraživanje se nastavlja prema dubljim regijama stabla ili grafa dok ne dostigne "list" ili vrh bez neistraženih susjeda. Tada se vraća unazad i istražuje sljedeću granu koja nije potpuno

istražena [5]. Važno je napomenuti da DFS nije uvijek najbolji izbor za pronalaženje najkraćeg puta ili optimalnog rješenja, jer može zaglaviti u beskonačnoj petlji ako postoje ciklusi u grafu. Međutim, DFS je jednostavan i intuitivan algoritam pretrage grafa koji se često koristi u mnogim situacijama.,

2. Algoritam popunjavanja polja (eng. Floodfill) uključuje dodjeljivanje vrijednosti svakoj ćeliji u labirintu, pri čemu te vrijednosti predstavljaju udaljenost od bilo koje ćelije u labirintu do krajnje ćelije [7].

3. Bellman-Ford algoritam - algoritam korišten za određivanje najkraćih puteva u težinskim grafovima [17]. Ovaj algoritam može se primijeniti u kontekstu mobilne robotike za navigaciju robota u labirintu ili mrežnom okruženju.

4. Širinska pretraga (eng. Breadth First Search - BFS) - osnovni algoritam pretraživanja koji se koristi u provjeri modela, kako bi se pokazalo da su određena stanja dostižna ili nedostižna, te kako bi se odredio radijus problema ili najduži odnosno najkraći put iz bilo kojeg zadanog stanja [18].

Algoritmi za rješavanje labirinta su metode koje su u većini slučajeva, a pogotovo kod kompleksnijih labirinta, manje učinkovite i manje pouzdane metode od metoda rješavanja labirinta pomoću teorije grafa ili umjetne inteligencije. Međutim, imaju prednost kod jednostavnijih labirinta zbog niže razine matematičke kompleksnosti. Neki od generalnih algoritama za rješavanje labirinta su:

1. Algoritam sljedbenika zida (eng. Wall follower algorithm) - jedna od najpoznatijih te najjednostavnija metoda za rješavanje labirinta. Ova metoda biti će objašnjenja detaljnije u kasnijem poglavlju [2],

2. Algoritam nasumičnog miša (eng. Random mouse algorithm) - ova metoda može se implementirati na bilo kojeg robota. Trivijalna je te jednostavna za implementaciju. Prati ravnu liniju dok ne dođe do prepreke. Zatim nasumičnim odabirom radi skretanje od 90 stupnjeva ulijevo ili udesno [6],

3. Algoritam zaloga (eng. Pledge algorithm) - Pogodna implementacija za pronalazak izlaza iz labirinta. Za određivanje orijentacije mobilnog robota na ulasku u labirint koristi se senzor

kompasa. Algoritam koristi koncept "zaloga" kako bi osigurao da robot neće stalno kružiti u beskonačnoj petlji [6].

Za rješavanje izazova ovog diplomskog rada definiran je labirit 8 x 8 ćelija, koje će biti definirane u poglavlju koje objašnjava hijerarhiju scene. Ta dimenzija labirinta je u praksi neznačajna te za rješavanje istoga nisu potrebne kompleksne metode. Iz ovog razloga, odabrala se jedna od generalnih metoda, a to je algoritam sljedbenika zida.

Algoritam sljedbenika zida jedna je od najjednostavnijih tehnika rješavanja problema labrinta, budući da sam algoritam djeluje na osnovi jednostavnog principa sljedbenika zida [2]. Uzeći to u obzir, nije pogodan kao pouzdan odabir za labirinte koji su veće te matematički kompleksnije građe, kao i za one koji imaju više od jednog rješenja, odnosno izlaza. Kod složenijih labirinta pogodnije je odabrati kompleksniji algoritam koji će svojim svojstvima zadovoljiti potrebe definiranog labiritna, poput algoritama teorije grafa.

2.5. Algoritam sljedbenika zida

Prije samog obrazloženja algoritma, potrebno je reći da algoritam sljedbenika zida ima dvije moguće varijante koje se mogu naći u realnom okruženju: praćenje lijevogzida te praćenje lijevog desnog zida. Iz razloga što vrijeme prolaska mobilnog robota kroz labirint kada se koristi praćenje lijevog ili desnog zida ovisi isključivo o strukturi pojedinog labirinta labirintu, a ne o odabiru jedne od te dvije opcije, svejedno je hoće li mobilni robot pratiti lijevi odnosno desni zid. U slučaju ovog rada odabrana je varijanta praćenja desnog zida. Sami princip rada algoritma sljedbenika zida slijedi u nastavku: Ako sa desne strane robota nije detektiran zid, mobilni robot radi zavoj udesno. Ukoliko se sa desne strane robota nalazi zid, a ispred robota je prazan prostor, robot se kreće ravno. Međutim, ako je i sa desne i sa prednje strane robota zid, robot se zaustavlja te okreće za 90 stupnjeva ulijevo. Sami algoritam prikazan je u Tablici 2.1.

Tablica 2.1. *Tablica odluka za algoritam sljedbenika desnog zida*

Lijevo	Ispred	Desno	Radnja
X	X	0	Zavoj udesno
X	0	1	Kretanje ravno
X	1	1	Skretanje lijevo

Vrijednost X predstavlja da prisutnost zida ne igra ulogu, 1 da je zid detektiran, te 0 da zida nema. Napravljena je distinkcija između zavoja i skretanja jer se zavoj udesno radi bez zaustavljanja robota, dok skretanje u lijevo zahtjeva potpuno zaustavljanje robota, okret za 90 stupnjeva u lijevo, te tek nakon toga pokretanje mobilnog robota. Također se može primjetiti da detekcija lijevog zida ne igra ulogu u nijednoj od 3 moguće radnje, što može biti gledano kao pozitivna strana iz razloga što je za takve robote potrebno korištenje duplo manje senzorike. Dakle, očito je zašto je ovaj algoritam pogodan za neke jednostavne slučajeve, a to je zbog njegove jako velike jednostavnosti primjene i integracije u razne sustave.

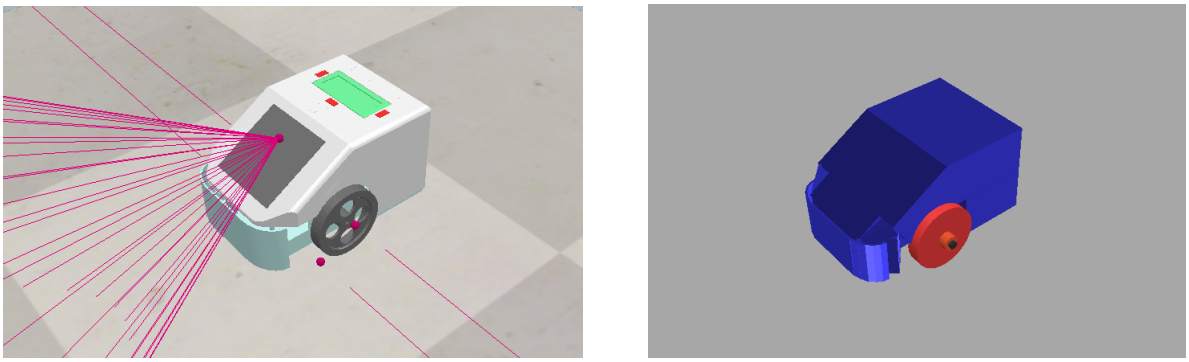
2.5.1. Građa mobilnog robota

Za simulaciju odabran je model mobilnog robota iz CoppeliaSim knjižnice pod nazivom "dr12". Ovaj mobilni robot sastoji se od tijela, pogonskog fiksnog kotača sa lijeve i desne strane robota te jedan pomoćni omnidirekcionalni kotač na stražnjoj strani robota. Pogon kojim se ovaj mobilni robot upravlja je diferencijalni pogon, te se upravljanje vrši logikom objašnjenom u poglavlju. Upravljanje aktuatorima i očitavanje senzora vrši se pomoću mikrokontrolera AVR ATmega128. Ovaj mikrokontroler je visokoučinkoviti, niskopotrošni 8-bitni mikročip baziran na RISC arhitekturi, koji kombinira 128 KB programibilne flash memorije, 4KB SRAM-a, 4 KB EEPROM-a i 8-kanalni 10-bitni A/D pretvarač. Bitne specifikacije mobilnog robota prikazane su u Tablici 2.2.

Tablica 2.2. *Specifikacije mobilnog robota "dr12"*

Specifikacije	
Pogon	Diferencijalni
Motor	STEP, RC servo motor
Kotači	2 pogonska konvencionalna, 1 pomoćni
Masa	2 kg
Maksimalna brzina	150 mm/sek
Mikrokontroler	AVR Atmega128 8-bitni mikročip

Potpune specifikacije mobilnog robota nalaze se u članku [28]. Na Slici 2.4 prikazana je vizualizacija modela mobilnog robota:



(a) ***(b)***

Slika 2.4. *Prikaz modela mobilnog robota dr12, a) kompleksni model, b) jednostavni (dinamički) model*

Za potrebe ovog rada na njega su naknadno dodana 6 senzora. 2 senzora blizine u obliku lasera sa lijeve i desne strane, jedan senzor na prednjoj strani mobilnog robota u obliku korneta te jednopikselni senzor usmjeren prema podu za detekciju ciljne linije. Senzor na prednjoj strani robota jest ultrazvučni senzor, bočni senzori su takozvani LIDAR (eng. Light Detection and Ranging) senzori a senzor za detekciju boja je TCS34725 senzor, koji je zapravo RGB senzor boja, te ga nije potrebno navesti u tablici.

Za razliku od optičkog senzora koji koristi prijemnik i odašiljač, ultrazvučni senzor koristi jedan ultrazvučni element i za odašiljanje i za primanje signala. U reflektirajućem modelu ultrazvučnog senzora, jedan oscilator naizmjenično emitira i prima ultrazvučne valove. To omogućuje minijaturizaciju glave senzora. Prikaz ultrazvučnog senzora nalazi se na Slici 2.5.

Slika 2.5. *Prikaz ultrazvučnog senzora*

LIDAR je metoda za određivanje dometa ciljanjem objekta ili površine laserom i mjerenjem vremena potrebnog za povratak reflektirane svjetlosti do prijemnika. Lidar može raditi u fiksnom smjeru (npr. horizontalno) ili može skenirati više smjerova, u kojem slučaju je poznat kao lidarsko

skeniranje ili 3D lasersko skeniranje, posebna kombinacija 3-D skeniranja i laserskog skeniranja. Za potrebe ovog rada, LIDAR senzori koristiti će se za određivanje udaljenosti zida od robota u fiksnome smjeru, i to u horizontalnoj orijentaciji. Prikaz jednog LIDAR senzora nalazi se na Slici 2.6

Slika 2.6. *Prikaz LIDAR senzora*

Senzori na bočnim stranama robota međusobno su udaljeni, što služi za izradu raznih proračuna kod kretanja mobilnog robota. Značajke senzora blizine nalaze se na Tablici 2.3:

Tablica 2.3. *Tablica značajki primjenjenih senzora blizine na mobilnom robotu*

	Oblik zrake	Offset [m]	Domet [m]	kut [°]
Ultrazvučni senzor	kornet	0	0.5	70
LIDAR senzor	laser	0.05	1.2	

Parametar "Offset" odnosno pomak, označava pomicanje početne točke detekcije senzora od modela senzora u metrima, domet označava udaljenost do koje doseže zraka senzora od početne točke detekcije a kut u stupnjevima označava kut između rubova konture zrake ultrazvučnog senzora.

2.5.2. Matematički model

Za praćenje desnog zida potrebni su bočni senzori sa desne strane kako bi se odredila udaljenost robota od desnog zida, te kut pod kojim se on nalazi s obzirom na zid. Te dvije vrijednosti dobivaju se sljedećim izrazima [8]:

$$\Phi_R = atan\left(\frac{d_{FR} - d_{RR}}{a}\right), \tag{2.3}$$

$$d_R = \frac{d_{FR} + d_{RR}}{2}, \tag{2.4}$$

gdje su:

- Φ_R - kut između mobilnog robota i desnog zida,
- d_R - udaljenost između mobilnog robota i desnog zida,
- d_{FR} - udaljenost između prednjeg desnog senzora i desnog zida,
- d_{RR} - udaljenost između zadnjeg desnog senzora i desnog zida i
- a - udaljenost između prednjeg i zadnjeg bočnog senzora.

Izravnavanje mobilnog robota točno između dvaju zidova vrši se adekvatnim podešavanjem kutne brzine vrtnje lijevog i desnog kotača. To se postiže jednadžbama [8]:

$$\alpha = \Phi_R + k_{WALL}(d_R - d_{wall}), \tag{2.5}$$

$$\omega_L = \frac{v}{R}(cos\alpha + \frac{b}{e}sin\alpha), \tag{2.6}$$

$$\omega_R = \frac{v}{R}(cos\alpha - \frac{b}{e}sin\alpha), \tag{2.7}$$

gdje su:

- α - kut potreban za računanje brzine vrtnje lijevog i desnog kotača,
- ω_L - brzina vrtnje lijevog kotača,
- ω_R - brzina vrtnje desnog kotača,
- k_{WALL} - konstanta za korekciju devijacije udaljenosti od zida,
- e - udaljenost kotača do prednjeg kraja mobilnog robota - off center kinematic control,

- $d_R - d_{wall}$ - razlika udaljenosti mobilnog robota od zida te minimalne udaljenosti robota od zida kod koje se mora zaustaviti,
- v - brzina kretanja mobilnog robota,
- R - radijus kotača mobilnog robota i
- b - udaljenost separacije kotača.

Sljedeće jednadžbe bitne su u slučaju kada mobilni robot i ispred sebe i sa svoje desne strane detektira zid.

Detekcija prednjeg zida [8]:

$$d_{sonar} < d_{sonarWall}, \tag{2.8}$$

gdje su:

- d_{sonar} - udaljenost na kojoj učtrazvučni senzor - sonar, detektira i
- $d_{sonarWall}$ - udaljenost ultrazvučnog senzora do zida.

Udaljenost sonara od zida računa se jednadžbom [8]:

$$d_{sonarWall} = \frac{C}{2} - \delta_{sonar} - \frac{\tau}{2}, \tag{2.9}$$

gdje su:

- C - veličina čelije labirinta,
- δ_{sonar} - udaljenost između baze kotača mobilnog robota i ultrazvučnog senzora i
- τ - širina zida.

Princip jednadžbe (2.8) je jednostavan. Ukoliko je postavljena udaljenost za detektiranje zida ispred mobilnog robota veća od trenutne udaljenosti robota mjerenu ultrazvučnim senzorom, tada se zaključuje detekcija prednjeg zida i vrši se skretanje u lijevo.

Dakle, kao što je navedeno u prethodnom odlomku, kod detekcije prednjeg zida uz prisutnost desnog zida, vrši se skretanje ulijevo. To se vrši postavljanjem brzina vrtnje kotača sljedećim jednadžbama [8]:

$$\omega_L = -s\frac{v_{turn}}{R}, \tag{2.10}$$

$$\omega_R = s\frac{v_{turn}}{R}, \tag{2.11}$$

gdje su:

- s - vrijednost koja ukazuje prati li se lijevi ili desni zid (s=1 - desni zid, s=-1 - lijevi zid) i
- v_{turn} - brzina mobilnog robota kod skretanja.

Vrijeme koje je potrebno da mobilni robot napravi skretanje određuje se jednadžbom[8]:

$$t_{turn} = \frac{\frac{b\pi}{2}}{v_{turn}}. \tag{2.12}$$

Uvjet da se izvršava proces pračenja desnog zida jest da i prednji i zadnji desni senzor detektiraju zid sa desne strane. U trenutku kada jedan od njih ne ispuni uvjet (a to je uglavnom prednji senzor jer je kretanje u većini slučajeva u naprijed) mobilni robot zaključuje da se sa desne strane robota nalazi slobodan prostor te on radi zavoj udesno. Detekcija desnog zida vrši se sljedećom jednadžbom [8]:

$$|d_{FR} - d_{wall}| < d_{wallTol} \&\& |d_{RR} - d_{wall}| < d_{wallTol}, \tag{2.13}$$

gdje je:

- $d_{wallTol}$ - udaljenost tolerancije kojom mobilni robot usporedno detektira prisutnost zida.

Udaljenost do zida računa se sljedećom jednadžbom [8]:

$$d_{wall} = \frac{C}{2} - \delta_{wall} - \frac{\tau}{2}, \tag{2.14}$$

gdje je:

- δ_{wall} - lateralna udaljenost između bočnog senzora i centra robota.

Kao što je navedeno, detekcijom slobodnog prostora sa desne strane robota, vrši se zavoj udesno, i to određivanjem brzina vrtnje kotača mobilnog robota dobivenim sljedećim jednadžbama[8]:

$$\omega_{curve} = \frac{v_{curve}}{C/2}, \tag{2.15}$$

$$\omega_L = \frac{(v_{curve} + s \cdot b \cdot \omega_{curve})}{R}, \tag{2.16}$$

$$\omega_R = \frac{(v_{curve} - s \cdot b \cdot \omega_{curve})}{R}, \tag{2.17}$$

gdje su:

- ω_{curve} - kutna brzina mobilnog robota u zavoju i
- v_{curve} - linearna brzina mobilnog robota u zavoju.

Vrijeme koje je potrebno za izvršavanje dobiva se sljedećim izrazom [8]:

$$t_{curve} = \frac{\frac{C \cdot \pi}{4}}{v_{curve}}, \tag{2.18}$$

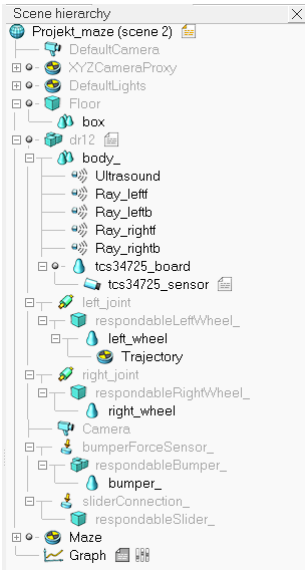
gdje je:

- t_{curve} - vrijeme potrebno za izvršenje zavoja mobilnog robota.

2.5.3. Simulacija mobilnog robota u CoppeliaSim programskom okruženju

U ovom diplomskom radu simulacija mobilnog robota vrši se unutar CoppeliaSim programskog paketa. Priprema simulacije sastoji se od nekoliko postpupaka koji će biti detaljnije opisani u nastavku. Ti postupci su izrada hijerarhijskog stabla, izrada modela mobilnog robota, izrada labirinta te pisanje LUA skripte mobilnog robota.

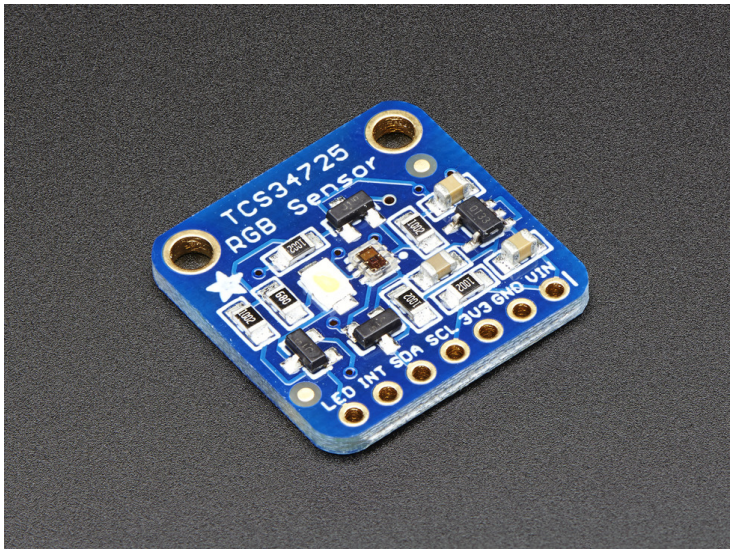
Hijerarhijsko stablo predstavlja temeljni koncept koji opisuje organizaciju i odnos između objekata u simulacijskoj sceni. Formira strukturu sličnu stablu, gdje su objekti raspoređeni hijerarhijski na temelju odnosa roditelj-dijete.



Slika 2.7. *Hijerarhija scene algoritma sljedbenika desnog zida*

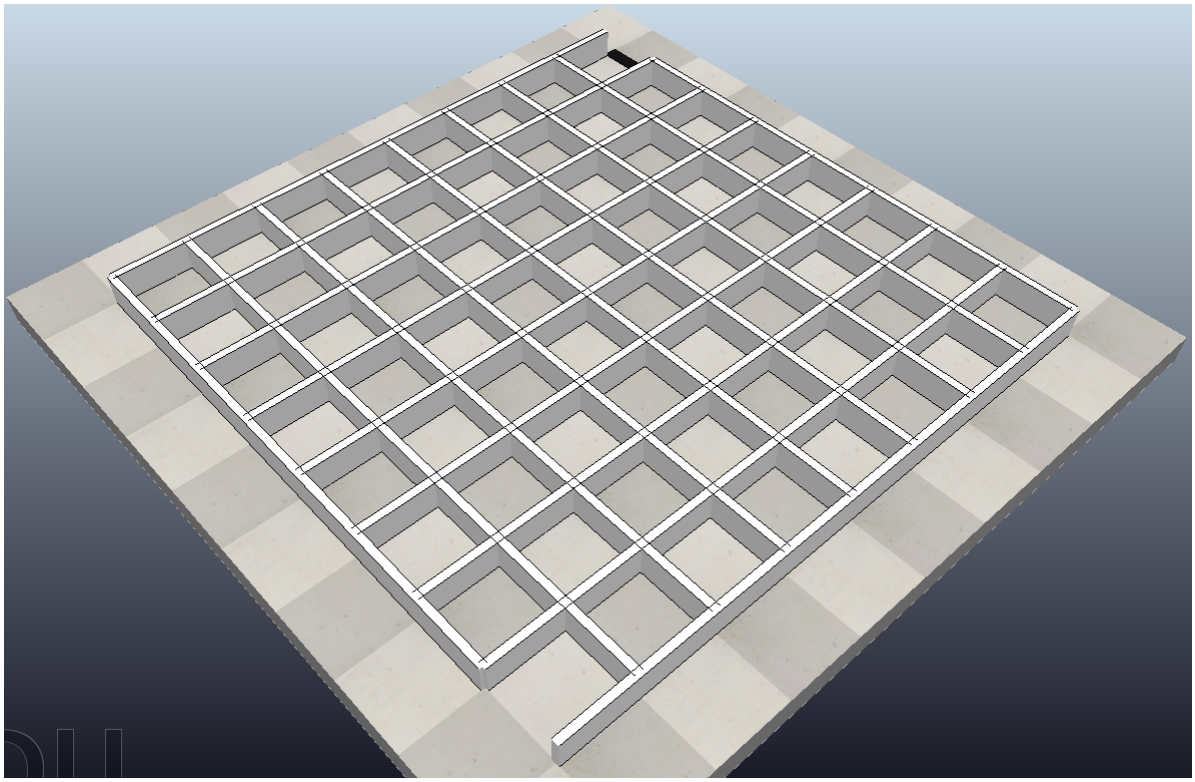
Na Slici 2.7 prikazano je hijerarhijsko stablo scene izrađene u CoppeliaSim-u koje predstavlja strukturu ove simulacije. Primjerice, objekt "body_" predstavlja model tijela robota, a on je "roditelj" objektima poput senzora, zglobova te kotača mobilnog robota. Iz prikazanog hijerarhijskog stabla vide se djelovi mobilnog robota. Sastoji se od tijela koje je sačinjeno od raznih primitivnih oblika, različitih senzora te dva revolutna zgloba na koja su spojena dva kotača. Nadalje, u stablu se nalaze objekt labirinta, kao i graf, koji se koristi za prikazivanje krivulje putanje mobilnog robota.

Prethodno je navedeno da je model mobilnog robota preuzet iz knjižnice CoppeliaSim programoskog paketa. Nadogradnje koje su napravljene na samom robotu su dodavanje 4 LIDAR senzora na bočne strane mobilnog robota, dodavanje jednog ultrazvučnog senzora na prednji kraj robota te dodavanje TCS34725 senzora na dno robota za prepoznavanje ciljne crte, i za dodatne upotrebe kasnije. Senzor TCS34725 ima infracrveni filter za blokiranje, integriran je na čipu i lokaliziran na fotodiode za senzore boja, smanjuje infracrvenu spektralnu komponentu dolaznog svjetla i omogućuje točna mjerenja boja. Izgled senzora prikazan je na Slici 2.8.



Slika 2.8. *Senzor TCS34725*

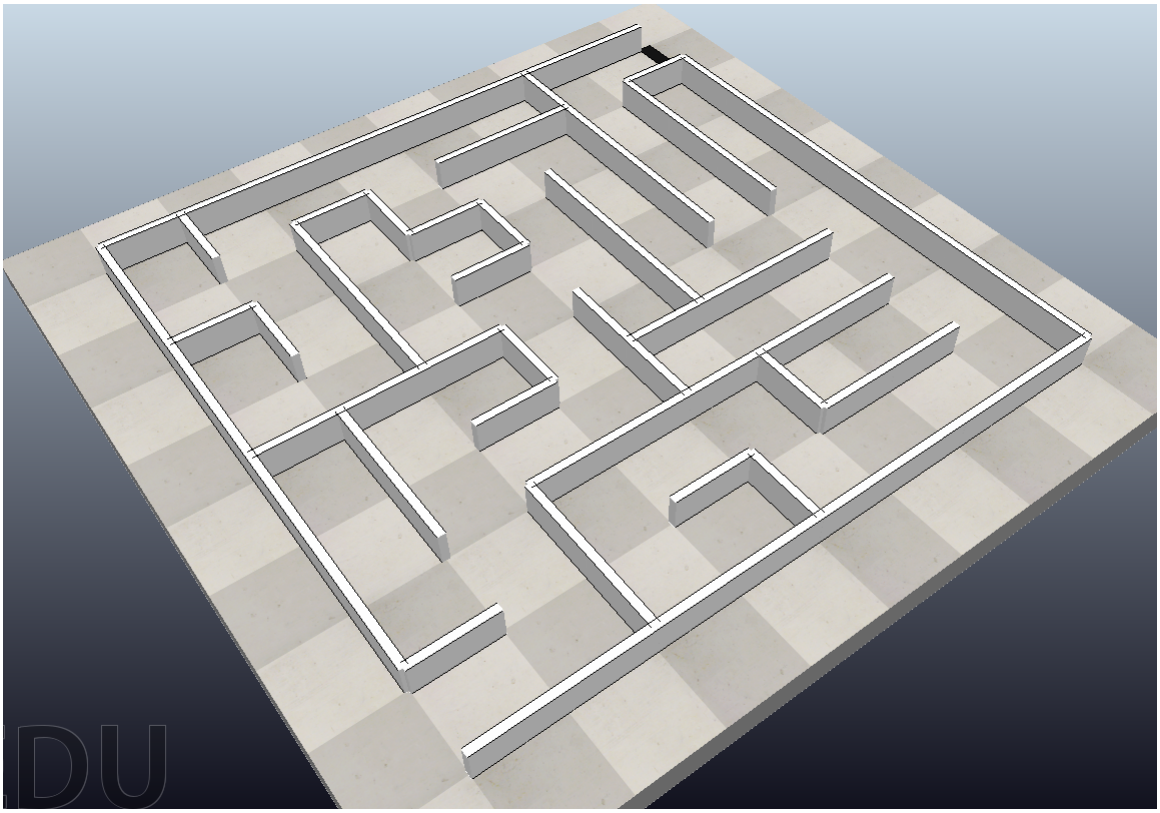
Model labirinta je polje dimenzije 8 redova i 8 stupaca ćelija ispunjeno zidovima. Taj model prikazan je na Slici 2.9.



Slika 2.9. *Model labirinta dimenzija 8 redova i 8 stupaca*

Labirint sa slike 2.9 izrađen je na način da se iz modela sa slike 2.9 uklone određeni zidovi čime se stvara put, pomoću kojega se robot može kretati od početka do cilja, ali i da se to polje

može smatrati labirintom. Završna verzija labirinta nalazi se na Slici 2.10.



Slika 2.10. *Prikaz labirinta za algoritam sljedbenika desnog zida*

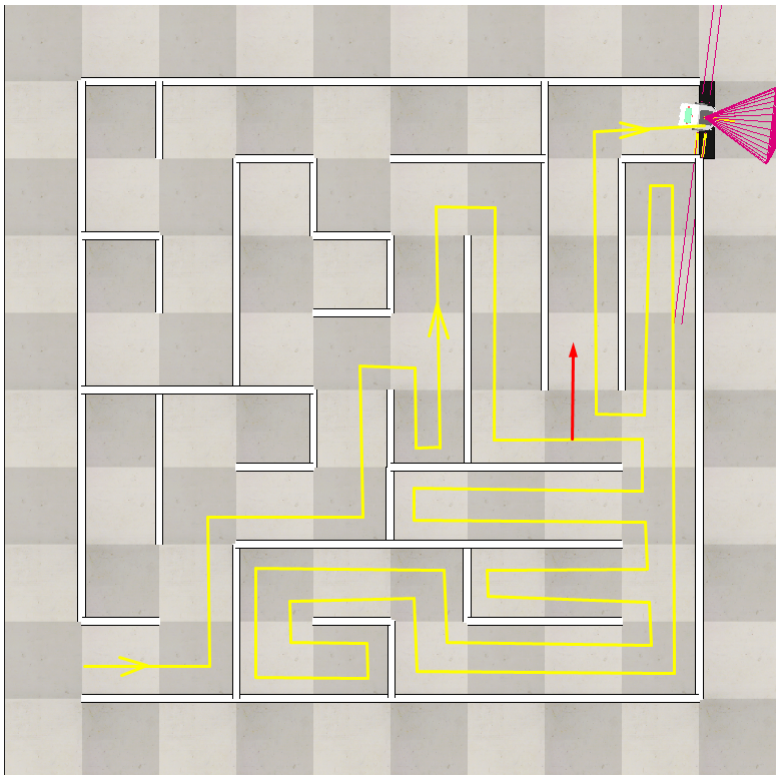
Kao što se na slikama vidi, sa donje strane vanjskih zidova labirinta nalazi se prolaz, što predstavlja početak labirinta, isto tako se sa gornje strane vanjskih zidova labirinta nalazi prolaz, koji predstavlja cilj, što se može i vidjeti crnom ciljnom crtom na podu.

Programiranje mobilnog robota sastoji se od postavljanja njegovih parametara (fizičkih, dinamičkih, kinematičkih), pisanja funkcija robota koristeći kinematičke zakone navedene u poglavlju Dinamičke i kinematičke značajke robota, postavljanje raznih uvjeta. Lua skripta za operaciju mobilnim robotom sa alogirmom sljedbenika zida nalazi se u Prilogu 9.1.

Pod programiranje mobilnog robota podrazumijeva se pisanje takozvane "child" skripte mobilnog robota. Taj proces počinje sa inicijalizacijom raznih elemenata mobilnog robota funkcijom "sysCall_init". Ti elementi su razni objekti poput lijevog i desnog kotača, ultrazvučnog, LIDAR i RGB senzora boja. Također se inicijaliziraju parametri poput polumjera kotača, referentnih brzina, međusobnoj udaljenosti između senzora i tako dalje. Tu se još nalaze i unaprijed izračunate vrijednosti za skretanje robota. To su vrijednosti ω_L i ω_R, koji su izvedeni individualno za potrebne radnje prethodno u radu.

Bitan element skripte je "sysCall_actuation". Ova funkcija se poziva kada je potrebno odrediti brzinu lijevog i desnog kotača mobilnog robota s obzirom na njegovo trenutno stanje (kreće li se ravno, skreće li). Sljedeća funkcija koja je bitna za rad skripte je "sysCall_sensing" koja se poziva kada je potrebno očitanje nekog od senzora na mobilnom robotu. Očitavanjem vrijednosti sa LIDAR senzora na bočnim stranama mobilnog robota nastoji se izjednačiti udaljenost sa jedne i druge strane, te se time omogućava kretanje po sredini puta. Funkcija se poziva i kada se ispred mobilnog robota pojavi zid, u kojem slučaju ultrazvučni senzor daje informaciju robotu da stane i zakrene se za 90 stupnjeva. Kada sa desne strane robota nema zida, LIDAR senzori daju tu informaciju i mobilni robot skreće udesno. Konačno, kada mobilni robot dosegne ciljnu točku, RGB senzor daje informaciju da je mobilni robot završio putanju i robot se zaustavlja. Sve ove radnje nalaze se u Tablici 2.3.

Nakon svih prethodnih koraka mobilni robot pozicionira se na ulaz labirinta te se pritisne tipka compile. Rezultat simulacije je linija po kojoj se mobilni robot kreće, a nalazi se na Slici 2.11.



Slika 2.11. *Rezultat simulacije algoritma sljedbenika desnog zida*

Na Slici je vidljivo da mobilni robot uspješno prolazi kroz labirint od početne točke do cilja, te se na njemu zaustavlja. Ono što je očito je da je put koji je mobilni robot prešao kako bi došao do ciljne točke labirinta suboptimalan, te bi se korištenjem neke od metoda teorije grafa ili UI put

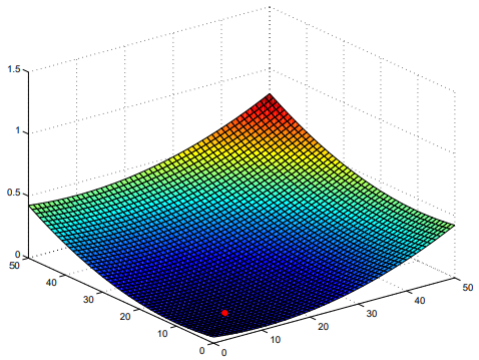
znatno skratio. Na dijelu gdje se nalazi crvena strelica usmjerena prema gore mobilni robot bi znatno skratio vrijeme odnosno put kretanja da je tu skrenuo prema lijevo. On tu po algoritmu sljedbenika desnog zida nastavlja ići ravno te putuje kroz dio labirinta koji nema izlaza, te se na kraju vraća na tu istu točku.

2.6. Planiranje putanje uz izbjegavanje prepreka

Izbjegavanjem prepreka smatra se oblikovanje putanje mobilnog robota kako bi savladao neočekivane prepreke [10]. Za navedeno izbjegavanje prepreka mobilnim robotom koriste se razne metode i algoritmi, od kojih su najpopularniji:

1. BUG algoritmi: Jednostavne metode za izbjegavanje neočekivanih prepreka na putanji mobilnog robota. Najpoznatije varijante bug algoritama su bug 1 i bug 2 algoritmi, o kojima će se više komentirati kasnije.

2. Metode potencijalnog polja: Mobilni robot smatra se česticom koja se kreće uronjena u potencijalnom polju generiranom ciljem i preprekama prisutnim u prostoru [10].



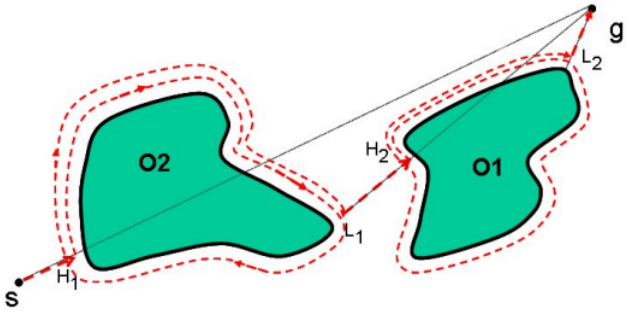
Slika 2.12. *Prikaz privlačnog potencijala kod metode potencijalnog polja [10]*

3. VFH algoritam (Vector Field Histogram): temeljen na VFF algoritmu (Virtual Force Field), metoda izbjegavanja prepreka u stvarnom vremenu, omogućava mobilnom robotu izbjegavanje nepoznatih prepreka dok istovremeno usmjerava robota prema cilju [10]. Metoda se vrši u 3 koraka: Prvo se izrađuje 2-dimenzionalni kartezijska histogramska mreža koja predstavlja prepreke, zatim se uzima prozor u kojem se nalazi mobilni robot, te se ta 2d mreža viltrira u 1d polarni histogram, a na kraju se računa kut usmjeravanja i kontrola brzine iz 1d polarnog histograma kao rezultat optimizacijske procedure.

2.6.1. Bug algoritmi

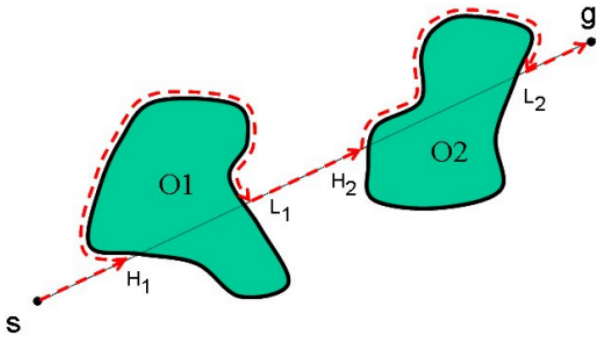
Cilj algoritama je generirati putanju mobilnog robota bez sudaranja sa okolinom od točke s "start" do točke g "goal". Kao što je već navedeno, postoji više varijanti bug algoritama, od kojih su najpopularnije bug 1 i bug 2.

Pojednostavljeno, bug 1 algoritam radi na način da se mobilni robot počne kretati sa startne točke prema ciljnoj. Kada naiđe na prepreku, mobilni robot obiđe cijelu konturu prepreke. Kada to završi, po istoj konturi se kreće prema točki koja je najbliža ciljnoj, te na njoj nastavi putanju prema ciljnoj točki. Proces se ponavlja dok robot ne stigne na odredište. Prikaz bug 1 algoritma nalazi se na Slici 2.13.



Slika 2.13. *Grafički prikaz postupka bug 1 algoritma [10]*

S druge strane, kod bug 2 algoritma mobilni robot se kreće po liniji koja spaja početnu i finalnu točku. Naiđe li mobilni robot na prepreku, počinje se kretati po konturi te prepreke sve dok ne presječe liniju po kojoj se prethodno kretao. Tada prestaje obilaziti konturu prepreke te se nastavlja kretati po liniji, a ovaj proces se ponavlja dok robot ne stigne do ciljne točke. Prikaz bug 2 algoritma nalazi se na Slici 2.14.



Slika 2.14. *Grafički prikaz postupka bug 2 algoritma [10]*

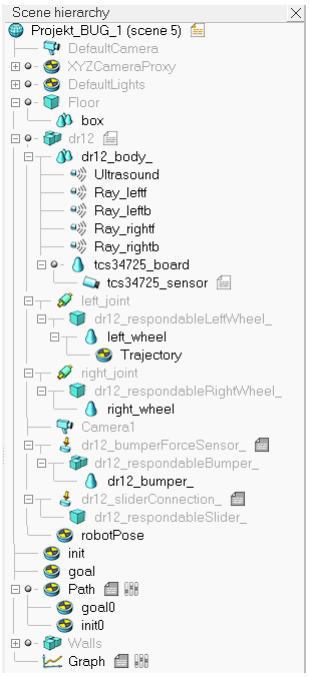
Generalno, bug 2 algoritam ima kraći put od bug 1 algoritma i više je efikasan u otvorenijem prostoru. Međutim, postoje situacije u kojima nije optimalan, a jedan taj primjer je da postoji mogućnost da robot zapne u labirintnoj strukturi. Ova usporedba može se sagledati sa još jednog ugla. Moglo bi se reći da je algoritam bug 2 "pohlepan" pošto uzima prvu moguću izlaznu točku sa konture prepreke [11]. Ta "pohlepnost može rezultirati jako dobro pronađenom rješenju putanje mobilnog robota, no to je uglavnom slučaj kod jednostavnijih prepreki. Kada se počinju pojavljivati kompleksnije prepreke može se desiti da bug 1 algoritam postane superiorniji nad bug 2 algoritmom. Za potrebe ovog diplomskog rada planiranje putanje uz izbjegavanje prepreka realizirati će se sa bug 2 algoritmom, te je on detaljnije opisan u nastavku.

Scena za izbjegavanje prepreka koristeći bug 2 algoritam postavljena je na način da se mobilni robot nalazi u prostoru. Postavljen je na pravac koji spaja početnu točku (onu na kojoj se mobilni robot nalazi inicijalno) i ciljnu točku, te se u tom prostoru između početne i ciljne točke nalaze razne prepreke. Algoritam kreće na način da se mobilni robot krene okretati oko svoje osi sve dok nije okrenut prema ciljnoj točki. Tada se počne kretati po tom pravcu sve dok se ne dogodi jedna od dvije moguće situacije. Prva situacija je da je mobilni robot stigao do ciljne točke. Tada se on zaustavlja i simulacija je gotova. Druga situacija je da je mobilni robot naišao na prepreku. Nakon toga on se okreće u lijevu ili desnu stranu nasumično. Kada su kontura prepreke te mobilni robot u paralelnom odnosu, robot se počne kretati uz konturu te prepreke. Kada vizualni senzor na mobilnom robotu detektira pravac po kojem se je inicijalno kretao, robot se prestane kretati te se opet okreće dok ne bude usmjeren prema ciljnoj točki. Proces se ponavlja sve dok robot ne stigne do ciljne točke.

Mobilni robot korišten za ovu simulaciju identičan je onome korišten za simulaciju sljedbenika desnog zida. Značajke njegovih senzora vidljive su u tablici 2.3. Također, matematički model izveden je po primjeru sljedbenika desnog zida.

2.6.2. Simulacija mobilnog robota u CoppeliaSim programskom okruženju

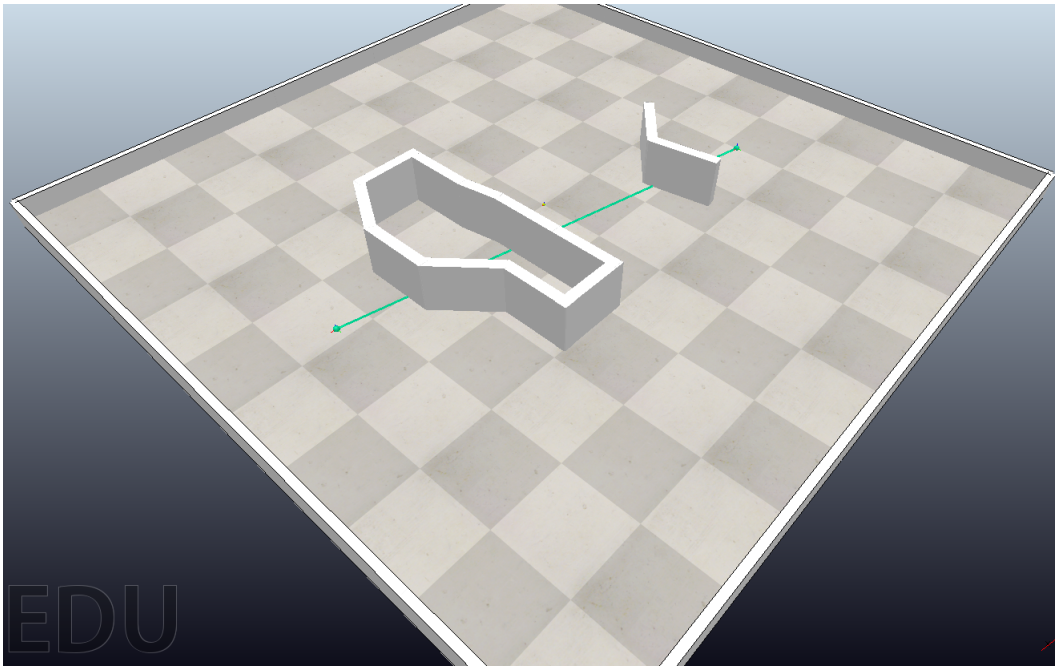
U nastavku će se objasniti proces izrade i simulacije scene rješenja izbjegavanja prepreka pomoću bug 2 algoritma.



Slika 2.15. *Hijerarhija scene bug 2 algoritma*

Na Slici 2.15 prikazano je hijerarhijsko stablo scene izrađene u CoppeliaSim-u koje predstavlja strukturu ove simulacije. S obzirom da se koristi isti mobilni robot kao i kod sljedbenika desnog zida, hijerarhija je većinom identična. Ono što je novo je jedan "node" imena "robotPose" koji služi za određivanje pozicije mobilnog robota u prostoru, točaka "init" i "goal" koje predstavljaju početnu i ciljnu točku te objekt puta "Path" koji vizualno prikazuje pravac po kojem se mobilni robot treba kretati kada ne obilazi konturu prepreke.

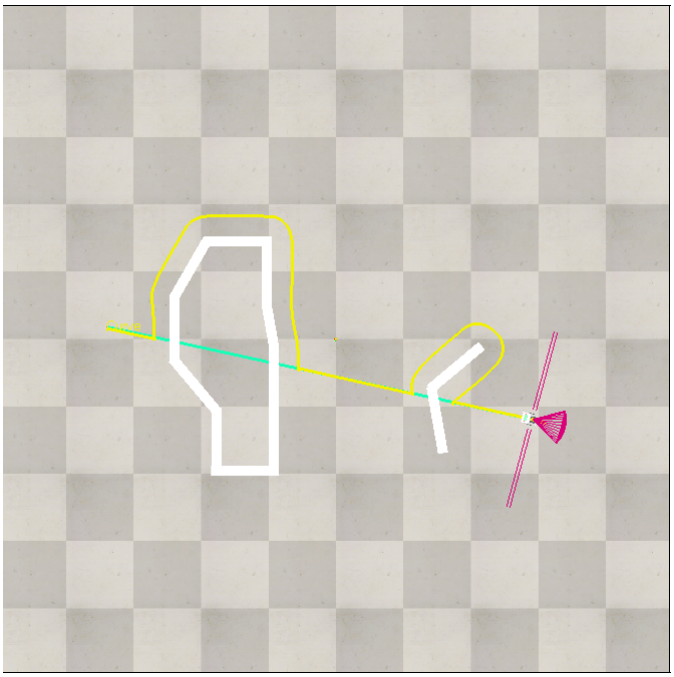
Prikaz scene sa preprekama i zadanom putanjom nalazi se na Slici 2.16.



Slika 2.16. *Prikaz scene bug 2 algoritma*

Kao što je prikazano scena se sastoji od jedne veće i jedne manje prepreke koje su sastavljene od višestrukih spojenih zidova. Lua skripta za operaciju mobilnog robota sa bug 2 algoritmom nalazi se u Prilogu 9.2.

Većina skripte za rješenje bug 2 algoritmom podudara se sa skriptom pisanom za prethodni problem. Inicijaliziraju se objekti, parametri i prethodno izračunate vrijednosti, pod aktuacijom se određuje brzina pojedinog kotača ω_L i ω_R, te se očitavaju razna očitavanja senzora. Razlika je u logici koju mobilni robot prati i njom stiže do cilja, koja je objašnjena prethodno. Pozicioniranjem mobilnog robota na početak pravca putanje i pokretanjem simulacije, on se kreće kroz scenu te izbjegava prepreke. Rezultat simulacije prikazan je na Slici 2.17.



Slika 2.17. *Rezultat simulacije bug 2 algoritma*

Zelena linija predstavlja pravac za kretanje prema cilju, a žuta linija predstavlja put koji je mobilni robot prešao. Kao što se vidi, on je uspješno zaobišao prepreke te je došao do ciljne točke.

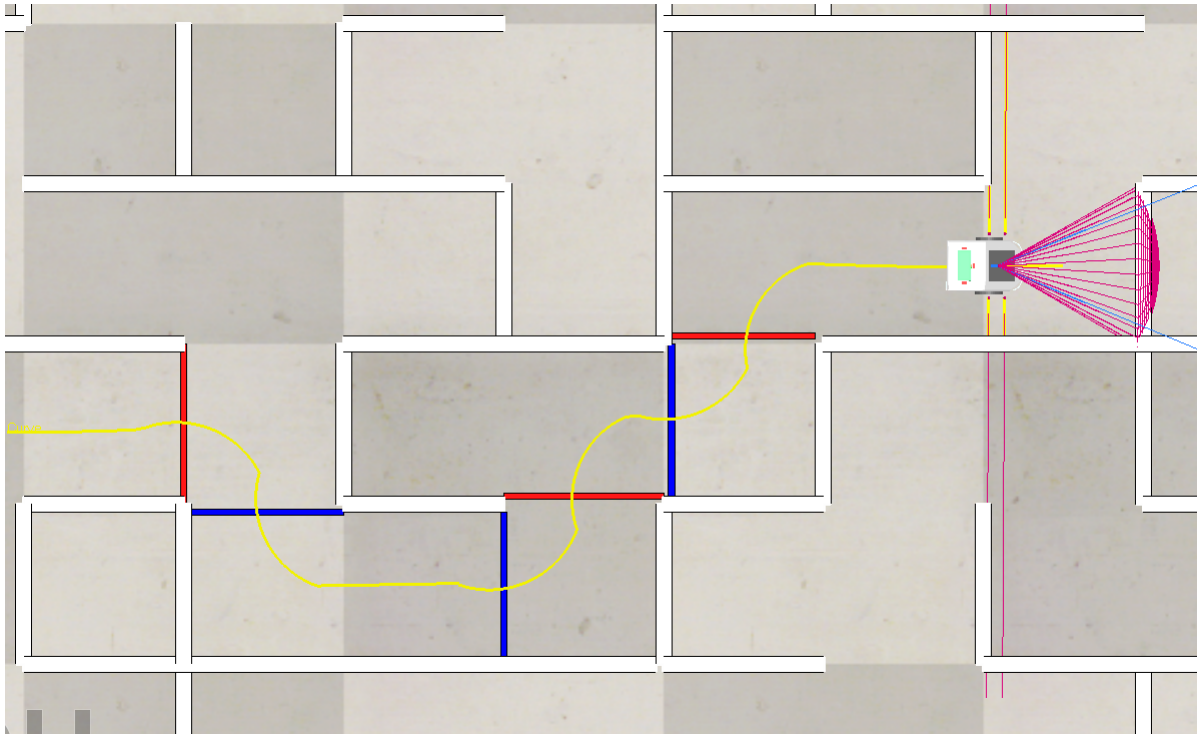
2.7. Rješenje za optimizaciju putanje

Kao što je već prethodno navedeno u uvodu, za putanju mobilnog robota primjeniti će se umjetna inteligencija. Kao rezultat te primjene biti će optimalan put kroz labirint koji mobilni robot u CoppeliaSim okruženju mora proći. Ono što je bitno napomenuti je da je kod ovog rješenja potrebno mobilnom robotu dati upute kojim putem se mora kretati, odnosno na kojem dijelu labirinta mora skrenuti. Zbog ovog detalja dio logike koja se nalazi kod sljedbenika desnog zida i bug algoritma postaje nepotrebna. Potrebno je samo razviti način na koji mobilni robot dobije informaciju o tome gdje mora skrenuti a gdje ne. To je lako izvedivo korištenjem njegovih senzora.

Jedno jednostavno rješenje je korištenje TCS34725 senzora za detekciju znaka za skretanje u labirintu. Pošto je ovaj senzor uperen prema podu, to znači da se znakovi moraju nalaziti na podu labirinta. Pošto odabrani senzor ima mogućnost prepoznati RGB spektar, odabrano rješenje je korištenje plave ili crvene trake na podu labirinta kao indikacija za skretanje mobilnog robota. Dakle, ako bi na podu bila crvena traka, mobilni robot bi tu crvenu traku detektirao pomoću TCS34725 senzora te bi potom skrenuo lijevo. Ako bi na podu bila plava traka, mobilni robot bi skrenuo

desno.

Implementacija ove modifikacije je jednostavna. Logika za praćenje desnog / lijevog zida se briše te se dodaje dio koda pomoću kojega se vrši detekcija boje sa senzorom, kao i upravljanje kotačima mobilnog robota za skretanje. Mobilni robot bi se kao i u primjeru sljedbenika zida kretao između dvaju zidova po sredini prolaza. Prikaz primjera ovog rješenja nalazi se na Slici 2.18.



Slika 2.18. *Kretanje mobilnog robota pomoću plavih i crvenih znakova na podu labirinta*

Lua skripta za operaciju mobilnog robota korištenjem algoritma za prepoznavanje znakova nalazi se u Prilogu 9.3..

3. UMJETNA INTELIGENIJA

Umjetna inteligencija je podskup računalne znanosti koji se bavi razvojem i proučavanjem algoritama i tehnika koje omogućuju strojevima da djeluju inteligentno. Cilj UI-a je razviti računalne sustave koji mogu obavljati zadatke koji zahtijevaju ljudsku inteligenciju, kao što su prepoznavanje uzoraka, zaključivanje, učenje, planiranje, jezik, percepcija i rješavanje problema. UI se primjenjuje u sve širim područjima, uključujući robotiku, autonomna vozila, medicinu, financije, igre, pretraživanje na webu, prirodni jezik obrada i mnoge druge industrije i disciplini. [12]. Kako bi se bolje shvatila umjetna inteligencija, potrebno je prvo definirati inteligenciju. Inteligencija je sposobnost učenja iz iskustva i prilagođavanja, oblikovanja i odabira okruženja [13]. Postoje mnoge i različite teorije inteligencije, koje su detaljno pregledane drugdje. Postoji nekoliko vrsta teorija inteligencije. Najpoznatije teorije bile su psihometrijske teorije, koje konceptualiziraju inteligenciju u terminima neke vrste "karte" uma. Takve teorije određuju temeljne strukture za koje se pretpostavlja da su temeljne za inteligenciju, temeljne na analizama individualnih razlika u uspješnosti ispitanika na psihometrijskim testovima. Novija vrsta teorije je teorija sustava, koja pokušava okarakterizirati sustav struktura i mehanizama uma koji čine inteligenciju.

Postoje različite grane UI-a, uključujući:

- Strojno učenje (eng. Machine Learning): To je podskup UI-a koji se fokusira na omogućavanje strojevima da uče iz podataka bez eksplicitnog programiranja. Algoritmi strojnog učenja analiziraju velike količine podataka kako bi identificirali uzorke, davali prognoze i donosili informirane odluke.

- Duboko učenje (eng. Deep Learning): Duboko učenje je specijalizirana grana strojnog učenja koja koristi umjetne neuronske mreže inspirirane ljudskim mozgom. Algoritmi dubokog učenja sposobni su učiti i izvlačiti kompleksne uzorke iz ogromnih količina nestrukturiranih podataka, poput slika, zvuka i teksta.

- Obrada prirodnog jezika (eng. Natural Language Processing): Obrada prirodnog jezika (skraćeno OPJ) uključuje omogućavanje strojevima da razumiju, interpretiraju i odgovaraju

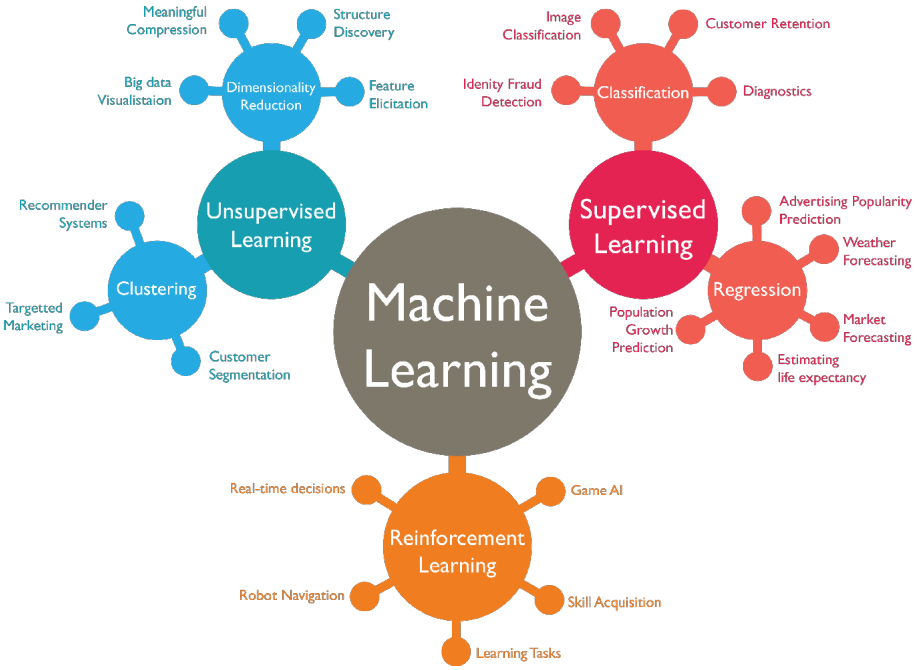
na ljudski jezik. Uključuje zadatke poput prijevoda jezika, analize sentimenta, prepoznavanja govora i razvoja chatbota.

- Računalni vid (eng. Computer Vision): Računalni vid usredotočen je na davanje strojevima sposobnosti "vida" i interpretaciju vizualnih podataka, poput slika i videozapisa. Uključuje zadatke poput prepoznavanja slike, klasifikacije slika i segmentacije slika.
- Robotika (eng. Robotics): Robotika kombinira UI i inženjering kako bi se dizajnirali i izgradili roboti koji mogu interaktirati s fizičkim svijetom. Roboti s umjetnom inteligencijom mogu izvršavati zadatke poput autonomne vožnje, manipulacije objektima i interakcije s ljudima.
- Genetski algoritam (eng. Genetic algorithm): Podskup evolucijskih algoritama inspirirani procesima prirodne selekcije i genetike. Uključuju optimizaciju i problem pretrage koristeći metode poput selekcije, mutacije te križanja.

Za okvire ovog rada razmatrati će se alati za optimizaciju rješenja, uzevši u obzir da je to glavni zadatak ovoga rada. Jedna od metoda često korištena za optimizaciju spada pod strojno učenje, koje će se detaljnije razraditi u nastavku.

3.1. Strojno učenje

Strojno učenje je dinamično područje unutar umjetne inteligencije koje omogućava računalima da uče i poboljšavaju se na temelju iskustva bez eksplicitnog programiranja [14]. Obuhvaća raznolik niz algoritama i tehnika s ciljem omogućavanja računalima da prepoznaju uzorke unutar podataka i donose informirane odluke ili predikcije na temelju te analize. Od predviđanja korisničkih preferencija do dijagnosticiranja bolesti, strojno učenje prožima se kroz različite industrije, transformirajući način na koji se pristupa složenim problemima i zadacima.

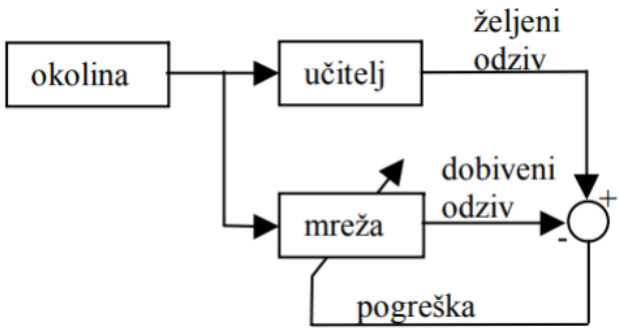


Slika 3.1. *Podjela strojnog učenja [18]*

Kao što je prikazano na Slici 3.1, tehnike strojnog učenja mogu se široko podijeliti na učenje pod nadzorom (eng. supervised learning), učenje bez učitelja (eng. unsupervised learning) i pojačano učenje (eng. reinforcement learning).

3.1.1. Učenje s učiteljem

Učenje s učiteljem uključuje treniranje modela na označenim podacima, gdje algoritam uči predviđati izlaz na temelju ulaznih značajki. Često se koristi u zadacima poput klasifikacije i regresije, gdje se algoritmu pruža skup podataka koji sadrži parove ulaza-izlaza te uči generalizirati uzorke radi predviđanja na novim, neviđenim podacima [14].



Slika 3.2. *Pojednostavljeni prikaz učenja s učiteljem*

Učitelj ima znanje o okolini u obliku parova ulaz-izlaz. Kao što je prikazano na Slici 3.2, pogreška je razlika između željenog i dobivenog odziva za neki ulazni vektor. Parametri mreže se mijenjaju pod utjecajem ulaznih vektora i signala pogreške. Ovaj proces se iterativno ponavlja sve dok mreža ne nauči imitirati učitelja. Nakon što je učenje završilo, učitelj više nije potreban i mreža može raditi bez nadzora.

3.1.2. Učenje bez učitelja

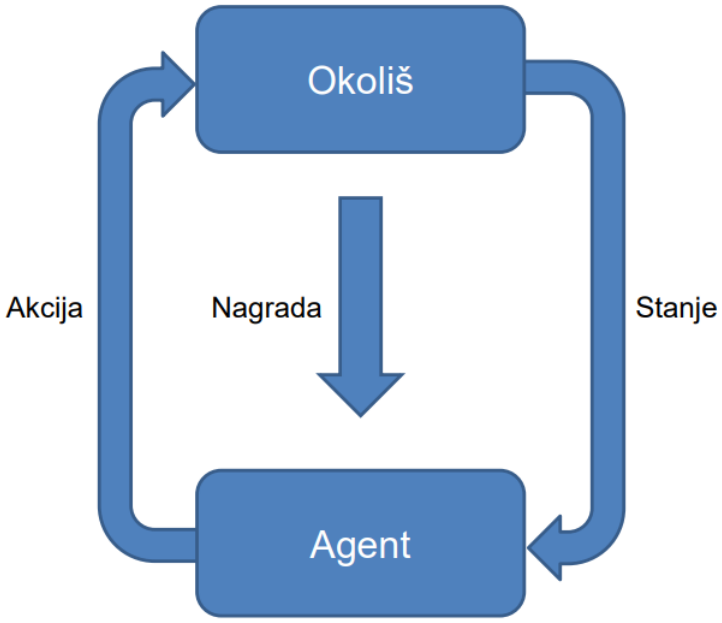
Učenje bez učitelja bavi se neoznačenim podacima, a algoritam ima za cilj otkriti skrivene obrasce ili strukture unutar podataka. Za razliku od učenja s učiteljem, ovdje nema predefiniranih izlaznih oznaka, pa algoritam mora sam zaključiti osnovnu strukturu podataka [14]. U nenadziranom učenju stroj jednostavno prima ulazne podatke, ali ne dobiva niti ciljne izlaze kao u učenju s učiteljem, niti nagrade iz svog okruženja. Može se činiti pomalo tajanstvenim zamišljati što bi stroj mogao naučiti s obzirom na to da ne dobiva nikakve povratne informacije iz svoje okoline [17]. Međutim, moguće je razviti formalni okvir za učenje bez učitelja na temelju ideje da je cilj stroja izgraditi prikaze ulaza koji se mogu koristiti za donošenje odluka, predviđanje budućih unosa, učinkovito komuniciranje ulaza drugom stroju itd. Jedan od najraširenijih zadataka učenja bez učitelja je grupiranje: otkrivanje potencijalno korisnih skupina ulaznih primjera. Npr., taksi agent može postupno razviti koncept "dana s dobrim prometom" i "dana s lošim prometom", a da mu učitelj nikada ne da označene primjere za svaki.

3.1.3. Pojačano učenje

Pojačano učenje uključuje treniranje agenta da interagira s okolinom kako bi postigao cilj kroz sustav pokušaja i pogreške (eng. trial and error). Agent uči navigirati okolinom prema akcijama koje poduzima i prima povratne informacije na temelju posljedica tih akcija. Kroz metodu pokušaja i pogreške, agent postupno uči optimalnu politiku za maksimiziranje kumulativnih nagrada tijekom nekog vremena [15].

Pojačano učenje u mnogo toga razlikuje se od učenja s učiteljem, kojim se smatra učenje iz skupa označenih primjera za obuku koje pruža iskusan vanjski učitelj. Svaki primjer je opis situacije zajedno sa specifikacijom - oznakom - ispravne akcije koju bi sustav trebao poduzeti u toj situaciji, što je često identificirati kategoriju kojoj situacija pripada [15]. Također se u mnoštvu

stvari razlikuje od učenja bez učitelja, koje se obično odnosi na pronalaženje strukture skrivene u zbirkama neoznačenih podataka. Jedan od izazova koji se javlja u pojačanom učenju, a ne u drugim vrstama učenja, kompromis je između istraživanja i eksploatacija (eng. exploration and exploatation). Da bi dobio veliku nagradu, agent pojačanog učenja mora preferirati akcije koje su pokušane u prošlosti i pokazale se učinkovitim u dobivanju nagrade. Ali da bi otkrio takve radnje, mora isprobati akcije koje nije odabrao prije. Agent mora iskoristiti ono što već zna kako bi dobio nagradu, ali također mora istražiti kako bi bolje odabrao akcije u budućnosti. Dilema je u tome što se ni istraživanje ni eksploatacija ne mogu nastaviti isključivo bez neuspjeha u zadatku. Agent mora isprobati niz radnji i postupno favorizirati one koje se čine najboljima. Na stohastičkom zadatku, svaka se radnja mora isprobati mnogo puta kako bi se dobila pouzdana procjena očekivane nagrade [15]. Pojednostavljeni prikaz pojačanog učenja nalazi se na Slici 3.3.



Slika 3.3. *Pojednostavljeni prikaz pojačanog učenja*

3.2. Q-učenje

Q-učenje (eng. Q-learning) je temeljni algoritam pojačanog učenja koji se pokazao učinkovitim u rješavanju širokog spektra problema sekvencijalnog donošenja odluka [16]. Pripada kategoriji model-free algoritama pojačanog učenja, što znači da uči izravno iz interakcije s okolinom bez potrebe za modelom dinamike okoline. Učenje se odvija slično metodi vremenskih razlika (eng.

Temporal difference learning): agent pokušava akciju u određenom stanju i procjenjuje njezine posljedice u smislu neposredne nagrade ili kazne koju prima i svoje procjene vrijednosti stanja u koje je odveden. Uzastopnim isprobavanjem svih radnji u svim stanjima, uči koje su sveukupno najbolje, ocjenjujući dugoročno sniženu nagradu. Q-učenje je primitivan oblik učenja, ali kao takav može djelovati kao temelj daleko sofisticiranijih uređaja [16].

Potrebno je zamisliti računalnog agenta koji se kreće po nekom diskretnom, konačnom svijetu, birajući jednu od radnji iz konačnog skupa radnji u svakom vremenskom koraku. Svijet čini kontrolirani Markovljev proces s agentom kao kontrolorom. U koraku n, agent je opremljen za registraciju stanja x_n svijeta i može izabrati svoju akciju a_n u skladu s tim. Agent dobiva probabilističku nagradu r_n, čija srednja vrijednost $r_{x_n}(a_n)$ ovisi samo o stanju i akciji, a stanje svijeta se vjerojatnosno mijenja u y_n prema zakonu:

$$Prob[y_n = y|x_n, a_n] = P_{x_n y}[a_n]. \tag{3.1}$$

Zadatak s kojim se agent suočava je određivanje optimalne strategije π, one koja maksimizira ukupnu umanjenu očekivanu nagradu. Pod umanjenom nagradom smatra se da su nagrade vrijedile manje od nagrada koje su primljene sada, i to za faktor $\gamma^s (0 < \gamma < 1)$. Drugim riječima, γ koji je bliži 1, prisiljava agenta da više uči sa dugoročnim planom, što znači da će proces učenja duže trajati, ali će model biti robusniji. Pod politikom π, dobrota stanja x je

$$V^{\pi}(x) = r_x(\pi(x)) + \gamma \sum P_{xy}[\pi(x)] V^{\pi}(y), \tag{3.2}$$

jer agent očekuje primiti $r_x(\pi(x))$ momentalno za odrađivanje akcije po preporuci od π, te zatim prelazi u stanje koje je njemu "vrijedno" $V^{\pi}(y)$ sa vjerojatnošću $P_{xy}[\pi(x)]$. Teorija dinamičkog programiranja (skraćeno DP) potvrđuje da postoji barem jedna optimalna stacionarna strategija π^* koja je takva da:

$$V^*(X) = V^{\pi^*}(x) = max_a \{R_x(a) + \gamma \sum_y P_{xy}[a] V^{\pi^*}(y)\}. \tag{3.3}$$

Iako ovo može izgledati kružno, zapravo je dobro definirano, a DP pruža brojne metode za izračunavanje V^* i jednog π^*, pod pretpostavkom da su $r_x(a)$ i $P_{xy}[a]$ poznati. Zadatak s kojim se suočava Q učenik je određivanje π^* bez početnog poznavanja tih vrijednosti. Postoje tradicionalne

metode za učenje $r_x(a)$ i $P_{xy}[a]$ dok se istovremeno izvodi DP, ali svaka pretpostavka ekvivalencije sigurnosti, tj. izračunavanje radnji kao da je trenutni model točan, skupo košta u ranim fazama učenja. Watkins (1989) klasificira Q-učenje kao inkrementalno dinamičko programiranje, zbog načina na koji korak po korak određuje optimalnu strategiju.

Za strategiju π, Q vrijednost se definira kao:

$$Q^{\pi}(x,a) = r_x(a) + \gamma \sum_{y} P_{xy}[\pi(x)]V^{\pi}(y). \tag{3.4}$$

Cilj Q-učenja je estimirati vrijednost Q za optimalnu strategiju. Unutar samog učenja, agentovo iskustvo sastoji se od slijeda različitih faza ili epizoda. U n-toj fazi, agent:

- promatra njegovo trenutno stanje x_n,
- odabire i izvodi radnju a_n,
- promatra naknadno stanje y_n,
- prima trenutnu nagradu r_n i
- prilagođava vrijednosti Q_{n-1} koristeći faktor učenja α_n prema:

$$Q_n(x,a) = \begin{cases} (1-\alpha_n)Q_{n-1}(x,a) + \alpha_n[r_n + \gamma V_{n-1}(y_n)] \text{ako su } x = x_n \text{ i } a = a_n \quad \text{, inače:} \\ Q_{n-1}(x,a), \end{cases} \tag{3.5}$$

gdje je

$$V_{n-1}(y) = max_b\{Q_{n-1}(y,b)\}. \tag{3.6}$$

Sljedeći teorem definira skup uvjeta pod kojima $Q_n(x,a) \to Q^*(x,a)$, ako $n \to \infty$. S obzirom na ograničene nagrade $|r_n| \leq R$, stope učenja $0 \leq \alpha < 1$ i

$$\sum_{i=1}^{\infty} \alpha_n^i(x,a) = \infty, \sum_{i=1}^{\infty} [\alpha_n^i(x,a)]^2 \infty, \forall x, a, \tag{3.7}$$

tada $Q_n(x,a) \to Q^*(x,a)$ ako $n \to \infty$, $\forall x, a$, sa vjerojatnošću 1.

4. PRIMJENA Q ALGORITMA ZA OPTIMIZACIJU PUTANJE MOBILNOG ROBOTA

Ideja za korištenje umjetne inteligencije, odnosno Q-učenja jest optimizacija putanje, odnosno pronalazak optimalnog puta mobilnog robota. Prepreka koju je potrebno savladati je kao i u primjeru rješenja algoritmom wall-followera, a to je labirint. Bitno je naglasiti da je sav kod korišten u svrhu implementacije Q-učenja za optimizaciju puta mobilnog robota pisan u programskom jeziku Python, koristeći VScodium, koji je besplatna licencirana verzija Microsoft-ovog tekstualnog editora Visual Studio Code. Rješenje je podjeljeno na tri skripte, u koje spadaju skripta za generaciju i grafički prikaz labirinta, skripta za treniranje Q-learning agenta, te skripta za korisničko sučelje.

4.1. Labirint

U postavljenom problemu zadan je labirint konstantne, fiksne veličine s r-brojem redaka i c-brojem stupaca (eng. rows - columns). Labirint je sastavljen od blokova koji mogu, ali ne moraju imati zidove između sebe. Blokovi su međusobno povezani na gornjem, donjem, lijevom i desnom rubu, osim kada se blok nalazi na vanjskom rubu labirinta ili postoji zid između dva bloka. Zidovi se nasumično generiraju tijekom stvaranja labirinta i statični su tijekom procesa učenja i testiranja. Labirint također sadrži ciljnu točku. Agent kojeg se obučava dobit će početnu točku i mora odrediti najkraći put od te početne točke do ciljne točke [19].

Za generiranje labirinta korišten je kod dan u prilogu 9.4.. Navedeni kod modificirana je verzija koda koji je dan u članku "Solving A Maze With Q-Learning" [19]. U nastavku je kod za generaciju i grafički prikaz labirinta detaljnije objašnjen.

Za potrebe pisanja skripte "maze.py" pomoću koje se generira i grafički prikazuje labirint, koriste se dvije biblioteke: numpy i random.

Numpy je popularna Python biblioteka za numeričko računanje, koja omogućuje rad s višedimenzionalnim poljima (nizovima) i pruža razne matematičke funkcije za izvođenje operacija na tim nizovima. Random je biblioteka koja pruža mogućnost generacije nasumičnog broja.

Nakon importanja biblioteka potrebno je definirati klasu "Maze" koja definira temeljne funkcionalnosti i atribute te klase, fokusirajući se na logičnu reprezentaciju i manipulaciju strukturom labirinta. Neki od atributa klase "Maze" su veličina ćelije, zidovi, početna i ciljna točke, put, te boja početne i ciljne točke, puta, označene ćelije i pozadine.

Zatim je potrebno definirati metodu "generate" kojom se generira labirint sa određenim brojem redova i stupaca. Taj broj je moguće odrediti po potrebi korisnika. Labirint se generira implementacijom DFS algoritma za generiranje labirinta. Ovaj proces opisan je u nastavku:

1. Slučajna ćelija odabrana je kao početna točka. Ta se ćelija stavlja u hrpu
2. Odabire se jedna nasumična ćelija među susjedima trenutne ćelije
3. Ako odabranu ćeliju još nije provjerio algoritam, razbijen je zid između trenutne ćelije i slučajne ćelije. Redoslijed radnji se ponavlja od prve točke, ali sa slučajnom ćelijom kao početnom točkom.
4. Ako je odabrana ćelija već posjećeni, odabire se druga ćelija iz susjedstva te ćelije i ponavlja se od koraka 3.
5. Ako su svi susjedi trenutne ćelije posjećeni, tada se prelazi na prethodnu ćeliju i ona postaje trenutna, a zatim se nastavlja od koraka 2.
6. Algoritam se zaustavlja kada se posjete sve ćelije [20].

Bitno je navesti i neke metode za postavljanje/dobivanje početnih i krajnjih točaka, pristup strukturi labirinta i manipuliranje odabranim blokovima.

Nakon toga se izrađuje grafički prikaz koji je bitan zbog daljnjeg korištenja optimizacije u CoppeliaSim-u. Koristi se metoda "generate_image" za generiranje vizualnog prikaza labirinta crtanjem njegovih komponenti na sliku. Postoje i pomoćne metode koje se koriste za grafičkih prikaz atributa klase "Maze" poput puta, početne i ciljne točke te zidova labirinta, kao i odabranog bloka. Bitno je naglasiti da se zidovi labirinta generiraju i brišu po "uputama" DFS algoritma generacije labirinta koristeći metode "__remove_walls". Drugim riječima, unutar koda potrebno je prvo nacrtati sve zidove koje je moguće, te zatim brisati određene zidove koji stoje između ćelija koje su povezane. Dakle, ako su dva bloka susjedna, potrebno je ukloniti zidove između njih.

Detaljnije pojašnjeno, prvo se provjerava jesu li redni indeksi dva bloka "first_point" i "second_point" jednaki. Ako su jednaki, to znači da su blokovi vertikalno poravnati. Unutar ove provjere:

- Ako je indeks stupca "first_point" veći od indeksa "second_point", to znači da je "first_point" desno od "second_point". U tom slučaju, uklanjaju se lijevi zid "first_point" i desni zid "second_point".
- Ako je indeks stupca "first_point" manji od indeksa "second_point", to znači da je "first_point" lijevo od "second_point". Ovdje se uklanjaju desni zid "first_point" i lijevi zid "second_point".

4.2. Učenje Q-agenta

Kao što je objašnjeno u prethodnom poglavlju, u Q-učenju agent prima povratnu informaciju iz okoline u obliku nagrade $R(A(s, s'))$ (nagrada za poduzimanje akcije prelaska iz stanja s u stanje s'). Cilj svakog algoritma podržanog učenja je maksimiziranje vrijednosti ove funkcije nagrađivanja tijekom vremena. Ovaj zadatak se postiže korištenjem matrice učenja, $Q(A(s, s'))$ (otuda i naziv 'Q-učenje'). Kada je obučen, agent može koristiti ovu Q matricu kako bi odredio koju radnju treba poduzeti odabirom stanja s' koje maksimizira $Q(A(s, s'))|_s$ (maksimiziranje očekivane dugoročne nagrade od poduzimanja radnje $A(s, s')$ pomicanjem agenta iz stanja s u stanje s' s obzirom da je agent trenutno u stanju s). To znači da nakon što se matrica učenja nauči, agent će moći odrediti optimalnu akciju iz bilo kojeg stanja. Ovaj proces je odrađen iterativnom aplikacijom Bellman-ove jednadžbe. Jednadžba 3.4 modificira se na način da:

$$V*(x) = max_a Q(x, a'), \tag{4.1}$$

stoga:

$$Q(x, a) = r(x, a) + \gamma max_a Q(\delta(x), a'). \tag{4.2}$$

Konačno, napiše li se izraz Q funkcije na način da se akcija a iz stanja x napiše u obliku akcije A iz stanja s u novo stanje s', Q funkcija izgleda ovako:

$$Q(A(s, s')) = R(A(s, s')) + \gamma max_{s''|s'}(Q(A(s', s''))). \tag{4.3}$$

s'' predstavlja stanje nakon novog stanja s', a izraz $max_{s''|s'}$ se čita kao ("pronađi sve očekivane nagrade za akcije koje prebacuju agenta iz stanja s' u stanje s" i uzmi maksimum"). Dakle, Q-agent trenira se maksimiziranjem Q matrice.

Za učenje Q-agenta korišten je kod dan u prilogu 9.5.. U nastavku je kod za treniranje Q-agenta pomoću Q-učenja detaljnije objašnjen.

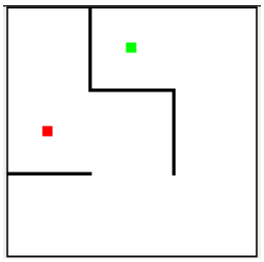
Za potrebe pisanja skripte "qlearn_agent.py" pomoću koje se trenira Q-agent, koriste se numpy i random knjižnice, kao i moduli sys i maze. Knjižnice numpy i random objašnjene su u prijašnjem poglavlju. Modul sys u Pythonu pruža razne funkcije i varijable koje se koriste za manipuliranje različitim dijelovima Python runtime okruženja. Omogućuje rad na interpreteru jer omogućuje pristup varijablama i funkcijama koje su u snažnoj interakciji s interpreterom. Nakon importanja modula "maze", skripta može pristupiti svim funkcijama, varijablama ili klasama definiranim unutar skripte "maze.py".

Nakon importanja biblioteka i modula, definira se klasa "QLearnAgent" koja predstavlja Q-agenta. Sastoji se od različitih metoda koje će biti opisane u nastavku. Metoda "__init__" služi za inicijalizaciju raznih atributa agenta poput matrice nagrade, matrice učenja, broja stanja i slično. Metoda "is_trained" vraća vrijednost kada Q-agent bude naučen. Iduća metoda je iznimno bitna, a to je metoda "initialize" koja inicijalizira Q i R matricu tako da Q matricu inicijalizira u nulu (na početku ne znamo ništa o labirintu), a R matricu (matricu nagrade) ispuni sa vrijednostima ovisno o tome kakva je akcija:

- Ako akcija pokreće agenta prema ciljnom stanju: R = 1,
- Ako je akcija valinda: R = 0 i
- Ako akcija nije validna: R = -1.

Dva su ograničenja koja treba imati na umu. Prvo, osim ako agent nije na cilju, ne može ostati u istom stanju između dvije iteracije. Dakle, ne smije se dopustiti da akcija $A(s, s)$ ikada bude izabrana. Sljedeće ograničenje je da ako postoji zid između dva bloka ili ako bi potez uzrokovao pomicanje agenta izvan rubova labirinta, tada je potez nevažeći.

Labirint za primjer dimenzija 3 reda i 3 stupca prikazan je na Slici 4.1.



Slika 4.1. *Primjer labirinta dimenzija 3 reda i 3 stupaca*

Za labirint prikazan na slici, matrica nagrade je:

$$R = \begin{bmatrix} -1 & -1 & -1 & 0 & -1 & -1 & -1 & -1 & -1 \\ -1 & 1 & 0 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 & -1 & 0 & -1 & -1 & -1 \\ 0 & -1 & -1 & -1 & 0 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & 0 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & 0 & -1 & -1 & -1 & -1 & -1 & 0 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & -1 & 0 & -1 & 0 & -1 & 0 \\ -1 & -1 & -1 & -1 & -1 & 0 & -1 & 0 & -1 \end{bmatrix}.$$

Slijedi metoda "train" koja služi za treniranje agenta Q-učenja. Uzima parametre "gamma" (γ iz Bellman-ove jednadžbe) i "min_change_per_epoch", koji određuje minimalnu promjenu u q matrici. Iteracijom po epohama nastoji dostignuti konvergenciju. Da bi se matrica istrenirala, počne se sa postavljanjem agenta u slučajno stanje s. Zatim se dobiva popis sljedećih mogućih stanja gledajući R matricu. Od važećih poteza, nasumično se odabire sljedeće stanje s'. S obzirom na sljedeće stanje s', gledaju se sve vrijednosti $Q(A(s', s"))$ za sve važeće $A(s', s")$ i uzima se najveća vrijednost. Zatim se to kombinira s funkcijom nagrađivanja $R(A(s, s'))$ kako je određeno Bellman-ovom jednadžbom 4.3 i ažurira se vrijednost $Q(A(s, s'))$. Skraćeno, algoritam izgleda ovako:

Dok konvergencija nije dostignuta:

1. Počni u nasumičnom stanju,

2. Dok se ne dostigne ciljno stanje:

    a) Odaberi nasumično sljedeće stanje,

    b) Dobij najbolju q vrijednost kroz sve validne akcije od sljedećeg stanja,

    c) Ažuriraj q matricu i

    d) Pomakni se u novo stanje.

Na kraju je potrebno q matricu normalizirati kako bi se izbjeglo prelijevanje (overflow). Drugim riječima, cijela matrica se dijeli sa najvećom vrijednosti iz matrice, tako da najveća vrijednost unutar matrice bude 1. Treniranje se zaustavlja kada promjena Q vrijednosti padne ispod praga "min_change_per_epoch". Sljedeća matrica prikazuje izgled istrenirane q matrice za labirint sa primjera:

$$Q = \begin{bmatrix} 0.00 & 0.00 & 0.00 & 0.26 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.64 & 0.00 & 0.00 & 0.00 \\ 0.21 & 0.00 & 0.00 & 0.00 & 0.33 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.26 & 0.00 & 0.00 & 0.00 & 0.41 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.51 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.41 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.33 & 0.00 & 0.33 & 0.00 & 0.51 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.64 & 0.00 & 0.41 & 0.00 \end{bmatrix}.$$

Konačno, metoda je "solve" koja predviđa optimalan put kroz labirint. Nakon što je agent istreniran, može se lako riješiti labirint s bilo koje početne točke uzimajući maksimalnu vrijednost $Q(A(s, s'))$ za bilo koje dano stanje dokse ne postigne ciljno stanje. Skraćeno, algoritam izgleda ovako:

1. Inicijaliziraj put,

2. Dok ciljno stanje nije dostignuto:

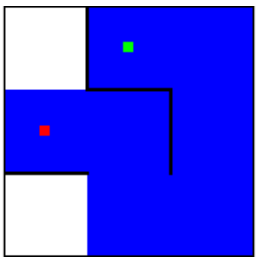
    a) Dobij sva moguća sljedeća stanja,

    b) Odaberi stanje koje maksimizira q vrijednost i

c) Pomakni se u to stanje i dodaj ga u put.

Na primjer, u gornjem labirintu, agent je inicijalno u stanju 3. Agent može ići u stanje 0 ili stanje 4. Uspoređuju se $Q(A(s, s'))$ vrijednosti za oba ta pomaka gledajući na Q matricu: $Q[3][0] == 0,21$, i $Q[3][4] == 0,33$. Budući da je $Q[3][4] > Q[3][0]$, agent odabire akciju A(3,4) i prelazi u stanje 4. Iz stanja 4 može se pomaknuti u stanje 3 ili stanje 7. $Q[4][3] = 0,26$, a $Q[4][7] = 0,41$, pa agent odabire A(4,7) i prelazi u stanje 7. Proces se ponavlja dok se ne dođe do terminalnog stanja, stanja 1, dajući konačni put [3,4,7,8,5,2,1].

Prikaz rješenog labirinta iz primjera sa nacrtanim optimalnim putem od početne do ciljne točke nalazi se na Slici 4.2



Slika 4.2. *Riješeni primjer labirinta dimenzija 3 reda i 3 stupaca*

U kodu se koriste još metoda "__maze_dims_to_state" za konverciju iz (x,y) koordinata u numeričko stanje, kao i metoda "__state_to_maze_dims" za konverciju numeričkih stanja u (x,y) koordinate.

4.3. Korisničko sučelje

Za izradu korisničkog sučelja korišten je kod dan u prilogu 9.6.. U nastavku je kod za izradu korisničkog sučelja detaljnije objašnjen.

Za potrebe pisanja skripte "main_window.py" pomoću koje se izrađuje korisničko sučelje, koriste se knjižnica GI, te je potrebno pozvati module sys, maze i qlearn_agent. Nakon importanja modula "maze" i "qlearn_agent", skripta može pristupiti svim funkcijama, varijablama ili klasama definiranim unutar skripta "maze.py" i "qlearn_agent.py". Knjižnica GI (GObject Introspection) pruža jezična povezivanja za različite biblioteke i API-je, dopuštajući njihovu upotrebu iz različitih programskih jezika. GI se primarno poziva za korištenje alatom GTK, pomoću kojega je razvijeno korisničko sučelje (eng. Graphical User Interface - GUI)

GTK je skup alata opće namjene za izradu korisničkih sučelja za desktop aplikacije. GTK pruža skup biblioteka i alata za izgradnju GUI-ja s različitim programskim jezicima, uključujući C, C++, Python i druge. Nudi widgete (elemente korisničkog sučelja), upravitelje izgleda, mehanizme za rukovanje događajima i još mnogo toga, omogućujući razvojnim programerima stvaranje interaktivnih i vizualno privlačnih aplikacija. Za instalaciju skupa alata GTK, potrebno je sa web strance [21] preuzeti MSYS2. MSYS2 je platforma za distribuciju i izgradnju softvera za Windows. Cilj mu je pružiti okruženje slično Unixu za izgradnju i pokretanje softvera na Windows operativnim sustavima. Sastoji se od tri glavne komponente:

1. MSYS2 Runtime: Ova komponenta pruža minimalno POSIX-kompatibilno UNIX-slično okruženje na Windowsima. Uključuje osnovne alate i biblioteke, kao što su bash, coreutils, sed, awk i još mnogo njih

2. MinGW-w64: Razvojno okruženje za stvaranje izvornih Windows aplikacija. MinGW-w64 pruža skup kompajlera, povezivača i zaglavlja koji programerima omogućuju izradu Windows aplikacija korištenjem GNU Compiler Collection (GCC). Podržava 32-bitne i 64-bitne Windows arhitekture

3. Pacman Package Manager: Paketni upravitelj koji koristi MSYS2 za instaliranje, ažuriranje i upravljanje softverskim paketima.

Sljedeći korak u kodu je definiranje klase "MainWindowController". Ova klasa sadrži metode koje obrađuju događaje potaknute interakcijom korisnika s GUI komponentama. Na primjer, kada se klikne gumb, poziva se odgovarajuća metoda u ovoj klasi za izvođenje željene radnje. Metode obrađuju različite radnje kao što su promjena veličine labirinta, generiranje novog labirinta, postavljanje početnih i ciljnih točaka, treniranje agenta za rješavanje labirinta, pokretanje treniranog agenta itd. Metode se pozivaju stiskom određene tipke na GUI-u.

Glavna logika programa može se opisati na sljedeći način:

1. Stvaranje instance "Gtk.Builder", koja se koristi za učitavanje GUI izgleda iz Glade datoteke "main_window.glade",

2. Stvaranje instance klase "MainWindowController", prosljeđujući instancu "Gtk.Builder" kao argument,

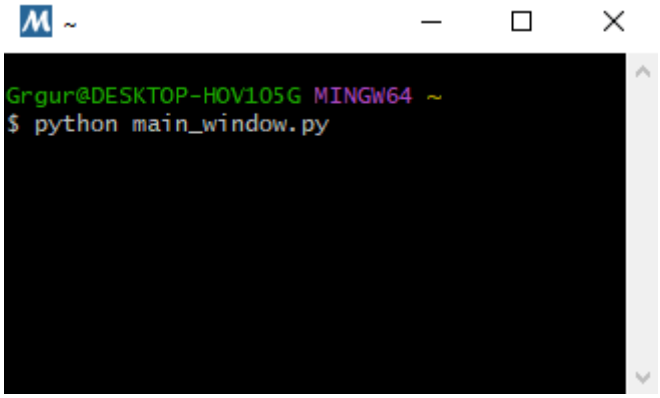
3. Povezivanje signala (događaja) koje emitiraju GUI komponente s odgovarajućim metodama u klasi "MainWindowController",

4. Dohvaćanje objekta glavnog prozora iz instance "Gtk.Builder", povezivanje događaja 'destroy' s funkcijom "Gtk.main_quit" i prikazivanje prozora i

5. Pozivanje "Gtk.main()" za pokretanje glavne petlje događaja, koja sluša događaje i upravlja GUI interakcijama.

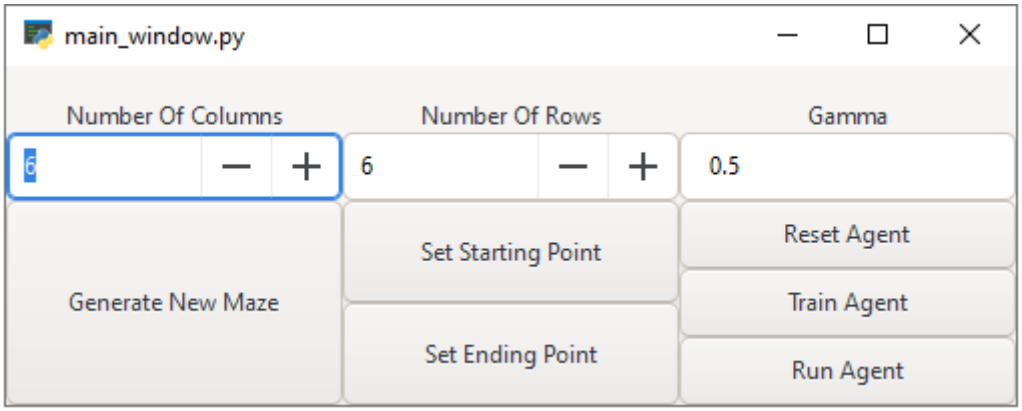
Kao što je već navedeno, korisničko sučelje izrađeno je u formatu .glade datoteke, pomoću Glade Interface Designer-a koji je zapravo GUI builder za GTK. Za izradu datoteke koristi se programski jezik Glade. Za ovaj rad preuzeto je napravljeno korisničko sučelje sa stranice [22].

Za korištenje korisničkog sučelja, potrebno je instalirati GTK3. Za to je potrebno u MSYS2 shell-u upisati nekoliko komandi, kao što su pacman -S mingw-w64-x86_64-gtk3 (bitno je odabrati instalaciju GTK3 a ne GTK4). Zatim je potrebno Python skripte kao i Glade datoteku korisničkog sučelja prebaciti u folder u kojemu GTK ima postavljen put. Otvara se datoteka Mingw64.exe, i upisuje komanda za otvaranje Python skripte "main_window.py". Proces je prikazan na Slici 4.3.



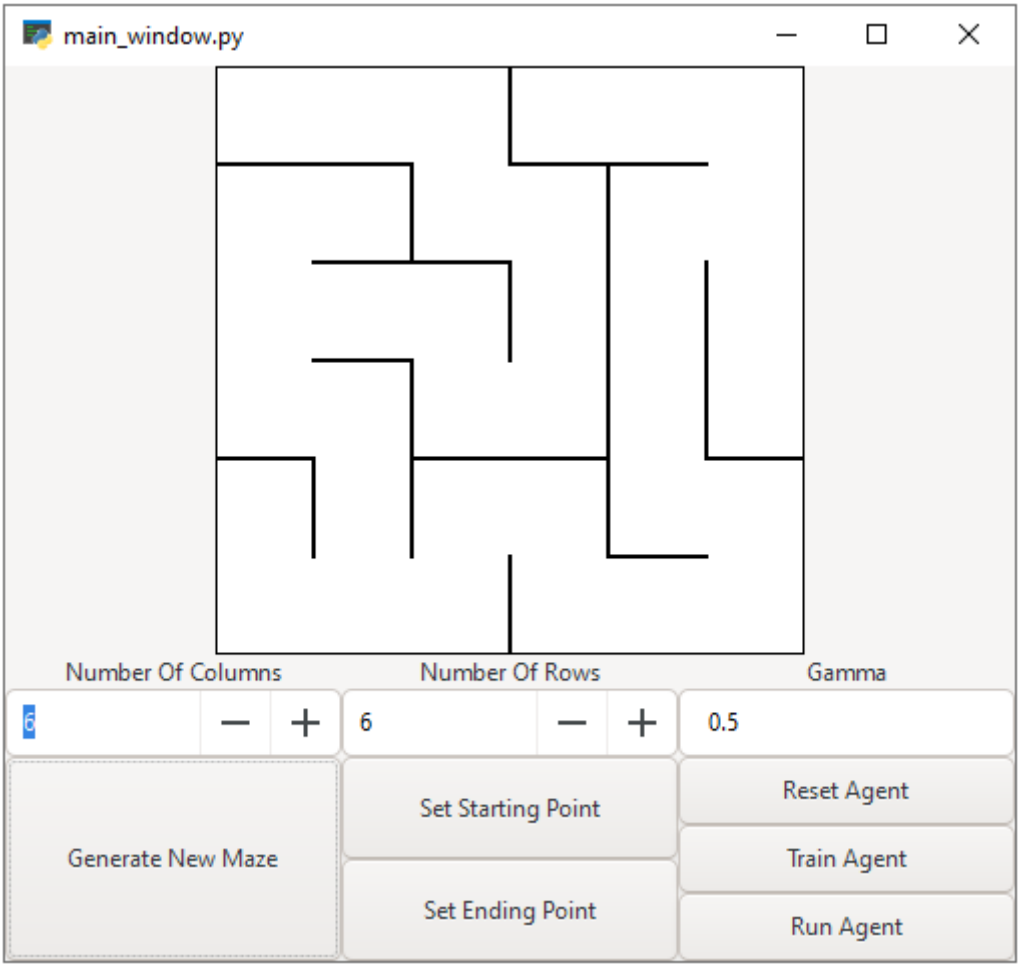
Slika 4.3. *Prikaz skočnog prozora mingw64.exe*

Upisom komande dobije se prozor korisničkog sučelja, koji je prikazan na Slici 4.4.



Slika 4.4. *Prikaz GUI prozora*

Kao što se vidi na slici, korisničko sučelje sastoji se od raznih tipki poput tipke za generiranje novog labirinta, treniranje agenta te polja u koja se upisuju vrijednosti poput broja stupaca i redova labirinta kao i vrijednosti Gamma, odnosno koeficijenta umanjenja nagrade. Odabirom vrijednosti prikazanih na slici, i pritiskom na gumb "Generate New Maze", generira se labirint sa odabranim parametrima. Primjer tog labirinta prikazan je na Slici 4.5.



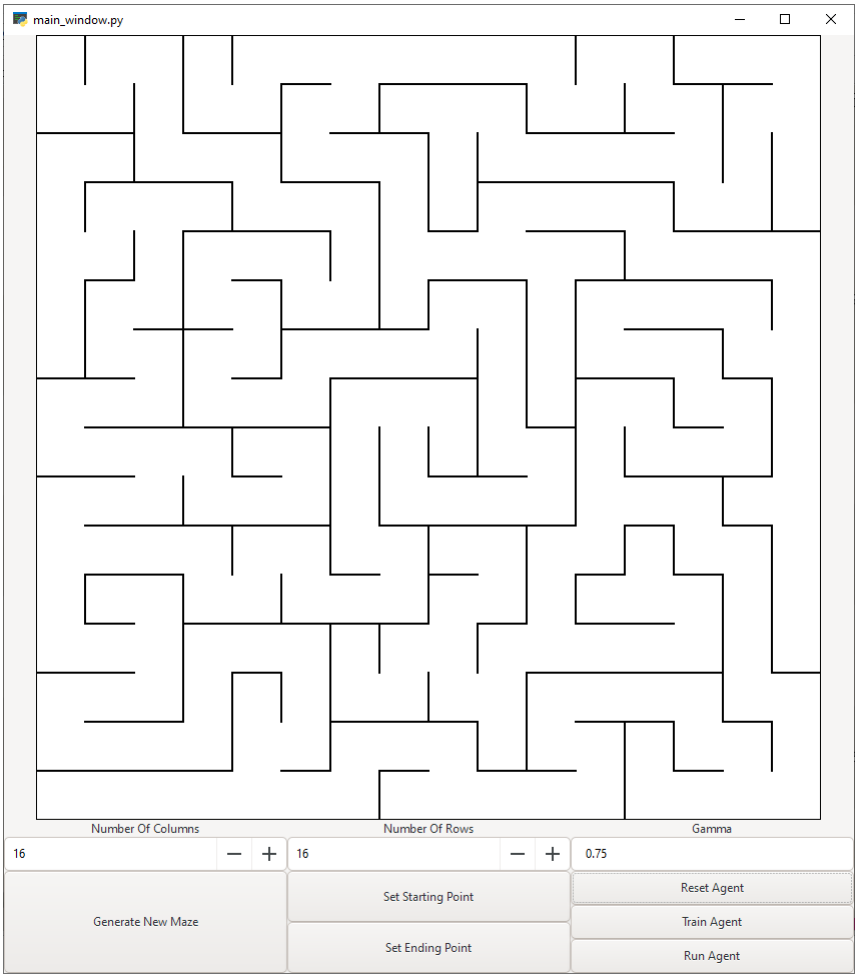
Slika 4.5. *Prikaz labirinta generiranog u GUI-u*

5. REZULTATI

U ovom poglavlju biti će razrađena realizacija problema postavljenog u zadatku diplomskog rada. Opisana je ideja za rješenje, kao i rezultati kod realiziranja te ideje. Ona se može opisati u koracima.

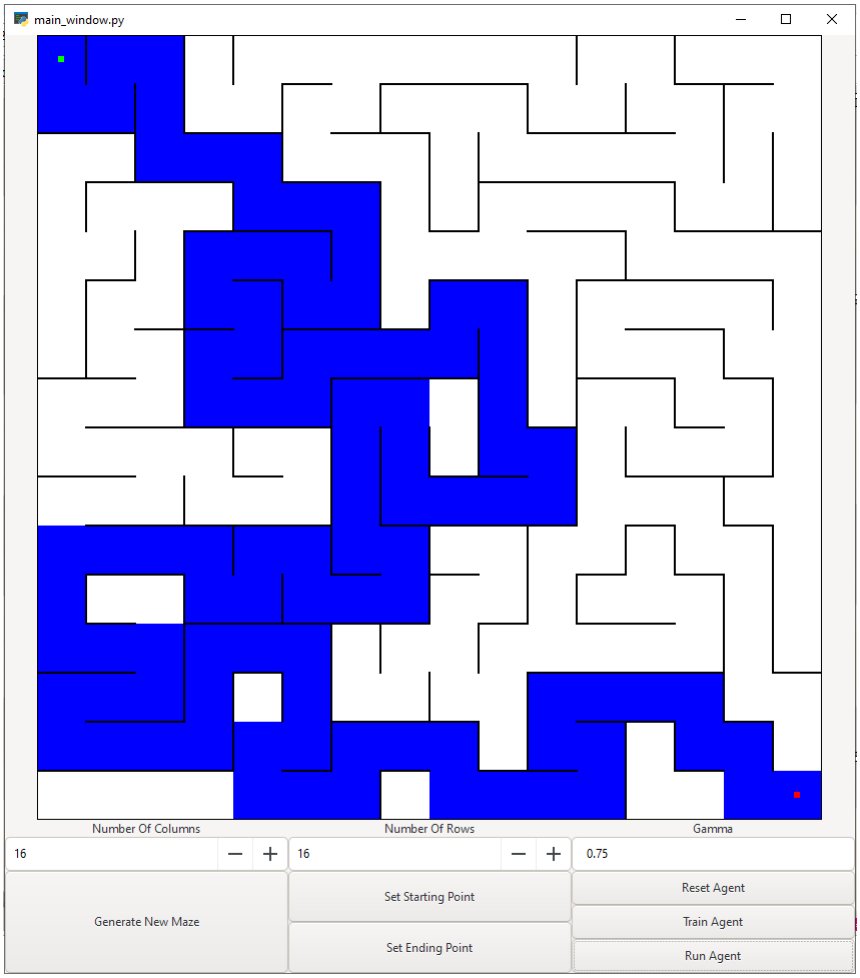
1. Generiranje labirinta,
2. Učenje Q-agenta,
3. Optimizacija puta (grafički prikaz optimalnog puta),
4. Modeliranje labirinta u CoppeliaSim-u i
5. Implementacija u CoppeliaSim-u.

Prvo je potrebno generirati labirint. Za rješavanje problema odabran je labirint dimenzija 16 redova i 16 stupaca. Generacija labirinta je objašnjena u prethodnom poglavlju, a njegov prikaz je na Slici 5.1.



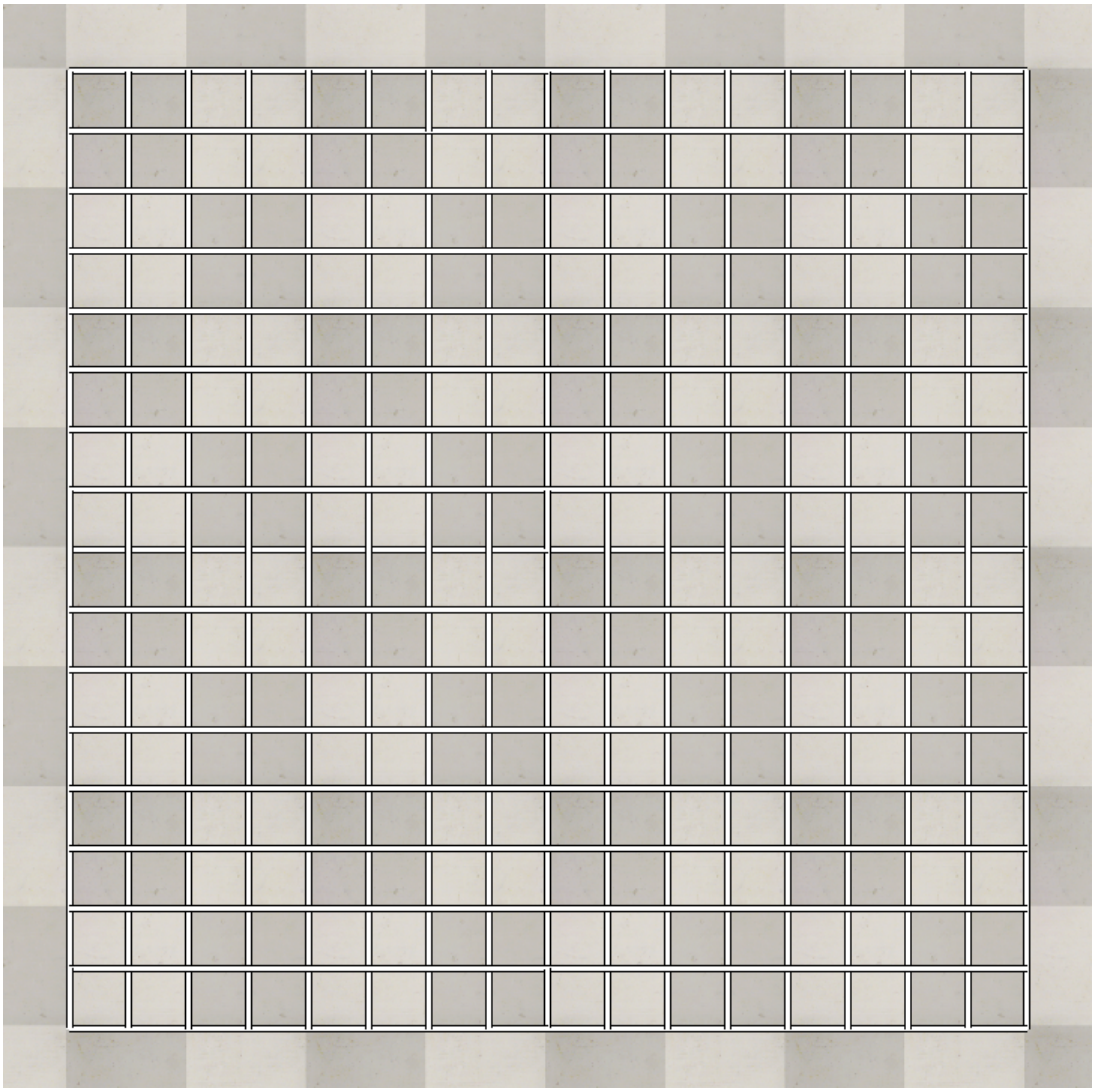
Slika 5.1. *Prikaz labirinta dimenzija 16 redova i 16 stupaca generiranog u GUI-u*

Zatim je potrebno istrenirati Q-agenta. Odabere se početna i ciljna točka, te se postavi koeficijent "gamma". Kao što je već objašnjeno, faktor umanjenja nagrade "gamma" određuje omjer trenutne i buduće nagrade. Kada se agent istrenira, klikom na tipku "run agent" generira se optimalan put labirinta koji je prikazan na Slici 5.2.



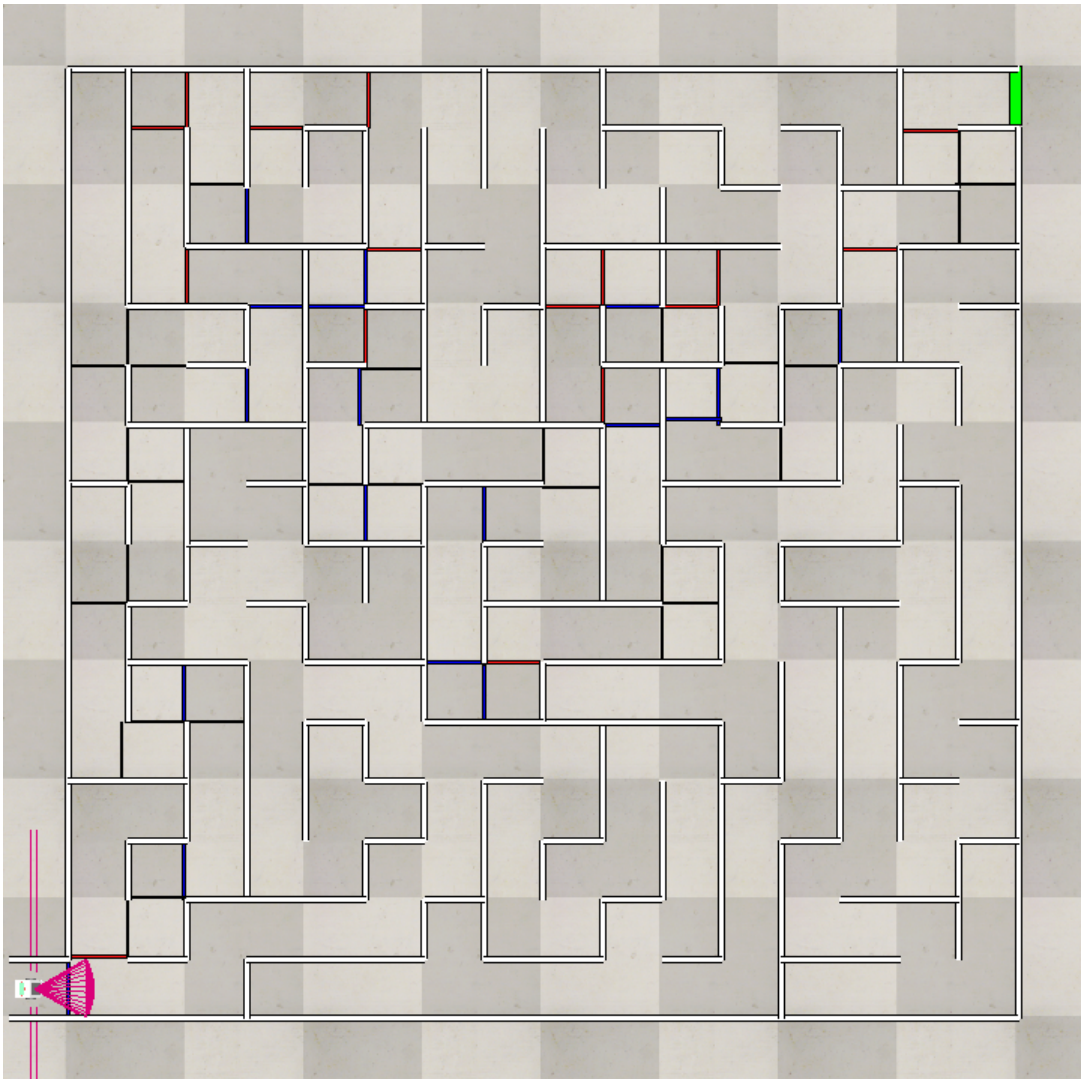
Slika 5.2. *Prikaz optimalnog puta labirinta dimenzija 16 redova i 16 stupaca generiranog u GUI-u*

Nakon dobivanja optimalnog puta, slijedi modeliranje labirinta u CoppeliaSim-u. Iz modela sa primjera sljedbenika desnog zida kreiran je "template" labirint dimenzija 16 redova i 16 stupaca koji je prikazan na Slici 5.3.

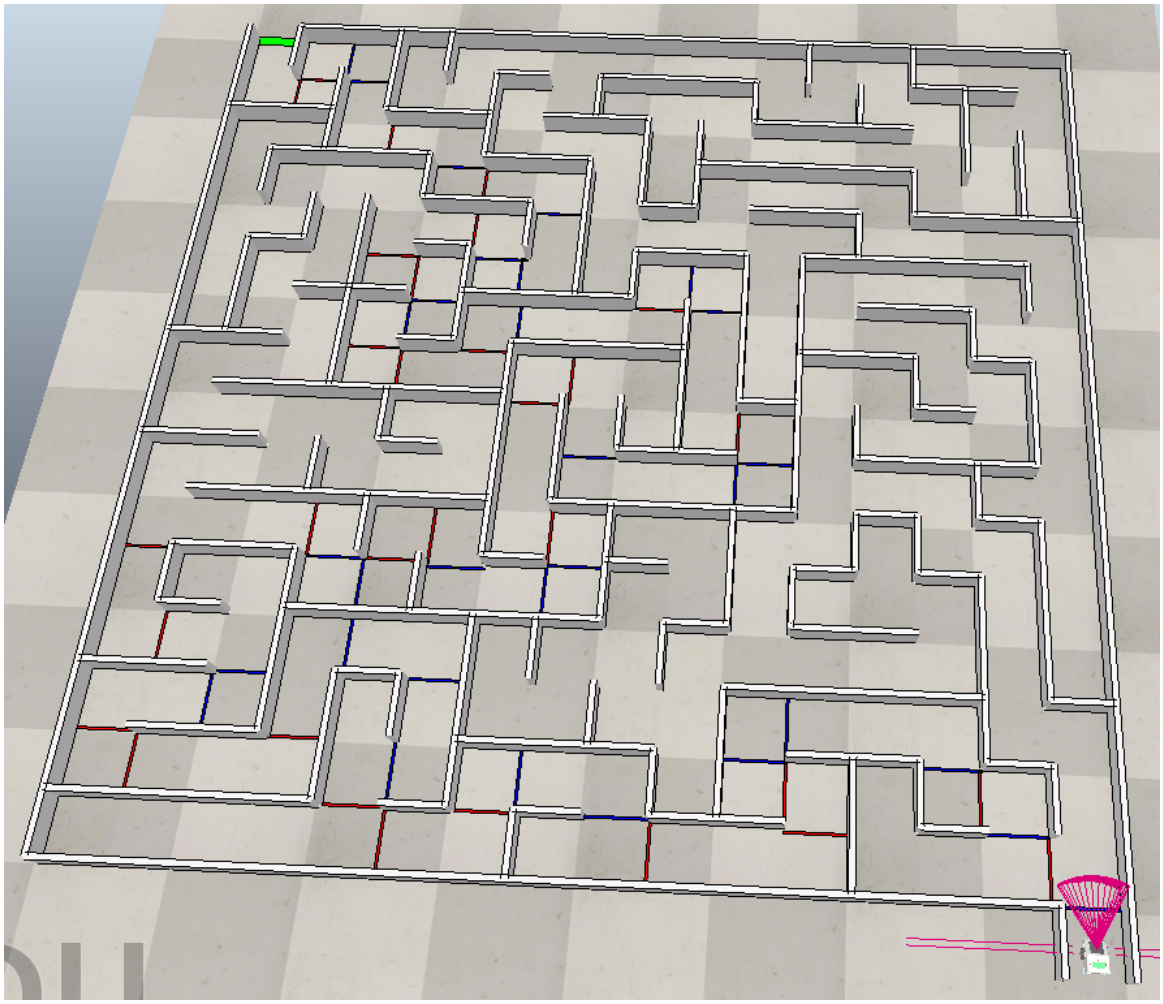


Slika 5.3. *Prikaz "template" labirinta dimenzija 16 redova i 16 stupaca u CoppeliaSim-u*

Uklanjajući određene zidove, labirint generiran u GUI-u "precrtan" je u CoppeliaSim-u.Također, dodani su i znakovi kako je prikazano na 2.18. Taj labirint prikazan je na Slici 5.4.

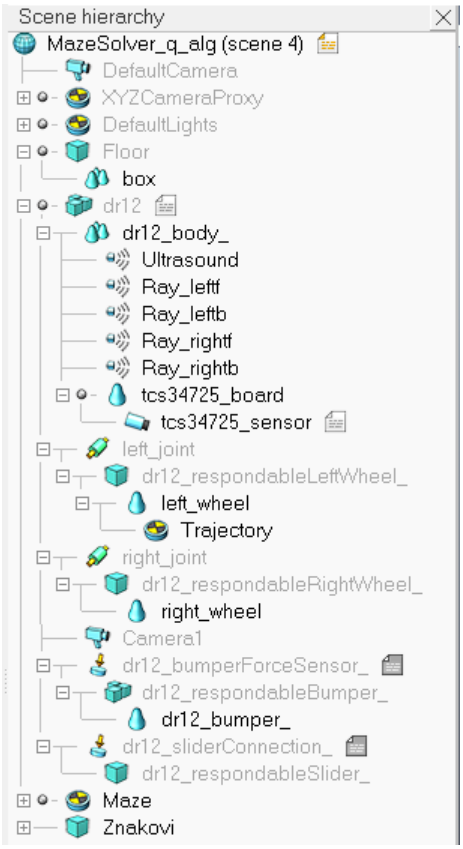


Slika 5.4. *Prikaz labirinta dimenzija 16 redova i 16 stupaca u CoppeliaSim-u*



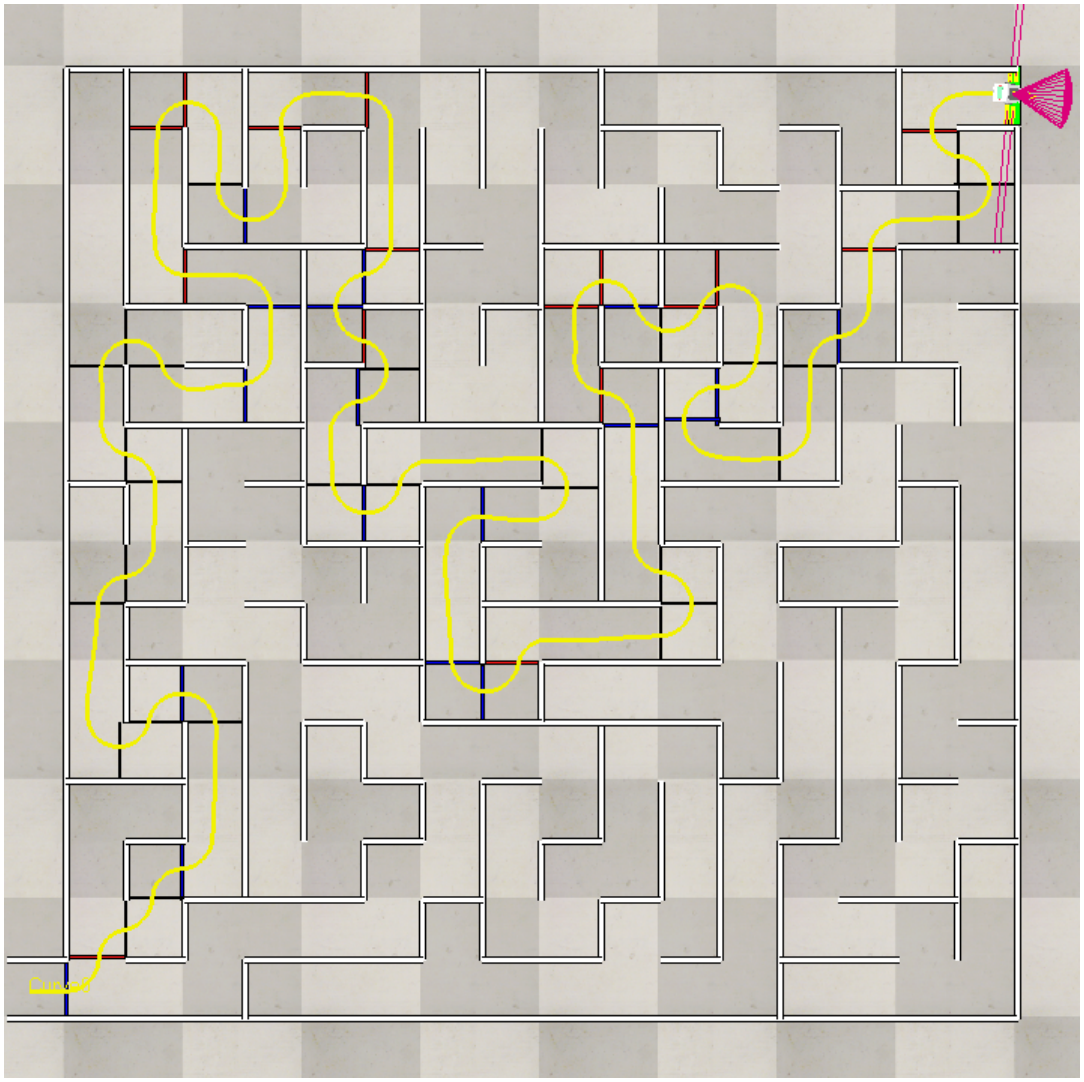
Slika 5.5. *Prikaz labirinta dimenzija 16 redova i 16 stupaca u CoppeliaSim-u iz drugog kuta*

Hijerarhijsko stablo prikazano je na Slici 5.6.

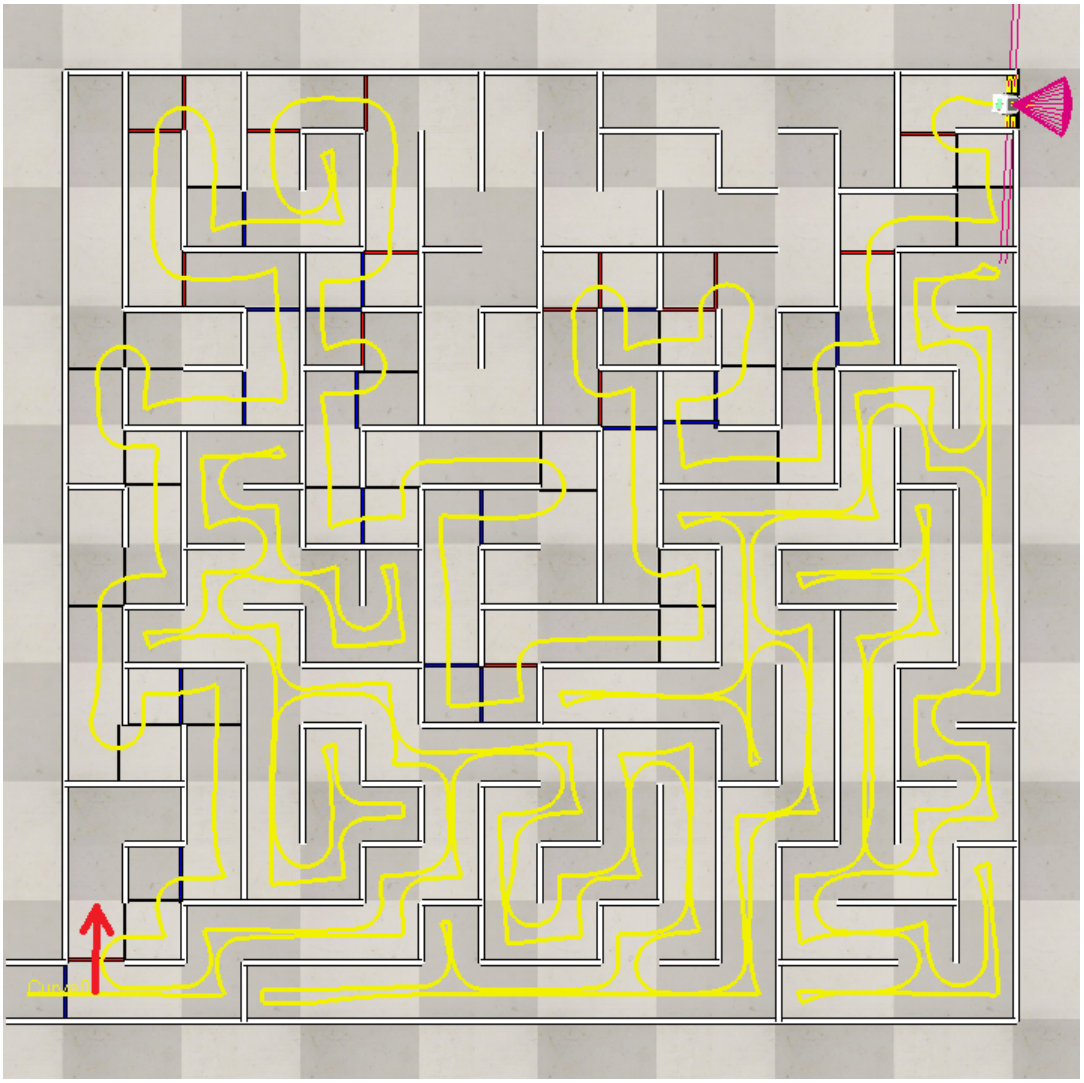


Slika 5.6. *Prikaz hijerarhijskog stabla za labirint dimenzija 16 redova i 16 stupaca u CoppeliaSim-u*

Na hijerarhijskom stablu je gotovo sve isto kao i kod primjera sljedbenika desnog zida. Jedina razlika je dodatak znakova. Rješenje za kretanje mobilnog robota kroz labirint razrađena je u poglavlju 2.7.. Rezultat simulacije nalazi se na Slici 5.7. U donjem lijevom kutu labirinta nalazi se početna, a u gornjem desnom kutu nalazi se ciljna točka labirinta. Žuta staza iscrtana na putu labirinta prikazuje put kojim se mobilni robot kretao. Radi preglednosti, prikaz je rotiran za 90 stupnjeva udesno.



Slika 5.7. *Rezultat simulacije primjenom umjetne inteligencije u labirintu dimenzija 16 redova i 16 stupaca u CoppeliaSim-u*



Slika 5.8. *Rezultat simulacije sljedbenika desnog zida u labirintu dimenzija 16 redova i 16 stupaca u CoppeliaSim-u*

Na Slici 5.8 prikazana je putanja mobilnog robota programiranog za sljedbenika desnog zida unutar okruženja prethodnog labirinta. Očito je da mobilnom robotu sa algoritmom sljedbenika desnog zida treba puno više vremena da riješi labirint od robota koji se kreće optimalnim putom. Crvenom strelicom naznačeno je mjesto labirinta na kojemu je mobilni robot "krivim" skretanjem rezultirao puno većim putom.

6. ZAKLJUČAK

U ovom diplomskom radu opisana je primjena umjetne inteligencije za optimizaciju putanje mobilnog robota. U prvom dijelu rada razradile su se osnove robotike, definicija robota, njihovi osnovni elementi te njihova podjela. To je bila podloga za razradu nekih osnovnih algoritama za savladavanje prepreka mobilnim robotima.

Objašnjene tehnike primjenjene su u programskom paketu CoppeliaSim, specifično algoritam sljedbenika desnog zida te bug 2 algoritam. Kod sljedbenika desnog zida za provjeru funkcionalnosti algoritma koristio se labirint dimenzija 8 redova i 8 stupaca, a za bug 2 algoritam koristio se set nasumičnih zidova. Uspješnom simulacijom oba algoritma, razvio se algoritam koji je namijenjen za optimizaciju primjenom umjetne inteligencije, i to korištenjem crvenih i plavih znakova koji su strateški postavljeni na podu na mjestima gdje mobilni robot mora skrenuti.

Zatim je razvijen model umjetne inteligencije za optimizaciju putanje mobilnog robota. Za optimizaciju se odabrao algoritam Q-učenja. Model je podijeljen u tri Python skripte. Prva skripta generira labirint željenih dimenzija te ga grafički prikazuje. Druga skripta trenira Q-agenta na generiranom labirintu, na kojemu na kraju iscrtava optimalan put. Zadnja skripta služi za pokretanje i manipulaciju korisničkim sučeljem.

Na kraju je na temelju dobivenog labirinta sa optimalnim putom izrađena simulacija mobilnog robota koji se kreće kroz taj labirint optimizirano. Prvo je razvijena scena u CoppeliaSim-u. Zatim je implementirana skripta mobilnog robota sa algoritmom za optimizaciju primjenom umjetne inteligencije. Mobilni robot uspješno rješava labirint krečući se optimalnim putom dobivenim primjenom umjetne inteligencije. Usporedbom dobivenih rezultata na kraju može se zaključiti da dobiveno rješenje zadovoljava zahtjeve ovog rada.

LITERATURA

[1] Tursynbek, Iliyas, i Almas Shintemirov: "Modeling and simulation of spherical parallel manipulators in CoppeliaSim (V-REP) robot simulator software." 2020 International Conference Nonlinearity, Information and Robotics (NIR). IEEE, 2020.

[2] Del Rosario, Jay Robert B., et al. "Modelling and characterization of a maze-solving mobile robot using wall follower algorithm." Applied Mechanics and Materials. Vol. 446. Trans Tech Publications Ltd, 2014.

[3] Garcia, Elena, et al.: "The evolution of robotics research." IEEE Robotics & Automation Magazine 14.1 (2007): 90-103.

[4] Tzafestas, Spyros G.: "Introduction to mobile robot control". Elsevier, 2013.: pp. 1-28

[5] Sadik, Adil MJ, et al.: "A comprehensive and comparative study of maze-solving techniques by implementing graph theory." 2010 International Conference on Artificial Intelligence and Computational Intelligence. Vol. 1. IEEE, 2010.

[6] Babula, Michał.: "Simulated maze solving algorithms through unknown mazes." Organizing and Program Committee 13, 2009.

[7] Manoj Sharma,“Algorithms for Micro-mouse”, Proc. 2009, International Conference on Future Computer and Communication, 2009.

[8] Chih-Yang Chen, Tzuu-Hseng S. Li, Ying-Chieh Yeh: "EP-based kinematic control and adaptive fuzzy sliding-mode dynamic control for wheeled mobile robots", Information Sciences, Volume 179, Issues 1–2, 2009, pp. 180-195

[9] Coppelia Robotics, s interneta: https://www.coppeliarobotics.com

[10] Ribeiro, Maria Isabel.: "Obstacle avoidance." Instituto de Sistemas e Robótica, Instituto Superio Técnico 1, 2005.

[11] Choset, Howie, et al. P"rinciples of robot motion: theory, algorithms, and implementations.", MIT press, 2005.

[12] Nilsson, Nils J. "Principles of artificial intelligence." Springer Science & Business Media, 1982.

[13] Sternberg, Robert J.: "Intelligence." The Palgrave Encyclopedia of the Possible 2023: pp. 793-800.

[14] Russell, Stuart J., and Peter Norvig.: "Artificial intelligence: a modern approach." Pearson, 2016.

[15] Sutton, Richard S., and Andrew G. Barto.: "Reinforcement learning: An introduction." MIT press, 2018.

[16] Watkins, Christopher JCH, and Peter Dayan. "Q-learning." Machine learning 8 1992, pp. 279-292.

[17] Ghahramani, Zoubin.: "Unsupervised learning." Summer school on machine learning. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. 72-112.

[18] Lorberfeld, Audrey: "Machine Learning Algorithms In Layman's Terms, Part 1", S interneta: https://towardsdatascience.com/machine-learning-algorithms-in-laymans-terms-part-1-d0368d769a7b

[19] Sprynn, Mitchell: "Solving A Maze With Q Learning", S interneta: https://mitchellspryn.com/2017/10/28/Solving-A-Maze-With-Q-Learning.html

[20] Kozlova, Aliona, Joseph Alexander Brown, and Elizabeth Reading.: "Examination of representational expression in maze generation algorithms." 2015 IEEE Conference on Computational Intelligence and Games (CIG). IEEE, 2015.

[21] GTK, S interneta: https://www.gtk.org/docs/installations/windows/#using-gtk-from-msys2-packages

[22] S interneta: https://github.com/mitchellspryn/QLearningMazeSolver, 2017.

[23] Niku, Saeed B.: "Introduction to robotics: analysis, control, applications.", John Wiley & Sons, 2020.

[24] Ben-Ari, Mordechai, Francesco Mondada: "Elements of robotics". Springer Nature, 2017.

[25] Rao, Ravi: "Robot Joints: An In-Depth Guide to Anatomy, Physics and Challenges in Design", 2023.

[26] Car Z.: "Uvod u robotiku", materijali s predavanja, 2020.

[27] Klancar, Gregor, et al.: "Wheeled mobile robotics: from fundamentals towards autonomous systems.", Butterworth-Heinemann, 2017.

[28] S interneta:
http://www.cubictek.com/contents/product/product07_intelligent
_mobile_robot.asp?a_site=1_1&m=1, 2008.

7. SAŽETAK I KLJUČNE RIJEČI

U diplomskom radu je opisan postupak optimizacije putanje mobilnog robota primjenom umjetne inteligencije. Objašnjeni su osnovni pojmovi robotike poput zglobova, krajnjeg djelovatelja i senzora pomoću kojeg robot obavlja zadatke. Razrađena je konstrukcija i načini manipulacije mobilnim robotom u radnom okruženju unutar programskog paketa CoppeliaSim. Zatim su opisane neke metode umjetne inteligencije kojima je moguće optimizirati put mobilnog robota kroz labirint, od kojih je odabrano Q-učenje. U programskom jeziku Python razrađeni su modeli generiranja labirinta pomoću Depth-first Search metode, kao i model za treniranje Q-agenta. Pomoću korisničkog sučelja dobiveni su rezultati optimizacije puta mobilnog robota u grafičkom obliku. Koristeći te rezultate, rekreirana je scena u CoppeliaSim okruženju izradom modela labirinta i puštanjem mobilnog robota kroz taj labirint. Mobilni robot je stigao do cilja optimalnim putem dobivenim modelom Q-učenja, i koristeći logiku koja je objašnjena u radu. Rezultat je uspoređen sa jednim primitivnim algoritmom te je očita da je duljina puta sa primjenom umjetne inteligencije izrazito kraća, odnosno optimalna.

Ključne riječi - **coppeliaSim, labirint, mobilni robot, optimizacija, Python, Q-učenje, senzori, umjetna inteligencija**

8. SUMMARY AND KEY WORDS

The thesis describes the process of optimizing the trajectory of a mobile robot using artificial intelligence. The basic concepts of robotics are explained, such as joints, end-effectors and sensors with which the robot performs tasks. The construction and ways of manipulating the mobile robot in the environment within the CoppeliaSim software package have been elaborated. Then, some artificial intelligence methods are described that can optimize the path of a mobile robot through a maze, from which Q-learning is chosen. Maze generation models using the Depth-first Search method, as well as a Q-agent training model, were developed in the Python programming language. Using the graphical user interface, the results of the optimization of the mobile robot's path were obtained in graphic form. Using these results, a scene was recreated in the CoppeliaSim environment by creating a maze model and running a mobile robot through that maze. The mobile robot reached the goal through the optimal path retrieved from the Q-learning model, and using the logic explained in the paper. The result was compared with a primitive algorithm, and it is obvious that the length of the path with the application of artificial intelligence is significantly shorter, that is, optimal.

Key words - **coppeliaSim, maze, mobile robot, optimization, Python, Q-learning, sensors, artificial intelligence**

9. PRILOZI

9.1. Prilog A1 - Lua skripta za simulaciju sljedbenika desnog zida

```
function sysCall_init()
-- do some initialization here
left_wheel=sim.getObject('./left_joint')
right_wheel=sim.getObject('./right_joint')
ir_sensor=sim.getObject('./tcrt5000_sensor')

us=sim.getObject('./Ultrasound')
ir_FL=sim.getObject('./Ray_leftf')
ir_FR=sim.getObject('./Ray_rightf')
ir_RL=sim.getObject('./Ray_leftb')
ir_RR=sim.getObject('./Ray_rightb')

resolution=sim.getVisionSensorResolution(ir_sensor)

wheel_radius=0.04 --wheel radius
vref=0.25 -- velocity when following the wall
speed_reduction=0.3 -- speed reduction factor when turning
cell_size=0.4 -- cell size of the maze
wallThick=0.05 -- wall thickness of the maze
b=0.0765 -- wheel base (wheel separation distance)
sepUS=0.04 -- distance of US with respect to the robot wheels

min_distUS=cell_size/2-sepUS-wallThick/2
-- Distance to stop when there's a wall in front of the robot
kWall=3 -- Wall following gain
e=0.24 -- Distance of an off-center point to
--perform the kinematic control for wall following
a=0.04 -- Separation distance between front and rear LIDAR sensors
sepLIDAR=0.061 -- Lateral separation distance
--of LIDAR sensors w.r.t. the robot
wallDist=cell_size/2-sepLIDAR-wallThick/2 -- The expected
--distance to detect a wall with LIDAR sensors
black=0.1 -- Line colour threshold for detecting black
objDetectedTol=2e-2  --Tolerance to
--detect an object in fron with the US
wallNotDetectedTol=2e-1 -- Tolerance to determine that there's no wall
wallDetectedTurnTol=5e-2 -- Tolerance to
--determine that there's a wall when rotating
wallDetectedCurveTol=5e-1 -- Tolerance to
--determine that tere's a wall when making a curve
```

```
lastTime=sim.getSimulationTime()  -- Variable use to
--measured the time elapsed since the 'last time'
States={"Follow wall","Turn slow","Make curve","Stop"}
-- Names for each defined state of the algorithm
state=1  -- Initial state
direction=1 -- Follow right 1, follow left -1

lineColour=0

max_distUS=0.5 -- Maximum distance returned by the US sensor
dUS=max_distUS  -- Initial US distance
max_distLIDAR=1.2 -- Maximum distance returned by the LIDAR sensors
dFR=max_distLIDAR -- Initial LIDAR distance
dRR=max_distLIDAR -- Initial LIDAR distance
dFL=max_distLIDAR -- Initial LIDAR distance
dRL=max_distLIDAR -- Initial LIDAR distance

wCurveL,wCurveR,minCurveTime=
makeCurve(vref,direction,cell_size,wheel_radius,b)
wTurnL,wTurnR,minPureRotationTime=
turnSlow(vref,direction,speed_reduction,wheel_radius,b)
end

function sysCall_actuation()
local wL,wR
if (state==1) then  --Follow wall
wL,wR=followWall(vref,direction,dFL,dRL,dFR,dRR,wallDist)
elseif (state==2) then  --Turn slow
wL=wTurnL --Precomputed values to make a turn
wR=wTurnR --Precomputed values to make a turn
elseif (state==3) then -- Make curve
wL=wCurveL  --Precomputed values to make a curve
wR=wCurveR  --Precomputed values to make a curve
else
wL=0
wR=0
end
sim.setJointTargetVelocity(left_wheel,wL)
sim.setJointTargetVelocity(right_wheel,wR)
end

function sysCall_sensing()
dUS=getDistance(us,max_distUS)
dFR=getDistance(ir_FR,max_distLIDAR)
dRR=getDistance(ir_RR,max_distLIDAR)
dFL=getDistance(ir_FL,max_distLIDAR)
dRL=getDistance(ir_RL,max_distLIDAR)
print(States[state])
```

```
if (state==1) then
lastTime=sim.getSimulationTime()
if dUS<min_distUS then
print("Object detected")
state=2
end
if not wallDetected(direction,dFL,dRL,dFR,dRR,
wallDist,wallNotDetectedTol) then
print("Wall not detected")
state=3 -- curve
end
elseif (state==2) then
local wall=wallDetected(direction,dFL,dRL,
,dFR,dRR,wallDist,wallDetectedTurnTol)
local timeElapsed=((sim.getSimulationTime()-
-lastTime)>minPureRotationTime)
if wall and timeElapsed then
print("Wall detected and time elapsed")
state=1 -- turn
end
elseif (state==3) then
local wall=wallDetected(direction,dFL,dRL,
,dFR,dRR,wallDist,wallDetectedCurveTol)
local timeElapsed=((sim.getSimulationTime()-lastTime)>minCurveTime)
if wall and timeElapsed then
print("Wall detected and time elapsed")
state=1 -- turn
end
end
if (readLineColour()<=black) then
print("Line detected")
state=4 --stop
end
end

function sysCall_cleanup()
-- do some clean-up here
end

function wallDetected(direction,dFrontL,
,dRearL,dFrontR,dRearR,dWallSide,tolerance)
local dFront,dRear,front,rear
if (direction==1) then
dFront=dFrontR
dRear=dRearR
else
dFront=dFrontL
dRear=dRearL
```

```
end
front=math.abs(dFront-dWallSide)<tolerance
rear=math.abs(dRear-dWallSide)<tolerance
return front and rear
end

function followWall(v,direction,dFL,dRL,dFR,dRR,dWallSide)
local phi,d,alpha,wL,wR
if (direction==1) then -- follow right wall
phi=math.atan((dFR-dRR)/a)
d=(0.5*(dFR+dRR)-dWallSide)
else -- follow left wall
phi=math.atan((dRL-dFL)/a)
d=(dWallSide-0.5*(dFL+dRL))
end
gamma=kWall*d
alpha=phi+gamma
wL=(v/wheel_radius)*(math.cos(alpha)+(b/e)*math.sin(alpha))
wR=(v/wheel_radius)*(math.cos(alpha)-(b/e)*math.sin(alpha))
return wL,wR
end

function getVelocity(d,vref,dmin,dmax)
if (d<dmin) then
return 0
else
return vref*d/dmax
end
end

function getDistance(sensor,max_dist)
local detected, distance
detected,distance=sim.readProximitySensor(sensor)
if (detected<1) then
distance=max_dist
end
return distance
end

function readLineColour()
local grey
value=sim.getVisionSensorImage(ir_sensor+sim.handleflag_greyscale)
grey=0
pixel=1
for i = 1, resolution[1] do
for j= 1, resolution[2] do
grey=grey+value[pixel]
pixel=pixel+1
```

```
end
end
grey=grey/(resolution[1]*resolution[2])
return grey
end

function turnSlow(vref,direction,speed_reduction,wheel_radius,b)
local wL,wR,t
wL=-direction*speed_reduction*vref/wheel_radius
wR=direction*speed_reduction*vref/wheel_radius
t=b*math.pi/2/(speed_reduction*vref)
return wL,wR,t
end

function makeCurve(vref,direction,cell_size,wheel_radius,b)
local wref,wL,wR,t
wref=(vref)/(cell_size/1.75)
wL=(vref+b*direction*wref)/wheel_radius
wR=(vref-b*direction*wref)/wheel_radius
t=(math.pi/2)/wref
return wL,wR,t
end
```

9.2. Prilog A2 - Lua skripta za simulaciju bug 2 algoritma

```
function sysCall_init()
-- do some initialization here
left_wheel=sim.getObject('./left_joint')
right_wheel=sim.getObject('./right_joint')
ir_sensor=sim.getObject('./tcrt5000_sensor')

us=sim.getObject('./Ultrasound')
ir_FL=sim.getObject('./Ray_leftf')
ir_FR=sim.getObject('./Ray_rightf')
ir_RL=sim.getObject('./Ray_leftb')
ir_RR=sim.getObject('./Ray_rightb')
init=sim.getObjectHandle('init')
goal=sim.getObjectHandle('goal')
robot=sim.getObjectHandle('robotPose')

wheel_radius=0.04 --wheel radius
vref=0.1 -- velocity when following the wall o moving forward
vTurn=0.1*vref -- velocity when turning
objectDist=0.25 -- distance to stop the robot when detecting an object
b=0.0765 -- wheel base (wheel separation distance)
```

```
sepUS=0.04 -- distance of US with respect to the robot wheels

min_distUS=objectDist-sepUS -- Distance to stop
--when there's a wall in front of the robot
kWall=3 -- Wall following gain
e=0.24 -- Distance of an off-center point to
--perform the kinematic control for wall following
a=0.03 -- Separation distance between front and rear LIDAR sensors
sepLIDAR=0.061 -- Lateral separation distance
--of LIDAR sensors w.r.t. the robot
wallDist=objectDist-sepLIDAR -- The expected
--distance to detect a wall with LIDAR sensors

goalDetectedTol=1e-2 --Tolerance to point to the goal
wallDetectedTol=1e-1 -- Tolerance to determine
--that there's a wall when rotating
goalReachedTol=1e-1
lastTime=sim.getSimulationTime()  -- Variable use to
--measured the time elapsed since the 'last time'
States={"Point to goal","Move forward","Turn",
        ,"Follow wall line not crossed","Follow wall","Stop"}
-- Names for each defined state of the algorithm
state=1  -- Initial state
initRandom()
direction=1 -- Follow right 1, follow left -1

max_distUS=0.5 -- Maximum distance returned by the US sensor
dUS=max_distUS  -- Initial US distance
max_distLIDAR=1.2 -- Maximum distance returned by the LIDAR sensors
dFR=max_distLIDAR -- Initial LIDAR distance
dRR=max_distLIDAR -- Initial LIDAR distance
dFL=max_distLIDAR -- Initial LIDAR distance
dRL=max_distLIDAR -- Initial LIDAR distance

wTurnL,wTurnR,turnTime=precomputeTurn(vTurn,direction,wheel_radius,b)
initPos=sim.getObjectPosition(init,-1)
goalPos=sim.getObjectPosition(goal,-1)
turnLeft=false
end

function sysCall_actuation()
-- put your actuation code here
local wL,wR,v
if (state==1) then
wTurnL,wTurnR,turnTime=precomputeTurn(vTurn,direction,wheel_radius,b)
wL=wTurnL
wR=wTurnR
elseif (state==2) then
```

```
wL=vref/wheel_radius
wR=vref/wheel_radius
elseif (state==3) then
wTurnL,wTurnR,turnTime=precomputeTurn(vTurn,direction,wheel_radius,b)
wL=wTurnL
wR=wTurnR
elseif (state==4) then
wL,wR=followWall(vref,direction,dFL,dRL,dFR,dRR,wallDist)
elseif (state==5) then
wL,wR=followWall(vref,direction,dFL,dRL,dFR,dRR,wallDist)
elseif (state==6) then
wL=0
wR=0
end
sim.setJointTargetVelocity(left_wheel,wL)
sim.setJointTargetVelocity(right_wheel,wR)
end

function sysCall_sensing()
-- put your sensing code here
print(States[state])
dUS=getDistance(us,max_distUS)
dFR=getDistance(ir_FR,max_distLIDAR)
dRR=getDistance(ir_RR,max_distLIDAR)
dFL=getDistance(ir_FL,max_distLIDAR)
dRL=getDistance(ir_RL,max_distLIDAR)
pose=getRobotPose()

--TODO: Read proximity sensors
--TODO: Update robot position

--Evaluate transitions conditions
--State machine
if (state==1) then
local pointingToGoal
pointingToGoal,direction=PointingToGoal
(pose,initPos,goalPos,goalDetectedTol)
if pointingToGoal then
state=2
end
elseif (state==2) then
local pointingToGoal,goalReached
goalReached=GoalReached(pose,goalPos,goalReachedTol)
pointingToGoal,direction=PointingToGoal
(pose,initPos,goalPos,goalReachedTol)
if goalReached then
state=6
elseif not pointingToGoal then
```

```
state=1
elseif dUS<min_distUS then
lastTime=sim.getSimulationTime()
direction=selectRandomDirection()
state=3
end
elseif (state==3) then
local wall=wallDetected
(direction,dFL,dRL,dFR,dRR,wallDist,wallDetectedTol)
local timeElapsed=((sim.getSimulationTime()-lastTime)>turnTime)
if wall and timeElapsed then
state=4
end
elseif (state==4) then
local lineSide
lineSide=LineSide(pose,initPos,goalPos)
if lineSide==direction then
state=5
end
elseif (state==5) then
local lineSide
lineSide=LineSide(pose,initPos,goalPos)
if lineSide~=direction then
state=1
end
end
end

function sysCall_cleanup()
-- do some clean-up here
end

function getRobotPose()
-- Returns the robot pose (list with x,y and theta)
local pose
position=sim.getObjectPosition(robot,-1)
orientation=sim.getObjectOrientation(robot,-1)
pose={position[1],position[2],orientation[3]}
return pose
end

function PointingToGoal(pose,initPos,goalPos,tolerance)
-- pose: list with x,y,theta values of the robot
-- initPose: list with x,y of the initial position
-- goalPose: list with x,y of the goal position
-- tolerance: admissible angle (in radians)
-- Returns true or false and the turning direction (1 or -1)
local angle_diff,pointing,direction
```

```
angle_diff=math.atan2(goalPos[2]-initPos[2],
,goalPos[1]-initPos[1])-pose[3]
if (math.abs(angle_diff)<tolerance) then
pointing=true
else
pointing=false
end
if (angle_diff>0) then
direction=1
else
direction=-1
end
return pointing,direction
end

function precomputeTurn(vturn,direction,wheel_radius,b)
-- vturn: turning speed
-- direction: 1-> Follow right, 1-> Follow left
-- wheel_radius,b: constructive parameters
-- Returns wheels velocities and turning time
local wL,wR,t
wL=-direction*vturn/wheel_radius
wR=direction*vturn/wheel_radius
t=b*(math.pi/2)/vturn
return wL,wR,t
end

function wallDetected(direction,dFL,dRL,dFR,dRR,dWallSide,tolerance)
-- direction: 1 wall on the right, -1 wall on the left
-- dFL,dRL: distance of LIDAR sensors on the left
-- dFR,dRR: distance of LIDAR sensors on the right
-- dWallSide: expected separation distance of the wall
-- tolerance: admissible distance to consider that there's a wall
-- Returns true or false
local dF,dR,front,rear
if (direction==1) then
dF=dFR
dR=dRR
else
dF=dFL
dR=dRL
end
front=math.abs(dF-dWallSide)<tolerance
rear=math.abs(dR-dWallSide)<tolerance
return front and rear
end

function followWall(v,direction,dFL,dRL,dFR,dRR,dWallSide)
```

```
-- dFL,dRL: distance of LIDAR sensors on the left
-- dFR,dRR: distance of LIDAR sensors on the right
-- dWallSide: expected separation distance of the wall
-- wheel_radius,a,b: Constructuve parameters
-- v,kWall,e: Parameters for wall following algorithm
-- Returns wheels velocities
local phi,d,alpha,wL,wR
if (direction==1) then  -- follow right wall
phi=math.atan((dFR-dRR)/a)
d=(0.5*(dFR+dRR)-dWallSide)
else -- follow left wall
phi=math.atan((dRL-dFL)/a)
d=(dWallSide-0.5*(dFL+dRL))
end
gamma=kWall*d
alpha=phi+gamma
wL=(v/wheel_radius)*(math.cos(alpha)+(b/e)*math.sin(alpha))
wR=(v/wheel_radius)*(math.cos(alpha)-(b/e)*math.sin(alpha))
return wL,wR
end

function LineSide(pose,initPos,goalPos)
-- pose: list with x,y,theta values of the robot
-- initPose: list with x,y of the initial position
-- goalPose: list with x,y of the goal position
-- Returns 1 on the left side and -1 on the right side
local d,dGoalIni,xDiff1,yDiff1,xDiff2,yDiff2
xDiff1=initPos[1]-goalPos[1]
yDiff1=goalPos[2]-initPos[2]
xDiff2=initPos[1]-pose[1]
yDiff2=initPos[2]-pose[2]
dGoalIni=math.sqrt(xDiff1^2+yDiff1^2)
d=(xDiff2*yDiff1+yDiff2*xDiff1)/dGoalIni
print(d)
if d>0 then
return 1
else
return -1
end
end

function GoalReached(pose,goalPos,tolerance)
-- pose: list with x,y,theta values of the robot
-- goalPose: list with x,y of the goal position
-- tolerance: admissible distance to
--consider that the goal has been reached
-- Returns true or false
local dGoal
```

```
dGoal=math.sqrt((goalPos[1]-pose[1])^2+(goalPos[2]-pose[2])^2)
return dGoal<tolerance
end

function getDistance(sensor,max_dist)
local detected, distance
detected,distance=sim.readProximitySensor(sensor)
if (detected<1) then
distance=max_dist
end
return distance
end

function initRandom()
math.randomseed(os.time())
for i=1,5,1 do
math.random()
end
end

function selectRandomDirection()
local n,direction
n=math.random(0,1)
if (n<0.5) then
direction=1
else
direction=-1
end
return direction
end
```

9.3. Prilog A3 - Lua skripta za simulaciju algoritma kretanje mobilnog robota optimalnom putanjom dobivenom implementacijom umjetne inteligencije

```
function sysCall_init()
-- do some initialization here
left_wheel=sim.getObject('./left_joint')
right_wheel=sim.getObject('./right_joint')
ir_sensor=sim.getObject('./tcrt5000_sensor')

us=sim.getObject('./Ultrasound')
ir_FL=sim.getObject('./Ray_leftf')
ir_FR=sim.getObject('./Ray_rightf')
ir_RL=sim.getObject('./Ray_leftb')
ir_RR=sim.getObject('./Ray_rightb')
```

```
resolution=sim.getVisionSensorResolution(ir_sensor)
GREEN_THRESHOLD = 0.99
RED_THRESHOLD = 0.99
BLUE_THRESHOLD = 0.99

wheel_radius=0.045 --wheel radius
vref=0.15 -- velocity when following the wall
speed_reduction=0.3 -- speed reduction factor when turning
cell_size=0.4 -- cell size of the maze
wallThick=0.04 -- wall thickness of the maze
b=0.0765 -- wheel base (wheel separation distance)
sepUS=0.04 -- distance of US with respect to the robot wheels

min_distUS=0.2 -- Distance to stop when
--there's a wall in front of the robot
kWall=3 -- Wall following gain
e=0.24 -- Distance of an off-center point to
--perform the kinematic control for wall following
a=0.04 -- Separation distance between
--front and rear LIDAR sensors
sepLIDAR=0.061 -- Lateral separation distance
--of LIDAR sensors w.r.t. the robot
wallDist=cell_size/2-sepLIDAR-wallThick/2 -- The expected
--distance to detect a wall with LIDAR sensors
black=0.1 -- Line colour threshold for detecting black
objDetectedTol=2e-2  --Tolerance to detect an
--object in fron with the US
wallNotDetectedTol=2e-1 -- Tolerance to
--determine that there's no wall
wallDetectedTurnTol=5e-2 -- Tolerance to
--determine that there's a wall when rotating
wallDetectedCurveTol=5e-1 -- Tolerance to determine
--that tere's a wall when making a curve
lastTime=sim.getSimulationTime()  -- Variable use to
--measured the time elapsed since the 'last time'
States={"Follow wall","Turn slow","Make curve", "Stop"}
-- Names for each defined state of the algorithm
state=1  -- Initial state
direction=1 -- Follow right 1, follow left -1

lineColour=0

max_distUS=0.5 -- Maximum distance returned by the US sensor
dUS=max_distUS  -- Initial US distance
max_distLIDAR=1.2 -- Maximum distance returned by the LIDAR sensors
dFR=max_distLIDAR -- Initial LIDAR distance
dRR=max_distLIDAR -- Initial LIDAR distance
```

```
dFL=max_distLIDAR -- Initial LIDAR distance
dRL=max_distLIDAR -- Initial LIDAR distance

wCurveL,wCurveR,minCurveTime=makeCurve
(vref,direction,cell_size,wheel_radius,b)
wTurnL,wTurnR,minPureRotationTime=turnSlow
(vref,direction,speed_reduction,wheel_radius,b)
end

function sysCall_actuation()
local wL = 0
local wR = 0
local redDetected = false
local blueDetected = false
local crvena = readLineColour(1)
local zelena = readLineColour(2)
local plava = readLineColour(3)

-- Check if red or blue line is detected
if crvena >= RED_THRESHOLD then
redDetected = true
direction = 1
end
if plava >= BLUE_THRESHOLD then
blueDetected = true
direction = -1
end
if zelena >= GREEN_THRESHOLD then
greenDetected = true
end

-- Adjust wheel velocities based on detected lines
if redDetected then
print("skrecem desno")
wL, wR = wCurveL, wCurveR
-- Precomputed values to make a curve to the right
redDetected = false
lastTime = sim.getSimulationTime()
-- Reset lastTime for curve timing
curveInProgress = true
elseif blueDetected then
print("skrecem lijevo")
wL, wR = wCurveR, wCurveL
-- Precomputed values to make a curve to the left
blueDetected = false
lastTime = sim.getSimulationTime()  -- Reset lastTime for curve timing
curveInProgress = true
elseif greenDetected then
```

```
wL = 0
wR = 0
elseif curveInProgress then
-- Continue the curve until the minimum curve time is reached
if sim.getSimulationTime() - lastTime < minCurveTime then
-- Continue with curve velocities
wL, wR = curveWheelVelocities(direction)
else
curveInProgress = false  -- Curve completed
end
else
-- No line detected, continue following the wall
wL, wR = followWall(vref, direction, dFL, dRL, dFR, dRR, wallDist)
end

-- Set wheel velocities
sim.setJointTargetVelocity(left_wheel, wL)
sim.setJointTargetVelocity(right_wheel, wR)
end

function sysCall_sensing()
-- Get the distance readings from all sensors
dUS = getDistance(us, max_distUS)
dFR = getDistance(ir_FR, max_distLIDAR)
dRR = getDistance(ir_RR, max_distLIDAR)
dFL = getDistance(ir_FL, max_distLIDAR)
dRL = getDistance(ir_RL, max_distLIDAR)

-- Output the current state for debugging
print("Follow wall")

-- Read color values for line detection
local redValue = readLineColour(1)
local greenValue = readLineColour(2)
local blueValue = readLineColour(3)

-- Detect green line and stop
if greenValue >= GREEN_THRESHOLD then
print("Green line detected, stopping")
-- Stop the robot
sim.setJointTargetVelocity(left_wheel, 0)
sim.setJointTargetVelocity(right_wheel, 0)
end
end

function sysCall_cleanup()
-- do some clean-up here
end
```

```
simRemoteApi.start(19999)
function wallDetected(direction,dFrontL,dRearL,dFrontR,
,dRearR,dWallSide,tolerance)
local dFront,dRear,front,rear
if (direction==1) then
dFront=dFrontR
dRear=dRearR
else
dFront=dFrontL
dRear=dRearL
end
front=math.abs(dFront-dWallSide)<tolerance
rear=math.abs(dRear-dWallSide)<tolerance
return front and rear
end

function followWall(v,direction,dFL,dRL,dFR,dRR,dWallSide)
local phi,d,alpha,wL,wR
if (direction==1) then  -- follow right wall
phi=math.atan((dFR-dRR)/a)
d=(0.5*(dFR+dRR)-dWallSide)
else -- follow left wall
phi=math.atan((dRL-dFL)/a)
d=(dWallSide-0.5*(dFL+dRL))
end
gamma=kWall*d
alpha=phi+gamma
wL=(v/wheel_radius)*(math.cos(alpha)+(b/e)*math.sin(alpha))
wR=(v/wheel_radius)*(math.cos(alpha)-(b/e)*math.sin(alpha))
return wL,wR
end



function getVelocity(d,vref,dmin,dmax)
if (d<dmin) then
return 0
else
return vref*d/dmax
end
end

function getDistance(sensor,max_dist)
local detected, distance
detected,distance=sim.readProximitySensor(sensor)
if (detected<1) then
distance=max_dist
end
```

```
return distance
end

function readLineColour(color)
local totalColorValue = 0
local pixelIndex = 1
local image = sim.getVisionSensorImage(ir_sensor)
local numChannels = 3  -- Assuming RGB color space (Red, Green, Blue)

for i = 1, resolution[1] do
for j = 1, resolution[2] do
-- Each pixel consists of numChannels values (R, G, B)
local colorValue = image[pixelIndex + color - 1]
-- Extract the desired color channel
totalColorValue = totalColorValue + colorValue
pixelIndex = pixelIndex + numChannels
end
end

-- Calculate the average value of the chosen color channel
local averageColorValue = totalColorValue / (resolution[1] * resolution[2])

return averageColorValue
end

function turnSlow(vref,direction,speed_reduction,wheel_radius,b)
local wL,wR,t
wL=-direction*speed_reduction*vref/wheel_radius
wR=direction*speed_reduction*vref/wheel_radius
t=b*math.pi/2/(speed_reduction*vref)
return wL,wR,t
end

function makeCurve(vref,direction,cell_size,wheel_radius,b)
local wref,wL,wR,t
wref=(vref)/(cell_size/1.55)
wL=(vref+b*direction*wref)/wheel_radius
wR=(vref-b*direction*wref)/wheel_radius
t=(math.pi/2)/wref
return wL,wR,t
end

function curveWheelVelocities(direction)
if direction == 1 then
return wCurveL, wCurveR
else
return wCurveR, wCurveL
end
```

```
end
```

9.4. Prilog A4 - skripta "maze.py" za generaciju labirinta

```
import numpy as np
import random

# A class to represent a maze
#
class Maze:
def __init__(self):
self.maze = None
self.start_point = None
self.end_point = None
self.path = None
self.selected_block = None
self.block_size = 49
self.WALL_COLOR = [0,0,0]
self.BACKGROUND_COLOR_INT = 255
self.BACKGROUND_COLOR=[self.BACKGROUND_COLOR_INT for i in range(0, 3, 1)]
self.START_COLOR = [255, 0, 0]
self.GOAL_COLOR = [0, 255, 0]
self.PATH_COLOR = [0, 0, 255]
self.SELECTION_COLOR = [232, 244, 66]

# Public members
#

# Generates a new maze of (num_rows x num_cols) blocks
# Each item in the maze index will point to a cell
to which it is adjacent
# For example, if index [2,5] has the value (2,4),
then there is no wall between (2,5) and (2,4)
#
def generate(self, num_rows, num_cols):
self.maze = [[None for i in range(0, num_cols, 1)]
for j in range(0, num_rows, 1)]

# Straightforward implementation of the depth-first search
maze generation algorithm
# https://en.wikipedia.org/wiki/Maze_generation_algorithm
#
# Start by setting top-left corner as current point
#
current_point=(0,0)
self.maze[0][0] = current_point
```

```
maze_stack = [current_point]
unvisited_spaces = (num_rows * num_cols) - 1

# Keep iterating until all spaces have are joined to at
least one neighbor
#
while (unvisited_spaces > 0):

# Count the number of unvisited neighbors to the current point
#
current_unvisited_neighbors = self.__get_unvisited_neighbors
(current_point)

# If all are visited, backtrack until we find a
visited point with unvisited neighbors
#
if (len(current_unvisited_neighbors) == 0):
current_point = maze_stack.pop()

# If some are unvisited, randomly pick one to move to
# Mark that block as adjacent to current_point
else:
next_point = random.choice(current_unvisited_neighbors)
self.maze[next_point[0]][next_point[1]] = current_point
maze_stack.append(current_point)
current_point = next_point
unvisited_spaces -= 1

def set_start_point(self, start_point):
self.start_point = start_point

def get_start_point(self):
return self.start_point

def set_end_point(self, end_point):
self.end_point = end_point

def get_end_point(self):
return self.end_point

def get_maze(self):
return self.maze

def set_path(self, path):
self.path = path

def set_selected_block(self, selected_block_coords):
self.selected_block = selected_block_coords
```

```
def clear_selected_block(self):
self.selected_block = None

def get_block_size(self):
return self.block_size

# Generates an image to graphically represent the maze
#
def generate_image(self):
if self.maze is None:
raise ValueError('Maze is not initialized')

# First, draw the background
#
image = self.__initialize_image()

# The order of the drawing is important - otherwise,
we can get weird graphical artifacts
#
if (self.path is not None):
image = self.__draw_path(image, self.path)
if (self.selected_block is not None):
image = self.__draw_selected_block(image, self.selected_block)
if (self.start_point is not None):
image = self.__draw_start(image, self.start_point)
if (self.end_point is not None):
image = self.__draw_end(image, self.end_point)

# Draw all of the walls
#
image = self.__draw_all_walls(image)

# If two blocks are adjacent, remove the walls between them
#
for y in range(0, len(self.maze), 1):
for x in range(0, len(self.maze[0]), 1):
# 0, 0 is the starting point, so it has no walls to remove
#
if (x != 0 or y != 0):
image = self.__remove_walls(image, (y,x), self.maze[y][x])

return image

# Private members
#

# Given a block in a maze, determine how many
```

```
of its neighbors are 'None'
# This is used to generate the maze
#
def __get_unvisited_neighbors(self, point):
unvisited_neighbors = []
for offsets in [(0, -1), (0, 1), (1, 0), (-1, 0)]:
y_ind = point[0] + offsets[0]
x_ind = point[1] + offsets[1]

if y_ind >= 0 and y_ind < len(self.maze):
if x_ind >= 0 and x_ind < len(self.maze[0]):
if self.maze[y_ind][x_ind] is None:
unvisited_neighbors.append((y_ind, x_ind))
return unvisited_neighbors

# Initializes the image to a constant color
#
def __initialize_image(self):
image = np.full((len(self.maze) * self.block_size, len(self.maze[0]) *
self.block_size, 3), self.BACKGROUND_COLOR_INT, dtype=np.uint8)
return image

# Draws all of the possible maze walls on the board
#
def __draw_all_walls(self, image):
for y in range(0, image.shape[0], 1):
for x in range(0, image.shape[1], 1):
is_wall = False
if y % self.block_size == 0 or y % self.block_size
== self.block_size-1:
is_wall = True
elif x % self.block_size == 0 or x % self.block_size
== self.block_size-1:
is_wall = True
if is_wall:
image[y][x] = self.WALL_COLOR
return image

# Erases a wall between two points
#
def __remove_walls(self, image, first_point, second_point):
if (first_point[0] == second_point[0] and first_point[1]
== second_point[1]):
raise ValueError('Duplicate points: {0} and {1}'.format
(first_point, second_point))

# Vertical alignment
#
```

```
if (first_point[0] == second_point[0]):
if (first_point[1] > second_point[1]):
image = self.__remove_left_wall(image, first_point)
image = self.__remove_right_wall(image, second_point)
elif (first_point[1] < second_point[1]):
image = self.__remove_right_wall(image, first_point)
image = self.__remove_left_wall(image, second_point)

#Horizontal alignment
#
elif (first_point[1] == second_point[1]):
if (first_point[0] > second_point[0]):
image = self.__remove_top_wall(image, first_point)
image = self.__remove_bottom_wall(image, second_point)
elif (first_point[0] < second_point[0]):
image = self.__remove_bottom_wall(image, first_point)
image = self.__remove_top_wall(image, second_point)
else:
raise ValueError('points {0} and {1} cannot be connected.'.format
(first_point, second_point))
return image

# Removes the wall on the top of a block
#
def __remove_top_wall(self, image, point):
y_px = (point[0]) * self.block_size
x_start = (point[1] * self.block_size) + 1
x_end = ( (point[1]+1) * self.block_size ) - 1

color = self.BACKGROUND_COLOR
if self.path is not None and point in self.path:
color = self.PATH_COLOR

while x_start < x_end:
image[y_px][x_start] = color
x_start += 1
return image

# Removes the wall on the bottom of a block
#
def __remove_bottom_wall(self, image, point):
y_px = ((point[0]+1) * self.block_size) - 1
x_start = (point[1] * self.block_size) + 1
x_end = ((point[1]+1) * self.block_size) - 1

color = self.BACKGROUND_COLOR
if self.path is not None and point in self.path:
color = self.PATH_COLOR
```

```
while x_start < x_end:
image[y_px][x_start] = color
x_start += 1
return image

# Removes the wall on the left of a block
#
def __remove_left_wall(self, image, point):
x_px = (point[1]) * self.block_size
y_start = (point[0] * self.block_size) + 1
y_end = ( (point[0]+1) * self.block_size ) - 1

color = self.BACKGROUND_COLOR
if self.path is not None and point in self.path:
color = self.PATH_COLOR

while y_start < y_end:
image[y_start][x_px] = color
y_start += 1
return image

# Removes the wall on the right of a block
#
def __remove_right_wall(self, image, point):
x_px = ((point[1] + 1) * self.block_size) - 1
y_start = (point[0] * self.block_size) + 1
y_end = ((point[0]+1) * self.block_size) - 1

color = self.BACKGROUND_COLOR
if self.path is not None and point in self.path:
color = self.PATH_COLOR

while y_start < y_end:
image[y_start][x_px] = color
y_start += 1
return image

# Draws the icon for the starting point for the agent
#
def __draw_start(self, image, point):
return self.__draw_square_in_center_of_patch
(image, point, self.START_COLOR)

# Draws the icon for the ending point for the agent
#
def __draw_end(self, image, point):
return self.__draw_square_in_center_of_patch
```

```
(image, point, self.GOAL_COLOR)

# Draws a small square in the center of the block
#
def __draw_square_in_center_of_patch(self, image, point, color):
center_y_pixel = (point[0] * self.block_size) + (self.block_size // 2)
center_x_pixel = (point[1] * self.block_size) + (self.block_size // 2)

min_x = center_x_pixel - 3
max_x = center_x_pixel + 3
min_y = center_y_pixel - 3
max_y = center_y_pixel + 3

return self.__draw_filled_rectangle
(image, min_x, min_y, max_x, max_y, color)

def __draw_path(self, image, path):
for path_node in path:
min_x = path_node[1] * self.block_size
min_y = path_node[0] * self.block_size
max_x = (path_node[1] + 1) * self.block_size
max_y = (path_node[0] + 1) * self.block_size

image = self.__draw_filled_rectangle
(image, min_x, min_y, max_x, max_y, self.PATH_COLOR)
return image


def __draw_selected_block(self, image, selected_block):
min_x = selected_block[1] * self.block_size
min_y = selected_block[0] * self.block_size
max_x = (selected_block[1] + 1) * self.block_size
max_y = (selected_block[0] + 1) * self.block_size

return self.__draw_filled_rectangle
(image, min_x, min_y, max_x, max_y, self.SELECTION_COLOR)

def __draw_filled_rectangle
(self, image, min_x, min_y, max_x, max_y, color):
for y in range(min_y, max_y, 1):
for x in range(min_x, max_x, 1):
image[y][x] = color
return image
```

9.5. Prilog A5 - skripta "qlearn_agent.py" za treniranje agenta

```
import sys
import numpy as np
import maze
import random

# The main learning agent
# Given a maze, learns the optimal path from each
 starting point via Q-Learning
#
class QLearnAgent():
def __init__(self):
# Reward matrix
# R(A(s, s')) in the blog post
# s indexes the rows, s' the columns
# So R(A(s,s')) = R[s][s']
self.R = None

# State matrix
# Q(A(s,s')) in the blog post
# s indexes the rows, s' the columns
# So Q(A(s,s')) = Q[s][s']
self.Q = None

self.num_columns = None
self.end_state = None
self.num_states = None
self.next_states = {}
self.trained = False

# Public Members
#
def is_trained(self):
return self.trained

# Initializes the learning matricies
# Q matrix -> zeros
# R matrix -> 0 if connected, 1 if goal, -1 otherwise
#
def initialize(self, maze):
if maze is None:
return
maze_fields = maze.get_maze()
end_point = maze.get_end_point()
if end_point is None:
return
self.num_states = len(maze_fields)*len(maze_fields[0])
self.num_columns = len(maze_fields[0])
```

```
# Begin by initializing the reward matrix to zero connectivity
#
self.R = np.full((self.num_states, self.num_states),
-1, dtype=np.float64)

# For each point in the maze, connect it to its parent
#
for y in range(0, len(maze_fields), 1):
for x in range(0, len(maze_fields[0]), 1):
first_point = (y,x)
second_point = maze_fields[y][x]

# States may not point to themselves
#
if (first_point[0] != second_point[0] or
first_point[1] != second_point[1]):
first_state = self.__maze_dims_to_state
(first_point[0], first_point[1])
second_state = self.__maze_dims_to_state
(second_point[0], second_point[1])

# Create a hashtable for neighboring states to
avoid continuous iteration over matrix
#
if first_state not in self.next_states:
self.next_states[first_state] = []
self.next_states[first_state].append(second_state)

if second_state not in self.next_states:
self.next_states[second_state] = []
self.next_states[second_state].append(first_state)

self.R[first_state][second_state] = 0
self.R[second_state][first_state] = 0

# Set terminal state to point to self,
as well as all neighbors to point to it
#
self.end_state = self.__maze_dims_to_state
(end_point[0], end_point[1])
for i in range(0, self.num_states, 1):
if self.R[i][self.end_state] != -1:
self.R[i][self.end_state] = 1.0
self.R[self.end_state][self.end_state] = 1.0

# Initialize Q matrix to zeros
#
```

```
self.Q = np.full((self.num_states, self.num_states), 0.0)
self.trained = False

# Trains the agent
# initialize() should have been called
before this function is called
#
def train(self, gamma, min_change_per_epoch):
print('Training...')
epoch_iteration = 0
while True:
previous_q = np.copy(self.Q)

# Consider multiple states per epoch.
# Early termination can happen if same state is picked twice
#
for i in range(0, 10, 1):
# Pick a random starting position
#
current_state = random.randint(0, self.num_states-1)

# Keep iterating until goal is reached
#
while(current_state != self.end_state):

# Pick a random next state
#
possible_next_states = self.next_states[current_state]
next_state = random.choice(possible_next_states)

# Get the outgoing states from next state.
# Compute the max Q values of those outgoing states
#
max_q_next_state = -1
next_next_states = self.next_states[next_state]
for next_next_state in next_next_states:
max_q_next_state = max([max_q_next_state,
self.Q[next_state][next_next_state]])

# Set Q value for transition from current->
next state via bellman equation
#
self.Q[current_state][next_state] = self.R
[current_state][next_state] + (gamma * max_q_next_state)

# Move to next state
#
current_state = next_state
```

```
# Normalize the Q matrix to avoid overflow
#
self.Q = self.Q / np.max(self.Q)

#Check stopping criteria
diff = np.sum(np.abs(self.Q - previous_q))
print('In epoch {0}, difference is {1}'.format(epoch_iteration, diff))
if (diff < min_change_per_epoch):
break

epoch_iteration += 1

# Agent is trained!
#
self.trained = True

# Given a starting state, predict the optimal path to the ending state
# This should be called only on a trained agent
def solve(self, starting_state):
if not self.trained:
return []

# The first point in the path is the starting state
# Translate from (y,x) coordinates into state index
#
path = [starting_state]
current_state = self.__maze_dims_to_state
(starting_state[0], starting_state[1])

# Keep going until we reach the goal
while(current_state != self.end_state and len
(path) < self.num_states):

# For all of the next states, determine the
state with the highest Q value
#
possible_next_states = self.next_states[current_state]
best_next_state = possible_next_states[0]
best_next_state_reward = self.Q[current_state][best_next_state]

for i in range(1, len(possible_next_states), 1):
potential_next_state = possible_next_states[i]
if (self.Q[current_state][potential_next_state]
> best_next_state_reward):
best_next_state = potential_next_state
best_next_state_reward =
```

```
self.Q[current_state][potential_next_state]

# Move to that state, and add to path
#
current_state = best_next_state
path.append(self.__state_to_maze_dims(current_state))

return path

# Private Members
#

# Converts (y,x) coordinates to a numerical state
#
def __maze_dims_to_state(self, y, x):
return (y*self.num_columns) + x

# Converts a numerical state to (y,x) coordinates
#
def __state_to_maze_dims(self, state):
y = int(state // self.num_columns)
x = state % self.num_columns
return (y,x)
```

9.6. Prilog A6 - skripta "main_window.py" za GUI

```
import sys
import gi
gi.require_version('Gtk', '3.0')
from gi.repository import Gtk
from gi.repository import GdkPixbuf

import maze
import qlearn_agent

# The main event handler for the program
# The functions here get called when buttons are clicked in the GUI
#
class MainWindowController:
def __init__(self, builder):
self.maze = None

# Set these to the default values set in the Glade GUI
#
self.maze_size = (6,6)
```

```
        self.builder = builder
        self.gamma = 0.8
        self.maze_selected_block = None
        self.agent = None

    def GammaTextEntryValueChanged(self, text_entry):
        self.gamma = text_entry.get_text()

    def NumberOfColumnsAdjustmentValueChanged(self, adjustment):
        self.maze_size = (self.maze_size[0], int(adjustment.get_value()))

    def NumberOfRowsAdjustmentValueChanged(self, adjustment):
        self.maze_size = (int(adjustment.get_value()), self.maze_size[1])

    def GenerateNewMazeButtonPressed(self, button):
        self.maze = maze.Maze()
        self.maze.generate(self.maze_size[0], self.maze_size[1])
        self.__reset_agent()
        self.__redraw_maze()

    def SetStartingPointButtonPressed(self, button):
        if (self.maze_selected_block is not None):
            self.maze.set_start_point(self.maze_selected_block)
            self.maze.clear_selected_block()
            self.maze_selected_block = None
            self.maze.set_path(None)
            self.__redraw_maze()

    def SetEndingPointButtonPressed(self, button):
        if (self.maze_selected_block is not None):
            self.maze.set_end_point(self.maze_selected_block)
            self.maze.clear_selected_block()
            self.maze_selected_block = None
            self.maze.set_path(None)
            self.__reset_agent()
            self.__redraw_maze()

    def ResetAgentButtonPressed(self, button):
        self.__reset_agent()
        self.maze.set_path(None)
        self.maze.set_start_point(None)
        self.maze.set_end_point(None)
        self.__redraw_maze()

    def TrainAgentButtonPressed(self, button):
        try:
            gamma = float(self.gamma)
            if (gamma < 0 or gamma > 1):
```

```
raise ValueError('gamma')
except:
self.__show_popup('Error: gamma must be a value on the range [0, 1]')
return

if self.maze is None:
self.__show_popup('Error: Maze is not generated.
Please generate maze.')
return

if self.maze.get_end_point() is None:
self.__show_popup('Error: End point not specified. Please specify
end point by selecting a block and clicking "Set Endpoint"')
return

if self.agent is None:
self.agent = qlearn_agent.QLearnAgent()
self.agent.initialize(self.maze)

self.agent.train(gamma, 0.001)
self.__show_popup('Agent successfully trained.')

def RunAgentButtonPressed(self, button):
if (self.agent is None or not self.agent.is_trained()):
self.__show_popup('Error: Agent not trained.
Please train the agent before attempting to run it')
return

start_point = self.maze.get_start_point()
if start_point is None:
self.__show_popup('Error: Start point not specified. Please specify
start point by selecting a block and clicking "Set Starting Point"')
return

path = self.agent.solve(start_point)
self.maze.set_path(path)

self.__redraw_maze()

# Find the block that the user clicked on and highlight it
#
def MazeImageEventBoxPressed(self, event_box, click_event):
block_size = self.maze.get_block_size()
self.maze_selected_block = (int(click_event.y // block_size),
int(click_event.x // block_size))
self.maze.set_selected_block(self.maze_selected_block)
self.__redraw_maze()
```

```
def __reset_agent(self):
if self.maze is None:
self.agent = None
else:
self.agent = qlearn_agent.QLearnAgent()
self.agent.initialize(self.maze)

# Draws the maze image in the GUI
#
def __redraw_maze(self):
#
image = self.maze.generate_image()
header = bytes('P6 {0} {1} 255 '.format(image.shape[1],
image.shape[0]), 'ascii')
iloader = GdkPixbuf.PixbufLoader.new_with_type('pnm')
iloader.write(header)
iloader.write(bytes(image))
pixbuf = iloader.get_pixbuf()
iloader.close()
image_ref = builder.get_object('MazeImage')
image_ref.set_from_pixbuf(pixbuf)

# Shows a small popup window with a message to the user
#
def __show_popup(self, message):
md = Gtk.MessageDialog()
md.set_markup(message)
md.run()
md.destroy()


# The main entry point of the program
# Builds the window, attaches the
MainWindowController, and shows the window
builder = Gtk.Builder()
builder.add_from_file('main_window.glade')

controller = MainWindowController(builder)
builder.connect_signals(controller)

window = builder.get_object('MainWindow')
window.connect('destroy', Gtk.main_quit)
window.show_all()

Gtk.main()
```