Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Robert Slak

Primerjava različnih lastnosti večagentnih sistemov v simulacijskem okolju

diplomsko delo
na univerzitetnem študiju

Mentor: doc. dr. Danijel Skočaj

Ljubljana 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01863/2012
Datum: 04.09.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ROBERT SLAK**

Naslov: **PRIMERJAVA RAZLIČNIH LASTNOSTI VEČAGENTNIH SISTEMOV V SIMULACIJSKEM OKOLJU**

COMPARISON OF DIFFERENT PROPERTIES OF MULTIAGENT SYSTEMS IN SIMULATED ENVIRONMENT

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Nekatere lastnosti večagentnih sistemov lahko učinkovito raziskujemo v simulacijskem okolju. V diplomski nalogi razvijte ustrezno simulacijsko okolje ter implementirajte večagentni sistem, ki se bo s konkurenčnim sistemom boril za vire v simuliranem okolju. Preučite različne algoritme za preiskovanje prostora ter eksperimentalno preizkusite kako na rezultate vplivajo različne lastnosti večagentnega sistema, kot so število agentov v sistemu ter vpliv komunikacije med agenti.

Mentor:

doc. dr. Danijel Skočaj

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Robert Slak, z vpisno številko **63070147**, sem avtor diplomskega dela z naslovom:

Primerjava različnih lastnosti večagentnih sistemov v simulacijskem okolju

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Danijela Skočaja,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 15. marca 2013 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Danijelu Skočaju za njegovo strokovno in mentorsko podporo pri pripravi diplomske naloge. Hvala tudi Luki Čehovinu za razvit simulacijski sistem in njegove nasvete pri nadgradnji le-tega. Zahvala gre tudi Mateji, ki mi je pri pisanju naloge nudila moralno in lektorsko podporo. Nenazadnje pa se zahvaljujem staršema, ki sta me podpirala tekom celotnega študija.

Kazalo

Povzetek

V diplomski nalogi je predstavljeno simulacijsko okolje GridLand, v katerem simuliramo dva večagentna sistema, ki se potegujeta za vire v tem okolju.

Razvili smo agenta, ki je zmožen delovanja v večagentnem sistemu in ima nabor različnih lastnosti, s katerimi vplivamo na njegovo delovanje. Preučili in implementirali smo dva iskalna algoritma - Dijkstra in A*, ki ju lahko agent uporabi pri iskanju virov in pri navigaciji po prostoru. Delovanje agenta smo preizkusili v simulacijskem okolju, kjer smo opazovali, kako določena lastnost agenta vpliva na rezultat simulacije.

Zanimalo nas je, ali komunikacija vpliva na rezultat simulacije in kako se ta vpliv odraža s spreminjanjem radija komunikacije. Prav tako smo merili vpliv števila agentov v sistemu, načina preiskovanja prostora in taktike napadanja agentov nasprotne ekipe.

Rezultate opravljenih simulacij smo preučili in ovrednotili pomembnost posamezne lastnosti glede na prispevek h količini nabranih virov. Potrdili smo hipotezo o pozitivnem vplivu povečanja števila agentov v sistemu in o pozitivnem vplivu komunikacije na delovanje sistema.

Ključne besede: večagentni sistem, simulacijsko okolje, preiskovalni algoritmi, A*, agent

Abstract

The thesis presents the simulation environment GridLand, where we simulate two multi-agent systems, that are competing for resources.

We have developed an agent that can operate in a multi-agent system and has various properties. These properties have an affect on its actions. We examined and implemented two search algorithms - Dijkstra and A*, which the agent can use for discovering resources and navigating in the environment. The agent's activity was tested in the simulation environment, we observed how specific properties of the agent effected the final simulation result.

We were interested in whether the communication has an influence on the result and how this influence is reflected by changing the radius of communication. We also measured the influence of the number of agents in the system, the search method and the attacking strategy.

We examined the simulation results and evaluated the importance of each property on the basis of the quantity of gathered resources. We have confirmed the hypothesis that increasing the number of agents and the radius of communication has a positive influence on an agent's performance.

Key words: multi-agent system, simulation environment, search algorithms, A*, agent

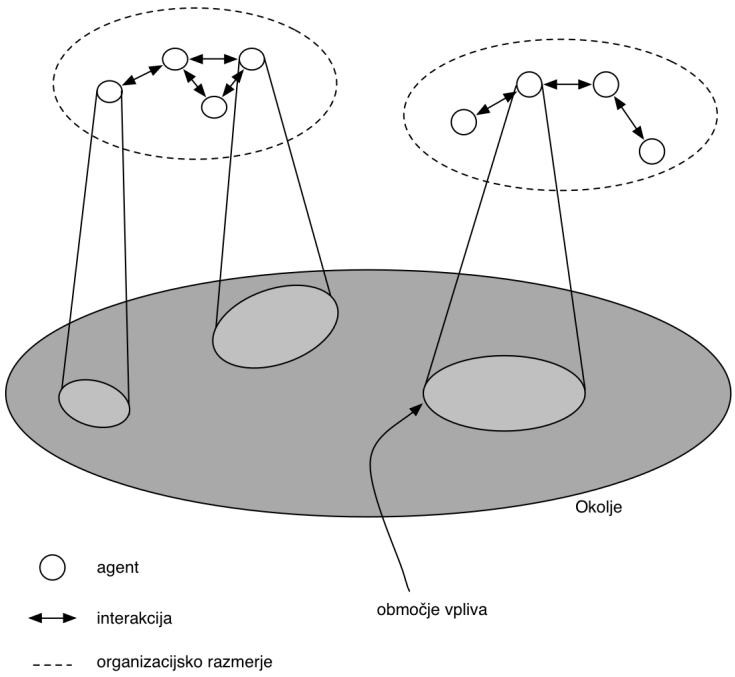
Poglavje 1

Uvod

Večagentni sistemi so relativno novo področje v računalniški znanosti. Zanimanje za take sisteme se je začelo okoli leta 1980, do njihove široke prepoznavnosti pa je prišlo šele sredi devetdesetih let prejšnjega stoletja. Področje je v zadnjih letih pridobilo veliko veljavo, saj je z razširjenostjo interneta omogočilo reševanje in analizo kompleksnih problemov.

Večagentni sistemi so sistemi, sestavljeni iz več interakcijskih računskih elementov, znanih pod imenom agentje, ti pa so umeščeni v neko okolje [9]. Uporabljamo jih lahko za reševanje težkih problemov, ki jih en sam agent ne zmore rešiti ali pa jih reši le s težavo. Tipično z večagentnimi sistemi povezujemo programske agente, čeprav lahko tak sistem sestoji tudi iz agentov, ki so roboti, ljudje ali pa skupine ljudi. Agentje so entitete, bodisi računalniški bodisi robotski sistemi z dvema pomembnima zmožnostma. V prvi vrsti so vsaj v neki meri zmožni avtonomnih akcij. Sprejemajo lahko samostojne odločitve, s katerimi bodo dosegli zastavljene cilje. Druga zmožnost agentov pa je sposobnost interakcije z drugimi agenti. Interakcija je lahko preprosta, kot na primer izmenjava znanja ali pa kompleksnejša, kot so sodelovanje, koordinacija in pogajanje[9]. Agentje so lahko pasivni (brez ciljev), aktivni (s preprostimi cilji) ali pa kompleksni, kjer imajo določene kognitivne sposobnosti.

Slika 1.1, povzeta iz [9], prikazuje tipično strukturo večagentnega sis-



Slika 1.1: Tipična struktura večagentnega sistema [9].

tema. Sistem vsebuje nekaj agentov, ki medsebojno sodelujejo s pomočjo komunikacije. Agentje so sposobni delovanja v okolju - različni agentje imajo različna območja vpliva (ang. spheres of influence), kar pomeni da kontrolirajo oziroma da imajo vpliv na različne dele okolja. Ta območja vpliva se lahko včasih prekrivajo, kar lahko med agenti povzroči medsebojna odvisna razmerja.

Cilj diplomske naloge je narediti aktivnega agenta, ki bo sposoben delovanja v večagentnem sistemu. Ta sistem naj bi bil simuliran tako, da bi bilo agentovo okolje nek virtualen prostor, v katerem bi bilo na voljo končno mnogo virov, ki jih agent lahko pobere. Da agent pridobi vire, jih je potrebno najprej poiskati. Operacijo iskanja sprva potrebujemo za preiskovanje prostora, ko najdemo vir pa ga uporabimo za določitev poti do vira in kasneje še za določitev najkrajše poti do točke, kjer vir odložimo. Zanima nas, kako bi se v tem okolju odrezali dve skupini agentov, kjer bi si vsaka želela zagotoviti

čim več virov. Ali bi se končen rezultat spremenil, če bi agentje ene skupine komunicirali med sabo, medtem ko druga skupina ne bi? Kako je rezultat odvisen od radija komunikacije? Ali ima način iskanja poti in raziskovanja vpliv na uspeh ekipe? Kakšno bi bilo razmerje v pobranih virih pri določeni spremembi v delovanju agenta? Ali je ekipa napadalnih agentov kaj v prednosti pred ekipo, ki ne napada drugih agentov? Zanima nas, kakšne lastnosti mora imeti agent, da se čim bolje odreže pri ekipnem tekmovanju za vire v simulacijskem okolju. Katera lastnost je najpomembnejša, kako se izraža v količini nabranih virov in kako je odvisna od prostora, v katerem se agent nahaja - to so nekatera vprašanja, na katera želimo odgovoriti v naši nalogi. Naša hipoteza je, da velikost radija komunikacije in število agentov pozitivno vpliva na rezultat. Prav tako predvidevamo, da ima način raziskovanja določen vpliv.

Da poiščemo odgovore na zgornja vprašanja, si moramo izbrati simulacijsko okolje, ga preučiti in ga po potrebi še razširiti, tako da bo podpiralo želen način simulacije. Implementirati je potrebno agenta, ki je sposoben avtonomnega in ekipnega delovanja. Agent mora imeti vse lastnosti, katerih vpliv želimo preveriti.

Diplomska naloga je razdeljena na štiri dele. V prvem delu se spoznamo s simulacijskim okoljem, ga izboljšamo in prilagodimo svojim zahtevam. V drugem delu preučimo nekatere preiskovalne algoritme in izberemo ustreznega, ki ga bomo implementirali. Tretji del predstavlja implementacijo agenta, ki je sposoben delovanja v simulacijskem okolju. Na koncu izvedemo še simulacije različnih konfiguracij agentov ter preverimo, kako vplivajo na končni ekipni rezultat.

Poglavje 2

Simulacijsko okolje

Za simulacijo delovanja agenta potrebujemo simulacijsko okolje. To okolje mora imeti možnost simuliranja več agentov, oziroma več skupin agentov. Biti mora prilagodljivo, tako da lahko določimo samo zgradbo - zemljevid okolja, ter nastavljivo do te mere, da lahko spreminjamo parametre simulacije (hitrost simulacije, radij zaznavanja agenta). Imeti mora podporo za medagentno komunikacijo, tako da si lahko agentje pošiljajo poljubna sporočila.

Po pregledu obstoječih simulacijskih okolij smo naleteli na dve, ki sta ustrezali našim zahtevam. Prvo okolje je *MAS Simulator* [4], razvito v laboratoriju za večagentne sisteme na univerzi v Massachusetts-u. Okolje je namenjeno testiranju koordinacije in pogajanju v večagentnih sistemih. Drugo okolje je okolje *GridLand* [1]. Namenjeno je simulaciji igre pobiranja zastavic (ang. capture the flag), kjer se več agentov poteguje za določen vir v okolju. Ker je namen tega okolja podoben naši ideji o simulaciji skupin agentov, ki se potegujejo za vire, smo se odločili, da ga izberemo za simulacijsko okolje, za katerega bomo implementirali agenta in testirali njegovo delovanje.

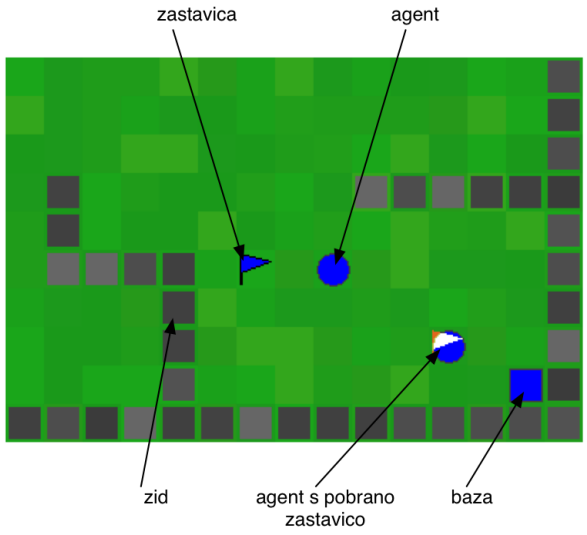
Okolje *GridLand* je zasnovano na principu arhitekture strežnik - odjemalec. Napisano je v programskem jeziku Java in izdano pod licenco GPL. Dostopno je na GitHub-ovem repozitoriju [1]. Avtor okolja je Luka Čehovin.

V našem delu uporabimo okolje *GridLand* kot osnovo, ki jo nadgradimo in posodobimo z nekaterimi funkcionalnostmi, ki so potrebne za uspešno

izvedbo simulacij. V nadaljevanju bomo opisali zgradbo okolja, njegovo konfiguracijo in nadgradnjo, ki je bila potrebna, da je zadostil našim potrebam.

2.1 Predstavitev okolja

Simulacijsko okolje je predstavljeno z dvodimenzionalno mrežo polj, ki jo imenujemo zemljevid. Začetno stanje zemljevida je podano na začetku simulacije, nato pa se spreminja glede na akcije agentov. Del zemljevida simulacijskega okolja skupaj z opisom posameznih polj lahko vidimo na sliki 2.1. Polje zemljevida je lahko prazno, lahko vsebuje enega od gradnikov, ali pa se na njem nahaja agent. Gradniki so: zid, zastavica in baza. Zid predsta-



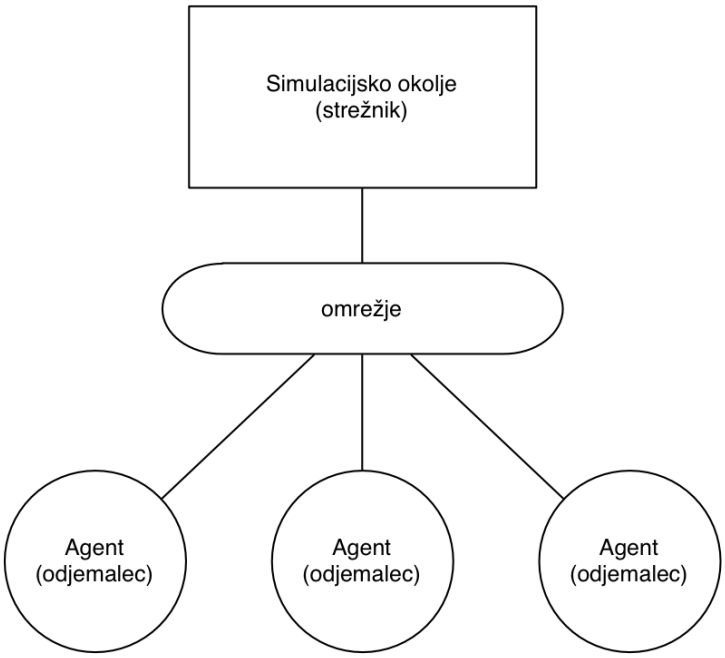
Slika 2.1: Del zemljevida kjer se nahajata agenta, njuna baza in dve zastavici.

vlja neprehoden prostor na zemljevidu, ki ga ne moremo premakniti in skozi celotno simulacijo zavzema isto mesto. Zastavica predstavlja vir v okolju, ki ga lahko agent pobere. Baza je gradnik ob katerem se pojavijo agentje, ko vstopijo v simulacijo. Vsak agent pa mora do polja s tem gradnikom prinesti pobrano zastavico. Vsaka ekipa agentov ima eno bazo, zato lahko zemljevid

vsebuje le toliko baz, kolikor je ekip. Tako kot baza, je tudi zastavica dodeljena določeni ekipi. Vsaka ekipa ima lahko po več zastavic, medtem ko je zastavica lahko le od ene ekipe. Ko agent pride na polje z zastavico svoje ekipe pravimo, da je agent pobral zastavico oziroma pridobil vir. Ko agent nosi zastavico, je njegova hitrost manjša kot običajno. Agent nosi zastavico dokler ta ne pride na polje z gradnikom baze oziroma se zaleti v zid ali v drugega agenta. Premik na polje, kjer se nahaja baza, se odraža s povečanjem rezultata ekipe, kateri je pripadal agent, ki se je premaknil na to polje. Če se agent premakne na polje, ki vsebuje neprehoden gradnik oziroma je na njem drug agent, se ta agent izloči iz igre.

Arhitektura sistema je zasnovana na principu strežnik - odjemalec. Kot vidimo na sliki 2.2, simulacijsko okolje predstavlja strežniški del. Strežniški del nudi okolje za agentovo delovanje in skrbi za potek same simulacije. Strežnik vodi tudi seznam priklopljenih agentov (agentov, vključenih v simulacijo) in če kateri izmed agentov naredi napako v delovanju, ga kaznuje z odklopom (izključitvijo). Agent predstavlja odjemalca, ki se poveže na strežnik z zahtevo za vključitev v simulacijsko okolje. Agent kot odjemalec pošilja zahteve na strežnik, ta pa mu na vsako zahtevo odgovori. Agentove zahteve so akcije, ki jih ta želi opraviti - na primer premik v okolju ali odpošiljatev sporočila. Opravljene akcije lahko vplivajo na stanje simulacijskega okolja, kar se odraža v spremembi vsebine odgovorov. Agent si na podlagi odgovorov, ki jih prejme iz strežnika, zgradi podobo o stanju okolja in se na podlagi tega odloča o nadaljnjih akcijah.

Komunikacija med agentom in simulacijskim okoljem poteka preko omrežja. Če sta strežnik in odjemalec nameščena na istem računalniku, komunikacija poteka preko povratnega naslova (ang. loopback address). V kolikor sta strežnik in odjemalec nameščena na različnih napravah, komunikacija poteka preko lokalnega omrežja (LAN) ali pa interneta. Ker je v tej arhitekturi strežnik centralna točka, mora komunikacija med dvema agentoma potekati preko njega. To predstavlja slabost z vidika zmogljivosti, saj s tem obremenimo omrežje in povečamo čas dostave sporočil. Po drugi strani pa pred-



Slika 2.2: Strežnik - odjemalec arhitektura.

stavlja prednost, saj lahko strežnik spremlja vso komunikacijo med agenti in nanjo vpliva.

2.2 Strežniški del

Strežniški del simulacijskega okolja služi kot priklopna točka za vse odjemalce – agente. Na njem se regulirajo pravila simulacije, kot na primer število agentov, parametri dopustne komunikacije in izločanje agentov iz okolja.

2.2.1 Parametri simulacije

Vsako simulacijo definiramo z množico parametrov, podanih v inicializacijski datoteki, ki jo ob zagonu kot parameter podamo programu (strežniku). Datoteka je v formatu *INI*, kar pomeni, da so parametri zapisani v obliki ključ-vrednost.

Parametri simulacije so:

- Hitrost simulacije (*gameplay.speed*) – hitrost odvijanja simulacije N; en korak simulacije je 1s / N.
- Simulacijski zemljevid (*gameplay.field*) – ime datoteke, kjer je definiran zemljevid simulacijske površine.
- Število agentov (*gameplay.agents*) – maksimalno število agentov v vsaki ekipi.
- Interval prihajanja (*gameplay.respawn*) – število korakov med posameznim prihodom (rojstvom) agenta.
- Vrsta zastavic (*gameplay.flags*) – način pojavljanja zastavic (hrane).
- Število zastavic (*gameplay.flags.pool*).
- Interval pojavljanja zastavic (*gameplay.flags.respawn*) – število korakov med pojavitvami novih zastavic.
- Teža zastavice (*gameplay.flags.weight*) – teža posamezne zastavice. Vsaka pobrana zastavica upočasni hitrost premikanja agenta.

Okolje nam omogoča sledeče načine pojavljanja zastavic:

- Unikatno (*unique*) – zastavicam se določi lokacija, ki je podana na zemljevidu.
- Naključno (*random*) - zastavice se naključno porazdelijo po zemljevidu.
- Ponovna pojavitev (*respawn*) - zastavice se naključno porazdelijo po zemljevidu in se ob dosegu točke ponovno pojavijo.
- Način za merjenje (*benchmark*) – poseben način, dodan za naše potrebe. Gre za način *unikatno*, z možnostjo da so vse zastavice skupne (lahko jih pobere katerakoli ekipa)

Parametri protokola:

- Maksimalna velikost sporočila (*message.size*) – največja dovoljena velikost poslanega sporočila drugemu agentu, izražena v bajtih.
- Velikost okolice (*message.neighborhood*) – radij okolice, ki jo agent vidi.
- Hitrost sporočanja (*message.speed*) – Hitrost pošiljanja sporočila med agenti. Vrednost ne predstavlja časa, dejansko potrebnega za prenos sporočila preko protokola TCP/IP, ampak le zamik med sprejetjem in pošiljanjem sporočila, ki je umetno ustvarjen na strežniški strani.

Za vsako ekipo je potrebno določiti ustrezno identifikacijsko oznako skupaj z imenom ekipe. Identifikacijska oznaka se začne z nizom *team*, ki ji sledi zaporedna številka ekipe (npr. *team1 = ekipaRdeči*).

Vsaki ekipi lahko določimo tudi geslo, ki ga potrebujejo agentje za uspešno priključitev ekipi. Geslo za posamezno ekipo določimo s parametrom *team[številka ekipe].passphrase*.

Simulacija se odvija na zemljevidu, ki ga določimo z ustreznim parametrom (*gameplay.field*). Zemljevid je *N*M* velika mreža, sestavljena iz polj, ki so lahko naslednjih tipov:

- prazno polje,
- zid,
- baza ekipe,
- zastavica ekipe.

Zemljevid je lahko podan v dveh različnih formatih – bodisi kot tekstovna datoteka bodisi kot slika v formatu PNG. Če je podana v tekstovni datoteki je le-ta kodirana kot serija znakov ACSII. Število vrstic predstavlja višino, dolžina najdaljše vrstice pa širino zemljevida. Krajše vrstice so podaljšane z znakom za prazno polje, tako da so vse vrstice enako široke. Prazno polje je predstavljeno z znakom presledek, zid z znakom lojtra (#), baza ekipe in ekipne zastave pa so predstavljene z znakom iz angleške abecede. Vsaka

ekipa ima svojo črko pri čemer velika črka predstavlja bazo ekipe, mala pa zastavo ekipe.

Zemljevid lahko podamo tudi kot kot 8-bitno sliko RGB, zaželen je format PNG. Vsaka točka na sliki predstavlja svoje polje na zemljevidu. Barva točke določa tip celice. Prazno celico predstavimo s katerokoli sivino z vrednostjo nad 200. Vrednost sivine pod 200 predstavlja zid. Vsako ne-sivo barvo pretvorimo v barvni prostor HSB (hue saturation brightness). Odtenek te barve predstavlja ekipo, svetlost pa ali gre za bazo (vrednost pod 128) ali zastavico (vrednost nad 128).

2.2.2 Delovanje strežnika

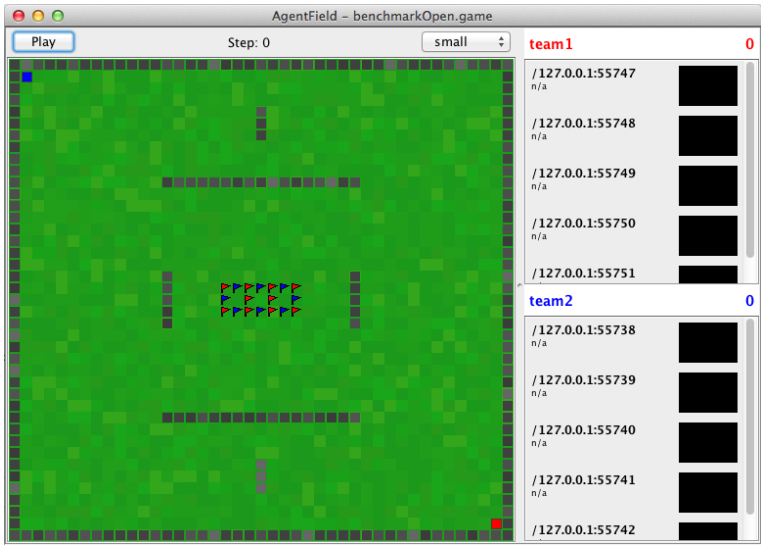
Strežnik zaženemo v javanskem stroju s podanim parametrom, ki predstavlja ime nastavitvene datoteke. Ko se ta naloži, se strežnik ustrezno inicializira, prikaže se grafični vmesnik, strežnik pa začne na vratih 5000 poslušati za dohodnimi povezavami. Ob prejeti povezavi se preveri, ali je pošiljatelj agent, kateri ekipi pripada in morebitno geslo ekipe. V kolikor se agent uspešno identificira, ga dodamo na seznam povezanih agentov. Nadaljnja komunikacija med strežnikom in odjemalcem je opisana v poglavju 2.4.

Slika 2.3 prikazuje strežnikov grafični vmesnik. Glavno zeleno polje na sredini predstavlja simulacijski zemljevid. Na desni strani so prikazane ekipe z igralci. Za vsakega agenta vidimo njegov naslov IP in vrata. S klikom na agenta se nam prikaže pot, ki jo je agent prehodil in polja, ki jih je videl. Imamo tudi možnost izločitve agenta iz igre oz. eliminacije instance izbranega agenta. Simulacijo pričnemo s klikom na gumb *Play*.

2.3 Odjemalec

V našem okolju je odjemalec vsak program, ki se poveže na strežnik ter se ustrezno identificira. Takemu odjemalcu pravimo agent.

Okolje *GridLand* nam priskrbi osnovni abstraktni razred agenta. Ta vsebuje glavno metodo, s katero agenta zaženemo, prav tako pa ta skrbi za vso



Slika 2.3: Grafični vmesnik strežnika.

komunikacijo s strežnikom in za življenjski cikel agenta.

Ker je razred abstrakten, ga moramo razširiti in implementirati vse abstraktne metode. Osnovni razred *Agent* nam ponuja naslednje metode, ki jih lahko uporabimo pri implementaciji lastnega agenta:

- *send(int id_naslovnika, byte[] sporocilo),*
- *send(int id_naslovnika, String sporocilo),*
- *move(Direction smer),*
- *scan(int zig),*
- *isAlive(),*
- *getId(),*
- *getSpeed(),*
- *getMaxMessageSize().*

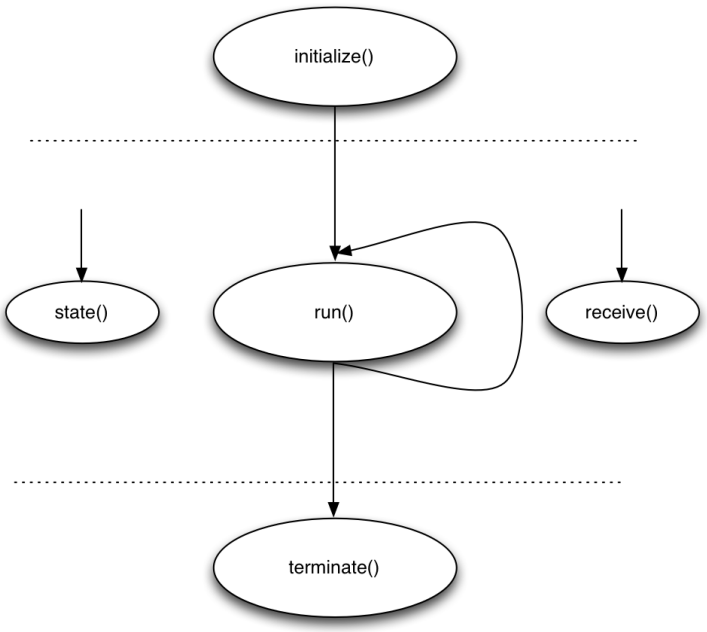
Abstraktne metode, ki jih moramo implementirati:

- *terminate(),*
- *initialize(),*
- *receive(int posiljatelj, byte[] sporocilo),*
- *state(int zig, Neighborhood okolica, Direction smer, bool imam_zastavo),*
- *run().*

V življenjskem ciklu agenta se najprej pokliče metoda *initialize*, s katero inicializiramo potrebne spremenljivke za pravilno delovanje agenta. Metoda se pokliče, ko strežnik sporoči, da je agent prišel iz svoje baze. Po inicializaciji se pokliče metoda *run*, kjer je implementirano samo delovanje agenta. Med delovanjem agenta lahko ta dobi sporočila o njegovi lokaciji in sporočila drugih agentov. Za prebiranje sporočil o agentovem stanju v simulacijskem okolju moramo implementirati metodo *state*. V tej metodi kot parametre dobimo podatke o agentovi okolici, v katero smer se premika in če agent nosi zastavico. Branje sporočil, ki jih pošljejo drugi agentje iz ekipe, nam omogoča metoda *receive*. Tu kot parametre dobimo identifikacijsko številko agenta pošiljatelja in samo sporočilo. Ti dve metodi se pokličeta v novi niti takoj, ko agent dobi sporočilo. Ob izločitvi agenta iz igre se pokliče metoda *terminate*, s katero lahko poskrbimo za sprostitev agentovih sredstev. Izločitev oz. smrt agenta nam strežnik sporoči preko posebnega klica. Opisan življenjski cikel agenta je ponazorjen s sliko 2.4.

2.4 Komunikacija

Komunikacija med strežnikom in agentom je zgrajena s pomočjo javanskih knjižnic *ServerSocket* in *Socket*. Strežniški del ustvari *ServerSocket* objekt, ki posluša na vratih 5000. *ServerSocket* je *java.net* razred, ki zagotavlja sistemsko neodvisno implementacijo strežniškega dela strežnik-odjemalec povezave (ang. server/client socket connection). Strežnik čaka na odjemalčev zahtevek za povezavo na strežniškem naslovu in vratih. Ko je povezava uspešno



Slika 2.4: Abstraktni življenjski cikel agenta.

vzpostavljena, pridobi strežnik nov objekt tipa *Socket* - vtičnico, ki vsebuje naslov in vrata odjemalca ter se bo uporabljal za vso nadaljnjo komunikacijo. Za pošiljanje in sprejemanje podatkov preko pridobljene vtičnice (objekt *Socket*) se uporabljata javanska razreda *DataOutputStream* in *DataInputStream*. S pomočjo teh razredov si lahko med strežnikom in odjemalcem pošiljamo podatke kateregakoli tipa. V okolju *GridLand* se podatki pošiljajo tako, da se s pomočjo javanske serializacije serializira celoten objekt sporočila. Vsi tipi sporočil razširijo razred *Message*, ki implementira razred *Serializable* - ta omogoči serializacijo objekta z metodo *writeObject*. Na strani sprejemnika objekt rekonstruiramo (deserializiramo) z metodo *readObject*, za kateri tip sporočila gre pa enostavno preverimo s pomočjo rezervirane besede *instanceOf*. Način pošiljanja podatkov smo za naše potrebe spremenili, kar je opisano v poglavju 2.5.2. Da spoznamo, kako se strežnik in odjemalec sporazumevata med seboj, si oglejmo kakšne tipe sporočil imamo na voljo.

- **Registriraj:** Sporočilo za registracijo pošlje odjemalec strežniku, ko se ta hoče priključiti v simulacijsko okolje kot agent določene ekipe. Sporočilo je sestavljeno iz imena ekipe in ekipnega gesla, če ga le-ta ima.

- **Potrditev:** Sporočilo tipa potrditev pošlje strežnik agentu v potrditev, da se je odjemalec uspešno priključil v okolje kot agent predstavljene ekipe.To sporočilo lahko pošlje tudi agent strežniku kot odgovor, da je prejel nastavitveno (inicializacijsko) sporočilo.

- **Inicializacija:** Strežnik uporabi to sporočilo ob pojavitvi agenta v simulacijskem okolju. S tem agentu sporoči, da je sedaj aktiven agent. Sporočilo vsebuje agentovo identifikacijsko številko, velikost največjega dovoljenega poslanega sporočila in hitrost simulacije.

- **Prekinitev:** Ob izločitvi agenta iz simulacije se odjemalcu pošlje to sporočilo.

- **Skeniranje:** Ta tip sporočila uporabi agent, ko želi izvedeti, kakšno je stanje sveta okoli njega. Sporočilo vsebuje identifikacijsko številko, ki jo uporabi strežnik pri odgovoru.

- **Premik:** Agent uporabi to sporočilo za sporočitev premika v simulacijskem okolju. V sporočilo vključi želeno smer premika.

- **Stanje:** Sporočilo stanje se pošlje kot odgovor na sporočili skeniranje in premik. Vsebina sporočila vsebuje posnetek agentove bližnje okolice, smer agentovega gibanja in podatek, ali agent nosi zastavico. Posnetek okolice predstavlja vsa polja v določenem radiju, ki je definiran na strežnikovi strani.

- **Pošiljanje:** Agent uporabi ta tip sporočila za pošiljanje sporočila drugemu agentu iz njegove ekipe. Sporočilo vsebuje identifikacijsko številko agenta prejemnika in ustrezno kodirano sporočilo.

- **Prejetje:** To sporočilo je identično tipu sporočila pošiljanje, le da je v njem identifikacijska številka agenta pošiljatelja skupaj z ustrezno kodiranim sporočilom.

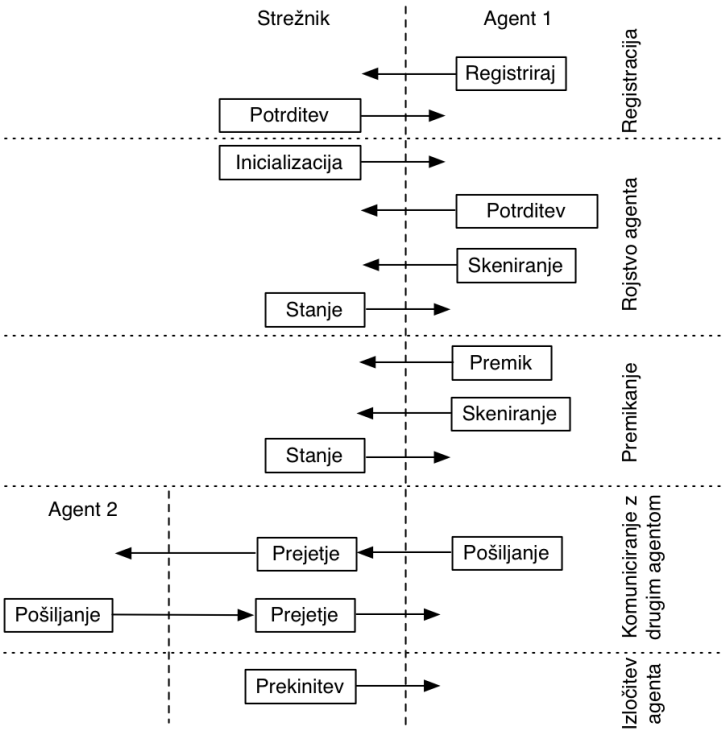
Vsa ta sporočila se uporabljajo skozi celoten življenjski cikel agenta. Oglejmo si sliko 2.5, ki prikazuje običajen potek sporočil med agentom in strežnikom. Na začetku agent opravi registracijo z ustreznim sporočilom in če je bila ta uspešna, dobi v odgovor potrditev. Agent sedaj čaka, da mu strežnik pošlje znak (inicializacija) za pričetek delovanja. Agent na to sporočilo odgovori s sporočilom za potrditev in pošlje zahtevek za skeniranje okolice. Strežnik odgovori z ustreznim sporočilom za stanje okolice. Sedaj navadno poteka komunikacija tako, da agent pošlje strežniku premik in zahtevek za skeniranje okolice, ta pa mu na slednjega odgovori s stanjem okolice. V primeru, da želi agent poslati sporočilo drugemu agentu iz svoje ekipe, pošlje strežniku sporočilo tipa *pošiljanje*. Strežnik to sporočilo posreduje naslovnemu agentu s sporočilom tipa *prejetje*. Strežnik lahko pošlje agentu sporočilo *prekinitev*, kar pomeni, da je agent izločen iz simulacije. Sporočilo se pošlje v primeru, da se je agent premaknil na nedovoljeno polje, oziroma je preko grafičnega vmesnika uporabnik zahteval prekinitev delovanja.

2.5 Nadgradnja simulacijskega okolja

Za potrebe diplomske naloge nam simulacijsko okolje *GridLand* ne omogoča vseh želenih funkcionalnosti, zato smo okolje nadgradili. Dodali smo nov način pobiranja zastavic, implementirali meritve določenih parametrov, odpravili nekatere nepravilnosti pri komunikaciji in nadgradili komunikacijo med strežnikom in agentom.

2.5.1 Meritveni način pobiranja zastavic

Simulacijsko okolje je namenjeno simuliranju igre pobiranja zastavic. Pri tej igri velja pravilo, da lahko vsak agent pobere le zastavico svoje ekipe.



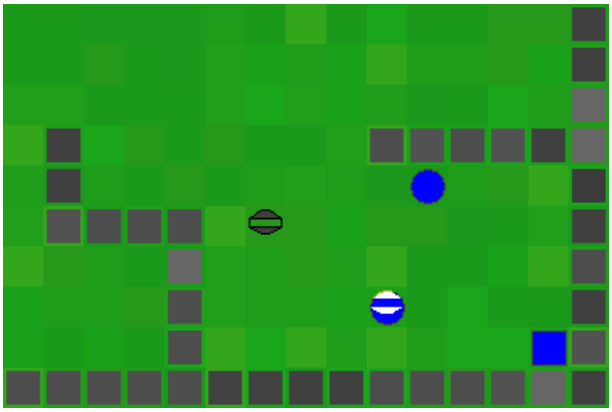
Slika 2.5: Potek komunikacije med agentom, strežnikom in drugim agentom.

V naši nalogi želimo simulirati agente v okolju z omejeno količino virov. Viri v tem primeru predstavljajo hrano, za katero velja, da jo lahko pobere agent katerekoli ekipe in jo nese v svojo bazo. Iz tega razloga smo morali implementirati dodatni način simulacije, ki nam bo omogočal to funkcionalnost. Nov način imenujemo meritveni način (ang. benchmark). Nastavimo ga na strežniku s parametrom za vrsto zastavic na sledeč način:

$$gameplay.flags = benchmark$$

Sedaj lahko namesto o zastavicah govorimo o hrani, saj strežnik z izločitvijo ne kaznuje agenta, ki je pobral zastavico nasprotne ekipe. Iz tega razloga na grafičnem vmesniku strežnika prikazujemo podobo hrane namesto zastavice, kar vidimo na sliki 2.6. V tem načinu je tudi izklopljena simulacija zakasnitve sporočil (parameter hitrost sporočanja), ki si jih agenti pošiljajo

med sabo. Za potrebe analize simulacij v dnevniško (log) datoteko beležimo število pobrane hrane za vsako ekipo, koliko časa je simulacija trajala in število ustvarjenih agentov.



Slika 2.6: Izgled grafičnega vmesnika v meritvenem načinu.

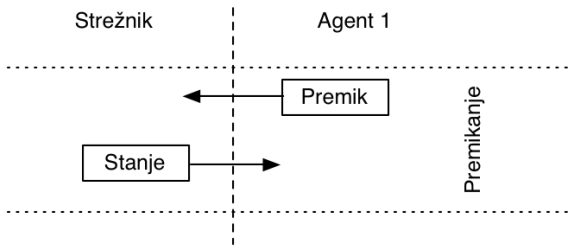
2.5.2 Nadgradnja komunikacije

Komunikacija med agentom in strežnikom je realizirana na osnovi povezave TCP/IP, preko katere se v okolju *GridLand* pošiljajo podatki v obliki serializiranega objekta.

Odločili smo se, da v okviru diplomske naloge spremenimo način pošiljanja podatkov. Razlog za to je večja neodvisnost agentov glede jezika in platforme, na kateri so implementirani in zmanjšanje velikosti sporočil. Javanska serializacija deluje le znotraj javanskega virtualnega stroja, zato je implementacija agenta v drugem programskem jeziku težja. Za ta namen bi sicer lahko uporabili serializacijo XML, ki je platformsko neodvisna, vendar smo se zaradi zmanjšanja nepotrebnih dodatnih informacij odločili za lastno kodiranje.

Kodiranje je zelo poenostavljeno, saj pošiljamo podatke v formatu CSV, kjer so vrednosti ločene z vejico. Implementacija je podobna prvotni, tako da vsak tip sporočila razširja razred *NewMessage* in za vsak parameter sporočila obstaja metoda, ki vrne njegovo vrednost. Kodirano sporočilo v obliki niza

znakov, ki je primerno za pošiljanje, dobimo s klicem metode *encodeMessage* razreda *NewMessage*. Na prejemnikovi strani sprejmemo sporočilo kot niz znakov iz katerega kreiramo objekt tipa *NewMessage*, ki nam lahko določi tip sporočila. Sedaj dostopamo do željenih vrednosti sporočila s pomočjo dostopkovnih metod (ang. getter methods).



Slika 2.7: Spremenjen potek komunikacije med agentom in strežnikom.

Optimizirali smo tudi klic *premik*, na katerega nam sedaj agent odgovori z sporočilom *stanje*. Spremenjen potek komunikacije za izvedbo premikanja nam prikazuje slika 2.7. Originalna implementacija je okorna za uporabo, saj moramo vedno po klicu za premik izvesti klic za skenikanje okolice, ker drugače ne vemo, kakšno je novo stanje okolja. Takšna sprememba protokola sporazumevanja nam prinese še izboljšavo z zmogljivostnega vidika, saj nam sedaj ni potrebno izvajati dodatnega klica.

Z vsemi opravljenimi nadgradnjami smo si zagotovili simulacijsko okolje, ki ga potrebujemo za izvedbo zadane naloge. Z vpeljavo novega načina delovanja smo v okolju definirali pravila, ki si jih želimo v našem sistemu. S spremembo načina komunikacije pa smo izboljšali zmogljivostni vidik delovanja in odprli nadaljnjo možnost za implementacijo agenta v kateremu drugemu programskemu jeziku poleg Jave.

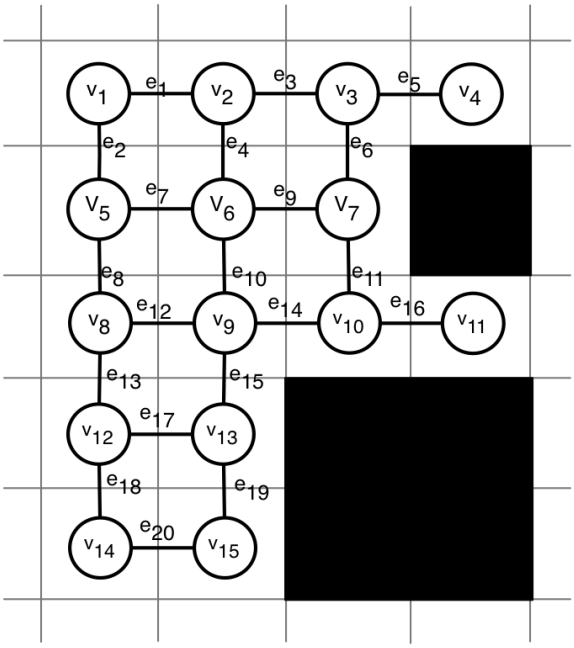
Poglavje 3

Pregled preiskovalinh algoritmov

Osnovna naloga agenta je raziskovanje okolja in iskanje hrane. Pomembno je, da je izbira cilja premikanja in načrtovanje premikanja optimalno, tako da za premik na želeno mesto porabimo najmanjše možno število korakov. V tem poglavju bomo preučili nekatere preiskovalne algoritme in določili, katere bi uporabili pri našem agentu.

Zemljevid, po katerem se agent premika, lahko predstavimo kot neusmerjen graf *G(V, E)*, kjer je *V* neprazna množica točk oziroma vozlišč, *E* pa množica povezav. V našem primeru je vozlišče posamezno polje na zemljevidu, kar pomeni, da množica *V* predstavlja raziskana polja na zemljevidu. Na sliki 3.1 lahko vidimo predstavitev dela zemljevida z neusmerjenim grafom. Povezave med vozlišči predstavljajo možne premike na sosednja polja. Vsako vozlišče ima lahko največ štiri povezave, saj ima lahko polje največ štiri sosednja polja, iz katerih je možen premik na to polje. V našem primeru množici *V* in *E* nista ves čas enaki, saj agent preiskuje prostor in v graf dodaja nova vozlišča in povezave. Cena vseh povezav je konstantna, saj obstajajo le povezave med sosednjimi polji.

Problem iskanja poti po našem zemljevidu predstavimo kot problem iskanja najkrajše poti v grafu. Za začetno vozlišče vzamemo trenutno lokacijo



Slika 3.1: Predstavitev neusmerjenega grafa na podlagi zemljevida.

agenta, za končno pa vozlišče, kjer se nahaja hrana. V primeru, da nam lokacija hrane ni znana, iščemo pot do najbližjega nepreiskanega polja. Iskanje poti v grafu lahko realiziramo z iskalnimi algoritmi, ki jih bomo opisali v tem poglavju.

3.1 Dijkstrov algoritem

Dijkstrov algoritem [2] uporabljamo za iskanje najkrajše poti z enim začetim vozliščem na uteženem usmerjenem grafu, pri katerem so vse uteži povezav pozitivne. Za podano začetno izhodišče v grafu algoritem najde najcenejšo pot iz tega vozlišča do vseh ostalih vozlišč. Uporabimo ga lahko tudi za iskanje najkrajše poti med dvema vozliščema, tako da algoritem ustavimo, ko le-ta najde najkrajšo pot do ciljnega vozlišča.

Algoritem skozi svoje delovanje vzdržuje množico vozlišč S, za katere

imamo izračunane dolžine najkrajših poti. Na začetku bo vsem vozliščem določil neke začetne dolžine poti, ki jih bo skozi delovanje poskušal izboljšati. Oglejmo si delovanje algoritma korak za korakom.

1. Vsakemu vozlišču določimo začetno vrednost (trenutne) razdalje do izhodišča. Za izhodiščno vozlišče nastavimo to razdaljo na nič, medtem ko za ostala vozlišča privzamemo neskončno (oz. jeziku primerno) vrednost.

2. Za trenutno vozlišče vzamemo izhodišče, ustvarimo množico neobiskanih vozlišč, ki vsebuje vsa vozlišča razen izhodiščnega.

3. Za vsako sosednjo (povezano) vozlišče trenutnega vozlišča izračunamo njegovo trenutno razdaljo do izhodišča. V našem primeru so razdalje med vozlišči enake 1, zato je njegova trenutna razdalja za eno vrednost večja kot vrednost v trenutnem vozlišču. Če je izračunana razdalja manjša, kot je vrednost razdalje v tem sosednjem vozlišču, jo prepišemo, sicer pa zavržemo.

4. Ko smo obdelali vsa sosednja vozlišča, označimo trenutno vozlišče kot obiskano in ga odstranimo iz množice neobiskanih vozlišč.

5. Če smo kot obiskano vozlišče označili našo končno vozlišče, ustavimo algoritem, saj smo našli pot iz izhodiščnega vozlišča do iskanega. Algoritem ustavimo tudi v primeru, da je najmanjša vrednost trenutnih razdalj neobiskanih vozlišč enaka neskončnosti. V tem primeru algoritem ni našel rešitve.

6. Kot trenutno vozlišče izberemo vozlišče iz množice ne obiskanih vozlišč, ki ima najmanjšo trenutno vrednost razdalje do izhodišča. Vrnemo se na 3. korak.

Opisani koraki nas pripeljejo iz začetnega do končnega vozlišča, dolžina poti pa predstavlja vrednost trenutne razdalje zadnjega (trenutnega) vozlišča. Želimo si, da bi lahko določili vozlišča, preko katerih poteka pot od začetnega

do končnega vozlišča. To pot lahko določimo, če si v koraku 3 za vozlišče, čigar trenutno dolžino prepišemo, zapomnimo, kdo je njegov predhodnik (trenutno vozlišče).

Algoritem 1 Dijkstrov algoritem

```
1:  function DIJKSTRA(Graf, zacetek, cilj)
2:      razdalja={}, prejsnji={}
3:      neobiskana={Vsa vozlišča v Graf}
4:      for vsako voz v Graf do
5:          razdalja[voz] = neskončno, prejsnji[voz] = nedefiniran
6:      end for
7:      razdalja[zacetek] = 0
8:      while neobiskana ni prazen do
9:          min = vozlišče v neobiskana z najmanjšo vrednostjo v razdalja
10:         odstrani min iz neobiskana
11:         if razdalja[min] == neskončno then return
12:         else if min == cilj then
13:             return izpiši_pot(prejsnji,min)
14:         end if
15:         for vsakega soseda sosed vozlišča min do
16:             r = razdalja[min] + razdalja_med(sosed,min)
17:             if r < razdalja[sosed] then
18:                 razdalja[sosed] = r
19:                 prejsnji[sosed] = min
20:             end if
21:         end for
22:     end while
23: end function
```

Funkcija *izpiši_pot()*, v vrstici 13 prikazana na algoritmu 1, nam na koncu izvajanja algoritma Dijkstra iz množice predhodnikov izpiše pot od začetnega do končnega vozlišča.

Algoritem 2 Izpis poti iz podanih predhodnikov

```
1: function IZPISI_POT(Prejsnji, koncno)
2:     pot={koncno}
3:     trenutno = Prejsnji[koncno]
4:     while trenutno ni nedefiniran do
5:         vstavi trenutno na začetek pot
6:         trenutno = Prejsnji[trenutno]
7:     end while
8:     return pot
9: end function
```

3.2 Algoritem A*

A* [3] je zelo razširjen algoritem za iskanje poti in preiskovanje grafov. Je razširitev Dijkstrovega algoritma in ga zaradi uporabe hevristike prekaša v hitrosti.

Algoritem izbere najprimernejše vozlišče za nadaljevanje iskanja na podlagi trenutne oddaljenosti od izhodišča in hevristične ocene, ki predstavlja predvideno razdaljo od vozlišča do cilja. Funkcija, s katero algoritem ovrednoti posamezno vozlišče, je definirana kot

$$f(x) = g(x) + h(x) \tag{3.1}$$

kjer je *g(x)* funkcija, ki izračuna ceno poti od začetnega vozlišča do trenutnega in *h(x)* hevristična funkcija, ki izračuna ceno poti, ki je potrebna do ciljnega vozlišča. Zaradi uporabe hevristike je iskanje bolj usmerjeno, tako da za pridobitev rešitve preiščemo manj vozlišč. Pomembno je, da je izbrana hevristika optimistična oziroma sprejemljiva (ang. admissible), ker nam le tako algoritem zagotavlja optimalno - najkrajšo rešitev. Optimistična hevristika je tista, ki optimistično oceni razdaljo do končnega cilja - torej ne precenjuje cene poti, kar pomeni da velja

$$\forall n, h(n) \leq C(n) \tag{3.2}$$

kjer C(n) predstavlja dejansko ceno, ki je potrebna za premik od trenutnega vozlišča n do ciljnega vozlišča.

Ena izmed enostavnejših, a vendar optimističnih hevristik za določitev razdalje, ki smo jo uporabili tudi pri implementaciji algoritma, je manhattanska razdalja. Manhattanska razdalja med dvema točkama T_1 in T_2 je definirana kot

$$d(T_1, T_2) = |x_1 - x_2| + |y_1 - y_2| \tag{3.3}$$

3.3 Real-Time A*

Cilj algoritma Real-Time A* [8, 7] je zmanjšati (izboljšati) čas izvajana algoritma A*. Algoritem RTA* uporablja pristop k reševanju problema, ki ga srečamo pri igrah z dvema igralcema. Ta pristop temelji na omejenem preiskovalnem prostoru in na zavezi k izvedbi posamezne akcije (enega premika) v konstantnem času. Odločitev o naslednji akciji (premiku) moramo narediti brez celotnega načrta (poti od začetnega do ciljnega vozlišča), pri tem pa doseči ciljno vozlišče. Algoritem je sestavljen iz dveh delov. V prvem delu izvedemo individualno odločitev za premik, v drugem pa izvedemo serijo teh odločitev, tako da dosežemo cilj.

Za izvedbo prvega dela (individualna odločitev za premik) uporabimo iskanje minimin z alfa rezi (ang. Minimin Search with Alpha-Pruning). Iskanje minimin je podobno kot iskanje minimaks, z razliko, da tu vrednosti hevristično ocenjenih vozlišč le minimiziramo in nikoli ne maksimiramo. Alfa rezi se uporabijo za optimizacijo samega minimin iskanja.

Zgolj s ponavljanjem minimin iskanja za vsak premik ignoriramo informacijo, pridobljeno v prejšnjem iskanju, zato izvajanje sčasoma postane ciklično. Ker je premik opravljen na podlagi omejenih informacij, je pogosto najboljši naslednji premik, premik nazaj. Vračanje nazaj bi se moralo zgoditi samo v primeru, da pričakovana cena v primeru nadaljevanja poti presega ceno vračanja na prejšnje mesto in pričakovane cene iz tega mesta do cilja. RTA* je učinkovit algoritem za implementacijo take osnovne strategije. Za oceno

Algoritem 3 Algoritem A*

```
1: function A_Star(Graf, zacetek, cilj)
2:     Množici: zaprta = {}, odprta = {zacetek}
3:     Slovar: prisel_iz = {}
4:     g_ocena[zacetek]=0
5:     h_ocena[zacetek]= hevristična_ocena(zacetek, cilj)
6:     f_ocena[zacetek] = g_ocena[zacetek] + h_ocena[zacetek]
7:     while odprta ni prazna do
8:         trenutno = vozlišče v odprta z najmanjšo f_ocena[]
9:         if trenutno == cilj then
10:            return izpiši_pot(prisel_iz, prisel_iz[cilj])
11:        end if
12:        odstrani trenutno iz odprta
13:        dodaj trenutno v zaprta
14:        for vsakega soseda sosed vozlišča trenutno do
15:            if sosed ∈ zaprta then
16:                continue
17:            end if
18:            pogojna_g_ocena = g_ocena[trenutno] + razdalja_med(trenutno, sosed)
19:            if sosed ∉ odprta then
20:                dodaj sosed v odprta
21:                h_ocena[sosed] = hevristična_ocena(sosed, cilj)
22:                pogojna_g_je_boljsa = true
23:            else if pogojna_g_ocena < g_ocena[sosed] then
24:                pogojna_g_je_boljsa = true
25:            else
26:                pogojna_g_je_boljsa = false
27:            end if
28:            if pogojna_g_je_boljsa == true then
29:                prisel_iz[sosed] = trenutno
30:                g_ocena[sosed] = pogojna_g_ocena
31:                f_ocena[sosed] = g_ocena[sosed] + h_ocena[sosed]
32:            end if
33:        end for
34:    end while
35: end function
```

vozlišča n vzemimo isto formulo (3.1), kot pri algoritmu A*, le da tukaj *g(n)* predstavlja razdaljo vozlišča n od trenutnega vozlišča in ne od izhodišča. Ključna razlika med A* in RTA* je v tem, da je pri RTA* cena vsakega vozlišča izmerjena relativno glede na trenutno vozlišče v katerem se nahajamo, zato začetno volišče ni pomembno.

Algoritem shranjuje seznam dejansko obiskanih vozlišč skupaj s h vrednostjo v zgoščeni tabeli (ang. hash table). Ob vsaki ponovitvi algoritma razširimo trenutno vozlišče, tako da generiramo njegove sosede in hevristično funkcijo. Hevristična funkcija temelji na vnaprejšnji preiskavi (ang. lookahead search) in jo uporabimo za vsako sosednje vozlišče, ki ni v zgoščeni tabeli obiskanih vozlišč. Za vozlišča, ki so v tabeli, uporabimo h vrednost iz zgoščene tabele. Upoštevajoč formulo (3.1) k vrednosti h dodamo razdaljo od trenutnega do sosednjega vozlišča. Vozlišče z najmanjšo f vrednostjo je izbrano kot naslednje vozlišče na katerega se premaknemo. Za konec še shranimo prejšnje vozlišče v zgoščeno tabelo, skupaj z drugo najboljšo (najmanjšo) f vrednostjo. Druga najmanjša f vrednost je najboljša od vseh alternativ, ki niso bile izbrane in predstavlja pričakovano h vrednost rešitve problema v primeru, da se vrnemo v to (sedaj prejšnje) stanje iz trenutnega vozlišča. Implementacija algoritma zahteva samo seznam obiskanih vozlišč. Velikost tega seznama je linearna - notri so le vozlišča, ki so bila dejansko obiskana, saj vnaprejšnje preiskovanje shrani le vrednost izhodiščnega vozlišča. Čas izvajanja je prav tako linearen glede na število opravljenih premikov. Res je, da vnaprejšnje preiskovanje zahteva eksponentni čas pri iskanju v globino, vendar je le-ta omejena z neko konstantno vrednostjo. Algoritem nam ne zagotavlja pridobitve optimalne rešitve, se ji pa lahko, z ustrezno določeno mejo vnaprejšnjega preiskovanja, zelo približamo. Povzetek algoritma je sestavljen iz štirih točk:

1. Nastavimo spremenljivko N kot začetno vozlišče

2. Generiramo vse naslednike vozlišča N. Če je kateri naslednik ciljno vozlišče, se premaknemo v to vozlišče in končamo.

3. Z uporabo omejenega iskanja v širino, izračunamo hevristično oceno za vsakega naslednika S.

4. Privzamemo, da ima naslednik S_1 najboljšo (najobetavnejšo) hevristiko. V_2 je vrednost hevristike drugega najboljšega naslednika. Premaknemo se v vozlišče S_1. V zgoščeno tabelo shranimo vozlišče N s ključem V_2. V primeru, da vozlišče N ponovno generiramo v koraku 2, se namesto koraka 3 uporabi vrednost hevristike zapisana v tabeli.

Na ideji algoritma je osnovan tudi algoritem *Learning Real Time A**. Njegovo delovanje je identično RTA*, le da v zgoščevalno tabelo namesto druge najboljše hevristike shranjujemo najboljšo. Ko naredimo premik, shranimo vrednosti hevurističnih ocen in jih uporabimo za izhodišče pri izbiri naslednjega premika. Čeprav je LRTA* manj učinkovit kot RTA*, nam v primeru, ko imamo na začetku optimistično hevristiko, hevrističe ocene konvergirajo k pravim vrednostim, kar nas posledično privede k optimalni rešitvi.

3.4 Inkrementalno hevristično iskanje

Inkrementalno hevristično iskanje [5] je preiskovalna tehnika, ki združuje pristope inkrementalnega in hevurističnega iskanja z namenom pohitritve iskanja pri domenah (prostorih), katere niso popolnoma poznane oziroma se dinamično spreminjajo. Algoritmi inkrementalnega iskanja pri svojem iskanju za dosego cilja ponovno uporabijo informacije iz prejšnjih iskanj. Cilj je poiskati rešitev problema s pomočjo zaporedja podobnih iskalnih problemov, kar lahko naredimo hitreje, kot če bi vsak takšen podproblem reševali posamično od začetka. Za razliko od nekaterih drugih preiskovalnih tehnik, ki pohitrijo iskanje, ta tehnika zagotavlja optimalnost najdene rešitve, če ta obstaja. Nekateri najbolj znani inkrementalni algoritmi so D*, Focused D*, D* Lite, SWSF-FP in LPA* .

3.4.1 Lifelong Planning A*

Eden izmed predstavnikov inkrementalno hevrističnih algoritmov je algoritem Lifelong Planning A* (LPA*) [6], ki združuje ideji algoritmov A* in DynamicSWSF-FP. Algoritem za iskanje poti uporablja hevristiko in če je ta optimistična, vedno najde optimalno pot. Pri prvem preiskovanju grafa je hitrost enaka kot pri A*, vsa nadaljnja preiskovanja pa so precej hitrejša. Takšna pohitritev je mogoča zato, ker algoritem pri drugem in vseh nadaljnjih preiskovanjih grafa uporabi tiste dele v grafu, ki se ne spremenijo.

Glavna funkcija algoritma LPA* je funkcija *Main()*, ki najprej pokliče funkcijo *Initialize()*, da vzpostavi začetno stanje za reševanje problema. Funkcija *Initialize()* nastavi začetno vrednost g vsem vozliščem na neskončno in vrednost *rhs* glede na enačbo

$$rhs(s) = \begin{cases} 0 & \text{ce } s = s_{start} \\ min_{s' \in pred(s)}(g(s') + c(s', s)) & \text{sicer.} \end{cases} \tag{3.4}$$

Iz tega sledi, da je začetno vozlišče (s_{start}) edino lokalno nekonsistentno vozlišče, zato ga dodamo v prioritetno vrsto U. Prioritetna vrsta U vsebuje lokalno nekonsistentna vozlišča, katerih cena mora biti posodobljena, da bodo postala lokalno konsistentna. Vozlišče, ki je lokalno konsistentno, ima enaki vrednosti *g(s)* in *rhs(s)*. Ključ *k(s)* vozlišča s predstavlja prioriteto v prioritetni vrsti U in je definiran kot vektor z dvema komponentama

$$k(s) =< k_1(s); k_2(s) >, \tag{3.5}$$

kjer je k_1*(s)* = min*(g(s), rhs(s))* + *h(s)* in k_2*(s)* = min*(g(s), rhs(s))*.

Funkcija *Main()* nato pokliče funkcijo *ComputeShortestPath()*, čigar naloga je poiskati najkrajšo pot od začetnega do končnega vozlišča. Tu izračunamo g vrednost vsem lokalno nekonsistentnim vozliščem v nepadajočem vrstnem redu glede na njihov ključ. Funkcija *Initialize()* je poskrbela, da bo prva iteracija funkcije razširila ista vozlišča, kot bi jih algoritem A*.

Lokalno nekonsistentno vozlišče s se imenuje lokalno nad-konsistentno, če velja *g(s)* > *rhs(s)*. Med razvitjem vozlišča se nastavi g vrednost vozlišča

Algoritem 4 Funkcija Main algoritma LPA*

```
1: function MAIN
2:     Initialize()
3:     while True do
4:         ComputeShortestPath()
5:         Počakaj na spremembe cen na povezavah
6:         for vse povezave (u, v), katerim se je spremenila cena do
7:             Posodobi ceno povezave c(u, v)
8:             UpdateVertex(v)
9:         end for
10:    end while
11: end function
```

Algoritem 5 Funkcija Initialize algoritma LPA*

```
1: function INITIALIZE
2:     U = {}
3:     for vse s ∈ S do
4:         rhs(s) = g(s) = ∞
5:     end for
6:     rhs(s_start) = 0
7:     v U vstavi < s_start, izracunaj_kljuc(s_start)>
8: end function
```

Algoritem 6 Funkcija ComputeShortestPath algoritma LPA*

```
1: function ComputeShortestPath
2:     while U.TopKey() <izracunaj_kljuc(s_goal) ali rhs(s_goal) ≠g(s_goal) do
3:         u = U.Pop()
4:         if g(u) > rhs(u) then
5:             g(u) = rhs(u)
6:             for vsak s ∈ Succ(u) do
7:                 UpdateVertex(s)
8:             end for
9:         else
10:            g(u) = ∞
11:            for vsak s ∈ Succ(u) ∪ {u} do
12:                UpdateVertex(s)
13:            end for
14:        end if
15:    end while
16: end function
```

Algoritem 7 Funkcija UpdateVertex algoritma LPA*2

```
1: function UpdateVertex(u)
2:     if u ≠ s_start then
3:         rhs(u) = min_{s'∈Pred_(u)}(g(s') + c(s', u))
4:     end if
5:     if u ∈ U then
6:         odstrani u iz U
7:     end if
8:     if g(u) ≠ rhs(u) then
9:         v U vstavi < u, izracunaj_kljuc(u)>
10:    end if
11: end function
```

na vrednost njegove *rhs* vrednosti, kar pomeni, da je vrednost g enaka vsoti cene poti od začetnega vozlišča s_{start} do tega vozlišča in s tem naredi vozlišče lokalno konsistentno. Njegova vrednost g se nato ne spreminja več, razen ob ponovnem klicu funkcije *ComputeShortestPath()*.

Lokalno nekonsistentno vozlišče s se imenuje lokalno pod-konsistentno, če velja $g(s) < rhs(s)$. Ko razvijemo tako vozlišče, nastavimo njegovo vrednost g na neskončno. S tem naredimo vozlišče bodisi lokalno konsistentno, bodisi nad-konsistentno.

Če je bilo razvito vozlišče lokalno nad-konsistentno, potem lahko sprememba njegove vrednosti g vpliva na lokalno konsistentnost njegovih naslednikov. Podobno velja tudi za vozlišče, ki je lokalno pod-konsistentno. Da se ohranja skladnost vrednosti *rhs*, prioritetne vrste in funkcije $k(s)$, funkcija *ComputeShortestPath()* posodobi vozliščem njihove vrednosti *rhs*, preveri njihovo lokalno konsistentnost in jih po potrebi doda ali odstrani iz prioritetne vrste.

Algoritem LPA* razvija vozlišča, dokler ni vozlišče s_{goal} lokalno konsistentno in ključ $k(s)$ naslednjega vozlišča, ki bi moralo biti razvito, ni manjši od ključa $k(s_{goal})$. V primeru, da po iskanju velja $k(s_{goal}) = \infty$, potem pot pot od začetnega do končnega vozlišča ne obstaja. V nasprotnem primeru algoritem najde najkrajšo pot med s_{start} in s_{goal} tako, da se vedno premakne od trenutnega vozlišča s, čigar začetek je v s_{goal}, do njegovega predhodnika s', ki minimizira vrednost $g(s') + c(s', s)$, dokler ni doseženo začetno vozlišče s_{start}.

Ob zaključku funkcije *ComputeShortestPath()* glavna funkcija *Main()* čaka na spremembe cen na povezavah. Če se cene spremenijo, se za vsako vozlišče, na katerega bi ta sprememba lahko vplivala, pokliče funkcija *UpdateVertex()*. Ta za vozlišče s posodobi vrednost $rhs(s)$ in ključ $k(s)$, s čimer ohrani skladnost vrednosti *rhs* in prioritetne vrste U. Zaradi sprememb cen je potrebno ponovno izračunati najkrajšo pot s klicem funkcije *ComputeShortestPath()*.

3.5 Izbira cilja iskanja

Naloga našega agenta je raziskati neznana območja zemljevida in pri tem poiskati hrano. Pot, po kateri se agent premika, je pomembna tako z vidika raziskanosti prostora, kot optimalnosti premikanja. Odgovoriti moramo na vprašanje, katero polje naj agent izbere za cilj iskanja, da bo čim hitreje preiskal prostor. Cilj je raziskati čim večjo površino prostora v čim krajšem času, saj se z večjo raziskanostjo prostora viša verjetnost najdbe hrane. Idealno bi bilo poiskati hrano tako, da je pri tem preiskana površina prostora minimalna, vendar ker agent vnaprej ne pozna lokacije hrane, ta način ni možen. Agent ima omejen pogled na okolico, saj ob vsakem premiku zazna le prostor v radiju n polj (privzeto je ta vrednost nastavljena na 5 polj). Ob tako omejeni predstavi celotnega sveta je težavno določiti pravi naslednji korak.

Odločili smo se za enostaven pristop, pri katerem agent za cilj naslednjega premika določi neraziskano polje, ki mu je najbližje. V primeru, da je takih polj več, agent naključno izbere eno izmed najbližjih. Nato z enim izmed iskalnih algoritmov poiščemo pot do izbranega polja. Če agent med preiskovanjem prostora najde hrano, bodisi jo zazna v svoji okolici ali pa pridobi njeno lokacijo s sporazumevanjem z drugim agentom, si za cilj iskanja izbere polje na katerem se le-ta nahaja. Ko agent pobere hrano (premik na polje, kjer se nahaja hrana), pa si za cilj iskanja izbere izhodiščno točko, ki je njegova baza.

Opisani način lahko izboljšamo tako, da poskusimo maksimirati ceno premika, v kolikor pa to ne bi bilo možno, bi uporabili zgornji pristop. Ceno premika definiramo kot število novih polj, ki jih pri posameznem premiku odkrijemo.

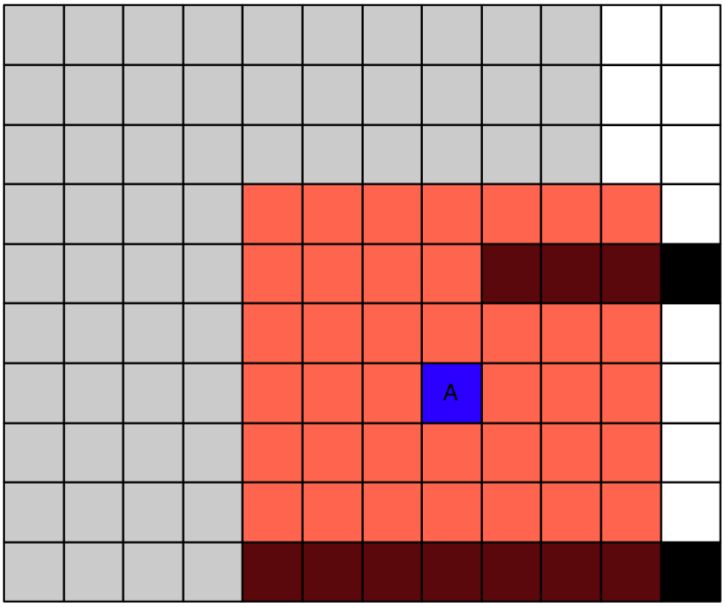
Oglejmo si pseudo kodo algoritma 8, ki določi premik glede na število na novo raziskanih polj. Na začetku v tretji vrstici s pomožno metodo *GenerirajMoznePremike()* generiramo vse možne premike iz trenutne pozicije. Premik ni možen, če bi z njim zavzeli mesto, kjer se nahaja zid, drug agent ali pa nasprotnikova baza, saj bi bila s tem trenutna instanca agenta izločena

iz simulacije. Sedaj z zanko preverimo, kako se obnese vsak možni premik. S pomožno metodo *PridobiLokacijeRobnihPolj()*, ki jo uporabimo v 6. vrstici, pridobimo robna polja za podan zemljevid, premik in radij. Robna polja so robna polja kvadrata s stranico polovice dolžine radija. Podan premik določi želeno stranico kvadrata. Sprva za vsa dobljena polja preverimo, ali vsebujejo zid. V primeru, da so vsa polja zidovi, se za premik v trenutno smer ne odločimo, saj ta ne bi bil optimalen. Če vsa dobljena polja niso zidovi, ponovno uporabimo metodo *PridobiLokacijeRobnihPolj()*, ki jo tokrat pokličemo z radijem povečanim za ena. Tako nam bo metoda vrnila lokacije vseh polj, ki bi jih s premikom v podani smeri preiskali. Sedaj se z zanko v 16. vrstici sprehodimo čez vsa polja in preverimo, ali so vsebovana v našem lokalnem zemljevidu. Število nevsebovanih polj si zapomnimo. V spremenljivki *max_nepreiskanih* si shranjujemo največje število nepreiskanih polj, v spremenljivki *premik* pa premike, s katerimi lahko raziščemo tolikšno število polj. Ko za vse možne premike izračunamo število novo preiskanih polj, izberemo premik, za katerega smo določili, da odkrije največ polj. Če je takih premikov več, naključno izberemo enega izmed njih. V primeru, da takega premika ni, algoritem ne vrne nobenega premika.

Za lažjo predstavo si oglejmo opisano delovanje še na sliki 3.2. Lokacija agenta je označena z modro barvo in črko A. Rdeča barva predstavlja polja, ki so v radiju zaznavanja ($n = 3$). Bela polja so preiskana, siva nepreiskana, črna oziroma temno rdeča pa predstavljajo zid. Agent se odloči za naslednji premik tako, da izračuna, koliko nepreiskanih polj bo pridobil za vsak možen premik in izbere tistega, pri katerem je to število največje. Če so na koncu katere smeri zaznavanja zaznani sami zidovi, premik v tej smeri preskočimo. Za preskok se odločimo zato, ker premik v tej smeri ni optimalen - lahko, da smo prišli do roba zemljevida okolja oziroma odseka zemljevida, ki je na tem mestu blokiran. Za primer na sliki 3.2 premik spodaj spustimo, premik levo ima ceno 7, premik gor 6 in premik desno 0. Ker ima premik v levo največjo ceno, ga izberemo. V primeru, da bi imelo več premikov isto ceno (različno od 0), naključno izberemo enega izmed njih. Če je cena za vsa polja

Algoritem 8 Algoritem za iskanje optimalnega premika

```
1: function IzracunajOptimalniPremik(zemljevid, radij_preiskovanja)
2:     premik = {}, max_nepreiskanih = 0
3:     mozni_premiki = GenerirajMoznePremike()
4:     for vsak premik p iz mozni_premiki do
5:         st_zidov = 0, st_nepreiskanih = -1
6:         lokacije = PridobiLokacijeRobnihPolj(p, zemljevid, radij_preiskovanja)
7:         for vsaka lokacija l v lokacije do
8:             if l ∈ zemljevid in zemljevid[l] == zid then
9:                 st_zidov = st_zidov + 1
10:            end if
11:        end for
12:        if st_zidov == radij_preiskovanja then
13:            continue
14:        end if
15:        lokacije = PridobiLokacijeRobnihPolj(p, zemljevid, radij_preiskovanja+
   1)
16:        for vsaka lokacija l v lokacije do
17:            if l ∉ zemljevid then
18:                st_nepreiskanih = st_nepreiskanih + 1
19:            end if
20:        end for
21:        if st_nepreiskanih > max_nepreiskanih then
22:            max_nepreiskanih = st_nepreiskanih
23:            premik = {p}
24:        else if st_nepreiskanih == max_nepreiskanih then
25:            premik.dodaj(p)
26:        end if
27:    end for
28:    if premik ≠ {} then
29:        return NakljucnoIzberiEnElement(premik)
30:    else
31:        return {}
32:    end if
33: end function
```



Slika 3.2: Agentov trenutni pogled na okolje.

0, poiščemo naslednji premik z izbranim iskalnim algoritmom do najbližjega neraziskanega polja.

3.6 Izbira preiskovalnega algoritma

Do sedaj smo si ogledali nekatere preiskovalne algoritme ter način, kako določiti ciljno polje pri iskanju. Pri konfiguraciji strežnika, kjer je radij preiskovanja *n*, bo med preiskovanjem zemljevida v veliki večini primerov izbrano ciljno polje v oddaljenosti *n+1*. V tem primeru bo dolžina poti reda velikosti *n*. Pri radiju preiskovanja 5 polj je ta dolžina relativno kratka, z večanjem radija pa se dolžina poti povečuje. Pri preiskovanju prostora, ko je le-ta že v veliki meri preiskan in pri vračanju agenta nazaj v bazo, bo dolžina poti veliko daljša, še posebej, če je zemljevid okolja kompleksen. Potrebno je, da je iskalni algoritem čim boljši ter tako najde optimalno pot v čim bolj kratkem času. Z dobrim algoritmom je naš agent veliko bolj splošen v smislu, da

lahko deluje v različnih okoljih.

Pri izbiri iskalnega algoritma smo se osredotočili na optimalnost in enostavnost algoritma. Vsi pregledani algoritmi razen RTA* najdejo optimalno rešitev. Najenostavnejši algoritem je zagotovo Dijkstra, A* pa je eden izmed najbolj razširjenih in široko uporabljenih iskalnih algoritmov. Iz teh razlogov in na osnovi ovrednotenja, ki bo predstavljeno v poglavju 5.2.1, smo se odločili, da implementiramo omenjena algoritma.

Poglavje 4

Delovanje agenta

Agent je entiteta, ki je umeščena v simulacijsko okolje, v katerem zavzema določeno pozicijo. Osnovna akcija, ki jo lahko agent v simulacijskem okolju *GridLand* neposredno izvede in se odraža v spremembi stanja okolja, je premik. S to akcijo lahko realiziramo druge višje nivojske akcije, saj premiki na različna polja sprožijo različne spremembe v okolju. Premik na polje s hrano se odraža kot pobiranje hrane, premik na nasprotnikovo polje pa kot napad na nasprotnega agenta. V simulacijsko okolje se lahko poveže več agentov posamezne ekipe. Vsi agentje so v našem primeru entitete istega razreda, ki smo ga v sklopu diplomske naloge implementirali. Seveda bi lahko bil vsak agent entiteta svojega lastnega razreda, kar bi nam omogočilo različne vrste agentov. Za ta pristop se v delu nismo odločili, saj hočemo narediti nekega splošnega agenta, ki je nastavljiv do te mere, da ga lahko prilagodimo za vsako ekipo posebej. To nam omogoča, da lahko izvajamo eksperimente, pri katerih ima vsaka ekipa različne agente in opazujemo, kako določen parameter vpliva na njihovo obnašanje. V tem poglavju si bomo ogledali delovanje agenta skozi njegov celoten življenjski cikel.

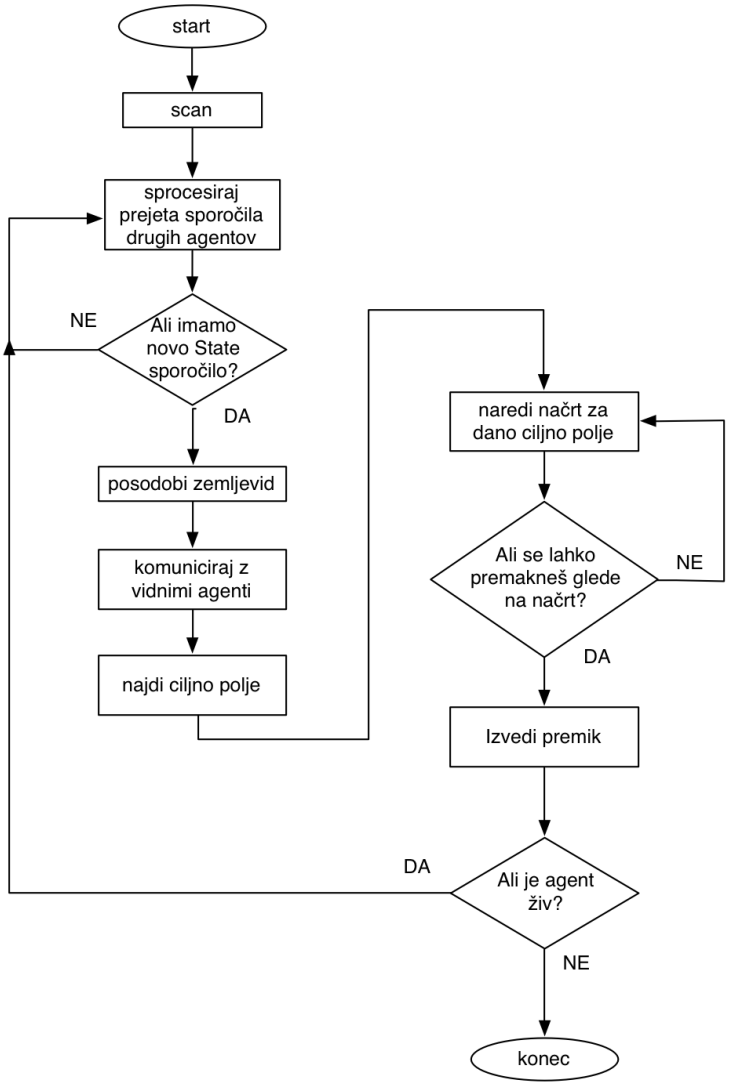
4.1 Izvedba agenta

Naša implementacija agenta, poimenovanega *Explorer*, razširja abstraktni razred Agent, ki smo ga opisali v poglavju 2.3. Ena od najpomembnejših metod, ki jih moramo implementirati, je metoda *run()*. V njej implementiramo obnašanje agenta med njegovim delovanjem v simulacijskem okolju. Ko se agent poveže na strežnik, se pokliče metoda *initialize()*, v kateri poskrbimo za nastavitev primernih vrednosti spremenljivk, ob zaključku agentove prisotnosti v simulacijskem okolju pa se pokliče metoda *terminate()*.

Naša implementacija agenta ima sedem načinov delovanja, ki agentu omogočajo različno delovanje glede na dano situacijo v sistemu. Tej načini so:

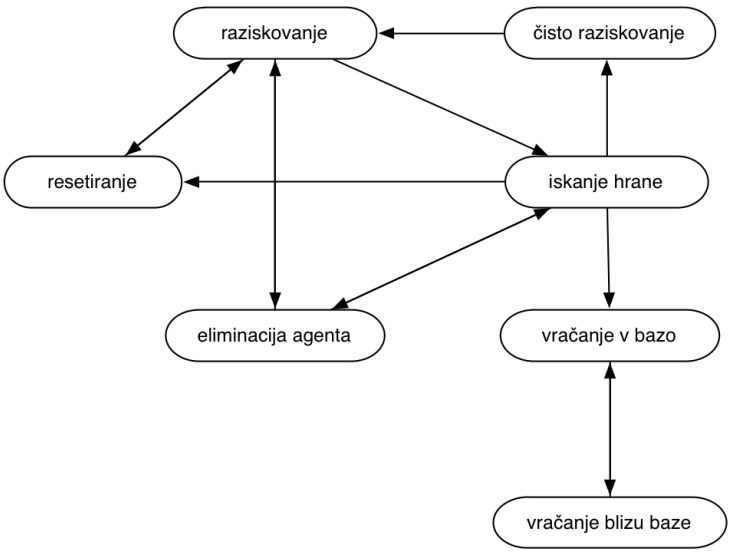
- raziskovanje,
- čisto raziskovanje,
- iskanje hrane,
- vračanje v bazo,
- resetiranje,
- eliminacija agenta,
- vračanje blizu baze.

Na začetku delovanja je agent v načinu *raziskovanje*. Delovanje agenta se začne v metodi *run()*, ki je predstavljena z grafom poteka na sliki 4.1. Na začetku agent pošlje zahtevo za pregled okolice z ukazom *scan()*. Na to sporočilo strežnik odgovori s sporočilom tipa *stanje*, ki vsebuje podatke o poljih, ki so v neposredni bližini agenta. Ukaz *scan()* agent uporabi samo v tem primeru, kasneje pa uporablja ukaz za premikanje, za katerega odgovor vedno dobi novo stanje okolice. Po izvršitvi ukaza za pregled okolice pride do vstopa v zanko, ki se izvaja, dokler je agent živ - prisoten v simulacijskem okolju. V tej zanki se izvaja večji del agentove logike. Na začetku se preveri,



Slika 4.1: Graf poteka metode *run()*.

ali je v vrsti za dohodna sporočila kakšno prejeto sporočilo, ki je bilo poslano od agenta iste ekipe. V kolikor vrsta ni prazna, se prejeta sporočila obdelajo. Nato se preveri, ali je agent prejel kakšno sporočilo tipa *stanje*, ki se hranijo v posebni vrsti. To sporočilo prejme agent kot odgovor na vsa poslana sporočila tipa *skeniranje* in *premik*. Če je vrsta za ta sporočila prazna, se izvajanje vrne na začetek zanke, kjer se preveri, ali se agent v okolju še vedno izvaja. V nasprotnem primeru se sporočilo vzame iz vrste in obdela. V sporočilu agent preveri, kakšna je njegova okolica, z njo posodobi svoj lokalni zemljevid stanja okolice in preveri, ali nosi hrano. V kolikor strežnik v sporočilu sporoči, da agent nosi hrano, ta spremeni način delovanja v *vračanje v bazo*. V ta način delovanja lahko zato teoretično preide agent iz vsakega načina, vendar se v praksi to vedno zgodi iz načina *iskanje hrane*. Na sliki 4.2 je zato prehod v način *vračanje v bazo* označen samo iz stanja *iskanje hrane*. Po posodobitvi lokalnega zemljevida agent preveri, ali se v bližini nahaja kakšen agent iz njegove ekipe. Če se nahaja, mu agent lahko pošlje sporočilo. Več o pošiljanju in sprejemanju sporočil bomo izvedeli v naslednjem poglavju.



Slika 4.2: Graf prehajanja med agentovimi načini delovanja.

Sedaj si agent izbere cilj premika, poišče pot (zgradi načrt) in naredi ustrezen premik. Cilj premika izbere na podlagi načina delovanja in zaporedne številke iteracije iskanja. Za način *raziskovanje* in *iskanje hrane* poteka iskanje cilja v sledečem zaporedju. Sprva se poskusi poiskati najbližjo enoto hrane, v kolikor pa je takih več, se izbere najbližjo glede na podano iteracijo. Nato agent preide v stanje *iskanje hrane* in zaključi z iskanjem cilja. Če hrane ne najde, poišče agenta nasprotne ekipe in v kolikor je v konfiguraciji eliminacija agentov dovoljena, preide v stanje *eliminacija agenta*. V primeru, da tudi to iskanje ni bilo uspešno, oziroma eliminacija ni dovoljena, poskusi z iskanjem najbližjega neraziskanega polja. V skoraj vseh primerih to iskanje najde ciljno vozlišče. Tako kot pri iskanju hrane se tudi tukaj za ključ, po katerem agent izbira med vozlišči, ki so enako oddaljena, uporabi številko podane iteracije. Če je celoten zemljevid preiskan in ni nobenega neraziskanega polja več, agent preide v stanje *resetiranje*. V tem načinu si agent izbriše celoten lokalni zemljevid, razen poti od trenutnega polja do baze in preide v način *raziskovanje*. Na ta način zagotovimo, da agent še naprej preiskuje prostor, saj se je lahko v tem času zaradi trka dveh agentov, od katerih je eden nosil hrano, na zemljevidu pojavila hrana. Agent ponovno začne s preiskovanjem celotnega zemljevida, pot do baze pa si pusti za primer, če hrano najde in jo želi prinesti v bazo.

Pri načinu *eliminacija agenta* agent poišče cilj tako, da najprej poskusi najti hrano, saj je to njegov glavni cilj. Če hrano najde, opusti sledenje agentu in spremeni način delovanja v *iskanje hrane*. Šele v primeru, ko hrane ne more najti, išče najbližjega agenta nasprotne ekipe. Če takega agenta najde, postane novi cilj lokacija najdenega agenta, drugače pa poišče najbližje neraziskano polje in preide v način delovanja *raziskovanje*.

Za način *vračanje v bazo* je iskanje cilja enostavno, saj agent le poišče lokacijo svoje baze. Pri načinu *vračanje blizu baze* agent poišče lokacijo baze, nato pa za ciljno polje izbere naključno prazno polje v njeni neposredni bližini. Pri načinu *čisto raziskovanje* pa za ciljno polje izbere najbližje neraziskano polje.

Sedaj, ko je agent določil ciljno polje, ga poda metodi za iskanje najkrajše poti - Dijkstri ali A*. Metoda vrne celoten načrt od trenutnega do ciljnega vozlišča. Če je dobljen načrt prazen, iskalni algoritem ni mogel najti poti. V tem primeru agent premika ne bo mogel izvesti, zato poveča števec iteracij, zaradi česar bo v naslednjem obhodu zanke pri izbiri ciljnega polja izbral drugo polje. V primeru, da je agent v stanju *iskanje hrane* in ni uspel najti načrta, spremeni način delovanja v *čisto raziskovanje*. Tako agentov algoritem v naslednjem obhodu prisilimo, da ne išče hrane, ampak naredi le premik proti neraziskanemu polju in nato nadaljuje z navadnim preiskovanjem prostora v načinu *raziskovanje*. Če je agent v načinu *vračanje v bazo*, se ta spremeni v *vračanje blizu baze*. S tem pokrijemo situacijo, kjer je pot do baze blokirana. Agent si za ciljno polje izbere polje blizu baze in se na ta način poskusi premakniti v smeri baze. V primeru, da agent uporablja optimalno premikanje z načinom maksimiranja cene premikanja, ki smo ga opisali v poglavju 3.5 in pridobi naslednji premik na ta način, opisanega iskanja poti in ciljnega polja ne izvedemo.

Do sedaj je agent določil ciljno točko in poiskal najkrajšo pot do nje, oziroma pridobil smer naslednjega premika. Čeprav ima agent v trenutku sprejetja sporočila tipa *stanje* na voljo trenutno lokalno stanje okolja, mora biti pazljiv pri lokacijah ostalih agentov, saj ne more predvideti, kakšna bo njihova lokacija v trenutku njegovega premika. Zato je potrebno dodatno preverjanje, ali je premik v izbrano smer varen. Premik je varen v kolikor pri njem ne more priti do trka. Pred premikom se preveri, ali je nova lokacija trenutno prazna in ali se lahko v N korakih na njo premakne kateri izmed drugih agentov. Število korakov N se določi glede na način, v katerem se agent trenutno nahaja. Če agent nosi hrano - torej je v načinu *vračanje v bazo* ali *vračanje blizu baze*, je vrednost N dva koraka, drugače pa en korak. Razlog za večjo vrednost v primeru, da agent nosi hrano, je v njegovi hitrosti, saj se lahko med enim premikom agenta s hrano, drugi agent premakne tudi za dve polji. Prav tako tukaj ločimo primer, ali je drugi agent iz agentove ali iz nasprotne ekipe. Če je agent iz nasprotne ekipe in je oddaljen manj

kot N korakov, se agent za premik odloči naključno z verjetnostjo 0.1 za premik. S tem se izogne morebitnemu smrtnemu objemu (ang. dead-lock), kjer bi oba agenta obstala in se zaradi medsebojnega blokiranja ne bi premaknila. V primeru, da je agent iz agentove ekipe, pa določimo, kateri agent se lahko premakne glede na prioriteto. Za določitev prioritete uporabimo identifikacijsko številko agenta, določeno s stani strežnika. Agent, ki ima manjšo številko, ima prednost pred agentom z večjo. Na ta način nedvoumno določimo agenta, ki se lahko premakne. Agent izvede premik s klicem metode *move()*, ki kot parameter sprejme smer premika. Premik je lahko ena od naslednjih vrednosti: *gor, dol, levo, desno* ali *brez*. Če se agent ne more varno premakniti, izvede klic *move()*, s parametrom za premik *brez*. Ta klic je enakovreden klicu *scan*().

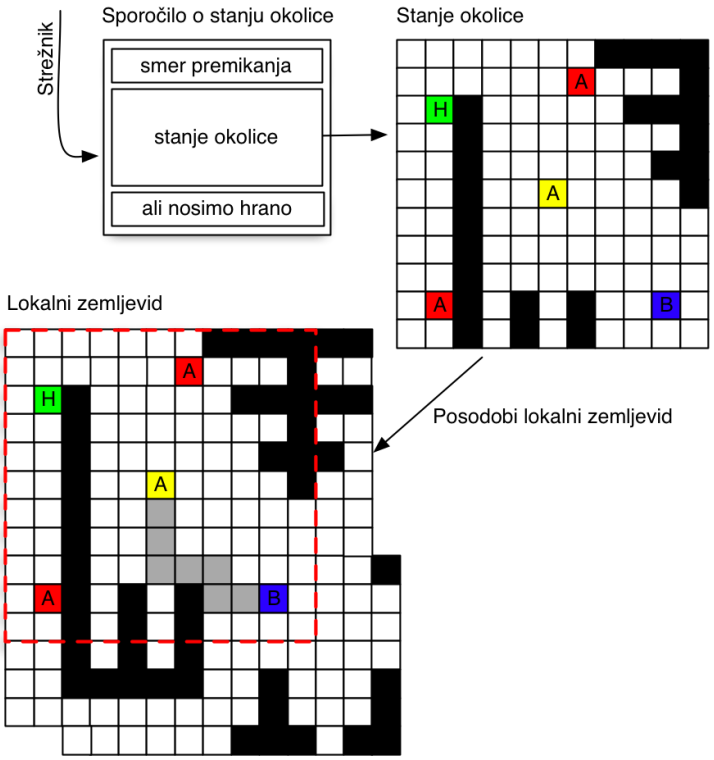
Sedaj se agent vrne na začetek zanke in čaka na prihod sporočila *stanje*, ki bo vsebovalo stanje okolja po izvedenem premiku.

4.2 Lokalni zemljevid

Agent se premika po zemljevidu glede na podatke o njegovi okolici, ki jih dobi po vsakem premiku. Iz teh podatkov si agent gradi lokalni zemljevid sveta, ki ga med drugim uporablja tudi za določitev naslednjega premika.

Slika 4.3 predstavlja potek gradnje lokalnega zemljevida od sprejetja sporočila o stanju okolice pa do posodobitve le-tega. Za vsak premik, ki ga agent izvede, ta dobi povratni odgovor o novem stanju v okolju. Sporočilo o stanju okolice vsebuje poleg stanja še podatek o smeri premikanja in podatek o tem, ali agent ima hrano, kar je zanj pomembna informacija pri odločanju o načinu delovanja. Dobljeno sporočilo mora agent uskladiti s svojo lokalno predstavitvijo zemljevida, zato najprej iz sporočila izloči del o stanju okolice. Na desni strani slike 4.3 vidimo dobljeno stanje okolice v radiju petih polj. Dobljena okolica je popolna, saj vedno dobimo vsa polja v radiju, tudi tista, ki so v okolju za zidom. Zidovi so predstavljeni s črnim kvadratom, prosta polja pa z belim. Agent je označen z rumeno črko A in je v središču okolice.

Modro polje, označeno s črko B, označuje agentovo bazo, zeleno s črko H hrano, rdeča polja pa so agentje nasprotne ekipe. Sedaj moramo združiti dobljen del zemljevida z lokalnim. To storimo tako, da izračunamo vektor odmika agentove lokacije od njegove baze. Združitev obeh zemljevidov je prikazana na levi strani slike z rdečo prekinjeno črto. Na lokalni zemljevid se prenesejo vsa polja iz okolice, vendar nekatera le začasno. Tak primer predstavlja polje, ki vsebuje agenta naše ali nasprotne ekipe. Agent se po poljih premika, kar zahteva posebno obravnavo pri posodabljanju zemljevida. Operiranje s temi polji in implementacijske podrobnosti delovanja zemljevida so opisane v naslednjem podpoglavju.



Slika 4.3: Potek gradnje lokalnega zemljevida.

4.2.1 Implementacija osveževanja zemljevida

Pomembna informacija, na katero se med grajenjem lokalnega zemljevida agent zanaša, je sporočilo tipa *state*. Sporočilo vsebuje naslednje podatke:

- smer premikanja,
- trenutno stanje okolice,
- ali agent nosi hrano.

Za grajenje zemljevida je najpomembnejši podatek o trenutnem stanju okolice. Podan je kot razred tipa $Neighborhood$, ki vsebuje enodimenzionalno tabelo s celoštevilskimi vrednostmi, ki predstavljajo tip posameznega polja. Tipi polj so naslednji: *prazno polje*, *zid*, *baza*, *nasprotnikova baza*, *hrana* in *agent nasprotne ekipe*. Vsak agent loči med agenti svoje in nasprotne ekipe. Vse agente agentove ekipe predstavimo z identifikacijskimi številkami, ki so večje od nič, agente nasprotne ekipe pa s številom, ki prestavlja tip polja *nasprotni agent*. Za enostavno dostopanje do posameznega polja iz tabele nam razred priskrbi funkcijo $getCell(x, y)$, ki vrne vrednost polja na zahtevani koordinati.

Implementirali smo razred Map, v katerem imamo poleg lokalnega zemljevida še vse potrebne metode, ki operirajo z njo. Lokalni zemljevid je implementiran kot zgoščena tabela, kjer je ključ objekt $Position$ in predstavlja lokacijo polja, vrednost pa vrsto polja, predstavljeno s celoštevilskim tipom $Integer$. Za uporabo zgoščene tabele smo se odločili zaradi hitrega preverjanja vsebovanosti polja v tabeli in dodajanja novega elementa. Ob vsakem prejetju sporočila $State$, agent iz njega vzame informacijo o okolici in z ustrezno metodo razreda Map posodobi lokalni zemljevid. Ob prvi posodobitvi zemljevida (ko je ta še prazen) se izračuna odmik od baze, s čimer se zagotovi, da je agentova baza v izhodiščnem polju (0,0). Pri naslednjih posodobitvah se določi lokacija prejetih okoliških polj glede na lokacijo agenta in odmika.

Na poljih so lahko statični ali dinamični objekti. Med dinamične objekte štejemo agente, ki so lahko iz ene ali druge ekipe. Te moramo obravnavati

drugače, saj vemo, da na zemljevidu ne bodo ves čas zavzemali polja na istih lokacijah, za razliko od zidu, ki je statičen objekt. Prejete lokacije nasprotnih agentov si zato agent shrani le začasno in jih na koncu vsake iteracije v metodi *run*() odstrani iz zemljevida. Razlog za tako delovanje izvira iz dejstva, da nasprotnih agentov ne ločimo med seboj in tako ne moremo slediti premikom posameznega agenta ter pravilno posodabljati njegove lokacije. Zato imamo po vsaki posodobitvi zemljevida na njem le lokacije nasprotnih agentov, ki so v neposredni bližini. Agent lahko loči agente svoje ekipe po identifikacijskih številkah, zato smo implementirali drugačen mehanizem shranjevanja njihovih lokacij. Pri vsaki posodobitvi zemljevida se shrani lokacija agenta v lokalni zemljevid, v posebno zgoščeno tabelo pa se shrani njegova identifikacijska številka skupaj s preostalim časom življenja (ang. TTL - time to live). Ob dodajanju ali posodobitvi lokacije agenta se vrednost TTL poenostavi, pri vsaki posodobitvi zemljevida pa se ta vrednost zmanjša. Ko agentov čas doseže vrednost 0, se ga odstrani iz tabele in iz zemljevida. Na ta način smo poskrbeli, da se agentova lokacija pravilo posodablja in jo agent po zadnjem srečanju še nekaj časa hrani, kar pride prav pri medsebojni komunikaciji agentov.

4.3 Komunikacija med agenti

Komunikacija je proces, pri katerem oddajni agent pošlje določeno informacijo sprejemnemu agentu in si tako izmenjata informacije. Brez komunikacije vsak agent deluje le na podlagi lastnih prepričanj in neodvisno od drugih agentov. S pomočjo komunikacije se lahko agentje obnašajo veliko bolj inteligentno, saj z njo zagotovimo pretok in izmenjavo znanj.

Za uspešno izvedbo komunikacije morajo imeti agentje sposobnost pošiljanja in sprejemanja sporočila. Vedeti morajo, komu bodo poslali sporočilo in kakšna bo vsebina sporočila. Prav tako morajo vedeti, kaj storiti s prejetim sporočilom. Pri izvedbi našega agenta smo za pošiljanje sporočil med agenti uporabili podedovano metodo *send*(). Agent pošlje sporočila vsem okoliškim

agentom. Informacijo o bližnjih agentih agent pridobi s sporočilom o stanju okolice, ki ga pošlje strežnik. Kateri agentje so v zadostni bližini, da jim agent lahko pošlje sporočilo, nastavimo s parametrom *radij komunikacije*. Da sporočil ne bi bilo preveč, lahko agent pošlje le eno sporočilo na premik.

Za sprejem sporočil je potrebno implementirati abstraktno metodo *receive()*, ki se pokliče pri prejetju sporočila. Na ta način agentu zagotovimo možnost prejemanja sporočil. Po prejetju sporočila, se to shrani za kasnejšo obdelavo. Po vsakem premiku agent preveri, ali je prejel kakšno sporočilo. Vsako prejeto sporočilo mora obdelati, tako da ugotovi njen pomen in izvede ustrezne akcije.

Odgovoriti moramo na vprašanje, kaj naj si agentje med seboj sporočajo. Sporočilo mora vsebovati informacijo, ki bo pripomogla k hitrejši rešitvi problema - iskanju hrane. Naša izvedba agenta omogoča pošiljanje agentovega lokalnega zemljevida v katerem je njegovo trenutno stanje okolja. Poslan zemljevid vsebuje vse statične objekte lokalnega zemljevida skupaj s praznimi polji in agenti njegove ekipe, ki so v neposredni bližini. Lokacije agentov, ki so v nasprotni ekipi, ne pošiljamo. S prejemom zemljevida drugega agenta se poveča agentov raziskan prostor, kar privede do hitrejšega raziskanja okolja in posledično hitrejšega lociranja hrane. Za sporočanje bi lahko uporabili tudi hevristične ocene do določenih polj ali pa samo agentovo lokalno okolico.

Čeprav prejeti zemljevidi vsebujejo trenutni lokalni zemljevid drugega agenta, ni nujno, da je ta ažuren glede na stanje v okolici. Zaradi velikosti zemljevida je velika verjetnost, da se prejeto stanje okolice v sporočilu *okolica*, ki ga dobimo kot odgovor na premik, rahlo razlikuje od prejetega zemljevida. Razlog najdemo v strežniku, saj mora ta poslano sporočilo sprejeti, ga obdelati in poslati naslovnemu agentu, kar se odraža v določenem časovnem zamiku med pošiljanjem in sprejetjem sporočila. Rešitev te težave je preprosta: najprej obdelamo vsa prejeta sporočila drugih agentov, nato pa posodobimo lokalni zemljevid s prejetim sporočilom trenutne okolice. Na ta način bomo odpravili morebitne nepravilnosti, nastale v naši okolici.

Agent mora prejet lokalni zemljevid na pravilen način združiti z svojim,

saj ima ta lahko ista polja na drugih lokacijah. To stori tako, da izračuna zamik zemljevidov glede na lokacijo baze (postavi jo v izhodišče). Sledi prehod čez vsa prejeta polja, ki jih dodaja v lokalni zemljevid ali pa v njem samo posodobi njihove vrednosti. Polje, kjer se nahaja agent, preskoči (zaradi zamika je lahko še na prejšnji lokaciji), polja drugih agentov pa doda v zgoščeno tabelo, kot to stori pri posodobitvi zemljevida z okolico.

4.4 Konfiguracija agenta

Do sedaj smo opisali delovanje našega agenta, sedaj pa si oglejmo, katere lastnosti agenta lahko spreminjamo in kakšen vpliv imajo na delovanje.

Na voljo imamo naslednje lastnosti, ki jih lahko določamo s parametri:

- iskalni algoritem,
- radij pošiljanja sporočil,
- optimalno preiskovanje,
- število agentov,
- eliminacijo agentov.

Na voljo imamo dva algoritma, ki ju lahko agent uporabi za iskanje poti po zemljevidu. To sta Dijkstra in A*, ki se uporabita v primeru, da je optimalno preiskovanje (maksimizacija preiskanih polj) izključeno z ustreznim parametrom. Če je optimalno preiskovanje vključeno, agent poskusi z iskanjem poti na ta način in v kolikor ne more določiti optimalnega premika, uporabi izbran iskalni algoritem. Nastavljivo je tudi število agentov, ki se kot ločene instance razreda povežejo na simulacijsko okolje. Komunikacijo med agenti lahko reguliramo s parametrom za radij pošiljanja sporočil. Z njim povemo, koliko je lahko oddaljen agent, da mu še lahko pošljemo sporočilo. Običajno je ta vrednost na intervalu med nič in velikostjo radijem okolice, ki jo dobimo v strežniškem sporočilu *state*. Če je vrednost nastavljena na

nič, agentje med sabo ne komunicirajo. Zadnji parameter, ki je na voljo, je parameter za omogočanje eliminacije agentov. V primeru, da je eliminacija omogočena, lahko agent začne slediti nasprotnemu agentu in se poskuša zaleteti vanj, kar ima za posledico izločitev obeh agentov iz simulacije.

Ker bomo v nadaljevanju naloge preverili obnašanje agentov z različnimi lastnostmi, potrebujemo enostaven način za njihovo nastavitev. Na posamezno lastnost agenta lahko vplivamo s parametrom, ki je zapisan v konfiguracijski datoteki. Konfiguracijska datoteka mora zato vsebovati vrednosti parametrov za vse agentove lastnosti. Razred *Agent* smo predelali, tako da sedaj kot prvi argument sprejme konfiguracijsko datoteko. To datoteko prebere in shrani njene vrednosti v spremenljivko, do katere lahko dostopajo vsi razredi, ki dedujejo razred *Agent*. Na ta način lahko sedaj iz naše implementacije agenta dostopamo do konfiguracije in tako nastavimo želeno delovanje agenta.

Poglavje 5

Eksperimentalna primerjava različnih agentov

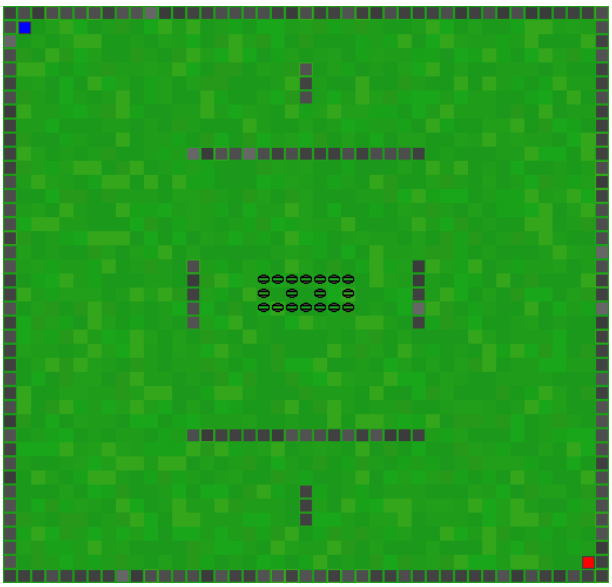
V tem poglavju si bomo ogledali, kako se naš agent odreže v praksi. Simulirali bomo več iger med dvema ekipama agentov z različnimi lastnostmi. Dobljene rezultate bomo primerjali in poskusili utemeljiti, kako določena lastnost vpliva na delovanje agentov in na razplet igre.

5.1 Pravila in postopek simulacije

Da zagotovimo med seboj primerljive rezultate simulacij, moramo določiti pravila, ki bodo veljala za vsako simulacijo. Pravila so naslednja: V eni simulaciji sodelujejo agenti dveh ekip, katerih bazi se nahajata v kotih zemljevida. Zemljevid je simetričen, kar postavi obe ekipi v enakovreden položaj. Pri vsaki ekipi agenti prihajajo iz baze na vsakih 30 korakov. Posamezen korak traja 20 ms in je določen na strežniški strani. Agent se odstrani iz simulacije v primeru trka z drugim agentom, zidom ali bazo. Ko se agent odstrani iz simulacije, se njegova instanca izbriše in se naredi nova, ki ne poseduje nikakršnega znanja prejšnje. Nova instanca agenta se pojavi v simulacijskem okolju, ko le-ta pride iz baze. Agent pridobi hrano, če se premakne na polje, na katerem se hrana nahaja. V kolikor se agent med nošenjem hrane odstrani

iz simulacije (v primeru trka), se njegova hrana pojavi na enem od bližnjih polj. V primeru, da ima agent hrano in se zaleti v svojo bazo, se njegova instanca odstrani iz simulacije, njegova ekipa pa dobi točko za pobrano hrano. V tem primeru hrana izgine iz okolja. Simulacija se zaključi, ko v okolju ni nobene hrane - seštevek ekipnih točk je enak začetnemu številu enot hrane.

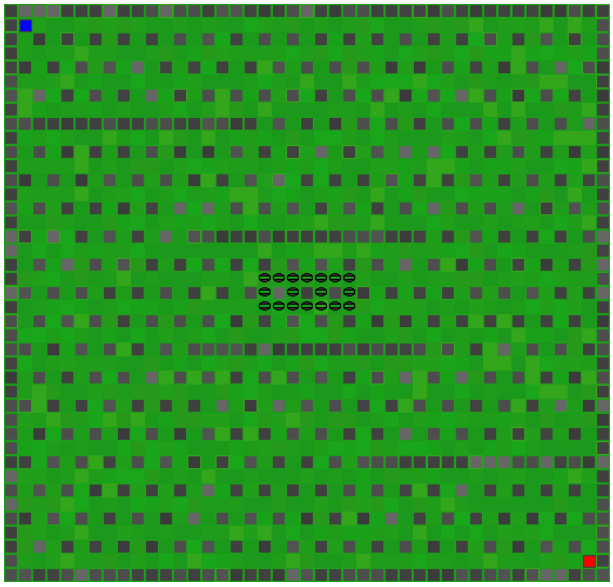
Pri testiranju obnašanja agentov z različnimi parametri izvedemo več simulacij pod enakimi pogoji in pogledamo, kakšne rezultate dobimo. Konfiguracija strežnika je za vse simulacije ista, razen če ni navedeno drugače. Za testiranje smo napisali lupinsko skripto, ki požene strežnik in agente obeh ekip. Za obe ekipi agentov določimo nastavitveno datoteko z nastavljenimi želenimi parametri. Ob koncu simulacije se zapiše rezultat (število pobrane hrane, čas izvajanja) v datoteko. Skripta požene več zaporednih simulacij z enakimi parametri, tako da dobimo čimbolj nepristranske rezultate. Ko smo izvedli vse simulacije, poženemo skripto, ki na podlagi generiranih dnevniških datotek izračuna povprečne vrednosti rezultatov.



Slika 5.1: Zemljevid odprtega tipa.

Delovanje agentov je odvisno tudi od samega zemljevida, kjer se simula-

cija izvaja. Ta je lahko zaprtega ali pa odprtega tipa. Odprti tip zemljevida vsebuje malo ovir in veliko odprtega prostora. Primer takega zemljevida, na katerem bomo izvajali simulacije, je zemljevid na sliki 5.1. Pri zemljevidih zaprtega tipa pa imamo veliko ovir - zidov, ki otežujejo premikanje agentov. Primer zaprtega zemljevida vidimo na sliki 5.2. Pri obeh zemljevidih imamo na voljo enako količino hrane (18 enot) na identično istih lokacijah. Pričakujemo, da se bodo rezultati simulacij razlikovali glede na tip zemljevida, zato vsako skupino simulacij izvedemo na obeh zemljevidih.



Slika 5.2: Zemljevid zaprtega tipa.

5.2 Primerjava lastnosti agentov

Primerjali bomo, kako vpliva algoritem iskanja, številčnost, komunikacija in eliminacija na delovanje agenta. Za vsako skupino simulacij bomo najprej predstavili parametre, s katerimi nastavimo lastnosti agentov vsake ekipe. Nato si bomo ogledali, kakšna je bila povprečna količina nabrane hrane za vsako ekipo na zemljevidu odprtega in zaprtega tipa.

5.2.1 Vpliv izbire iskalnega algoritma

Najprej bomo primerjali dve ekipi, ki štejeta vsaka po 10 agentov, ki med seboj ne komunicirajo. Obe ekipi se med sabo ne napadata - eliminacija je izključena. Prva ekipa uporablja za iskanje poti Dijkstrov algoritem, druga pa A*.

Pri izvajanju 20 zaporednih simulacij smo dobili rezultate, prikazane v tabeli 5.1. Algoritem Dijkstra se je odrezal slabše od A* pri zaprtem in odprtem zemljevidu. Razlika v pobrani hrani je še posebno vidna pri zemljevidu zaprtega tipa. Dobljen rezultat je sprva malo nenavaden, saj vemo, da nam oba algoritma vedno vrneta isto (optimalno) pot. Razlog za slabši rezultat bi lahko pripisali hitrosti izvajanja algoritmov. A* uporablja za preiskovanje hevristiko, kar mu močno zmanjša število preiskanih vozlišč in posledično tudi čas izvajanja. Čas izvajanja bi lahko bil ključen dejavnik za slabši rezultat pri agentih, ki uporabljajo Djikstrov algoritem. V tem primeru je simulacija

	Odprt zemljevid	Zaprt zemljevid
Dijkstra	8.2	7.4
A*	9.8	10.6

Tabela 5.1: Povprečno število pobranih enot hrane pri simulacijah z ekipo, ki uporablja Djikstrov algoritem in ekipo z algoritmom A*.

tekla s hitrostjo 50 korakov na sekundo, kar tudi predstavlja največjo možno hitrost agenta. Da bi preverili našo hipotezo, ponovno izvedemo simulacijo z istimi parametri, vendar z manjšo hitrostjo simulacije. Hitrost simulacije znižamo na 10 korakov na sekundo. Rezultati ponovne simulacije so vidni v tabeli 5.2.

Kot vidimo, se je razlika med ekipami agentov z enim in drugim algoritmom zmanjšala. Veliko spremembo v številu enot pobrane hrane opazimo pri zemljevidu zaprtega tipa (1.4 hrane). Razlog za večjo spremembo je boljše delovanje algoritma A* v zaprtih prostorih, saj hevristika omeji iskanje, medtem ko algoritem Dijkstra za izračun optimalne rešitve preišče več prostora.

	Odprt zemljevid	Zaprt zemljevid
Dijkstra	8.5	8.8
A*	9.5	9.2

Tabela 5.2: Povprečno število pobranih enot hrane pri zmanjšani hitrosti (10 korakov/s) simulacij z ekipo, ki uporablja Djikstrov algoritem in ekipo z algoritmom A*.

Ko smo zmanjšali hitrost simulacije, je razlika v hitrosti postala manj pomembna. Pri zemljevidu odprtega tipa tako velike razlike ni, ker tukaj A* ni v tako veliki prednosti zaradi manjšega števila ovir.

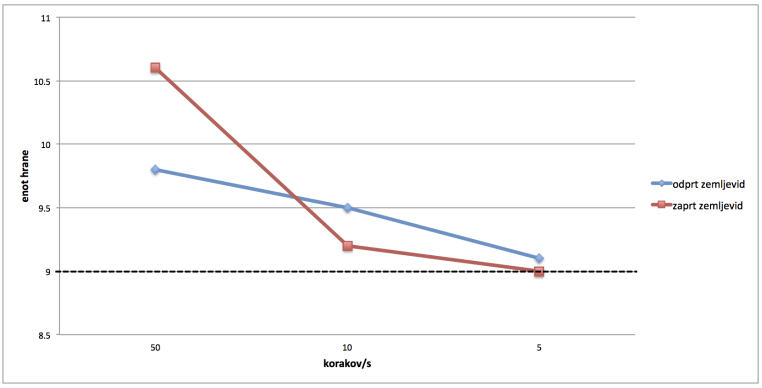
Za konec opravimo še simulacijo s hitrostjo 5 korakov na sekundo, da dokončno potrdimo našo hipotezo. Rezultate vidimo v tabeli 5.3.

	Odprt zemljevid	Zaprt zemljevid
Dijkstra	9.0	9.1
A*	9.0	8.9

Tabela 5.3: Povprečno število pobranih enot hrane pri zmanjšani hitrosti (5 korakov/s) simulacij z ekipo, ki uporablja Djikstrov algoritem in ekipo z algoritmom A*.

Graf na sliki 5.3 prikazuje število nabranih enot hrane za agente z algoritmom A*, glede na hitrost simulacije, pri čemer je nasprotna ekipa uporablja Dijkstrov algoritem. V okolju je vedno 18 enot hrane, kar pomeni, da je boljši algoritem tisti, ki ima število pobrane hrane več kot 9. Iz tega razloga je na grafu pri vrednosti 9 prekinjena črta, ki prikazuje mejno vrednost - ekipa, ki se nahaja nad to črto je v prednosti. Pri hitrosti simulacije 5 korakov na sekundo sta pri odprtem zemljevidu algoritma izenačena, pri zaprtem zemljevidu pade prednost ekipe z A* na vsega 0.1 enote hrane, kar je zanemarljivo.

Povprečna hitrost trajanja simulacije ene igre je v prvem primeru (50 korakov/s) trajala 111.75 sekund pri odprtem zemljevidu oziroma 151.2 sekund pri zaprtem. V drugem primeru (10 korakov/s) se je ta čas povečal na 286.5



Slika 5.3: Število pobranih enot hrane skupine agentov z algoritomom A*, glede na hitrost simulacije. Nasprotna ekipa uporabja Dijkstrov algoritem.

sekund pri odprtem in 380.5 sekund pri zaprtem zemljevidu. V zadnjem primeru (5 korakov/s) pa se je povprečni čas trajanja povečal na 720 sekund pri zaprtem oziroma 500 sekund pri odprtem zemljevidu. V nadaljevanju bomo pri simulacijah za iskanje poti uporabljali le A*, saj je hitrejši in nam omogoča, da se simulacije lahko izvajajo z višjo hitrostjo.

Oglejmo si še, kako se obnesejo agenti, ki iščejo pot po načinu maksimizacije preiskanih polj (parameter optimalno preiskovanje) v primerjavi z agenti z algoritmom A*. Omeniti moramo, da v primeru, ko se agent na podlagi maksimizacije ne more odločiti za (optimalen) premik, poišče pot s pomočjo algoritma A*. Vsaka ekipa ima 10 agentov, ki med sabo ne komunicirajo in se ne napadajo. V tabeli 5.4 so rezultati simulacije. Vidimo, da se povprečno

	Odprt zemljevid	Zaprt zemljevid
Optimalno preiskovanje	8.9	9.2
A*	9.1	8.8

Tabela 5.4: Število pobranih enot hrane pri simulaciji Optimalno preiskovanje proti A*

število pobrane hrane skoraj ne razlikuje glede na algoritem preiskovanja. Pri zemljevidih odprtega tipa je A* rahlo v prednosti, optimalno preiskovanje pa

se bolje odreže pri zaprtih zemljevidih. Optimalno preiskovanje poskusi narediti premik, ki poveča raziskan prostor, zato ima agent s to lastnostjo večjo verjetnost za najdbo hrane. Pričakovali bi, da je zato optimalno preiskovanje superiorno algoritmu A* in se bo agent s to lastnostjo v najslabšem primeru odrezal enako kot agent z algoritmom A*. Razlog za nepričakovan rezultat bi lahko bil v razporeditvi hrane na zemljevidu. Vsa hrana se sedaj nahaja v središču zemljevida, kot prikazuje slika 5.1. Da bi potrdili naše predvidevanje, opravimo simulacijo, v kateri bo hrana razporejena naključno po celotnem zemljevidu. Ostali parametri so enaki kot pri predhodni simulaciji. Rezultate vidimo v spodnji tabeli 5.5. Dobljeni rezultati so tokrat v skladu s pričakovanji, saj je količina nabrane hrane enaka ali večja od količine, ki jo naberejo agenti z A* iskanjem.

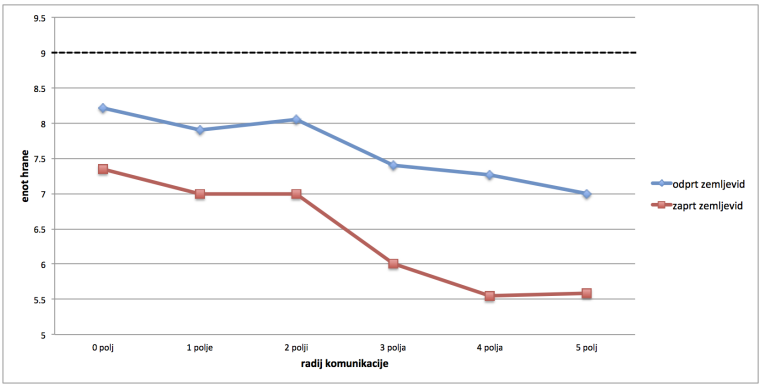
	Odprt zemljevid	Zaprt zemljevid
Optimalno preiskovanje	9.4	9.0
A*	8.6	9.0

Tabela 5.5: Rezultat simulacij Optimalno preiskovanje proti A* pri naključno porazdeljeni hrani

5.2.2 Vpliv komunikacije

Sedaj si oglejmo, kako komunikacija vpliva na uspešnost agentov pri nabiranju hrane. Ponovno primerjajmo algoritma A* in Dijkstro, le da tokrat lahko agenti, ki uporabljajo Dijkstro, med seboj komunicirajo. V vsaki ekipi je 10 agentov in med sabo se ne napadajo. Izvedemo več sklopov ponovitev simulacij, pri čemer za vsak sklop povečamo razdaljo, na kateri lahko agenti med sabo komunicirajo. Rezultati so prikazani na sliki 5.4. Zaradi preglednosti je prikazano povprečno število nabrane hrane le za skupino agentov z Dijkstra algoritmom. Iz grafa vidimo, da je algoritem Dijkstra slabši pri obeh tipih zemljevida. Tudi komunikacija med agenti ne povzroči izboljšanja rezultata. Zanimivo je, da se pri povečanju radija komuniciranja rezultat agentov, ki

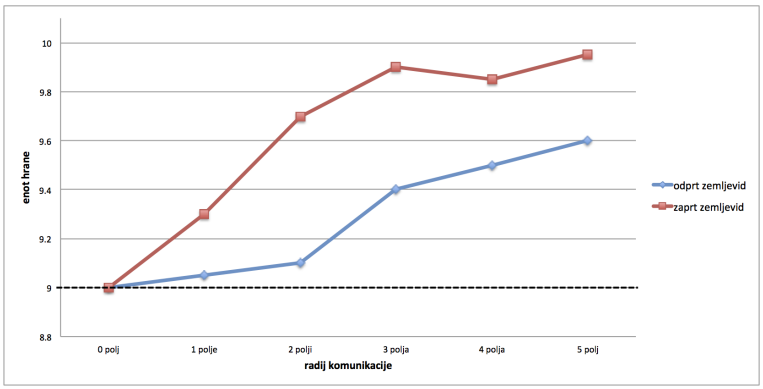
uporabljajo Dijkstro, slabša. Pričakovali smo, da bo komunikacija med agenti pozitivno vplivala na izid igre, vendar to ni vedno tako. Razlog v poslabšanju ekipe, ki lahko komunicira, ponovno poiščemo v času izvajana. Kot smo videli v prejšnjem poglavju, je algoritem Dijkstra počasnejši od A*, sedaj pa ga dodatno obremenimo še s procesiranjem komunikacije. Z večanjem radija komuniciranja se poveča število prejetih in poslanih sporočil, kar predstavlja dodatno zakasnitev pri reagiranju agenta. Večja zakasnitev reagiranja predstavlja manj premikov, kar usodno vpliva na rezultat.



Slika 5.4: Število pobranih enot hrane ekipe agentov z algoritmom Dijkstra in spremenljivim radijem komunikacije.

Poskusimo s simulacijo dveh ekip agentov, kjer obe ekipi uporabljata isti algoritem planiranja, medtem ko samo ena ekipa lahko komunicira. Zanima nas ali bo s pomočjo komunikacije ekipa lahko premagala drugo, tako da bo pobrala več hrane. Tokrat smo primerjali ekipi petih agentov, kjer smo eni ekipi povečevali radij komunikacije. Rezultate vidimo na sliki 5.5. Prikazan je graf za ekipo agentov z omogočeno komunikacijo, kjer vidimo količino pobrane hrane glede na radij komunikacije. Kot vidimo, je skupina agentov, ki imajo omogočeno komunikacijo boljša od skupine, ki ne uporablja komunikacije (radij 0 polj). S povečevanjem radija se izboljšuje tudi število pobrane hrane, vendar le do neke mere. Povečanje komunikacije iz treh na pet polj ne prinese več tolikšne spremembe v količini nabrane hrane. Iz opravljenega

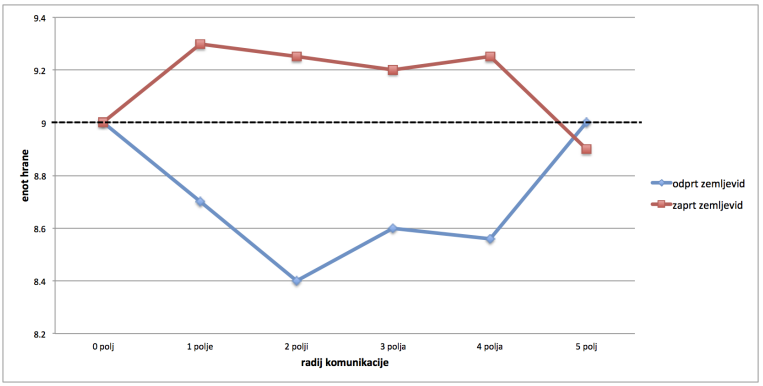
eksperimenta lahko potrdimo, da povečevanje radija komunikacije pozitivno vpliva na delovanje agentov.



Slika 5.5: Število pobranih enot hrane ekipe agentov, ki lahko komunicirajo, glede na nasprotno ekipo, ki ne komunicira.

Opravimo še simulacije za primer, pri katerem lahko obe ekipi komunicirata, vendar imata drugačen način preiskovanja. Prva ekipa agentov uporablja način optimalnega preiskovanja, druga pa algoritem A*. Obe ekipi štejeta po deset agentov. Med simulacijami povečujemo radij komunikacije in opazujemo, koliko hrane pobere vsaka ekipa. Rezultat za ekipo agentov z optimalnim preiskovanjem vidimo na sliki 5.6. Ta se razlikuje glede na zemljevid, v katerem je simulacija potekala. Pri zemljevidu zaprtega tipa je bilo optimalno preiskovanje skoraj vedno uspešnejše od samega A*. Za slabše se pokaže le v primeru, kjer obe ekipi komunicirata z radijem 5 polj. Podoben rezultat (pri zaprtem zemljevidu) smo dobili tudi pri simulaciji z istimi algoritmi preiskovanja in brez komunikacije. Razlog za poslabšanje pri komunikaciji z radijem 5 polj je v tem, da agent v veliki meri uporablja algoritem A*, saj optimalen premik ni mogoč. S sporočili drugih agentov velikokrat pridobi tudi raziskano okolico okoli njega, kar onemogoča optimalen premik. Pri zemljevidu odprtega tipa pa je situacija obrnjena. Agenti z optimalnim preiskovanjem se odrežejo slabše od agentov z A*, razen pri radiju komunikacije 5 polj. Pri tem radiju sta obe ekipi, ne glede na tip ze-

mljevida, zelo primerljivi, kar pripisujemo dejstvu, da se veliko iskanja izvaja z algoritmom A*, kar daje primerljiv rezultat.

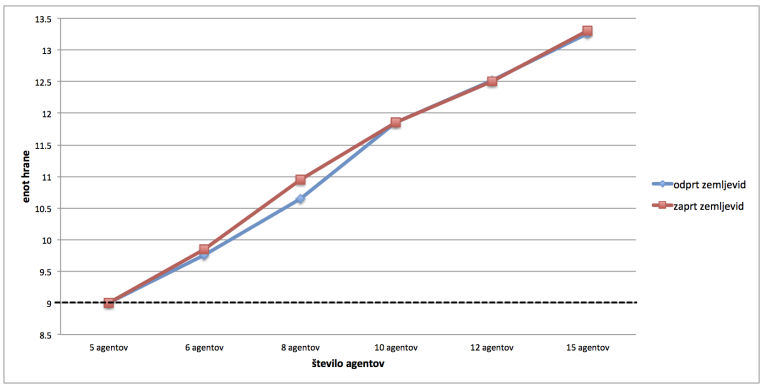


Slika 5.6: Število pobranih enot hrane ekipe agentov z optimalnim preiskovanjem glede na agente z A*, kjer komunicirata obe ekipi.

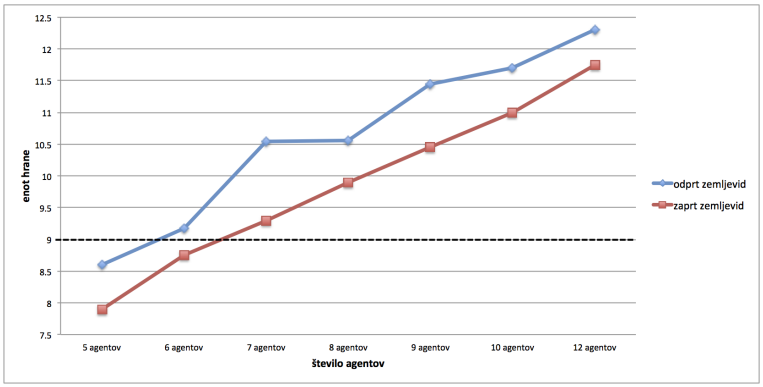
5.2.3 Vpliv številčnosti

V tem podpoglavju si bomo ogledali, kako vpliva število agentov na rezultat simulacij. Tokrat smo opravili simulacijo med ekipama agentov z optimalnim preiskovanjem, ki med seboj ne komunicirata. V eni ekipi je bilo ves čas le pet agentov, medtem ko smo število agentov v drugi ekipi povečevali. Rezultati simulacije za zemljevid odprtega in zaprtega tipa so prikazani na sliki 5.7. Graf na sliki prikazuje število pobranih enot hrane ekipe s spremenljivim številom agentov. Rezultat je skoraj neodvisen od tipa zemljevida in se konstantno izboljšuje glede na število agentov. To nas ne preseneča, ker nam vsak dodatni agent predstavlja novega nosilca hrane. Vrednosti v grafu kažejo, da s povečanjem števila agentov za skoraj enako vrednost povečamo število nabrane hrane.

Zanima nas, kako se lahko rezultat spremeni, če agenti med seboj komunicirajo. Izvedli smo simulacijo, kjer se tako kot v prejšnji, pomerijo agenti z istim načinom preiskovanja (optimalno preiskovanje), le da lahko skupina petih agentov med sabo komunicira v radiju petih polj. Druga skupina, ki ji



Slika 5.7: Rezultat simulacije pri povečevanju števila agentov ene ekipe.

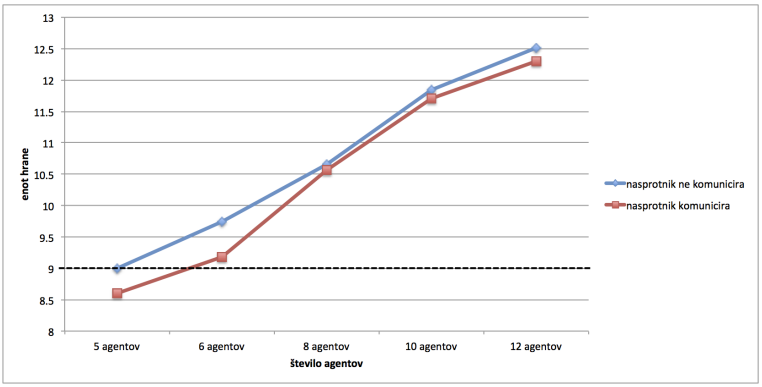


Slika 5.8: Rezultat simulacije pri povečevanju števila agentov ene ekipe, pri čemer nasprota ekipa lahko komunicira.

povečujemo število agentov, pa nima možnost komuniciranja. Rezultati simuliranja so prikazani na sliki 5.8, kjer so podane količine nabrane hrane za skupino agentov, ki ji povečujemo število le-teh. Pri enako velikih skupinah je po pričakovanjih v prednosti skupina, ki lahko komunicira. Razlika se pojavi v tipu zemljevida, kjer se agenti, ki komunicirajo, bolje izkažejo pri zaprtem tipu. S povečevanjem števila agentov se ekipi rezultat izboljšuje. Prelomna točka nastopi že pri šestih agentih, kjer je odločilen zemljevid simulacije. Pri zaprtem je še vedno v prednosti ekipa s komunikacijo, medtem ko pri odprtem ekipa šestih agentov že pobere več hrane. Nadaljnjo povečevanje števila

agentov le še izboljšuje rezultat v prid ekipi z večjim številom agentov.

Oglejmo si primerjavo med zgornjima simulacijama. Slika 5.9 prikazuje primerjavo rezultatov na odprtem zemljevidu med ekipo, ki komunicira in ekipo, ki ne. Pri enakem številu agentov, oziroma pri enem agentu več, je razlika v 0.4 enote več pobrane hrane za ekipo, kjer nasprotnik ne komunicira. Z večanjem števila agentov pa se razlika zelo zmanjša, vendar je še vedno opazna prednost komunikativne ekipe.

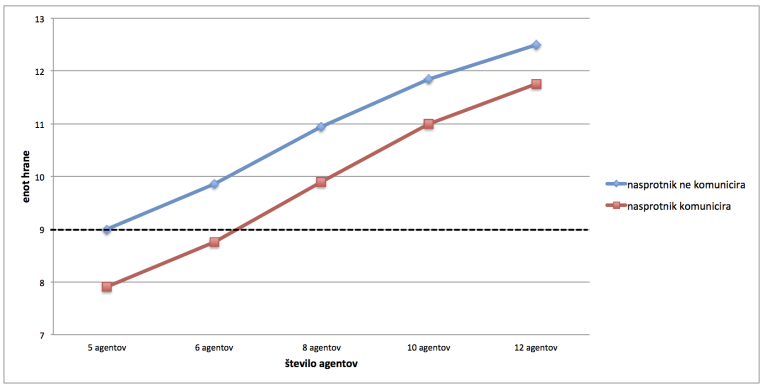


Slika 5.9: Primerjava rezultatov na odprtem zemljevidu glede na ekipo z možnostjo komuniciranja in brez.

Rezultati za isti primer, le da je zemljevid zaprt, so podani na sliki 5.10. Razlika simulacij med ekipo s komunikacijo in brez je tukaj izrazitejša in znaša 0.9 enote hrane. Ekipa, ki ima za nasprotnika ekipo, ki komunicira tudi pri odprtem zemljevidu, doseže slabši rezultat. Razlika med eno in drugo ekipo je ne glede na dodano število agentov konstantna. Razlog za to je tip zemljevida, saj se pri zaprtem agenti težje premikajo. Iz tega razloga predstavlja vsaka dodatna informacija o zemljevidu, pridobljena od drugih agentov, veliko prednost ekipe.

5.2.4 Vpliv eliminacije

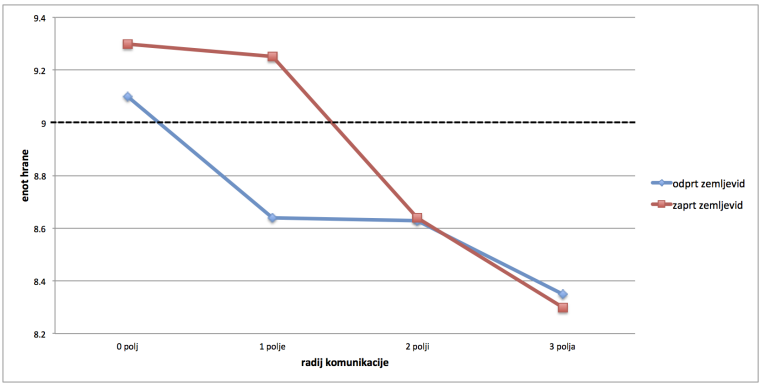
Zadnja lastnost, za katero smo preverili vpliv na delovanje, je lastnost eliminacije. Ta omogoča, da se agent z določeno verjetnostjo odloči za napad na



Slika 5.10: Primerjava rezultatov na zaprtem zemljevidu glede na ekipo z možnostjo komuniciranja in brez.

bližnjega agenta nasprotne ekipe. V opravljeni simulaciji se agent odloči za napad v 20% primerov, če zazna nasprotnega agenta v radiju petih polj. V simulaciji sta nastopili dve ekipi, ki štejeta vsaka po pet agentov. Vsi agenti so uporabljali način preiskovanja A*. Prva ekipa je lahko napadala nasprotne agente, medtem ko je druga ekipa lahko med sabo komunicirala. Ker je komunikacija pozitivna lastnost, ki pripomore k boljšemu rezultatu, smo drugi ekipi povečevali radij komunikacije. Rezultat je prikazan na sliki 5.11, kjer vidimo, kako se odreže ekipa agentov, ki lahko napada. Simulacije smo opravili na zemljevidu zaprtega in odprtega tipa. V kolikor lahko agentje ene ekipe napadajo, drugi pa ne, ima napadajoča ekipa prednost. To vidimo na grafu v točki *0 polj*, kjer se ekipa, ki lahko napada, nahaja nad prekinjeno črto, kar pomeni, da je ta ekipa boljša od nasprotne ekipe, ki ne napada in niti ne komunicira. Napadalna ekipa se pri zaprtem zemljevidu odreže veliko bolje, kot pri odprtem. Ko omogočimo nasprotnim agentom komunikacijo v radiju enega polja, se rezultat za ekipo, ki napada, poslabša. Pri zemljevidu zaprtega tipa se napadajoča ekipa še vedno odreže bolje kot nasprotna ekipa, pri odprtem pa je situacija obrnjena in so agentje, ki komunicirajo, že v prednosti. Razlog za to lahko najdemo v dejstvu, da pri odprtem zemljevidu agentje veliko prej pridejo v radij enega polja, kot pri zemljevidu zaprtega

tipa, ki ima veliko ovir in je zato komunikacija na tako kratki razdalj manj verjetna. Pri povečanju radija komunikacije na dve polji je rezultat enak, ne glede na tip zemljevida. Napadajoča ekipa je slabša. Prag preloma je nekje med radijem enega in dveh polj pri zaprtem zemljevidu, pri odprtem pa se pojavi, ko vsaj malo povečamo komunikacijo. Ker je radij komunikacije diskretna spremenljivka, lahko rečemo, da je prag preloma pri zaprtem zemljevidu dve polji, pri odprtem pa eno polje. Iz tega sklepamo, da nam dobra komunikacija med agenti prinese boljši rezultat kot taktika napadanja nasprotne ekipe.



Slika 5.11: Primerjava dveh ekip agentov, kjer ima ena možnost napadanja, druga pa komuniciranja. Na grafu so prikazane vrednosti za ekipo, ki komunicira.

Pri vseh izvedenih simulacijah smo podali povprečne vrednosti števila enot pobrane hrane. V povprečju je bil standardni odklon 0.7 enote hrane. Ta je posledica nekaterih izrojenih rezultatov simulacije, kjer se je določena lastnost agentov bolje izrazila ter tako imela večji vpliv na rezultat. Ker lahko ekipa agentov pobere le celo število enot hrane, se vpliv določene lastnosti (sploh, če je ta manjši) ne bo pokazal v vsakem primeru, v nekaterih primerih pa bo ravno zaradi te lastnosti ekipa pridobila kakšno enoto hrane več.

5.3 Diskusija

Sedaj, ko smo opravili zgoraj opisane simulacije, lahko analiziramo rezultate in podamo najuspešnejšo strategijo iskanja hrane v našem simulacijskem okolju.

Primerjava preiskovalnih algoritmov nam je pokazala, da ta ni tako pomemben, če je čas simulacije dovolj velik. V kolikor simulacijsko okolje dopušča velik reakcijski čas agenta in agentov algoritem poišče optimalno pot do izbranega polja, je vseeno, kateri algoritem izberemo. V primeru, da je čas premika kratek oz. je hitrost simulacije visoka, je bolje izbrati hitrejši algoritem. Tako situacijo smo lahko videli pri simulaciji, kjer sta ekipi uporabljali Dijsktro in A*. S povečanjem časa premika, se je algoritem Dijkstra približal algoritmu A*. Pri simulaciji ekip z optimalnim preiskovanjem in algoritmom A* smo videli vpliv različnih tipov zemljevidov na sam rezultat. Na zemljevidu zaprtega tipa se je optimalno preiskovanje odrezalo bolje kot pri odprtem. Razlog za tako delovanje poiščemo v razporeditvi hrane, ki je bila v tem primeru centrirana. V naslednji simulaciji imamo hrano porazdeljeno naključno, kar postavi algoritem optimalnega preiskovanja v prednost. Vzrok za to je izbira naslednjega premika, ki pri optimalnem preiskovanju predstavlja polje, s katerim bomo pridobili največ preiskanega območja. Agent s takim preiskovanjem bo veliko prej preiskal prostor in posledično prej poiskal hrano.

Komunikacija med agenti se je pokazala kot izrazito pozitivna lastnost. Slabo se je izkazala samo v primeru, kjer je bila hitrost simulacije problematična. Sama komunikacija zahteva določen čas na strani agenta, kar v primeru, da nasprotnik uporablja hitrejši algoritmom preiskovanja, lahko predstavlja neželen učinek na uspešnost agenta. V primeru, kjer sta ekipi vsebovali agente z istim načinom preiskovanja, je ekipa, ki je komunicirala, imela prednost. Ta prednost se je višala z radijem komuniciranja, vendar le do radija treh polj, kjer je dosegla maksimalno vrednost.

Vpliv števila agentov v ekipi je bila naslednja stvar, ki smo jo hoteli s simulacijo izmeriti. Število agentov nedvomno vpliva na boljši rezultat ekipe,

saj lahko ekipa z več agenti v enem življenju agenta prinese v bazo več hrane. Simulacija ekip z istim iskalnim algoritmom je pokazala, da se rezultat ekipe izboljšuje glede na povečanje števila agentov in ne glede na tip zemljevida. Še enkrat smo opravili enako simulacijo, le da je tokrat ekipa petih agentov lahko komunicirala. Hoteli smo preveriti, ali bo imela komunikacija kakšen vpliv pri večjem številu agentov nasprotne ekipe in kakšen bo ta vpliv. Izkazalo se je, da vpliv ima, vendar se ta spreminja glede na tip zemljevida. Pri zaprtem zemljevidu je število pobrane hrane ekipe, ki komunicira, večje kot pri odprtem zemljevidu. Primerjava simulacije, kjer nasprotnik lahko komunicira in simulacije, kjer ne more, glede na tip zemljevida pokaže, da je pri odprtem zemljevidu razlika očitna le do stanja, ko nasprotnikova ekipa šteje 8 agentov. Komunikacija torej pomaga le dokler nasprotnik ni preštevilčen, nato pa je razlika zanemarljiva. Zaprt zemljevid pokaže drugačno stanje. Tu je razlika ves čas konstantna, ne glede na povečevanje števila agentov. Razlog lahko poiščemo v premajhnem številu agentov - pri simulaciji je imela nasprotna ekipa največ 12 agentov. To pomeni, da je pri zaprtem zemljevidu in številčnejšem nasprotniku komunikacija ključna lastnost za konkuriranje drugi ekipi.

Napadanje nasprotnih agentov je ena od strategij, ki smo jo preizkusili s simulacijo. Izkazalo se je, da je enakovredna komunikaciji z radijem enega polja pri odprtem zemljevidu in radijem dveh polj pri zaprtem. Če ima agent prve ekipe iste lastnosti kot agent druge, le da nima možnosti napadanja, je agent druge ekipe uspešnejši. Na prvi pogled je rezultat presenetljiv, saj ko agent napade (se zaleti v) drugega agenta, sta oba agenta izločena iz igre, kar postavlja oba v enak položaj. Agent se lahko odloči za napad samo v primeru, da še ni našel hrane. Napad je realiziran s sledenjem drugemu agentu, da pa je napad uspešen, se mora zasledovan agent premikati počasneje od agenta. To se lahko zgodi le v primeru, da zasledovan agent nosi hrano in sta algoritma iskanja pri obeh ekipah enaka. V simulaciji se napad na nasprotnega agenta, ki ne nosi hrane, prikaže kot navadno sledenje nasprotnemu agentu. V kolikor pa ima ta agent hrano, se prikaže kot trk med dvema agentoma. Iz dejstva,

da se zaletimo le v primeru, ko ima nasprotni agent hrano, izhaja prednost napadalne ekipe, saj napadalec izgubi le življenje, napadeni pa tudi hrano.

Iz pridobljenih simulacijskih rezultatov smo določili pomembnost določene lastnosti agenta. Kot najpomembnejša se je izkazala številčnost, saj na njeno uspešnost skoraj ne moremo vplivati s simulacijskimi parametri (hitrost, tip zemljevida...). Druga najpomembnejša je radij komunikacije. S samo komunikacijo, oziroma z dovolj velikim radijem komunikacije lahko v skoraj vseh primerih izboljšamo ekipni rezultat. Naslednja lastnost je eliminacija, ki ima majhen vpliv na rezultat simulacije. Zadnja lastnost, ki nam ostane, je iskalni algoritem. Ob predpostavki, da je za premik na voljo dovolj časa (korak simulacije je dolg), je preiskovalni algoritem v veliki meri nepomemben. V nasprotnem primeru, kjer je korak simulacije kratek, pa lahko postavimo izbiro algoritma iskanja po pomembnosti na prvo mesto. Za konec ne smemo pozabiti na način maksimizacije preiskanih polj, ki nam skupaj z enim od preiskovalnih algoritmov v večini primerov izboljša rezultat iskanja.

Poglavje 6

Zaključek

V tem diplomskem delu smo si ogledali simulacijsko okolje *GridLand*, opisali njegovo delovanje in ga nadgradili. Okolju smo dodali podporo za izvedbo simulacij, s katerimi smo primerjali agente z različnimi lastnostmi.

Implementirali smo agenta, ki se lahko vključi v to simulacijsko okolje in deluje kot avtonomna entiteta. Za agenta smo razvili vrsto nastavljivih lastnosti, kot so komunikacija, napadanje in način iskanja. Podrobno smo si ogledali štiri iskalne algoritme, ki so primerni za iskanje poti v prostoru in dva od teh tudi implementirali.

Na koncu smo opravili primerjave različnih lastnosti agentov. Primerjali smo, kako se ekipa agentov z določenim iskalnim algoritmom izkaže proti ekipi z drugim algoritmom. Raziskali smo, kakšen je vpliv komunikacije na delovanje agentov, koliko ta izboljša rezultat glede na vrsto simulacijskega zemljevida ter kako komunikacija skupaj z določenim algoritmom preiskovanja vpliva na končen rezultat. Opravili smo tudi simulacije, ki so pokazale, kako večje število agentov vpliva na potek igre in kako lahko z medsebojno komunikacijo agentov do določene mere ublažimo številčno prednost nasprotne ekipe. Za konec smo opravili simulacijo z napadalno ekipo agentov ter ugotovili, kakšen prispevek ima ta lastnost na končen rezultat. V diskusiji smo poskusili utemeljiti dobljene rezultate simulacij in razvrstiti implementirane lastnosti agenta po pomembnosti, glede na prispevek k rezultatu.

V delu smo ugotovili, da s povečanjem števila agentov v največji meri pozitivno vplivamo na uspešnost pridobivanja vira. Potrdili smo hipotezo, da ima komunikacija pozitiven vpliv in da povečanje radija komunikacije ugodno vpliva na uspešnost zagotovitve virov. Pokazali smo, da lahko s pomočjo metode maksimizacije nepreiskanih polj (optimalnega preiskovanja) povečamo uspešnost agenta in da vrsta okolja in način razporeditve virov vpliva na delovanje te metode. S simuliranjem različnih ekip agentov smo ugotovili, da je vpliv lastnosti eliminacije zelo podoben vplivu komunikacije z radijem enega polja.

6.1 Nadaljnje delo

Nadaljevanje tega diplomskega dela se lahko usmeri v izboljšanje komunikacije med agenti ali pa v implementacijo kakšnega drugega iskalnega algoritma oziroma načina preiskovanja prostora.

Medagentno komunikacijo bi lahko uporabili še za sporočanje trenutnih ciljev, s katerimi bi verjetno lahko dosegli bolj porazdeljeno preiskovanje prostora. Komunikacijo bi lahko uporabili za izogibanje trkom z drugimi agenti, oziroma za koordinirano premikanje v primeru premikanja po zelo ozkem prostoru. Prav tako bi bile zanimive primerjave, koliko te izboljšave pripomorejo k boljšemu ekipnemu rezultatu.

Z uporabo katerega drugega iskalnega algoritma (na primer Real-Time A*), bi izboljšali uspešnost agenta v hitrejšem simulacijskem okolju. Z algoritmom za iskanje suboptimalne rešitve bi lahko tudi preverili, kolikšen vpliv na delovanje ima iskanje točne rešitve za nek premik v primerjavi z neko suboptimalno rešitvijo. Pozornost bi bilo vredno usmeriti tudi v sam način raziskovanja prostora, tako da bi izboljšali predstavljen način optimalnega raziskovanja, kjer bi s pomočjo medagentne komunikacije usklajevali porazdeljeno preiskovanje prostora.

Literatura

[1] L. Čehovin. (2011). Gridland - Simple grid-world simulation environment for distributed systems [Online]. Dostopno na: https://github.com/lukacu/gridland

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs", v *Numerische Mathematik 1*, 1959, str. 269-297.

[3] P. E. Hart, N. J. Nilsson, B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Transactions on Systems Science and Cybernetics SSC4*, str. 100-107, 1968.

[4] B. Horling, V. Lesser in R. Vincent, "Multi-Agent System Simulation Framework", *16th IMACS World Congress 2000 on Scientific Computation, Applied Mathematics and Simulation*, EPFL, Lausanne, Switzerland, 2000.

[5] S. Koenig, M. Likhachev, Y. Liu in D. Furcy, "Incremental Heuristic Search in Artificial Intelligence", *Artificial Intelligence Magazine*, št. 25, zv. 2, str. 99-112, 2004.

[6] S. Koenig, M. Likhachev and D. Furcy, "Lifelong Planning A*", *Artificial Intelligence Journal*, št. 155, zv. 1–2, str. 93–146, 2004.

[7] R. E. Korf. (1996). *Artificial Intelligence Search Algorithms* [Online]. Dostopno na: http://lvk.cs.msu.su/ bruzz/articles/IR/korf96artificial.pdf

[8] R. E. Korf, "Real-Time Heuristic Search", *Artificial Intelligence*, št. 42, zv. 2-3, str. 189-211, 1990.

[9] M. Wooldridge, *An Introduction to MultiAgent Systems - Second Edition*, Liverpool: John Wiley, 2009, pogl. 6.