SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Vunak

AUTOMATSKO PLANIRANJE U VIŠEAGENTNOJ OKOLINI

DIPLOMSKI RAD

Varaždin, 2018.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Vunak

Matični broj: 44492/14–R

Studij: Baze podataka i baze znanja

AUTOMATSKO PLANIRANJE U VIŠEAGENTNOJ OKOLINI

DIPLOMSKI RAD

Mentor/Mentorica:

Izv. prof. dr. sc. Markus Schatten

Varaždin, srpanj 2018.

Tomislav Vunak

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

U ovome diplomskom radu napravit će se uvod u višeagentne sustave te će se objasniti pojmovi poput agenata i višeagentnih sustava, isto tako napraviti će se usporedba definicija različitih pojmova prema različitim autorima. Primjerice definicije, agenata i višeagentne okoline. Nakon toga slijedi pojašnjavanje domene automatskog planiranja i povezanosti i različitosti s drugim znanostima i znanstvenim disciplinama. Nakon upoznavanja s glavnim problemom slijedi teoretsko objašnjavanje algoritama koji se mogu koristiti za automatsko planiranje. Iza toga slijedi opisivanje tehničke dome gdje ćemo se upoznati s alatima koji su korišteni za izradu ovog diplomskog rada. Na samom kraju nalazi se zaključak s osvrtom na sve što je napravljeno u ovom radu.

Ključne riječi: umjetna inteligencija; planiranje; automatsko planiranje; agenti; višeagentni sustavi; Python; SPADE;

Sadržaj

1. Uvod

U uvodu upoznati ćemo se s pojmovima poput umjetne inteligencije, agenata, višeagentnih sustava. Nakon toga ćemo se detaljnije upoznati s tim pojmovima te poviješću umjetne inteligencije, značenjem planiranja i automatskog planiranja i na kraju ovog diplomskog rada biti će objašnjeni primjeri agenata te algoritma koji se koristi za automatizaciju planiranja.

Agenta možemo definirati kao računalni sustav koji je u stanju samostalno djelovati u skladu s ciljevima svojeg korisnika ili vlasnika, pri čemu samostalno određuje načine na koje treba zadovoljiti postavljene ciljeve, umjesto da mu se načini kontinuirano objašnjavaju. Kod ove definicije želi se istaknuti agentova podređenost vlasniku i autonomnost samih agenata.

Višeagentni sustavi, prema Jacquesu Ferberu, su sustavi koji se sastoje od okoline, objekata i agenata, odnose između svih entiteta, skup operacija koje entiteti mogu izvršiti i promjene u vremenu i prostoru koje mogu nastati kao posljedica tih akcija. Bitno je istaknuti da od navedenih entiteta koji se nalaze u višeagentnim sustavima jedino agenti su oni koji mogu djelovati. (Ferber, 1999)

Na početku autor želi pojasniti pojam namjernog djelovanja (engl. *deliberative acting*) u kontekstu višeagentnih sustava. Za početak ćemo promotriti riječi namjerno i djelovanje. Riječ namjerno je prilog koji ima korijen iz imenice namjera. Imenica namjera prema definiciji s Hrvatskog jezičnog portala predstavlja: ono što je tko odlučio učiniti, zamisao koja će se ostvariti radnjom ili činom. Druga riječ koju ćemo pojasniti je djelovanje. Djelovanje je glagolska imenica koja je nastala iz glagola djelovati. Prema definiciji iz istog izvora, djelovanje znači učinak nečega, ono što proizvodi primjena lijeka, kemijskog spoja, stroja, čijih riječi itd. (Hrvatski jezični portal, 2018)

Namjerno djelovanje sastoji se od donošenja odluka koje radnje je potrebno poduzeti i kako te radnje napraviti kako bi se ostvario cilj. Odnosi se na proces zaključivanja prije i tokom ponašanja, tj. izvršavanje neke radnje. Kako bi si ovo bolje predočili možemo si postaviti sljedeća pitanja:

- Ako agent poduzme neku radnju, koji će biti rezultat?
- Koje radnje treba poduzeti agent i kako agent treba izvesti odabrane radnje kako bi dobio željeni učinak?

Takvo zaključivanje omogućuje agentu da predvidi, odluči što i kako napraviti i kako povezati nekoliko akcija kako bi se ostvario cilj. Zaključivanje se sastoji od korištenja modela za

predviđanje okoline i sposobnosti simuliranja što će se dogoditi ako agent poduzme neku radnju.

Djelovanje je riječ koja se koristi kada se želi opisati nešto što agent radi, poput vršenja određene sile, kretanja, percepcije ili komunikacije s ciljem da napravi promjenu u vlastitoj okolini i svoje stanje. Za agenta možemo reći da je entitet koji je u stanju napraviti interakcije s svojom okolinom. Agenti se ponašaju namjerno kako bi ostvarili zadane ciljeve. Zadane ciljeve postižu tako što poduzimaju određene radnje.

Razumijevanje namjernog djelovanja je cilj za većinu kognitivnih znanosti. Ono što je specifično za umjetnu inteligenciju je stvaranje modela, putem računalnog pristupa, koji nam omogućuju da objasnimo i stvaramo modele sposobnosti. Tehnološka motivacija za davanje namjernog djelovanja umjetnom agentu je sljedeća:

- autonomnost, što znači da agent obavlja određene operacije bez da ga netko kontrolira ili navodi
- raznovrsnost u zadacima koje agent obavlja i okolinama u kojima se ti zadaci odvijaju

Bez autonomnosti, agentom bi upravljao netko udaljenim pristupom. Za primjer možemo uzeti pristupanje udaljenoj radnoj površini (engl. *remote desktop*) gdje je moguće sa svojega računala pristupiti i upravljati drugim računalom. S druge strane, imamo autonomne sustave koji su specificirani za određenu okolinu u kojoj se nalaze. Primjerice, roboti koji se koriste za bojanje automobila u tvornicama. Kod drugih sustava ne postoji raznovrsnost kod okoline, već je okolina isključivo ono što su tvorci imali na umu, a obično je to vrlo specifična domena. Ovdje dolazimo do problema jer se autonomni agent mora suočiti sa širokim spektrom zadataka, okolina i interakcija kako bi postigao svoj cilj, a za to će mu biti potrebno namjerno djelovanje. Primjene ovakvih agenata možemo pronaći kod osobnih robota, simulacijskih video igara itd.

Autonomnost, raznovrsnost u zadacima i okolinama te potreba za razmatranjem nisu binarna svojstva koja imaju vrijednosti da ili ne. Umjesto toga, što su više potrebe za autonomnosti i raznovrsnosti to je veća potreba za razmatranjem. Ovo se ne odnosi samo na sustave umjetne inteligencije, već primjere možemo naći u biologiji jer vrste kod kojih se događaju veće promjene u okolini moraju se prilagoditi promjenama koje su nastale za što im nužno potrebno neki oblik razmatranja, kako bi opstale. (Ghallab, Nau, Traverso, 2016)

2. Umjetna inteligencija

Ovo poglavlje će biti posvećeno umjetnoj inteligenciji (engl. *artificial intelligence*). Ovdje ćemo pojasniti što je umjetna inteligencija, na kojima temeljima počiva umjetna inteligencija i nabrojati ćemo neke važne točke u njenom razvoju.

Prema Lugeru, umjetna inteligencija se može definirati kao grana računalne znanosti koja brine o automatizaciji inteligentnog ponašanja. Problem kod ove definicije proizlazi iz činjenice da inteligencija sama po sebi nije dovoljno jasan i definiran pojam. Iako većina ljudi može prepoznati inteligentno ponašanje, vrlo je teško definirati ga. Posebice ako uzmemo u obzir da želimo definirati inteligenciju na način da bude dovoljno konkretan da je moguće evaluirati računalni program, a zadržati u obziru vitalnost i složenost ljudskog uma. (Luger, 2009)

Autor Russel i Norvig ne daju svoju konkretnu definiciju umjetne inteligencije već se pozivaju na druge autore i njihove definicije umjetne inteligencije. U svome primjeru navode osam definicije koje su podijeljene u četiri kategorije: ljudsko razmišljanje (engl. *thinking humanly*), ljudsko ponašanje (engl. *acting humanly*), racionalno razmišljanje (engl. *thinking rationally*) i racionalno ponašanje (engl. *acting rationally*). Podjelom definicija želi se postići sinteza definicija i pokazati kako postoje različita razmišljanja ljudi oko definiranja umjetne inteligencije. Naveli smo kategorije, a sada ćemo navesti po jednu definiciju za svaku od navedenih kategorija:

- ljudsko razmišljanje: „Uzbudljivi novi pokret koji želi napraviti računala koja misle… *računala s umom* u punom i doslovnom smislu “ (prema: Haugeland, 1985)
- ljudsko ponašanje: „Proučavanje kako napraviti računala da rade ono u čemu su ljudi trenutno bolji“ (prema: Rich and Knight, 1991)
- racionalno razmišljanje: „Proučavanje izračuna koji mogu omogućiti opažanje, zaključivanje i ponašanje“ (prema: Winston, 1992)
- racionalno ponašanje: „Umjetna inteligencija … se usredotočuje na inteligentno ponašanje u artefaktima“ (prema: Nilsson, 1998)

Kod razdvajanja ljudskog i racionalnog ne želi se reći da se ljudi ponašaju iracionalno, ali moramo imati u vidu da primjerice, svi igrači koji igraju šah nisu velemajstori te da svi učenici koji sudjeluju na nekom testu neće nužno dobiti odličnu ocjenu iz njega.

Problem kod definiranja umjetne inteligencije je što se u procesu definiranja postavljaju neka pitanja na koja ne postoji konkretan odgovor. Sada ćemo navesti primjere nekih pitanja. Što se točno događa za vrijeme učenja? Što je kreativnost? Što je intuicija? Što je samosvjesnost i koja je uloga samosvjesnosti kod inteligencije? Da li je uopće moguće

postići inteligenciju na računalu ili je inteligencija nešto što može biti vezano jedino uz živa bića? Nažalost, niti na jedno navedeno pitanje nije još moguće odgovoriti, ali sva ta pitanja su utjecala na razvoj umjetne inteligencije koje poznajemo.

Umjetna inteligencije se oduvijek bila više usredotočena na proširenja mogućnosti u računalnoj znanosti nego što se vodilo računa o strogom definiranju njezinih ograničenja. (Russell, Norvig, 2010)

2.1. Povijest umjetne inteligencije

U idućim odlomcima osvrnuti ćemo se na važna razdoblja u povijesti koja su obilježila umjetnu inteligenciju kakvu danas poznajemo. Opisana će biti moderna povijest umjetne inteligencije i preskočene će biti teme poput Diferencijskog stroja Charlesa Babbagea ili osnova Booleove logike.

2.1.1. Počeci umjetne inteligencije (1943 - 1956)

Prvo značajno djelo koje autor želi izdvojiti je „A Logical Calculus of the Ideas Immanent in Nervous Activity“ autora Warrena McCulloch i Waltera Pittsa koje datira iz 1943. godine. To djelo temelji se na trima glavnim vodiljama: poznavanju osnova psihologije i ulozi neurona u mozgu, formalnoj analizi propozicijske logike i Turingovoj teoriji računanja. (Russel, Norvig, 2010) 1950. godine Alan Turing je napisao jedno od najranijih djela u kojemu ispituje računalnu inteligenciju, iako je poznatiji po svome doprinosu teoriji računarstva, razmatrao je pitanje mogu li ili ne strojevi razmišljati. Turingovo djelo ima naslov „Computing Machinery and Intelligence“. Alan Turing u svome djelu iznosi argumente za i protiv teze o mogućnosti kreiranja inteligentnog računalnog stroja. Isto tako, predlaže da se na pitanje inteligencije pokuša odgovoriti dobro definiranim testom koji je danas poznat kao Turingov test. Turingov test spada među ponajbolje testove za mjerenje performansi inteligentnih strojeva i računala. U testu sudjeluju dva čovjeka i računalo tj. stroj. Svaki od njih se nalazi u zasebnoj sobi i nisu u mogućnosti vidjeti se i komunicirati. Jedan od ljudi ima ulogu ispitivača i njemu nije poznato u kojoj se sobi nalazi računalo, a u kojoj sobi je drugi živući sudionik. Ispitivač postavlja pitanja putem terminala ili nekim drugim putem i na njemu je da postavlja pitanja. Ako ispitivač nije u stanju razlikovati odgovore na svoja pitanja između ostalih sudionika i ukoliko ne može zaključiti koje odgovore je dalo računalo ili stroj, Turingov test kaže da o tome računalo možemo govoriti kao o inteligentnom računalu. Odvajanjem sudionika u zasebne sobe otežava se posao ispitivaču jer bi inače mogao lako odrediti tko je tko. No, ispitivač može postaviti bilo koje pitanje tako primjerice može postaviti složeno računsko pitanje i računalo bi tu moglo biti u vrlo velikoj prednosti jer bi puno prije

čovjeka izračunalo rješenje zadatka i time odalo svoj identitet. Isto tako, ispitivač može postaviti pitanje kako im se dojmio neki ulomak književnog djela ili pjesme, što bi značilo da računalo mora imati vrlo razvijen emotivni pristup prema dobivenim zadacima. Unatoč dobrim stranama Turingovog testa, postoje i zamjerke. Jedna od najvažnijih zamjerki je što je to čisti test koji se bavi rješavanjem određenog problema. Test ne propituje percepcijske vještine iako su to važne komponente ljudske inteligencije. Nadalje, Turingovim testom se računalna inteligencija želi uklopiti u ljudske okvire, što nije nužno dobra osobina. Primjerice, da li stvarno želimo imati računalo kojemu će trebati vremena koliko i čovjeku da napravi složeni matematički izračun? (Luger, 2009)

2.1.2.Konferencija u Dartmouthu i zlatne godine (1956 - 1974)

Konferencije u Dartmouthu je održana u ljeto 1956. godine i uvelike je utjecala na razvoj umjetne inteligencije. (McCarthy et al., 1955) Na konferenciji je sudjelovalo deset ljudi i trajala je dva mjeseca. Među sudionicima su bili profesori s uglednih sveučilišta poput Harvarda, MIT-a, ali i pripadnici iz velikih kompanija poput IBM-a ili Bell labsa. (Russel, Norvig, 2010) Zanimljivo je da termin umjetna inteligencija je skovan u pozivu na ovu konferenciju iz 1955. godine i osmislio ga je John McCarthy koji je u to vrijeme bio profesor matematike na Sveučilištu u Dartmouthu. (McCorduck, 2004) Na toj konferenciji posebno su se istaknula dvojica istraživača s Carnegie Techa, današnjeg Sveučilišta Carnegie Mellon, njihova imena su: Allen Newell i Herbert Simon. Njih dvojica su predstavili svoj program za zaključivanje koji se zvao *Logic Theorist* ili skraćeno LT. (Russel, Norvig, 2010) LT se smatrao izvanrednim postignućem, koje je ostavilo svijet u čudu tako što je bio u stanju napraviti dokaze logičkih teorema, a to je zadatak koji nedvojbeno iziskuje inteligenciju i kreativnost. (Buchanan, 2005)

Umjetna inteligencija je doživjela vidljivi napredak jer se tek nekoliko godina ranije smatralo da je jedina primjena računala kod aritmetike, ali to treba uzeti s razumijevanjem jer su u to vrijeme ljudi raspolagali s vrlo skromnim računalima i alati za programiranje su bili vrlo primitivni po današnjim standardima. Isto tako, u to vrijeme se razmišljalo da stroj tj. računalo nije sposobno napraviti mnogo toga. U nastavku su nabrojeni nedostaci računala prema Alanu Turingu, tj. sve ono za što računalo neće biti sposobno: biti ljubazno, snalažljivo, lijepo, prijateljsko, pokrenuti inicijativu, imati smisao za humor, razlikovati dobro od lošeg, raditi greške, zaljubiti se, uživati u jagodama sa šlagom, zavesti nekoga, učiti iz vlastitog iskustva, koristiti ispravno riječi, biti subjekt vlastitog razmišljanja, imati jednako širok spektar ponašanja kao čovjek i napraviti nešto sasvim novo. Neke od ovih tvrdnji su se ispostavile kao netočne ili u najmanju ruku netočne. Primjerice, nedavno je jedan hrvatski davatelj komunikacijskih usluga korisnicima nije bio u mogućnosti isporučiti mobilni internet

na području cijele Republike Hrvatske. Nadalje, gotovo sigurno smo se susreli s nekom web stranicom koja je privremeno bila nedostupna zbog različitih razloga, kod računala postoji i mogućnost hardverskih grešaka primjerice rijetko će se netko začuditi ako mu nakon desetak godine prestane raditi tvrdi disk ili grafička kartica. Naravno, kod ovih primjera postoji mogućnost ljudskog faktora i vanjskih utjecaja na računalo, ali možemo reći da i kod računala postoji mogućnost pogreške. Isto tako, stručnjak za računalni šah David Levy predviđa da će se do 2050. javljati pojava da se čovjek zaljubi u robota koji je nalik ljudima. Nadalje, računala su napravila mala, ali značajna otkrića u astronomiji, matematici, kemiji, računalnoj znanosti itd. Tim otkrićima može se osporavati da računalo nije u stanju napraviti nešto sasvim novo.

Idući iskorak u polju umjetne inteligencije je bio predstavljanje *General Problem Solver* ili skraćeno GPS-a, a njegovi autori su Allen Newell, John Clifford Shaw i Herbert Simon. Ovaj program je oponašao ljudski pristup rješavanju problema. Iako je imao vrlo ograničen broj zadataka koji je mogao riješiti ispostavilo se da GPS ima sličan redoslijed rješavanja zadataka kao i kad ljudi rješavaju iste zadatke. (Russel, Norvig, 2010) General Problem Solver je napisan u programskome jeziku IPL. Neke od njegovih najzanimljivijih karakteristika su:

- rekurzivna priroda kod rješavanja problema,
- odvajanje sadržaja problema koji se želi riješiti od tehnike kojom će se riješiti problem, a to je napravljeno kako bi se povećala općenitost samog programa,
- koristile su se dvije tehnike za rješavanje problema, a to su „means-end“ analiza i planiranje (Newell, Shaw, Simon, 1958)

U jesen 1956. godine John McCarthy seli se sa sveučilišta u Dartmouthu i počinje raditi na Massachusetts Institute of Technology ili skraćeno MIT-u. 1958. McCarthy kreira programski jezik visoke razine i naziva ga Lisp. Lisp će postati dominantan jezik na području umjetne inteligencije sljedećih trideset godina. Iste godine McCarthy objavljuje djelo *Programs with Common Sense* u kojemu opisuje program *Advice Taker*. Advice Taker je hipotetski program na koji možemo gledati kao na prvi potpuni sustav za umjetnu inteligenciju. (Russel, Norvig, 2010)

2.1.3. Prva zima umjetne inteligencije (1974 - 1980)

Prva zima umjetne inteligencije nastupa oko 1974. godine. U prijašnjem razdoblju istraživači nisu se libili davati vrlo optimistična predviđanja koja se kasnije nisu u potpunosti obistinila. Istraživači su svoj optimizam temeljili na vrlo dobrim performansama ranih sustava za umjetnu inteligenciju. Vrlo dobre performanse su bile rezultat jednostavnih problema koje

su rješavali rani sustavi za umjetnu inteligenciju, no kada su se pred njih postavili složeniji zadaci ili ako je domena bila šira od prvotno zamišljene, performanse bi bile vrlo slabe. Tako primjerice, poznata je izreka iz 1950-ih godina da će za nekih desetak godina računala moći pobijediti bilo koga u šahu. Ova tvrdnja se kasnije obistinila, ali ne u vremenskom roku u kojem su istraživači to očekivali, već nekih tridesetak godina nakon toga, kada je 1997. godine računalo *Deep Blue* porazilo svjetskog prvaka Garryja Kasparova. Prvi problemi počeli su se javljati jer su prvi programi znali vrlo malo o vlastitoj domeni. Dobar primjer za ovo je rani sustav za prevođenje koji je financirao U.S. National Research Council, a s ciljem ubrzavanja prevođenja Ruskih znanstvenih radova nakon lansiranja satelita Sputnik. Početna razmišljanja kod ovog sustava bila su da se jednostavnim sintaktičkim transformacijama iz Ruske u Englesku gramatiku i zamjenom riječi iz elektroničkog rječnika mogu točno prevesti rečenice iz jednog jezika u drugi. Činjenica je da je kod prevođenja potrebno imati više znanja o samome jeziku koji se prevodi od puke zamjene riječi, dobro poznavanje jezika može omogućiti nedvosmislenost prevedenih rečenica i razjasniti njihov sadržaj. Dobar primjer za neuspješnost ovog sustava je prijevod rečenice „Duh je jak, ali tijelo je slabo“ (engl. *the spirit is willing but the flesh is weak*) u „Vodka je dobra, ali meso je trulo“ (engl. *the vodka is good but the meat is rotten*) iz engleske verzije rečenice možemo lakše razumjeti kako je i zašto došlo do ove pogreške. 1966. godine, odbor koji je bio zadužen za nadgledanje projekata i praćenje rezultata utvrdio je da ne postoji računalo koje može prevesti neki znanstveni tekst i da se rješenje problema prevođenja ne nadzire u bližoj budućnosti.

Drugi vrlo važan problem koji se pojavio je oblikovanje problema koji su rani sustavi za umjetnu inteligenciju rješavali ili pokušavali riješiti. Princip rada kod mnogih ranih sustava za umjetnu inteligenciju bio je isprobavanje različitih koraka sve dok se ne dođe do rješenja. Ta strategija je u redu kod manjih problema i okolina u kojima nema puno objekata i ne postoji previše radnji koje je moguće poduzeti. Ovaj problem se pokušao riješiti s više procesorske snage i većim količinama memorije. Nakon što je osmišljena teorija računalne složenosti mijenja se pristup ovome problemu. 1973. godine, britanski matematičar, Sir John Lighthill na zahtjev britanskog parlamenta piše dokument koji će kasnije ostati upamćen kao *Lighthillov izvještaj*, iako je originalni naziv djela *Artificial Intelligence: A General Survey* u kojemu se kritizira područje umjetne inteligencije. Najveća zamjerka u Lighthillovom izvještaju bila je na račun odnosa postojećih sustava za umjetnu inteligenciju s kombinatornom eksplozijom. Drugim riječima, sustavi su dobro funkcionirali u malim okruženjima, ali kada su se trebali nositi s problemima koji su realni jednostavno nisu postizali očekivane rezultate. Kao rezultat Lighthillovog izvještaja ukinuta je podrška

Britanske vlade za većinu sveučilišta. Nastavili su se financirati projekti na samo dva sveučilišta.

Treći problem se pojavio zbog ograničenja na osnovnim strukturama koje su se koristile za generiranje inteligentnog ponašanja. Primjerice, Minsky i Papert u knjizi *Perceptrons* iz 1969. godine su dokazali da Perceptroni mogu reprezentirati vrlo malo, iako mogu naučiti sve što reprezentiraju.

Krajem 1960-ih i tijekom 1970-ih godina, umjetna inteligencija naglasak stavlja na ekspertne sustave koji imaju ipak užu domenu od do tada korištenih rješenja. Primjer jednog takvog programa je *DENDRAL* koji je razvijen na američkome sveučilištu Stanford. On je rezultat rada grupe stručnjaka koji su okupili kako bi riješili problem izvođenja molekularne strukture iz podataka koji bi dobili putem masenog spektrometra. Važnost programa *DENDRAL* je u tome što je to bio prvi uspješni znanjem intenzivan (engl. *knowledge-intensive*) sustav. Njegova ekspertiza ili stručnost postignuta je pomoću velikog broja pravila specijalizirane namjene. Kasnije razvijeni sustavi također su slijedili glavnu nit vodilju McCarthyjevog *Advice Takera* tj. jasno razdvajanje znanja ili skupa pravila od komponente koja se bavi izvođenjem zaključaka. Nakon ovog uspjeha znanstvenici sa sveučilišta u Stanfordu započeli su projekt *Heuristic Programming Project* čiji je cilj bio istražiti može li se i u kojoj mjeri nova metodologija ekspertnih sustava primijeniti na druga područja. Idući iskorak je razvoj sustava MYCIN koji se bavio postavljanjem dijagnoza krvnih infekcija. MYCIN je sadržavao oko 450 pravila za dijagnozu i bio je sposoban izvoditi dijagnoze poput stručnjaka iz područja medicine i vjerojatno bolje od mladih liječnika. Za razliku od prethodnog sustava gdje su pravila mogla izvoditi, kod MYCIN-a su pravila bila prikupljena putem razgovora sa stručnjacima iz područja medicine, a koji su ta znanja usvojili učenjem iz knjiga, razgovora s drugim stručnjacima i konkretnim radom u praksi. Druga velika razlika je u tome što je MYCIN imao implementirane faktore koji su odražavali nesigurnosti kod medicinskog znanja. (Russel, Norvig, 2010)

2.1.4.Suvremeni trendovi kod umjetne inteligencije (1980 - danas)

Početkom 1980-ih godina prošloga stoljeća dolazi do promjena kod isplativosti investiranja u umjetnu inteligenciju. Prvi uspješni ekspertni sustav imena R1 započeo se koristiti 1981. godine u Digital Equipment Corporation. Digital Equipment Corporation je poduzeće koje se bavilo prodajom računala, računalnih programa i računalne periferije. Program je pomagao pri konfiguraciji računala i smatra se da je od 1984 kompaniji uštedio 40 miliuna $ na godišnjoj razini. U to vrijeme počinje zanimanje velikih kompanija za ekspertne sustave, bilo da su se kompanije raspitivale oko korištenja ili pokušale implementirati neki sustav ili da su aktivno koristile neki ekspertni sustav. Paralelno s velikim

kompanijama, vlade se opet počinju zanimati za područje umjetne inteligencije, a posebno treba istaknuti Japan, Sjedinjene Američke Države i Veliku Britaniju koja ponovo počinje financirati projekte iz područja umjetne inteligencije iako je to prestala nakon Lighthillovog izvještaja. Nažalost niti jedan od tri vrlo ambiciozna projekta se ne može smatrati uspješnim. U samo nekoliko godina industrija umjetne inteligencije je narasla s nekoliko milijuna dolara 1980. godine do preko milijardu dolara 1988. godine.

Sredinom 1980-ih godina najmanje četiri različite grupe ponovo koriste back-propagation algoritam za učenje koji se prvi put koristi 1969. godine. Algoritam se primjenjuje kod mnogih problema učenja u računalnoj znanosti i psihologiji. Takozvani *connectionist* modeli inteligentnih sustava smatraju se direktnom konkurencijom simboličkih modela za koje su se zalagali Newell i Simon te logističkom (engl. *logicist*) pristupu za koji se zalagao McCarthy i drugi. Moderne neuronske mreže istraživači razdvajaju u dvije kategorije, jednoj posvećenoj kreiranju efektivnih mrežnih arhitektura i algoritama uz to pokušavaju razumjeti njihova matematička svojstva. Druga kategorija stavlja naglasak na pažljivo modeliranje empirijskih svojstava stvarnih neurona i njihovom grupiranju.

U proteklim godinama svjedočili smo revoluciji kod sadržaja i metodologije rada kod umjetne inteligencije. Danas se sve češće nastavlja raditi na postojećim teorijama, nego što se predlažu nove teorije. Nadalje, tvrdnje se danas zasnivaju na strogim teoremima ili složenim eksperimentalnim dokazima, umjesto da se zasnivaju na intuiciji. Uz to, danas se relevantnost nekog sustava pokazuje na primjenama iz realnog svijeta umjesto na pojednostavljenim primjerima.

Konačno možemo reći da se umjetna inteligencija smatra znanstvenom metodom. Kako bi se prihvatila određena hipoteza, hipoteza mora biti ispitana strogim empirijskim eksperimentima i rezultati moraju biti ispitani statističkim metodama kako bi se dokazala njihova točnost. Rad je uvelike olakšan činjenicom da je moguće ponoviti neke eksperimente tako što se nekome mogu omogućiti prava pristupa podacima i računalnom kodu.

Za vrijeme 1970-ih godina u polju prepoznavanja govora koristio se je širok spektar različitih arhitektura i pristupa. Mnogi od njih su bili *ad hoc* i krhki. Mogli su biti demonstrirana na nekoliko posebno odabranih primjera. Danas, u ovom području dominira pristup koji se temelji na skrivenom Markovljevom modelu (engl. *hidden Markov Model*). Valja izdvojiti dva svojstva skrivenih Markovljevih modela, kao prvo, baziraju se na strogoj matematičkoj teoriji. To omogućuje istraživačima da rade na nekoliko desetljeća matematičkih rezultata koji su razvijeni u drugim područjima. Kao drugo svojstvo valja izdvojiti da se modeli generiraju nad velikim setom govornih podataka. Tehnologija prepoznavanja govora i tehnologija prepoznavanja rukopisa već se koriste u mnogim industrijskim i korisničkim aplikacijama.

Kod računalnog prevođenja, tijekom 1950-ih godina postojao je početni entuzijazam za pristup koji se bazirao na slijedu riječi s modelima koji se isto koriste u teoriji informacija. Taj pristup se napušta u 1960-ima, ali se vraća u kasnim 1990-ima i od tada prevladava ovim područjem.

Bayesove mreže su formalizam koji je osmišljen kako bi se omogućila efikasna reprezentacija te zaključivanje nad neodređenostima (engl. *uncertain-knowledge*). Ovaj pristup rješava mnoge probleme sustava za zaključivanje iz 1960-ih i 1970-ih. Tim pristupom omogućeno je učenje iz iskustva i kombiniranje najboljeg od klasične umjetne inteligencije i neuronskih mreža.

Potaknuti rješavanjem podproblema umjetne inteligencije istraživači su započeli razmatrati problem potpunih agenata. Rad Allena Newella, Johna Lairda i Paula Rosenblooma na *SOAR*-u je najpoznatiji primjer potpune arhitekture agenata. Jedna od najvažnijih okolina za inteligentne agente je Internet. Sustavi za umjetnu inteligenciju postali su česti kod aplikacija koje su orijentirane na web (engl. *Web-based*) u tolikoj mjeri da se sufiks „-bot“ koristi u svakodnevnom jeziku. Štoviše, tehnologije umjetne inteligencije čine temelj mnogih internetskih alata, poput internet pretraživača i agregatora internetskih stranica (engl. *web site aggregators*). Jedna od posljedica nastojanja izgradnje potpunih agenata je realizacija prethodno izoliranih podpolja umjetne inteligencije koja će se možda trebati reorganizirati kako bi njihovi rezultati bili povezani. Posebice, široko je rasprostranjen sustav senzora, bilo da se radi o senzorima vida, sluha, prepoznavanju govora ili drugim senzorima. Senzori ne mogu isporučiti potpuno pouzdane informacije o okolini u kojoj se nalaze. Iz tog razloga, sustavi za zaključivanje i planiranje mogu biti sposobni nositi se s nedorečenošću. Druga velika posljedica agentne perspektive jest, da je umjetna inteligencija mnogo povezanija s drugim područjima poput teorije kontrole (engl. *control theory*) i ekonomije.

Unatoč ovim uspjesima, značajne osobe iz područja umjetne inteligencije poput Johna McChartyja, Marvina Minskyja, Nilsa Nilssona i Patricka Winstona izrazili su nezadovoljstvo napretkom umjetne inteligencije. Oni smatraju da umjetna inteligencija ne treba stavljati naglasak na nikad bolje verzije aplikacija koje su dobre u pojedinim zadacima poput vožnje automobila, igranja šaha ili prepoznavanja govora. Umjesto toga, vjeruju da se umjetna inteligencija treba vratiti svojim korijenima, ono što je Simon nazvao „machines that think, that learn and that create“.

Kroz 60 godina povijesti računalne znanosti naglasak je bio na algoritmima kao glavnom objektu proučavanja. Nedavni radovi iz područja umjetne inteligencije predlažu da je za mnoge probleme korisnije više pažnje posvetiti podacima, nego odabiru algoritma koji

će se koristiti. Ovo je točno zbog sve veće dostupnosti ogromnih izvora podataka (engl. *big data*). Primjerice, mogu se preuzeti milijarde riječi engleskog jezika i milijarde slika s weba te se koristiti kod istraživanja. Yarowskyjev rad se posebno ističe iz ovog područja, a govori o utvrđivanju jednoznačnog značenja riječi u rečenici. On je radio na korištenju riječi „plant“ u rečenici i otkrivanju značenja te riječi. Riječ „plant“ u engleskom jeziku može imati dvojako značenje, a može značiti tvornicu ili biljku. Yarowsky je dokazao da ovaj zadatak može biti obavljen s preciznošću iznad 96% bez označenih primjera, jer su se u prošlosti koristili označeni primjeri s algoritmima strojnog učenja (engl. *machine learning*). Umjesto toga, ako se koristi vrlo veliki skup neanotiranog teksta i rječničke definicije s dva smisla „rad, industrijska tvornica“ i „flora, biljni svijet“ rezultati prepoznavanja riječi biti će bolji. Kao drugi primjer možemo uzeti problem popunjavanja praznina na fotografijama. Recimo da želimo izrezati nekoga s grupne fotografije, pojavljuje se problem kako ispuniti pozadinu, a da fotografija izgleda što realnije. Hays i Efros u svojem radu iz 2007. godine definirali su algoritam koji prolazi kroz zbirku fotografija kako bi pronašao nešto što će se uklopiti. Performanse njihovog algoritma bile su vrlo loše kada su koristili zbirku od svega 10 000 fotografija, ali su dobili odlične performanse kada su povećali zbirku na 2 000 000 fotografija. (Russel, Norvig, 2010)

Duboko učenje (engl. deep learning) je grana strojnog učenja koja se javlja u nedavim godinama. Cilj dubokog učenja je omogućavanje da računala uče iz iskustva te da razumiju svijet u vidu hijerarhije određenog koncepta. Kada računalo prikuplja znanje iz iskustva, ne postoji potreba za čovjekom koji će unijeti informacije koja su znanja potrebna tome računalu. Hijerarhija koncepata omogućuje računalima da uče složene koncepte tako što će ih graditi od jednostavnijih koncepata. Kada bismo hijerarhije pokušali prikazati grafom, taj graf bi imao više slojeva dubine. (Goodfellow, Bengio, Courville, 2016) Metode kojima se koristi duboko učenje uvelike su poboljšale alate poput prepoznavanja govora, prepoznavanja vizualnih objekata, detekcije objekata, ali i drugih domena poput otkrivanja opojnih droga ili genomike, tj grane genetike koje se bavi proučavanjem genoma. Uz pomoć dubokog učenja moguće je otkriti složenu strukturu u kod velikih skupova podataka korištenjem back-propagation algoritma. Uz pomoć back-propagation algoritma računalo indicira kako treba mijenjati interne parametre koji se koriste za izračun reprezentacije u svakom sloju tako što se koriste parametri iz prethodnog sloja. (LeCun, Bengio, Hinton, 2015)

3. Agenti i višeagentna okolina

Povijest računalstva je obilježena s pet velikih trendova:

- sveprisutnost,
- umrežavanje,
- inteligencija,
- delegacija i
- orijentacija na ljude.

Pod sveprisutnošću se misli na pad cijene računala i procesorske snage i samim time korištenje računala gdje prije nisu bila ekonomična a ponekad uopće i zamisliva. Primjerice, danas većina ljudi posjeduje smartphone koji ima veću procesorsku snagu nego neko serversko računalo od prije dvadesetak godina.

Nekada se komunikacija računala svodila na komunikaciju s operaterom odnosno osobom koja je bila zadužena za rad s računalom. Danas, su stvari mnogo drugačije i razvoj Interneta je puno doprinio sadašnjoj situaciji. Vrlo rijetko možemo pronaći računalo koje se koristi u akademske ili komercijalne svrhe, a da nije povezano na Internet.

Treći trend koji je obilježio računalstvo je inteligencija. Pod inteligencijom se misli na sve veću složenost zadataka koja se mogla automatizirati i delegirati s ljudi na računala. Napreduje se na području izrade računalni sustava koji su u stanju rješavati zadatke koji su do nedavno bili vrlo teško rješivi.

Sljedeći trend je sve veća okrenutost prema delegaciji. Primjerice, računalima se delegiraju vrlo složeni zadaci poput upravljanja zrakoplovom. Postoji stajalište da je bolje dati računalnom sustavu da upravlja zrakoplovom nego iskusnom pilotu. Delegacija implicira da dajemo kontrolu računalnom sustavu.

Posljednji trend je okretanje od orijentiranosti prema strojevima i okretanje prema ljudima, odnosno korištenje koncepata koji su bliži ljudima nego strojevima. Ovaj trend je vidljiv u svim načinima interakcije s računalima. Primjerice, u najranijim danima računalstva se računalom upravljalo pomoću prekidača i operater je itekako dobro morao znati strukturu samog računala kako bi mogao upravljati njime. Nakon toga su se pojavili terminali koju su omogućili interakciju između operatera i računala. Takva su računala dominirala sve do 80-ih godina prošloga stoljeća kada se počinju pojavljivati računala s grafičkim korisničkim sučeljem koja su mnogo intuitivnija i olakšavaju rad s direktorijima i datotekama. (Wooldridge, 2002)

Nažalost ne postoji opće prihvaćena definicija termina agent i postoji mnogo rasprave na tu temu, ali na agenta možemo gledati kao na entitet koji je u interakciji s drugim agentima i svojom okolinom. U ovome radu gledamo na agente samo s aspekta računalstva, iako taj termin možemo pronaći i u drugim znanostima.

Prema Wooldridgeu agent je računalni sustav koji je smješten u neku okolinu i koji je sposoban napraviti autonomne radnje u toj okolini kako bi ispunio određene ciljeve. (Wooldridge, 2002) U ovoj definiciji autor želi staviti naglasak na nezavisnost agenata te njegovu orijentiranost na postizanje ciljeva.

Jacques Ferber definira pojam agenta kao materijalni ili virtualni entitet koji može djelovati, percipirati okolinu, na djelomičan način, komunicirati s ostalima agentima, autonoman je i ima vještine za ostvarivanje svojih ciljeva i tendencija. (Ferber, 1999) Kod ove definicije autor govori o prirodi agenata, bilo da su fizički ili virtualni, te navodi neke karakteristike agenata i orijentiranost na ostvarivanje ciljeva. Ako usporedimo ovu definiciju s prethodnom, možemo reći da je druga definicija konkretnija, primjerice jer navodi da agent može biti fizički entitet, ali i diskutabilna.

Prema Maessu, autonomni agenti su računalni sustavi koji stanuju u složenoj dinamičkoj okolini, osjećaju i ponašaju se autonomno u okolini te time ostvaruju skup ciljeva ili zadatke za koje su kreirani. (Maess, 1995) Ova definicija naglašava odvajanje pojma agenta, u ovom kontekstu, od ostalih područja znanosti poput biologije, psihologije ili ekonomije i smješta ga u područje računalne znanosti. Isto tako kod ove definicije želi se naglasiti autonomnost kod izvođenja zadataka te postizanje ciljeva.

Inteligentnog računalnog agenta možemo opisati kao računalni sustav koji ima fleksibilno ponašanje u svojoj okolini. Ovdje pod fleksibilnosti podrazumijevamo tri komponente koje su bitne za agente, a to su:

- reaktivnost
- proaktivnost
- društvenost

Pod reaktivnošću se misli na interakciju koju agenti ostvaruju sa svojom okolinom te da pravovremeno reagiraju na promjene u okolini. Ukoliko bi okolina bila determinirana tada agenti ne bi morali voditi brigu o uspjehu ili neuspjehu određenje radnje jer bi ishodi bili unaprijed poznati. Međutim, u većini slučajeva to nije tako. Okoline su obično podložne promjenama, neke informacije nisu potpune ili iz njih nije jasno koju radnju je potrebno poduzeti.

Kod proaktivnosti su stvari malo složenije u odnosu na reaktivnost. Pod proaktivnosti se misli da je agent aktivan u smislu da ne reagira samo na podražaje koji dolaze iz okoline već ima svoje ciljeve koje želi ostvariti. Ovdje želimo naglasiti agentovu samostalnost i preuzimanje inicijative.

Treća komponenta kod fleksibilnosti agenta je društvenost. Pod društvenosti se podrazumijeva interakcija i suradnja agenata s drugim agentima putem nekog zajedničkog jezika za komunikaciju. Bitno je spomenuti kod društvenost da je potrebno uzeti u obzir ciljeve drugih, a ne samo vlastite ciljeve.

Reaktivnost i proaktivnost se nerijetko nalaze na suprotnim stranama i potrebno je pronaći sredinu između njih. Balansiranje između reaktivnosti i proaktivnosti je još uvijek otvoreni problem.

Moramo spomenuti još neke od važnijih osobina agenata poput: mobilnosti, iskrenosti, benevolentnosti, racionalnosti i učenja. Mobilnost je sposobnost agenta da se kreće unutar mreže. Primjerice, tako agent može se izvršavati na jednoj platformi, pa se premjestiti na drugu platformu i nastaviti sa svojim radnjama. Iskrenost znači da agent neće namjerno davati lažne informacije. Pod benevolentnošću se podrazumijeva da agent neće sam spriječiti ispunjavanje vlastitih ciljeva tj. da neće imati ciljeve koji su međusobno suprotni. Racionalnost kod agenata znači da će izvršavati radnje kako bi postigao zadani cilj. Učenje je osobina agenata koja znači da se agenti s vremenom mogu prilagoditi, poboljšati i steći iskustvo. (Wooldridge, 2002)

Nakon što smo naveli definicije agenata i naveli njihova svojstva sada ćemo gledati na agenta na konceptualnoj razini. Kako bismo pojednostavili ponašanje agenta reći ćemo da je agent u interakciji s okolinom i drugim agentima. U tom pojednostavljenom scenariju agent se sastoji od dvaju modula: funkcija razmatranja i platforme za izvršavanje.

Funkcije razmatranja predstavljaju implementaciju zaključivanja koje je potrebno kako bi se odabrale, organizirale i obavile radnje koje agent mora napraviti kako bi obavio svoj zadatak, reagirao na promjene koje su se dogodile u okolini i kako bi bio u interakciji s drugim agentima. Agent treba obaviti više funkcija razmatranja kako bi odabrao i obavio radnje koje će rezultirati postizanjem određenog cilja. Dvije glavne funkcije razmatranja su: planiranje i ponašanje tj. izvršavanje radnje.

Platforma za izvršavanje predstavlja motorički sustav svakog agenta. Platforma za izvršavanje pretvara naredbe u stvarne radnje. Drugim riječima platforma predstavlja doslovan pomak ruke po određenoj osi kod robotske ruke ili pomak virtualnog lika u računalnoj igri u ovisnosti od pritisnute tipke na tipkovnici ili kontroleru. Isto tako, platforma za izvršavanje pretvara signale koje je agent zadobio putem senzora ili nekim drugim

putem,primjerice programskim, u informaciju koja je poznata agentu. Primjerice, robot tako dobiva signal da se ispred njega nalazi prepreka ili agent dobiva upit da prikupi određene podatke putem interneta. (Ghallab, Nau, Traverso, 2016)

Sada ćemo navesti neke od definicija višeagentnih sustava, ali kod definiranja višeagentnih sustava postoji isti problem kao i kod definiranja agenata, odnosno, ne postoji opće prihvaćena definicija višeagentnih sustava.

Kao što smo u uvodu naveli, višeagentni sustavi su sustavi koji se sastoje od okoline, objekata i agenata, odnose između svih entiteta, skup operacija koje entiteti mogu izvršiti i promjene u vremenu i prostoru koje mogu nastati kao posljedica tih akcija. Bitno je istaknuti da od navedenih entiteta koji se nalaze u višeagentnim sustavima jedino agenti su oni koji mogu djelovati. (Ferber, 1999) U ovoj definiciji želi se naglasiti dijelovi od kojih se višeagentni sustav sastoji kao i interakcije između tih dijelova. Nadalje posebno se ističe, agentova sposobnost djelovanja gdje on predstavlja inicijatora promjena u sustavu.

Višeagentni sustavi mogu se definirati kao slabo povezana mreža entiteta za rješavanje problema tj. agenata koji rade zajedno kako bi pronašli odgovore na probleme koji su izvan njihovih individualnih mogućnosti ili iznad njihovog individualnog znanja. (Stone, Veloso, 2000) Kod ove definicije želi se istaknuti orijentiranost agenata na suradnju i rješavanje složenih zadataka kroz dekompoziciju gdje svaki agent obavlja svoju zadaću, a ta zadaća dovodi do rješenja prvobitnog problema.

4. Planiranje

Na početku ovoga poglavlja autor želi istaknuti definiciju riječi planiranje. Planiranje je glagolska imenica koja ima korijen u riječi plan. Plan je unaprijed utvrđen skup mjera kojima se predviđa ostvarenje određenih zadataka i vrijeme u kojemu ih treba izvršiti. Dok je prema drugoj definiciji plan zamisao o onome što treba izvršiti da se postigne željeni cilj. (Hrvatski jezični portal, 2018)

Planiranje je eksplicitni proces razmatranja (engl. *deliberation*). Planiranje se sastoji od odabira i organiziranja radnji kojima ćemo ispuniti zadani cilj, a kao konačni rezultat imati će plan. U tom se procesu izabire jedna ili više akcija od svih mogućih opcija. Izabiremo one akcije koje su nam potrebne za postizanje cilja. Akcije oblikujemo u strukturu koju nazivamo plan. To se izvodi na način da predviđamo ishode određenih akcija. Računalni plan je računalna proučavanje cijelog procesa planiranja.

Planiranje je jedan od najvažnijih aspekata inteligentnog ponašanja. Sposobnost da se identificiraju i odaberu odgovarajuće aktivnosti i sposobnost da se predvide posljedice tih aktivnosti je fundamentalna za ljudska bića i za inteligentne robote. (Ghallab, Nau, Traverso, 2004)

Kod planiranja sustavi se mogu klasificirati u sljedeće kategorije u ovisnosti da li se modu konfigurirati za rad u različitim domenama: planiranje specifično za domenu (engl. *domain-specific*) i planiranje nezavisno od domene (engl. *domain-independent*) Oblici planiranja nisu isključivi i može se dogoditi slučaj da ce više oblika planiranja biti korišteno. Moguće je koristiti jedan ili više oblika planiranja specifičnog za domenu na nižim razinama problema koji želimo riješiti dok ćemo na apstraktnijoj razini koristiti planiranje nezavisno od domene,ali to uvelike ovisi o problemu koji želimo riješiti.

Kada govorimo o planiranju specifičnome za domenu, govorimo o sustavima za planiranje koji su napravljeni za konkretnu domenu. Takvi sustavi vrlo vjerojatno neće raditi u drugoj domeni osim ukoliko se na njima ne provedu velike promjene. Najveći problemi koji se javljaju kod planiranja specifičnog za domenu su: troškovi i autonomnost. Troškovi predstavljaju problem jer je potrebno razviti rješenje za svaki novi problem koji nije predviđen a to iziskuje vrijeme i novac. Pristupi koji su specifični za domene zatvaraju mogućnost da govorimo o inteligentnome i autonomnom računalu jer će njegove sposobnosti biti ograničene na područja na kojima ima razvijene planere za konkretne domene, osim ukoliko stroj ili računalo ima sposobnost razviti novi pristup specifičan za domenu na temelju informacija koje je dobio u interakciji s okolinom ili drugim agentima. U sljedećim odlomcima možemo se ukratko upoznati s različitim vrstama planiranja specifičnog za domenu.

Planiranje pomaka i kretanja (engl. *Path and motion planning*) sastoji se od kretanja objekta od početne do konačne pozicije te upravljanja kretanjem na tom putu. Primjeri objekata su: robot, mehanička ruka, kamion ili neki virtualni entitet. Planiranje kretanja uzima u obzir model okoline te kinetička i dinamička ograničenja. Kod ove vrste planiranja postoje već pouzdane i efikasne metode.

Planiranje percepcije (engl. *perception planning*) predstavlja planove koji obuhvaćaju smislene radnje za prikupljanje informacija. Pojavljuje se u zadacima poput modeliranja okoline ili objekata, prepoznavanja objekata ili prepoznavanja trenutnog stanja u kojem se nalazi okolina. Planiranje percepcije postavlja pitanja poput: koje su informacije potrebne, kada si te informacije potrebne, gdje potražiti te informacije, koji senzori su najbolji za tu vrstu informacija. Prikupljanje podataka je posebna vrsta planiranja percepcije u kojoj nije naglasak na smislenim radnjama već na upitima, tj. koje upite je potrebno postaviti kako bi se dobio određeni odgovor.

Planiranje navigacije (engl. *navigation planning*) objedinjuje probleme planiranja kretanja i percepcije kako bi se postigao željeni cilj ili kako bi se istražilo određeno područje.

Planiranje operacijama (engl. *manipulation planning*) bavi se operacijama nad objektima, primjerice, kako se povezivanjem objekata sastavljaju nove strukture.

Planiranje komunikacije (engl. *communication planning*) bavi se problemima u dijalozima i suradnji između više agenata, bilo da se radio o ljudskim agentima ili o virtualnim agentima.

Planiranje nezavisno od domene predstavlja planiranje u kojem bi se sustav trebao prilagoditi bilo kojoj domeni. Glavni nedostatak ovog pristupa je što nije moguće napraviti sustav planiranja koji je nezavisan od domene, a opet da bude efikasan kao sustav koji je specificiran za tu domenu. Ovo je veliki problem jer je primjerice teško pronaći poveznice u planiranju kod igranja šaha, planiranju savijanja lima ili planiranju kretanja bespilotne letjelice kod istraživanja svemira. Jednostavno, to su suviše različite domene da bi se ista klasa planiranja mogla koristiti. Kako bi se planiranje nezavisno od domene koristilo potpuno u praksi potrebno je nametnuti skup pojednostavljenih pretpostavki koje su preveliko ograničenje za mnoge aplikacije za planiranje. No planiranje nezavisno od domene ne mora nužno isključivat planiranje specifično za domenu, već se mogu nadopunjivati i tu dolazimo do prednosti. (Nau, 2007) Planiranje nezavisno od domene oslanja se na apstraktne, opće modele radnji. Modeli radnji mogu varirati od vrlo jednostavnih koji uvelike ograničuju oblike zaključivanja do vrlo opširnih modela koji imaju mogućnosti predviđanja. Planiranje nezavisno od domene možemo podijeliti u tri kategorije: planiranje projekata (engl. Project planning), raspored i alokacija resursa (engl. Scheduling and resource allocation) i sinteza

plana (engl. plan synthesis). Planiranje projekata je oblik u kojemu je naglasak na vremenskim i prioritetnim komponentama plana; primjerice, vremena kada određena radnja može najranije ili najkasnije započeti ili da li određena radnja ima veći prioritet nego neka druga radnja. Raspored i alokacija resursa nam govori nešto više, ono obuhvaća vremenske i prioritetne komponente, ali obuhvaća resurse i ograničenja nad resursima kod svake radnje. Sinteza plana uz vrijeme, prioritet i resurse obuhvaća uvjete koji su potrebni za primjenjivanje određene radnje i rezultate koje će ta radnja proizvesti. (Ghallab, Nau, Traverso, 2004)

Ponašanje ljudi prema planiranju možemo podijeliti na dvije vrste: ponašanje s eksplicitnim planiranjem i ponašanje bez eksplicitnog planiranja. Koju vrstu ponašanja ćemo koristiti ovisi o situaciji u kojoj se nalazimo. Postoji velik broj situacija u kojima ćemo primijeniti jedno od ova dva ponašanja. Neki od primjera ponašanja bez eksplicitnog planiranja su:

- kada je svrha ponašanja trenutna,
- kada obavljamo dobro uvježbane radnje i
- kada se tijek radnje može slobodno prilagoditi.

Primjer kada je svrha ponašanja trenutna je primjerice ukoliko smo jako žedni i želimo popiti čašu vode, jednostavno ćemo ustati otići do slavine, uzeti čašu, natočiti vodu u nju i popiti ju. Za ovu radnju nam nije potrebno eksplicitni plan. Primjer za uobičajene radnje je, primjerice, vožnja automobila, ako ste vozač. Posebice, ukoliko imate mnogo godina vozačkog iskustva. Ljudi s vremenom steknu naviku i pritišću papučicu spojke prije nego što promjene stupnjeve prijenosa kod mjenjača. Drugi je primjer vožnja bicikla, ako često vozite bicikl nije potrebno razmišljati o tome da je potrebno okretati pedale ili zakretati volan tokom vožnje. Primjer za situaciju kada se tijek radnje može slobodno prilagoditi je ako smo u trgovačkom centu i kupujemo određeni predmet, naravno pretpostavljamo da u više trgovina možemo kupiti taj predmet s nekim drugim obilježjima. U toj situaciji nije potrebno planirati, jer se jednostavno možemo vratiti u trgovinu nakon što donesemo odluku.

Situacije u kojima je potrebno eksplicitno planiranje:

- kada se nalazimo u novoj situaciji,
- kada je zadatak koji je potrebno napraviti složen,
- kada su troškovi i rizici izuzetno veliki i
- kod timskog rada ili sporta.

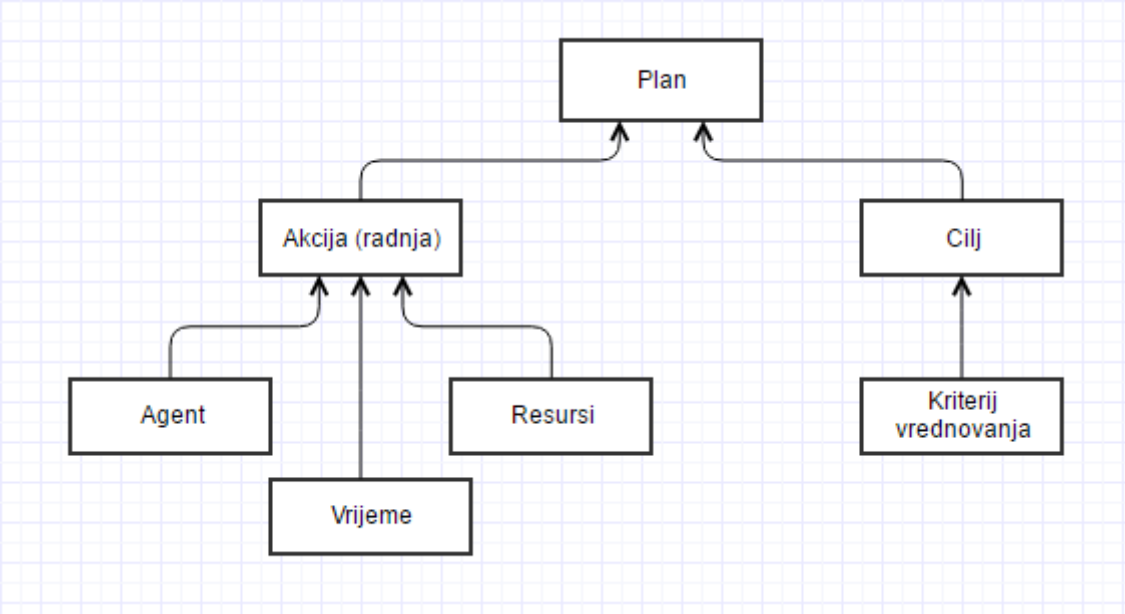
Primjer kada se netko nalazi u novoj situaciji je, kada se selimo. Potrebno je izabrati stvari koje ćemo ponijeti sa sobom, zapakirati ih da im se ne dogodi nešto, prevesti ih na

željenu lokaciju, raspakirati kada stignemo na odredište i posložiti na željeno mjesto. Dobar primjer eksplicitnog planiranja kada imamo složni zadatak je situacija kada se dizajnira novi programski proizvod, potrebno je dodijeliti ljude koji će raditi na tom projektu, potrebno je odrediti zahtjeve koje će rješenje sadržavati, tehnologije koje će biti korištene kako bi se napravilo programsko rješenje. Kad rješenje bude napravljeno, potrebno je napraviti testove te dokumentirati sve što je napravljeno kako bi se olakšale eventualne daljnje promjene na projektu. Za primjer velikih rizika možemo uzeti situaciju kada se odvijaju određeni radovi u nuklearnom postrojenju i svakako je potrebno dobro isplanirati svaku radnju kako ne bi došlo do problema, koji u ovoj situaciji mogu biti katastrofalnih razmjera. Prednosti kod timskog rada je podjela poslova tako što se jedna cjelina podjeli na manje dijelove i svakom pojedincu bude dodijeljen jedan dio. Tako se ostvaruje ušteda na vremenu, ali isto tako na troškovima. Naravno, kako bi se to dogodilo, potrebno je sve unaprijed isplanirati i dogovoriti se unutar tima koji član je zadužen za koje poslove.

Ljudi obično planiraju isključivo kada je potrebno. Najčešće se planiranje radi kada se zna da će planiranje donijeti neki oblik koristi. To proizlazi iz činjenice da je planiranje složen proces koji zahtjeva da se uloži vrijeme kako bi se napravio plan. (Ghallab, Nau, Traverso, 2016)

4.1. Konceptualni model planiranja

Kako bismo bolje mogli shvatiti što je to planiranje, potrebno je znati što je konceptualni model. Konceptualni model je teoretski zapis za opisivanje elemenata nekog problema. Neke od prednosti konceptualnog modeliranja su: objašnjavanje osnovnih koncepata, pojašnjenje pretpostavki, analiziranje zahtjeva, dokazivanje semantičkih svojstava. Primjer jednostavnog konceptualnog model možemo vidjeti na slici 4.1. Na konceptualnom modelu možemo vidjeti osnovne koncepte kod planiranja, ali i njihov međusobni odnos.



Slika 4.1 Konceptualni model

Deskriptivni modeli koji se koriste kod sustava za planiranje često se nazivaju planirajuće domene (engl. *planning domains*). Potrebno je imati na umu da planirajuća domena nije *a priori* definicija agenta i okoline nego nužno pojednostavljena aproksimacija koja mora uravnotežiti nekoliko međusobno suprotnih kriterija poput: preciznosti, računalnih performansi i biti razumljiva za krajnjeg korisnika.

Konceptualni model koji se koristi kod planiranja zove se sustav promjene stanja (engl. *state- transition system*). Kasnije ćemo se upoznati s nekima varijantama ovoga sustava koji su prilagođeni. Zanimljivo da sve varijante ovog sustava dolaze od istih autora (Ghallab, Nau, Traverso), ali iz različitih vremenskog perioda pa tako možemo vidjeti i promjene koje su se dogodile u modelu. Prvi model koji ćemo opisati datira iz 2004 godine i govori nam da je sustav promjene stanja sustav sastoji se četvorke Σ = (S, A, E, γ) gdje su:

S = $\{s_1, s_2, \ldots, s_n\}$

A = $\{a_1, a_2, \ldots, a_n\}$

E = $\{e_1, e_2, \ldots, e_n\}$

γ : S x A x E → 2^S

S predstavlja konačni ili rekurzivno beskonačan skup stanja. U skupu S nalaze se sva stanja koja su moguća u okolini. S dolazi kao akronim od engleski riječi za stanje (engl. state). Drugi član četvorke je A, koji predstavlja konačni ili rekurzivno beskonačan skup radnji ili akcija. Radnje su akcije koje agent može poduzeti kako bi promijenio stanja okoline u kojoj se nalazi. A dolazi od engleski riječi za radnju (engl. action). Treći član četvorke je E koji predstavlja skup događaja. Može biti konačan skup ili rekurzivno beskonačan. Događaji koji se događaju u okolini nisu po kontrolom agenta. Nadalje, događaji mogu promijeniti stanje u kojemu se okolina nalazi. E je akronim koji dolazi od engleske riječi za događaj (engl. event). Zadnja i najsloženija komponenta sustava je funkcija γ. Funkcija gama (γ) je zadužena za promjenu stanja sustava. Ona uzima dva faktora, stanje sustava S i radnju A ili događaj E, preciznije uzima stanja sustav S i uniju svih događaja A i svih događaja E. Rezultat funkcije promjene stanja je novi skup stanja. 2^S predstavlja skup svih mogućih stanja u kojima se sustav može nalaziti. (Ghallab, Nau, Traverso, 2004)

Važno je istaknuti kako radnje i događaji direktno utječu na promjene stanja u okolini, a razlikuju se utoliko što su radnje rezultat agentovog ponašanja dok su događaji pojave u okolini koje nisu prouzrokovane agentovim ponašanjem, već nekim drugim faktorima, primjerice uzrokovane ponašanjem drugog agenta.

Nakon što smo se upoznali s konceptnim modelima i sustavom promjene stanja. Reći ćemo kako koristiti grafove kod sustava promjene stanja. Na sustav promjene stanja možemo gledati kao na graf. Klasičan sustav promjene stanja Σ = (S, A, E, γ) može biti reprezentatiran kao usmjereni graf $G = (N_G, E_G)$. Graf se sastoji od čvorova N_G i listova E_G gdje čvorovi predstavljaju stanja sustava S, tj. $N_G = S$, a listovi predstavljaju promjene stanja, odnosno matematički zapisano $s \in N_G, s' \in N_G$ primjerice $s \rightarrow s' \in E_G$ označava $u \in (A \cup E)$ ako i samo ako je $s' \in \gamma(s, u)$.

Novija varijanta sustava promjene stanja, koja datira iz 2016 godine, kaže da se sustav promjene stanja sastoji od trojke Σ = (S, A, γ) ili četvorke Σ = (S, A, γ, cost) gdje:

- S predstavlja konačni skup stanja u kojima sustav može biti
- A predstavlja konačan skup radnji koje agent može poduzeti
- γ: S x A→ S predstavlja parcijalnu funkciju koja se naziva funkcija predviđanja ili funkcija promjene stanja. Ako je *(s, a)* u domeni *γ*, primjerice ako je *γ(s,a)* definirano, tada je *a* primjenjivo u *s*. Drugim riječima, ako vrijedi *γ(s,a)* tada možemo predvidjeti ishod, u suprotno slučaju, a nije primjenjivo u *s*.

- cost: S x A → [0,∞) je parcijalna funkcija koja ima istu domenu kao i *γ*. Iako se naziva funkcija troškova (engl. *cost*) njezino značenje je proizvoljno tj. može predstavljati doslovno novčani trošak, ali i trošak vremena ili bilo čega drugoga što želimo minimizirati. Ako funkcija troškova nije eksplicitno navedena, tada je ona jednaka jedan kada je *γ(s,a)* definirano.

Gore navedeni sustav promjene stanja zahtjeva restriktivne pretpostavke koje se nazivaju klasične pretpostavke planiranja. Postoje tri klasične pretpostavke i one su: konačna, statička okolina; nepostojanje eksplicitnog vremena i konkurentnosti; determinizam bez nesigurnosti. Sada ćemo pobliže objasniti svaku od pretpostavki:

1. *Konačna, statična okolina*. Kao dodatak zahtjevu da set radnji bude konačan, pretpostavljamo da se promjene događaju samo kao odgovor na radnje. Ukoliko agent ne vrši određenu radnju, stanje se ne mijenja. Ova pretpostavka isključuje mogućnost radnji koje poduzimaju drugi agenti ili mogućnost događaja koji nisu izazvani od strane agenata.
2. *Nepostojanje eksplicitnog vremena konkurentnosti*. Ne postojanje eksplicitnog vremena znači da ne možemo pokrenuti neku radnju u određeno vrijeme ili koliko neka radnja ili stanje vremenski traje ili bi trebalo trajati. Postoji samo diskretan niz stanja i akcija, primjerice $S = \{s_1, s_2, s_3, s_4\}\, A = \{a_1, a_2, a_3, a_4\}$
3. *Determinizam bez nesigurnosti*. Pretpostavljamo da možemo predvidjeti sa sigurnošću koje stanje će biti ako je radnja *a* poduzeta u stanju *s*. Ovo isključuje mogućnost pogrešaka ili pogrešaka kod izvođenja kao i mogućnost nedeterminističkih radnji poput bacanja para kockica, bacanja novčića ili izvlačenja karte iz špila karata.

U okolini koja ne zadovoljava klasične pretpostavke planiranja uvođenje gore navedenog modela može dovesti do pogrešaka kod agentovog razumijevanja, ali to ne znači da bismo se trebali odreći olako ovog modela u korist drugih. Pogreške kod klasičnog modela mogu biti prihvatljive ako se ne događaju često ili ako nemaju ozbiljne posljedice.

Razmislimo o računalnim aspektima ovog modela koristeći sustav promjene stanja. Ako su *S* i *A* dovoljno mali moglo bi biti izvedivo napraviti lookup tablicu koja sadrži *γ(s,a)* i *cost(s,a)* za svaki *s* i *a*, tako da je moguće dohvatiti svaki ishod radnje iz lookup tablice. Ukoliko je skup ipak prevelik da bi se navela svaka instanca *γ(s,a)*, uobičajen je pristup da se napravi generativna reprezentacija u kojoj postoje procedure za izračunavanje *γ(s, a)* za dane *s* i *a*. Specifikacija Σ može sadržavati eksplicitno navođenje jedne ili više stanja u *S*, dok se ostala stanja mogu izračunati koristeći *γ*.

Sada ćemo pokazati primjer reprezentacije koja ima specifičnu domenu, tj. one koja je napravljena za planiranu domenu. Nakon toga ćemo pokušati razviti pristup koji neće ovisiti i domeni za reprezentaciju bilo koje domene klasičnog planiranja.

Kako bismo probušili rupu u metalnoj ploči deskriptivni model bi sadržavao:

- Ime operacije i parametri, primjerice u našem primjeru bi to bile dimenzije, orijentacije itd.
- Preduvjeti operacije su uvjeti koje je nužno ispuniti kako bi se operacija izvršila, primjerice rupa treba biti okomita na ravninu, dakle, ne pod kutom različitim od 90°, ploča treba biti pričvršćena za vrijeme bušenja itd.
- Efekt operacije je ono što želimo postići, ovdje je to bušenje rupe, ali može uključivati i procjenu koliko vremena će nam biti potrebno ili koliko će nas to koštati

Prednost reprezentacija koje imaju specifičnu domenu je u tome što možemo koristit koju god strukturu podataka i algoritme koji se čine najbolji za danu planirajuću domenu. Mana reprezentacija specifičnih domena je što je za svaku planirajuću domenu potrebno razviti reprezentaciju. (Ghallab, Nau, Traverso, 2016)

4.1.1. Objekti i varijable stanja

U sustavu promjene stanja uobičajeno je da za svako stanje s iz skupa S je opis svojstava i različitih objekata u planiranoj okolini. Reći ćemo da svojstvo je strogo ako ostaje jednako u svakom stanju S te da je varirajuće ako se razlikuje između stanja u skupu S. Kako bismo reprezentirali objekte i njihova svojstva koristiti ćemo tri skupa B, R i X, uz napomenu da skupovi moraju bit konačni:

- B je skup imena za sve objekte i matematičke konstante, koje su potrebne kako bi se objasnila svojstva tih objekata. Obično ćemo skup B razdvojiti u odgovarajuće podskupove poput robota, lokacija, matematičkih konstanti i sl.
- Kako bismo reprezentirali stroga svojstva skupa Σ koristiti ćemo skup R. Svaki $r \in R$ biti će neka relacija nad B.
- Za reprezentaciju varirajućih svojstava skupa Σ koristiti ćemo skup X takav da je vrijednost svakog $x \in X$ ovisi samo o stanju s.

Koji objekti i svojstva se nalaze u skupovima B, R i X ovisi o kojem dijelu okoline planer mora donijeti zaključak. Uzmimo za primjer jednostavan zadatak otvaranja vrata u nekoj kući ili stanu, primjerice trenutni položaj ruke nam je važan kod zadatka niže apstrakcije otvori vrata, ali nam nije važan za zadatak veće apstrakcije idi i kuhinje u dnevni boravak.

Sada ćemo prikazati neka stanja u sustavu promjene stanja. Skup *B* se sastoji od dva robota, tri doka za ukrcavanje, tri kontejnera i tri hrpe tj. stogova kontejnera, bool konstanti koje imaj vrijednost istina ili laž i konstante nil:

B = roboti ∪ dokovi ∪ kontejneri ∪ hrpe ∪ booleani ∪ {nil};

Bool = {T, F};

roboti = {r1, r2};

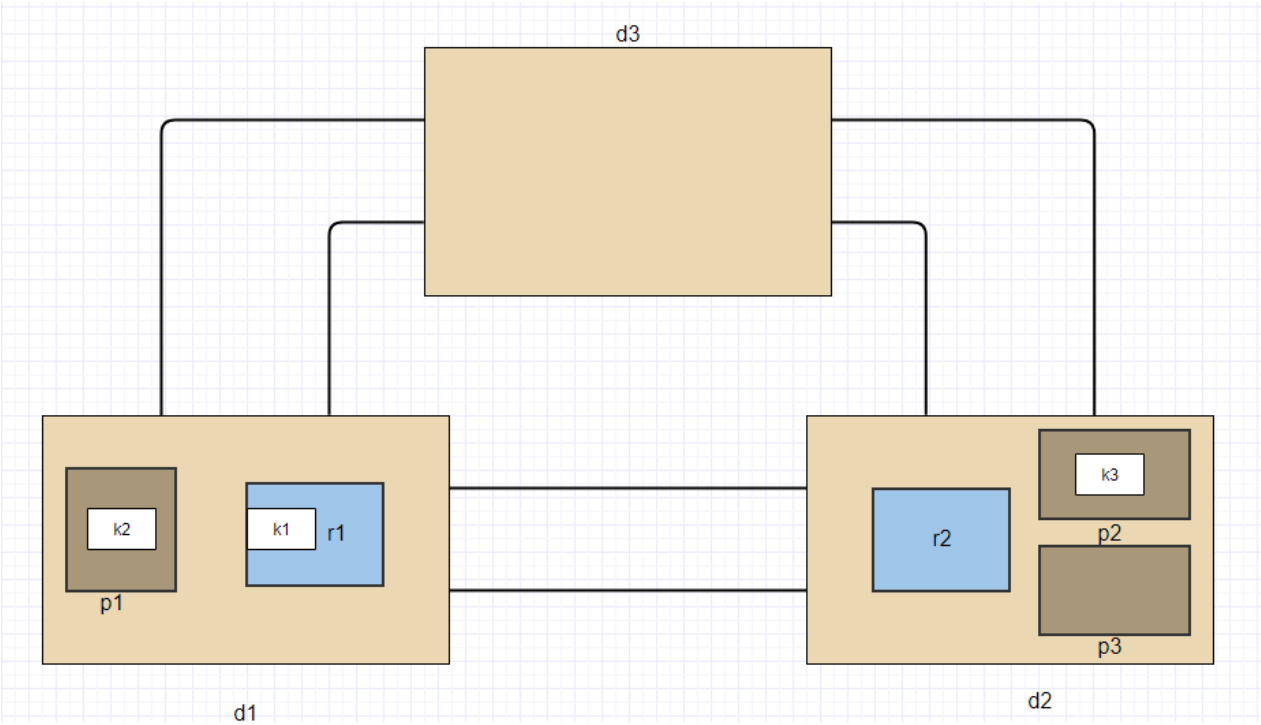
dokovi = {d1, d2, d3};

kontejneri = {k1, k2, k3};

hrpe = {h1 , h2 , h3}.

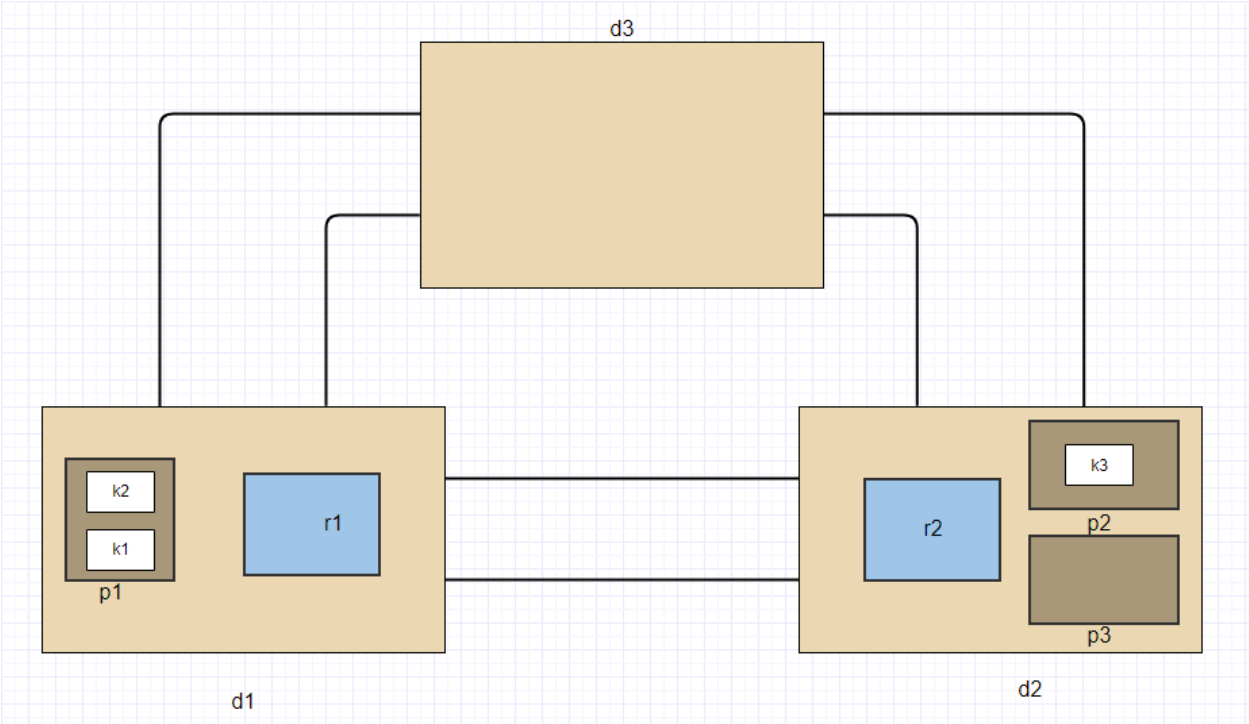
Definirati ćemo dva stroga svojstva: svaki par dokova za ukrcavanje je susjedan (engl. *adjacent*) ukoliko je cesta između njih i svaka hrpa pripada točno jednom doku za ukrcavanje. Isto tako, robot može držati samo jedan kontejner u nekom trenutku i na jednom doku može biti samo jedan robot. Napomenimo da se roboti imaju kotače te da se mogu kretati.

Na primjeru *s1* je stanje sljedeće: dok broj 3 je prazan, na doku broj dva nalaze se robot *r2* i platforme za ukrcavanje *p2* i *p3* na platformi *p2* nalazi se kontejner *k3*. Na doku *d1* nalazi se robot *r1* i on drži kontejner *k1* isto tako na doku se nalazi platforma *p1* na kojoj se nalazi kontejner *k2*. Da bismo si to mogli lakše vizualizirati možemo pogledati sliku 2.2.



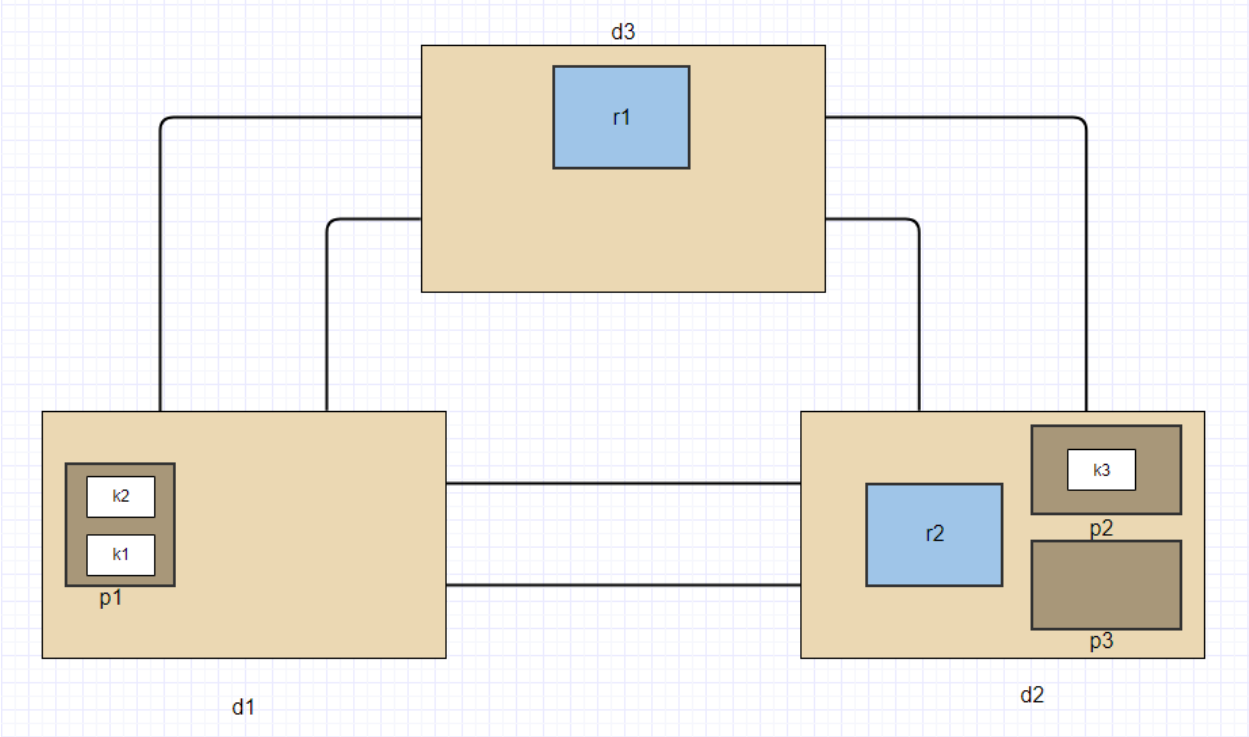
Slika 4.2 Prikaz stanja s1

Na slici 2.3 možemo vidjeti da je kontejner *k1* premješten s robota *r1* na platformu *p1*.



Slika 4.3 Prikaz stanja s0

Ukoliko bi se robot iz stanja *s0* s doka *d1* premjestio na platformu *d3* sustav bi došao u stanje *s2* koje možemo vidjeti na slici 2.4.



Slika 4.4 Prikaz stanja s2

Sustav može prijeći iz stanja *s0* u stanje *s1* ako robot *r1* podigne kontejner *k1* i obrnuto, sustav može iz stanja *s1* doći u stanje *s0* ako robot *r1* iskrca kontejner *k1*. Slično je sa stanjem *s2*, ukoliko se robot sa stanja *s0* premjesti s doka *d1* na dok *d3* sustav dolazi u stanje *s2*, vrijedi i obrat. Ovo možemo matematički zapisati pomoću uređene n-torke sustav

iz stanja *s0* uz pomoć operacije *ukrcaj(r1,k1,k2,p1,d1)* prelazi iz stanja *s0* u stanje *s1*. Obrat bi izgledao: sustav *s1* prelazi u stanje *s0* uz operaciju *iskrcaj(r1,k1,k2,p1,d1)*. Prijelaz sustava iz stanja *s0* u stanje *s2* možemo opisati pomoću operacije *pomakni(r1,d1,d3)*, a obrat tj. prelazak sustava iz stanja *s2* u stanje *s0* možemo opisati pomoću *pomakni(r1,d3,d1)*.

Kako bismo zapisali svojstva iz R = {susjedan, pored} gdje su:

Susjedan = {(d_1, d_2), (d_2, d_1), (d_2, d_3), (d_3, d_2), (d_3, d_1), (d_1, d_3)}

Pored = {(p_1, d_1), (p_2, d_2), (p_3, d_2)}

Definicija varijable stanja (engl. *state variable*) glasi: varijabla stanja iznad skupa *B* je termin

$x = sv(b_1, b_2, b_3, \dots, b_k)$

gdje *sv* predstavlja simbol koji nazivamo ime varijable stanja, a svaki b_i je član skupa *B*. Svaka varijabla *x* ima rang, $\mathrm{Rang(x)} \subseteq B$, koji predstavlja skup svih mogućih vrijednosti varijable *x*.

Zamislimo da imamo zadani sustav *X*:

X = {teret(*r*), lokacija(*r*), zauzetost(*d*), hrpa(*k*), pozicija(*k*), na(*k*) | *r* ∈ roboti, *d* ∈ dokovi, *k* ∈ kontejneri, *h* ∈ hrpa}

Varijable stanja imaju interpretacije:

- Svaki robot r može nositi jedan kontejner u nekom trenutku. Zapisati ćemo teret(*r*) = *k* ako *r* nosi kontejner *k* i teret(*r*) = nil ako r ne nosi teret tj. prazan je. Iz toga proizlazi Rang(teret(*r*)) = kontejneri ∪ {nil}.
- lokacija(*r*) je mjesto gdje se robot trenutno nalazi, tj. jedan od dokova na kojima se robot nalazi. Iz toga proizlazi Rang(lokacija(*r*)) = dokovi
- Svaki dok d može biti zauzet od najviše jednog robota u nekom trenutku. Kako bismo označili da je dok d zauzet zapisujemo Rang(zauzetost(*d*)) = bool vrijednosti.
- Pozicija(k) je mjesto gdje se trenutno kontejner nalazi. Pozicije gdje se kontejner nalazi mogu biti: robot, neki drugi kontejner ili nil ako se kontejner nalazi na dnu neke hrpe Rang(pozicija(*k*)) = kontejneri ∪ roboti ∪ {nil}.
- Ako kontejner *k* je na hrpi *h* tada hrpa(*k*) = *h*, ako je *k* nije niti na jednoj hrpi nema tada je hrpa(*k*) = nil. Iz toga proizlazi Rang(hrpa(k)) = hrpe ∪ {nil}.
- Svaka hrpa h, iako može biti i prazna, je stog kontejnera. Ako je stog prazan tada je na(*h*) = nil, ako na ako na hrpi ima kontejnera tada na(h). Iz toga proizlazi Rang(na(h)) = kontejneri ∪ {nil}.

Funkcija dodjeljivanja varijable nad skupom *X* je funkcija *s* koja povezuje svaki $x_i \in X$ sa vrijednosti $z_i \in \mathrm{Rang}(x_i)$. Ako je *X* ={$x_1, x_2, \dots, x_n$ }, a jest tada možemo reći da je:

s = {(x_1, z_1), (x_2, z_2), …, (x_n, z_n)}

često ćemo reći da je s skup tvrdnji:

s = $\{x_1 = z_1, x_2 = z_2, \dots, x_n = z_n\}$

Iduća definicija s kojom ćemo se upoznati je definicija prostora stanja varijable stanja (engl. *state-variable state space*) koja predstavlja skup *S* funkcija dodjeljivanja varijabli nad nekim skupom stanja varijabli *X*. Svaka funkcija dodjeljivanja varijable u *S* se naziva stanje u *S*. Ako je svrha *S* reprezentacija neke okoline *E* tada želimo da svako stanje *s* ima interpretaciju u *E*. Interpretacija I je funkcija koja povezuje skupove B, R i S objekata, strogih svojstava i svojstava varijabli u nekoj okolini E tako da je s ∈ S.

Sada ćemo definirati prostor stanja varijable stanja S koji smo maloprije definirali. Stanje s_0 ima sljedeću funkciju pridruživanja:

$$s_0 = \{teret(r_1) = nil, teret(r_2) = nil,$$
$$lokacija(r_1) = d_1, lokacija(r_2) = d_2,$$
$$zauzetost(d_1) = \top, zauzetost(d_2) = \top, zauzetost(d_2) = F,$$
$$hrpa(k_1) = h_1, hrpa(k_2) = h_1, hrpa(k_2) = h_2,$$
$$pozicija(k_1) = k_2, pozicija(k_2) = nil, pozicija(k_3) = nil,$$
$$na(h_1) = k_1, na(h_2) = k_3, na(h_3) = nil\}$$

U istom primjeru stanje s_1 je vrlo slično stanju s_0 samo što je $teret(r_1) = k_1, hrpa(k_1) = nil$, $pozicija(k_1) = r_1, na(h_1) = c_2$. Drugačije zapisano to izgleda:

$$s_1 = \{teret(r_1) = k_1, teret(r_2) = nil,$$
$$lokacija(r_1) = d_1, lokacija(r_2) = d_2,$$
$$zauzetost(d_1) = \top, zauzetost(d_2) = \top, zauzetost(d_3) = F,$$
$$hrpa(k_1) = nil, hrpa(k_2) = h_1, hrpa(k_3) = h_2,$$
$$pozicija(k_1) = r_1, pozicija(k_2) = nil, pozicija(k_3) = nil,$$
$$na(h_1) = c_2, na(h_2) = k_3, na(h_3) = nil\}$$

Rangovi su:

$|Rang(teret(r_1))| = |Rang(teret(r_2))| = 4,$

$|Rang(lokacija(r_1))| = |Rang(lokacija(r_2))| = 3,$

$|Rang(zauzetost(d_1))| = |Rang(zauzetost(d_2))| = |Rang(zauzetost(d_3))| = 2,$

$|Rang(hrpa(k_1))| = |Rang(hrpa(k_2))| = |Rang(hrpa(k_3))| = 4,$

$|Rang(pozicija(k_1))| = |Rang(pozicija(k_2))| = |Rang(pozicija(k_3))| = 6,$

$|Rang(na(h_1))| = |Rang(na(h_2))| = |Rang(na(h_3))| = 4.$

Iz ovoga proizlazi da je broj mogućih funkcija dodjeljivanja varijable jednak $4^2 * 3^2 * 2^3 * 4^3 * 6^3 * 4^3 = 1\,019\,215\,872$.

Iz gore navedenog izračuna možemo vidjeti kako za relativno jednostavan problem koji imamo s ne pretjerano velikim brojem varijabli koje se uzimaju u obzir dolazimo do vrlo velike brojke. (Ghallab, Nau, Traverso, 2016)

5. Implementacija algoritama za planiranje

U idućim poglavljima pokazati ćemo nekoliko implementacija algoritma A* te nekoliko primjera agenata koji su implementirani u platformi SPADE. Zadnji primjer koji će biti implementiran objedinit će više primjera koji će biti prethodno implementirani. Isto tako, A* algoritam biti će opisana na više primjera tj. na više različitih domena, konkretnije na dvije domene, od čega je prva pomalo ne tipična.

Kod implementacije A* algoritma korišten je programski jezik Python u verziji 2.7. Python je programski jezik visoke razine (engl. *high-level*), strukturiran (engl. *structured*) i otvorenog koda koji se može koristiti za mnogo različitih zadaća u programiranju. Programski jezik Python kreirao je Guido Van Rossum početkom 1990-ih godina. Popularnost Pythona raste s godinama i jezik je vrlo pogodan za nekoga tko uči programiranje jer njegova sintaksa spada među jednostavnije. Ime je dobio prema Britanskoj humorističnoj seriji Leteći cirkus Monty Pythona. Osim što je Python dobar za učenje programiranja, koristi se kod administracije sustava (engl. *system administration*), primjerice mnoge komponente Linux operacijskih sustava napisane su baš u tom programskom jeziku. Nadalje mnoge kompanije i organizacije, poput NASA-e, Googlea i Yahooa, koriste ovaj programski jezik. (Wikibooks, 2015)

5.1. A* (a zvjezdica) algoritam

A* algoritam se koristi za pronalazak najkraćeg puta od početnog do konačnog stanja. Ovaj algoritam se može primijeniti u vrlo velikom broju domena: od kretanja likova u računalnim igrama, rješavanju zagonetki, pronalasku niza znakova itd.

Kod umjetne inteligencije A* algoritam se najćešće koristi za planiranje puta, odnosno potragu za nizom logičkih radnji koje će promjeniti agentovo početno stanje u ciljno stanje. (Duchoň et al, 2014) A* algoritam u nekim primjerima ćemo koristiti za izradu plana kretanja od početne do konačne točke na rešetci. Prema definiciji, planiranje se sastoji od odabira i organiziranja radnji kojima ćemo ispuniti zadani cilj, a kao konačni rezultat kreirati će se plan. Iz ove definicije možemo povući paralelu sa A* algoritmom jer će algoritam odabrati radnje kojima će se ispuniti zadani cilj tj pomak na konačnu poziciju, dok će rezultat biti plan koji će se sastojati od svih pozicija koje je potrebno posjetiti kako bismo došli do ciljne pozicije na rešetci.

Algoritam radi na način da se na početku generiraju svi mogući koraci iz trenutne pozicije prema cilju. Mogući koraci predstavljaju djecu trenutne pozicije. Djeca se pohranjuju

u sortiranu listu ili prioritetni red. Lista djece će biti sortirana prema određenom kriteriju, a taj kriterij predstavlja način utvrđivanja da se približujemo određenom cilju. Idući korak je odabir djeteta iz liste koji se čini kao najbliži cilju i taj se korak ponavlja sve dok cilj nije postignut ili nema više djece. Dvije vrlo bitne stvari za ovaj algoritam su kako se mjeri udaljenost prema cilju i kako se generiraju djeca.

Neka od svojstava A* algoritma su: prekidanje, potpunost i optimalnost. Za svaki problem u klasičnom planiranju A* algoritam će završiti i vratiti rješenje ukoliko postoji, a u nekim slučajevima će vratiti optimalno rješenje. Najveći nedostatak ovog algoritma je zahtjev za prostorom. Drugim riječima, svako dijete je spremljeno u listu, no ovaj problem se može riješiti s dobrom heurističkom funkcijom i tada se skraćuje vrijeme rada algoritma i količina memorije koja je potrebna.

Ispod ovoga odlomka navedeni su primjeri implementacije algoritma A* u programskom jeziku Python. Problem koji će riješiti navedeni program je sortiranje slova kako bismo dobili konačnu riječ iz početne, naravno, ukoliko je to uopće moguće. Komentari koji će pobliže objasniti navedeni kod nalaze se ispod samog koda.

Prije nego bude naveden primjer, autor želi objasniti razloge odabira ovog primjera. Prvi razlog je što sama domena sortiranja slova iz početnog niza u željeni niz je jednostavnija u odnosu na kasnije primjere, a drugi razlog korištenja ovog primjera je isticanje kako ovaj algoritam može imati široku primjenu i koristit se kao moguće rješenja za širok spektar problema.

```
from Queue import PriorityQueue

class State(object):
    def __init__(self, value, parent, start = 0, goal = 0):
        self.children = []
        self.parent = parent
        self.value = value
        self.dist = 0
        if parent:
            self.path = parent.path[:]
            self.path.append(value)
            self.start = parent.start
            self.goal = parent.goal

        else:
            self.path = [value]
            self.start = start
            self.goal = goal

    def GetDist(self):
        pass
    def CreateChildren(self):
        pass

class State_String(State):
    def __init__(self, value, parent, start = 0, goal = 0):
```

```
        super(State_String, self).__init__(value, parent, start, goal)
        self.dist = self.GetDist()

    def GetDist(self):
        if self.value == self.goal:
            return 0
        dist = 0
        for i in range (len(self.goal)):
            letter = self.goal[i]
            dist += abs(i - self.value.index(letter))
        return dist

    def CreateChildren(self):
        if not self.children:
            for i in xrange(len(self.goal)-1):
                val = self.value
                val = val[:i] + val[i+1] + val[i] + val[i+2:]
                child = State_String(val, self)
                self.children.append(child)


class Astar_Solver:
    def __init__(self, start, goal):
        self.path = []
        self.visitedQueue = []
        self.priorityQueue = PriorityQueue()
        self.start = start
        self.goal = goal

    def Solve(self):
        startState = State_String(self.start, 0, self.start, self.goal)
        count = 0
        self.priorityQueue.put((0, count, startState))
        while (not self.path and self.priorityQueue.qsize()):
            closestChild = self.priorityQueue.get()[2]
            closestChild.CreateChildren()
            self.visitedQueue.append(closestChild.value)
            for child in closestChild.children:
                if child.value not in self.visitedQueue:
                    count += 1
                    if not child.dist:
                        self.path = child.path
                        break

                    self.priorityQueue.put((child.dist, count, child))

        if not self.path:
            print "Goal of" + self.goal + "is not possible"
        return self.path

if __name__ == "__main__":
    start1 = raw_input("Unesite pocetni string (npr abcd): ")
    goal1 = raw_input("Unesite konacni string (npr dcba): ")
    if len(start1) == len(goal1):
        print "Pocetak rada"
        a = Astar_Solver(start1, goal1)
        a.Solve()
        for i in xrange(len(a.path)):
            print "%d) " %i + a.path[i]
    else:
        print "Uneseni nizovi znakova nemaju istu duljinu.
```

```
Pokusajte ponovo."
```

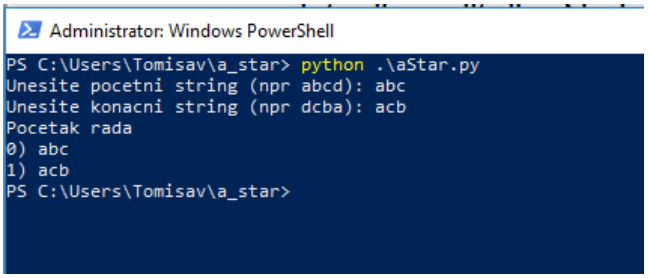
Na početku je potrebno reći Pythonu da iz biblioteke Queue koristi PriorityQueue, koji ćemo koristiti kasnije. Nakon toga imamo početnu klasu koja se zove State. Klasa State u konstruktoru ima parametre value, parent, start i goal. Parametri start i goal uz sebe imaju dodatak koji kaže jednako nuli, taj dodatak se koristi za slučaj kada roditelj ne postoji, tj. kada smo na inicijalnom stanju da algoritam može raditi. Nakon toga se definira lista djece koja je inicijalno prazna lista. Nakon toga se pohranjuju vrijednosti roditelja vrijednosti i postavlja se vrijednost varijable dist kojoj će vrijednost biti postavljena kasnije, a to će napraviti neka podklasa klase State. Ukoliko postoji roditelj tada će se postaviti njegovi atributi poput: liste s imenom path te varijabli start i goal. Ukoliko ne postoji parametar roditelji, lista path te varijable start i goal će dobiti drugačije vrijednosti. Nakon toga se definiraju dvije prazne funkcije koje će biti definirane u podklasi State_String.

Kod klase State_String inicijalizira se klasa State tj. bazna klasa. Nakon toga se poziva metoda GetDist koja je zadužena za izračunavanje udaljenosti od cilja. U metodi GetDist se prvo provjerava da li smo stigli do cilja i ako nismo stigli do cilja koliko smo udaljeni od njega. Udaljenost od cilja se mjeri na način da se provjerava svako slovo iz krajnjeg niza znakova. Za svako slovo ćemo tražiti njegovu trenutnu poziciju u trenutnome nizu znakova, ovime ćemo dobiti udaljenost tog slova od njegove željene tj. ciljane lokacije. Kod metode CreateChildren, u ovome primjeru, biti će kreirana nova riječ. S prvom linijom unutar metode se osigurava da se djeca ne kreiraju dva puta. Unutar for petlje se prolazi kroz sve mogućnosti poretka slova, na način da se zamjenjuju prvo i drugo slovo svakog para slova. Nakon toga se u varijablu child pohranjuje dobivena vrijednost i trenutni vrijednost koja predstavlja roditelja. Na kraju ove metode se kreirano dijete dodaje u listu djece.

U klasi AStarSolver varijabla path će sadržavati sva stanja koja su potrebna od početka prema željenom cilju. VisitedChildren sadrži informacije o svoj djeci koju smo „posjetili“, a start i goal varijable se koriste za spremanje vrijednosti koje su dobivene kao parametri prilikom inicijalizacije. Nakon toga slijedi definiranje metode Solve, kod koje prvo definiramo startState tj. početno stanje koje je opisano u prethodnome odlomku. Najzanimljiviji dio metode Solve je while petlja. While petlja se izvršava sve dok self.path nije prazan i dok priorityQueue nije prazan. Za vrijednost closestChild uzimamo vrijednost koja je na drugom indeksu u priorityQueueu, a to je StartState. Nakon toga se kreiraju djeca za to stanje. U idućoj liniji koda se trenutni čvor ili stanje dodaje u visitedQueue.

Sada ćemo pokazati nekoliko primjera izvršavanja programa. Program ćemo pokrenuti putem alata PowerShell pomoću naredbe python .\aStar.py, a prethodno smo se pozicionirali u mapi gdje se nalazi skripta koju ćemo pokrenuti. Na početku je potrebno unijeti

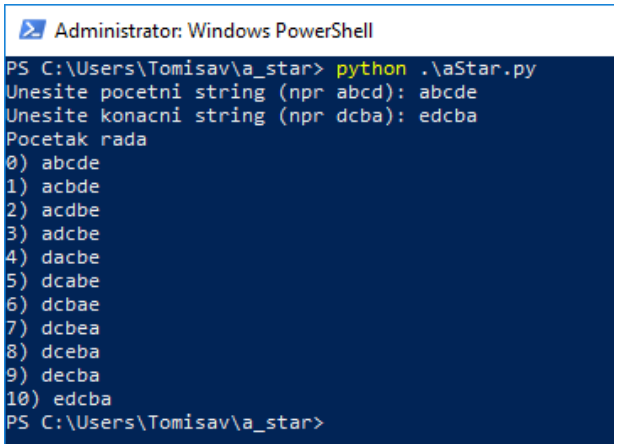
početni niz znakova, a nakon toga željeni ili ciljni niz znakova. Nakon što unesemo ciljani niz znakova, program će krenuti s izvršavanjem i završiti s radom ukoliko nađe rješenje problema. Za prvi primjer koristit ćemo „abc“ kao početni niz znakova, a „acb“ kao željeni niz znakova. (Trevor Payne, 2013)



Slika 5.1 Prvi primjer izvršavanja prve implementacije A* algoritma

Kao što možemo vidjeti sa slike program radi i već u idućem koraku dobivamo rješenje. Drugim riječima, bilo je potrebno samo zamijeniti poziciju drugog i trećeg slova, kako bismo dobili željeni niz znakova.

U idućem ćemo primjeru koristit „abcde“ kao početni niz znakova, a „edcba“ kao željeni niz znakova.



Slika 5.2 Drugi primjer izvršavanja prve implementacije A* algoritma

Kod ovog primjera možemo bolje vidjeti na koji način radi algoritam. Prvo je zamijenio pozicije znakovima c i b, nakon toga u iduća dva koraka stavlja slovo d ispred njih. Kad je to napravljeno, algoritam, u iduća tri koraka, pomiče slovo a iza „dcb“, te u zadnjih 5 koraka postavlja slovo e na sam početak niza znakova.

U idućim primjerima ćemo koristi drugačiju domenu u odnosu na prvi primjer. Domena će nam biti pronalazak puta od početnog ili startnog čvora do konačnog ili željenog čvora. Prvi primjer biti će jednostavniji i imati će neka ograničenja, dok će drugi primjer biti nešto realniji i neće imati toliko ograničenja. Za početak upoznati ćemo se s rešetkom (engl. *grid*) koju ćemo koristiti kako bismo mogli razumjeti sam rad algoritma. Autor ovdje pod rešetkom

misli na dvodimenzionalnu mrežu koja se može, primjerice, predočiti kao mapa po kojoj se moguće kretati s virtualnim likom.

Tablica 5.1 Vizualni prikaz rešetke

4	4 (1,4)	8 (2,4)	12 (3,4)	16 (4,4)
3	3 (1,3)	7 (2,3)	11 (3,3)	15 (4,3)
2	2 (1,2)	6 (2,2)	10 (3,2)	14 (4,2)
1	1 (1,1)	5 (2,1)	9 (3,1)	13 (4,1)
Y	1	2	3	4
X				

Tablicom 5.1 želi se prikazati identifikacija polja na mapi, jedinstveni identifikatori polja ili ćelije u tablici su označeni sa svijetlo plavom bojom. Veličina rešetke prikazane na slici je četiri retka i četiri stupca. Primjerice, svako polje ima svoj identifikator kako bi se izbjegli određeni problem do kojih je dolazilo tokom izrade programskog rješenja. Identifikatori rastu prvo prema y-osi, a nakon toga i prema x-osi. Dakle, vrijednosti identifikatora raste od dna stupca prema vrhu stupca. Isto tako, vrijednost identifikatora raste ukoliko uspoređujemo određeni stupac i stupac koji se nalazi desno od trenutno promatranoga. Autor je ovim rečenicama pokušao objasnit kako identifikatori rastu, ali ukoliko promotrimo sliku intuitivno nam je jasno kako se identifikatori povećavaju.

Nakon što smo se upoznali s rešetkom, koja predstavlja mapu po kojoj se je moguće kretati reći ćemo nešto više o problemu koji se želi riješiti u primjeru navedenom ispod. Dakle, cilj primjera je prikazati uz pomoć A* algoritma kako se određuje put između početne točke do konačne točke te kroz koje se polja prolazi na putu do cilja. Iako se na prvi pogled čini kako je ovaj problem jednostavan i očit, stvari se donekle kompliciraju kada je to potrebno objasniti računalu. U idućem primjeru, stvari će biti pojašnjene na način da je moguće kretati se isključivo u četiri smjera: lijevo, desno, naprijed i natrag. Realnije bi bilo da postoji više opcije, poput dijagonalnog kretanja, ali to neće biti implementirano u ovome primjeru.

Sada ćemo objasniti kako radi A* algoritam. Na ovome primjeru se logika samoga algoritma može jasnije vidi u odnosu na prvo pokazani primjer.

Algoritam započinje tako što uzimamo prvi čvor i gledamo čvorove koji su okolo početnog čvora. Za svaki od čvorova izračunat ćemo određene vrijednosti. Te vrijednosti su: G trošak (engl. *G cost*), H trošak (engl. *H cost*) i F trošak (engl. *F cost*).

Tablica 5.2 Prikaz G, H i F troškova kod početnog čvora

Y \ X	1	2	3	4
4				16 (4,2)
3		10 30 40		
2	10 40 50	6 (2,2)	10 30 40	
1		10 40 50		

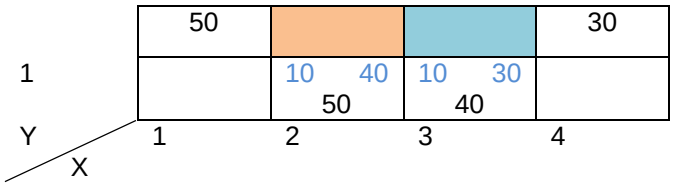
To možemo vidjeti na tablici 5.2 kako bismo si lakše predočili, u lijevom gornjem kutu ćelije prikazan je G trošak, u desnom gornjem kutu možemo vidjeti H trošak, dok se u sredini čvora nalazi F trošak. G trošak predstavlja udaljenost promatranog susjednog čvora u odnosu na trenutno promatrani čvor. U tablici 5.2 početni čvor ima jedinstveni identifikator 6, dok željeni ili ciljni čvor ima jedinstveni identifikator 16. Ta dva čvora su istaknuta s narančastom, odnosno crvenom bojom.

Iz tablice 5.2 vidimo da je G trošak 10, za sve susjedne čvorove, što predstavlja vrijednost pomicanja s trenutnog čvora na taj čvor. Možemo vidjeti da su za sve čvorove G trošak jednake. To je zbog pojednostavljenja ovog zadatka jer je gibanje moguće isključivo u četiri smjera. Kao što smo rekli H trošak se nalazi u gornjem desnom kutu, a ona predstavlja udaljenost susjednog čvora od ciljnog tj. željenog čvora. Jednostavnije rečeno ova udaljenost predstavlja udaljenost od cilja. F trošak, koji se nalazi u sredini čvora predstavlja zbroj G troška i H troška. Generalno gledano F trošak nam predstavlja udaljenost ukoliko se pomaknemo na taj čvor koliko ćemo biti udaljeni od cilja.

Algoritam će promatrati F trošak u potrazi za najnižim F trošku. Nakon što odaberemo jedan čvor ponovno se radi izračun svih troškova za taj čvor. U tablici 5.3 prikazan je izračun troškova ako se pomaknemo na čvor s jedinstvenim identifikatorom 10, taj čvor je istaknut plavom bojom pozadine.

Tablica 5.3 Prikaz G, H i F troškova kod čvora s jedinstvenim identifikatorom 10

4				16 (4,2)
3		10 30 40	10 20 30	
2	10 40	6 (2,2)	10 (3,2)	10 20



Na sljedećih nekoliko stranica prikazan je programski kojime je implementiran ovaj algoritam na domeni kretanja od početnog do željenog čvora.

```
class Node(object):
    def __init__(self, x, y, id):
        self.gCost = None
        self.hCost = None
        self.x = x
        self.y = y
        self.neighbours = []
        self.parent = None
        self.id = id

    def getFCost(self):
        return self.gCost - self.hCost


class Grid(Node):

    def __init__(self, x, y):
        self.gridX = x;
        self.gridY = y;
        self.nodes = []
        i = 0
        j = 0
        id = 1
        while i < x:
            i += 1
            while j < y:
                j += 1
                n = Node(i, j, id)
                self.nodes.append(n)
                id += 1
            j = 0
        self.addNeighbours()

    def returnNodeByID(self, id):
        for node in self.nodes:
            if node.id == id:
                return node

    def returnNodeIDByCoordints(self, x, y):
        for node in self.nodes:
            if node.x == x and node.y == y:
                return node.id

    def addNeighbours(self):
        for node in self.nodes:
            if node.x-1 > 0:
                node.neighbours.append(self.returnNodeIDByCoordints(node.x-
1, node.y))
            if node.x+1 <= self.gridX:
```

```
node.neighbours.append(self.returnNodeIDByCoordints(node.x+1, node.y))
            if node.y-1 > 0:
                node.neighbours.append(self.returnNodeIDByCoordints(node.x,
node.y-1))
            if node.y+1 <= self.gridY:
                node.neighbours.append(self.returnNodeIDByCoordints(node.x,
node.y+1))

    def setGCost(self, end):
        endNode = self.returnNodeByID(end)
        for node in self.nodes:
            if node == endNode:
                node.gCost = 0
            else:
                node.gCost = self.getDistance(node, endNode)

    def getDistance(self, nodeA, nodeB):
        distA = abs(nodeA.x - nodeB.x)
        distB = abs(nodeA.y - nodeB.y)
        return (distA + distB)*10


class astar_Solver:
    def __init__(self, gridX, gridY, start, goal):
        self.grid = Grid(gridX, gridY)
        self.start = start
        self.goal = goal
        self.openset = []
        self.closedset = []
        self.finalPath =[]
        self.grid.setGCost(self.goal)
        self.openset.append(self.grid.returnNodeByID(start))
        self.grid.doPrint()

    def solve(self):

         while (self.openset):
            currentNode = self.openset[0]
            for i in range(len(self.openset)-1):
                if (self.openset[i].hCost < currentNode.hCost or
self.openset[i].hCost == currentNode.hCost and self.openset[i].getFCost() <
currentNode.getFCost()):
                    currentNode = self.openset[i]
            self.openset.remove(currentNode)
            self.closedset.append(currentNode)

            if currentNode == self.grid.returnNodeByID(self.goal):
                self.finalPath = self.getFinalPath(self.start, self.goal)
                self.printFinalPath()
                return

            for x in currentNode.neighbours:
                neighbour = self.grid.returnNodeByID(x)
                if neighbour in self.closedset:
                    continue

                movementCostToNeighbour = currentNode.gCost +
self.getDistance(currentNode, neighbour)
                goalNode = self.grid.returnNodeByID(self.goal)
```

```
                if movementCostToNeighbour < neighbour.gCost or neighbour
not in self.openset:
                    neighbour.gCost = movementCostToNeighbour
                    neighbour.hCost = self.getDistance(neighbour, goalNode)
                    neighbour.parent = currentNode

                    if neighbour not in self.openset:
                        self.openset.append(neighbour)



    def getDistance(self, nodeA, nodeB):
        distA = abs(nodeA.x - nodeB.x)
        distB = abs(nodeA.y - nodeB.y)
        return (distA + distB)*10

    def getFinalPath(self, start, end):
        reversePath = []
        startNode = self.grid.returnNodeByID(start)
        currentNode = self.grid.returnNodeByID(end)
        while currentNode != startNode:
            reversePath.append(currentNode.id)
            currentNode = currentNode.parent
        reversePath.append(startNode.id)
        return reversePath[::-1]

    def printFinalPath(self):
        print self.finalPath

if __name__ == "__main__":
    print "Starting..."
    a = astar_Solver(5, 5, 2, 24)
    a.solve()
```

Na početku programskog koda možemo vidjeti definiranu klasu Node koja se koristi kao bazična klasa i sadrži informacije poput: G troška, H troška, x koordinate, y koordinate, susjeda, roditelja te ID određenog čvora. Ova klasa ima definiranu jednu metodu getFCost koja se koristi za izračunavanje F troška određenog čvora.

Nakon toga se nalazi definicija klase Grid koja predstavlja rešetku. U init metodi te klase se definira koliko ima redaka i stupaca te je definirano polje nodes u koje će se spremati čvorovi tj. objekti klase Node. Nadalje u toj metodi se kreiraju objekti klase node i pridružen im je jedinstveni identifikator ili ID, a pri kraju init metode se poziva metoda addNeighbours o kojoj ćemo više reći nešto kasnije. U ovoj klasi je definirana i metoda returnNodeByID koja kao parametar prima jedinstveni identifikator nekog čvora, a kao rezultat vraća objekt klase Node s tim identifikatorom. Poslije te metode je definirana metoda returnIDByCoordinates koja se koristi za dohvaćanje identifikatora nekog čvora ukoliko znamo njegove koordinate. Nakon toga slijedi definicija spomenute metode addNeighbours. Kod metode addNeighbours svakom čvoru se spremaju identifikatori njegovih susjeda u polje neighbours. Kod ove metode je potrebno obratiti pozornost gdje se trenutno promatrani čvor nalazi kako bi se izbjegli mogući problemi s dodavanjem pozicija u rešetki koje ne postoje.

Takva situacija je moguća kod čvorova koji se nalaze uz rubove. U ovoj klasi je definirana i metoda setGCost koja se koristi za postavljanje G troška kod određenog čvora, a to radi na način da postavlja G trošak na nulu ukoliko se radi o ciljnom čvoru, a u svim ostalim slučajevima poziva metodu getDistance. Posljednja metoda u ovoj klasi je već spomenuta getDistance metoda koja se koristi za izračunavanje udaljenosti između trenutnog i krajnjeg čvora. Ova metoda radi na način da se izračuna apsulutna udaljenost prvo između x koordinata trenutnog i ciljnog čvora, a nakon toga i y koordinata trenutnog i ciljnog čvora, izračunate apsulutne vrijednosti se zbroje te se pomnože s deset. Logika iza ove metode je vrlo jednostavna, zbrojimo koliko pomaka po x i y osi smo udaljeni od ciljnog čvora te tu udaljenost pomnožimo s 10. Množenje s deset nije nužno moralo biti implementirano i algoritam bi radio i bez toga.

Iduća klasa koja je definirana zove se astar_Solver. U init metodi ove klase se kreira rešetka pomoću dobivenih gridX i gridY parametara, uz to prosljeđuju se parametri start i goal koji predstavljaju jedinstvene identifikatore polja u rešetki. Tako start predstavlja identiifikator polja s kojega želimo krenuti, a goal predstavlja polje u koje se želimo pozicionirati kao krajnji cilj. Nakon toga se definiraju liste openset, closedset i finalpath. Lista openset predstavlja čvorove koje ćemo posjetiti, a u početku tamo dodajemo čvor koji ima jednak identifikator kao start parametar. Closed set je lista koja je inicijalno prazna, a u nju ćemo dodavati čvorove koje smo već posjetili. FinalPath je lista koja će se koristi za spremanje putanje od početnog do željenog čvora. Drugim riječima, finalPath je rješenje koje dobijem na kraju rada algoritma. Najvažnija metoda ovog algoritma je solve metoda iza koje se krije logika rada A* algoritma. U solve metodi postoji while petlja koja će se izvršavati sve dok lista openset nije prazna. Unutar petlje prolazimo kroz listu openset i provjeravamo da li postoji F trošak u listi koja bi bila manja od F troška trenutno promatranog čvora ili ako je F trošak iz liste jednak kao i kod trenutno promatranog čvora, ali je H trošak manji u odnosu na trenutno promatrani čvor. Ukoliko je taj uvjet zadovoljen trenutno promatrani čvor postaje onaj s najmanjim F troškom. Nakon toga se iz liste openset uklanja taj čvor, a isti čvor se dodaj u listu closedset. Drugim riječima, želimo promatrani čvor maknuti iz liste neposjećenih i dodati ga u listu posjećenih čvorova. Nakon toga se radi provjera, da li je trenutno promatrani čvor jednak željenom čvoru, te ukoliko je želimo ispisati putanju od početnog do željenog čvora te se program završava. Ukoliko to nije bio slučaj želimo pogledati susjedne čvorove od trenutno promatranog čvora, nakon toga se dohvaća susjedni čvor pomoću metode returnNodeByID, kada dohvatimo čvor ukoliko smo ga već posjetili, želimo ga preskočiti. Nakon toga radi se izračun troška pomicanja na susjedni čvor te trošak udaljenosti tog susjednog čvora do željenog čvora, u implementaciji se taj trošak nalazi u varijabli movementCostToNeighbour. Ukoliko je izračunati trošak manji od G troška trenutno promatranog čvora ili ako se susjedni

čvor ne nalazi u listi openset, tada postavljamo G trošak, F trošak susjednog čvora te mu kao roditelja postavljamo trenutno promatrani čvor. Na kraju, ukoliko se susjedni čvor ne nalazi u openset listi, želimo ga dodati u tu listu.

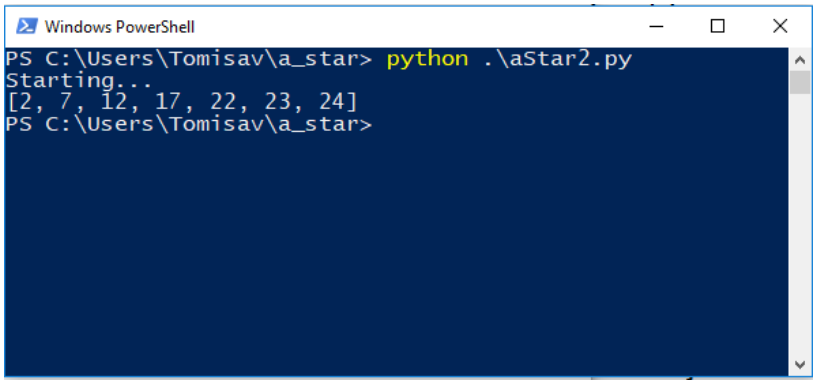
Metoda getDistance se koristi kako bi se izračunala udaljenost između dvaju čvorova, a to se radi na način da se izračuna absulutna udaljenost između x koordinata tih čvorova i absulutna udaljenost između y koordinata te se te dvije vrijednosti zbroje te pomnože s deset, kao rezultat metoda vrača izračunati umnožak.

Metoda getFinalPath služi za dohvaćanje konačnog puta od početnog do konačnog čvora. Metoda radi na način da dohvati prvo početni i konačni čvor, konačni čvor postaje trenutno promatrani i uspoređuje ga s početnim čvorom. Ukoliko su ta dva čvora različita, jedinstveni identifikator se dodaje u listu reversePath, a trenutno promatrani čvor postaje roditelj trenutnog čvora. Ovaj postupak se ponavlja sve dok ne dođemo do početnog čvora. Kada dođemo do njega izlazi se iz petlje te se početni čvor dodaje u listu koja se vraća kao rezultati, ali na način da zamjenjuju poredak elemenata u listi na način da prvi postaje zadnji, a zadnji element liste postaje prvi.

Metoda printFinalPath se koristi za jednostavan ispis liste koja je vraćena kao rezultat prethodno opisane metode.

Na samome kraju programa nalazi se main metoda kod koje se ispisuje tekst da je program započeo s radom te se kreira instanca klase astar_Solver s parametrima koji su uneseni kroz programski, ali mogu se jednostavno promijeniti te se poziva metoda solve s kojom se obavlja sav posao vezan uz rad algoritma.

Na kraju ovog primjera A* algoritma pokazati ćemo primjer izvođenja. Primjer ćemo pokrenuti pomoću Windows PowerShella, a rezultat možemo vidjeti na slici 5.3.



Slika 5.3 Primjer izvršavanja druge implementacije A* algoritma

Parametri kod ovoga primjera nisu promjenjivi i moguće ih je mijenjati jedino u programskom kodu. Veličina rešetke je 5 redaka i pet stupaca, početni čvor ima jedinstveni identifikator 2, a ciljni čvor ima jedinstveni identifikator 24. Sa slike možemo vidjeti kako je

program s početnog čvora s jedinstvenim identifikatorom 2 preko čvorova 7, 12, 17, 22, 23 stigao do ciljnog čvora 24.

U idućem primjeru način kretanja će biti složeniji. Pod složeniji autor misli na činjenicu da će odsad biti moguće kretati u osam smjerova te mogućnost postavljanja čvorova koji nisu dostupni tj. smatramo ih preprekama. U prethodno opisanom primjeru kretanje je bilo moguće u samo četiri smjera. Iz tablice vidimo da je moguće kretanje u više smjerova nego u prethodno opisanom primjeru, ali i da se troškovi drugačije izračunavaju. Primjerice, susjedni čvorovi koji se nalaze dijagonalno o trenutno promatranog čvora imaju veći G trošak u odnosu na ostale susjedne čvorove.

Tablica 5.4 Prikaz G, H i F troškova kod početnog čvora s osam smjerova kretanja

4				16 (4,2)
3	14 34 40	10 24 34	14 14 28	
2	10 40 50	6 (2,2)	10 24 34	
1	14 42 56	10 40 50	14 34 48	
Y / X	1	2	3	4

Primjer programskog koda nalazi se ispod ovog odlomka, ali neće biti opisan u cijelost već samo dijelovi koji su različiti u odnosu na drugi primjer A* algoritma. Važno je napomenuti kako se ispod ne nalazi potpuni kod već su neke metode koje su koriste, a iste su kao u prošlom primjeru, jednostavno izostavljene.

Kod klase Node nije bilo velikih promjera te je dodan atribut isObstical koji će sadržavati informaciju da li je neki čvor prepreka ili se radi o uobičajenom čvoru.

U klasi Grid, došlo je do promjene kod parametara, te se sada uz veličinu rešetke, odnosno broj redaka i stupaca, u metodu prosljeđuje i lista s jedinstvenim identifikatorima čvorova koji nisu prohodni. Novokreirana metoda setObsticals zadužena je za postavljanje parametra isObstical kod svakoga čvora. Neke metode su doživjele promjene u odnosu na prošli primjer te metode su addNeighbours i getDistance. Kao što smo rekli u uvodu, veći broj susjednih čvorova rezultirao je promjenama kod metode addNeighbours. Najveći broj susjeda nekog čvora u trenutnoj implementaciji iznosi osam, zbog nepreglednost provjeravanja sa selekcijama ova metoda je donijela promjene u logici rada, ali se krajnji rezultat nije mijenjao. Kod metode getDistance došlo je do promjena jer dijagonalno kretanje je „skuplje“ u odnosu na kretanje prema susjednima iz prošlog primjera, to je rezultiralo u promjeni izračunavanja udaljenosti između čvorova.

Nadalje, klasa astar_Solver je doživjela promjene. Promijenio se broj argumenata jer je dodana lista koja će sadržavati čvorove koji nisu prohodni. Isto tako napravljene su nužne promjene u logici kako bi algoritam radio.

Vjerojatno najveće promjene je doživjela main metoda jer je u ovome primjeru moguće promijeniti veličini rešetke, a od korisnika se očekuje da unese broj redaka i broj stupaca. Od korisnika se očekuje da unese jedinstveni identifikator početnog i konačnog ili ciljnog čvora, te postoji mogućnost da korisnik unese čvorove za koje želi da nisu prohodni te algoritam to mora uzeti u obzir. Napravljena je i jedna vrste verifikacije korisničkog unosa, kako bi se smanjila mogućnost pogrešnog unosa.

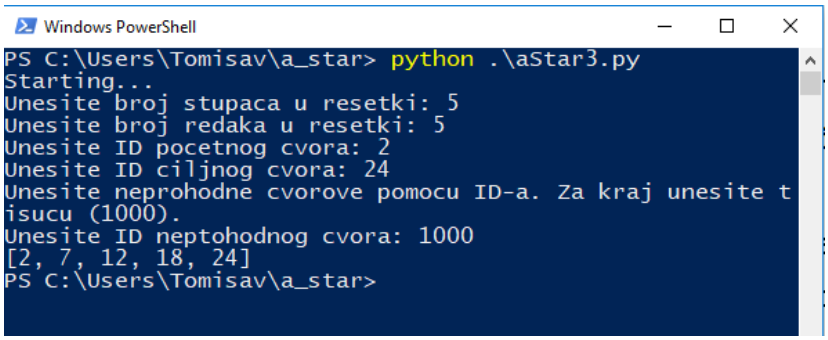
```
class Node(object):
    def __init__(self, x, y, id):
        self.isObsticle = False
        self.gCost = None
        self.hCost = None
        self.x = x
        self.y = y
        self.neighbours = []
        self.parent = None
        self.id = id

    def getFCost(self):
        return self.gCost - self.hCost


class Grid(Node):

    def __init__(self, x, y, obsticalList):
        self.gridX = x;
        self.gridY = y;
        self.nodes = []
        self.obsticalList = obsticalList
        i = 0
        j = 0
        id = 1
        while i < x:
            i += 1
            while j < y:
                j += 1
                n = Node(i, j, id)
                self.nodes.append(n)
                id += 1
            j = 0
        self.addNeighbours(x, y)
        self.setObsticals()


    def addNeighbours(self, gridx, gridY):
        for node in self.nodes:
           for xi in range(-1, 2):
               for yi in range(-1, 2):
                    if xi == 0 and yi == 0:
                       continue
                    else:
                        verifyX = node.x + xi
```

```
                        verifyY = node.y + yi
                        if verifyX > 0 and verifyX <= gridx and verifyY > 0
and verifyY <= gridY:

node.neighbours.append(self.returnNodeIDByCoordints(verifyX, verifyY))

    def setObsticals(self):
        if self.obsticalList:
            for obstical in self.obsticalList:
                for node in self.nodes:
                    if obstical == node.id:
                        node.isObsticle = True

    def getDistance(self, nodeA, nodeB):
        distA = abs(nodeA.x - nodeB.x)
        distB = abs(nodeA.y - nodeB.y)
        if distA > distB:
            return 14 * distB + 10*(distA - distB)
        return 14 * distA + 10*(distB - distA)


class astar_Solver:
    def __init__(self, gridX, gridY, start, goal, obsticalList):
        self.grid = Grid(gridX, gridY, obsticalList)
        self.start = start
        self.goal = goal
        self.openset = []
        self.closedset = []
        self.finalPath =[]
        self.grid.setGCost(self.goal)
        self.openset.append(self.grid.returnNodeByID(start))


    def solve(self):

         while (self.openset):
            currentNode = self.openset[0]
            for i in range(len(self.openset)-1):

                if (self.openset[i].getFCost() < currentNode.getFCost() or
self.openset[i].getFCost()       ==       currentNode.getFCost()       and
self.openset[i].hCost < currentNode.hCost):
                    currentNode = self.openset[i]
            self.openset.remove(currentNode)
            self.closedset.append(currentNode)

            if currentNode == self.grid.returnNodeByID(self.goal):
                self.finalPath = self.getFinalPath(self.start, self.goal)
                self.printFinalPath()
                return

            for x in currentNode.neighbours:
                neighbour = self.grid.returnNodeByID(x)
                if neighbour in self.closedset or neighbour.isObsticle:
                    continue

                movementCostToNeighbour      =      currentNode.gCost      +
self.getDistance(currentNode, neighbour)
                goalNode = self.grid.returnNodeByID(self.goal)
                if movementCostToNeighbour < neighbour.gCost or neighbour
not in self.openset:
```

```
                    neighbour.gCost = movementCostToNeighbour
                    neighbour.hCost = self.getDistance(neighbour, goalNode)
                    neighbour.parent = currentNode

                    if neighbour not in self.openset:
                        self.openset.append(neighbour)


    def getDistance(self, nodeA, nodeB):
        distA = abs(nodeA.x - nodeB.x)
        distB = abs(nodeA.y - nodeB.y)
        if distA > distB:
            return 14 * distB + 10*(distA - distB)
        return 14 * distA + 10*(distB - distA)

def verifyinput(userInput):
        try:
            val = int(userInput)
            if val > 0:
                return True
            else:
                print"Broj ne smije biti manji ili jednak nuli"
                return False
        except ValueError:
            print "Uneseni znak: \"",userInput,"\" nije broj."

if __name__ == "__main__":
    print "Starting..."
    gridx = raw_input("Unesite broj stupaca u resetki: ")
    gridy = raw_input("Unesite broj redaka u resetki: ")
    vefifyGridX = verifyinput(gridx)
    vefifyGridY = verifyinput(gridy)
    if vefifyGridX and vefifyGridY:
        intgridx = int(gridx)
        intgridy = int(gridy)
        start = raw_input("Unesite ID pocetnog cvora: ")
        goal = raw_input("Unesite ID ciljnog cvora: ")
        verifyStart = verifyinput(start)
        verifyGoal = verifyinput(goal)
        if verifyStart and verifyGoal:
            intStart = int(start)
            intGoal = int(goal)
            if intStart > 0 and intStart <= intgridx * intgridy and intGoal
> 0 and intGoal <= intgridx * intgridy:
                print "Unesite neprohodne cvorove pomocu ID-a. Za kraj
unesite tisucu (1000)."
                intobstical = 555
                obsticalList = []
                while (intobstical !=0):
                    obstical = raw_input("Unesite ID neprohodnog cvora: ")
                    verifyobstical = verifyinput(obstical)
                    if verifyobstical:
                        intobstical = int(obstical)
                        if intobstical == 1000:
                            break
                        else:
                            obsticalList.append(intobstical)
                a = astar_Solver(intgridx, intgridy, intStart, intGoal,
obsticalList)
                a.solve()
            else:
```

```
                print "Neisparavan unos startnog ili ciljnog cvora. ID ne
postoji unutar resetke. "
        else:
            print "Neisparavan unos startnog ili ciljnog cvora. Uneseni
znak nije broj"
    else:
        print "Neisparavan unos velicine resetke"
```

Nakon što smo naveli programski kod, sada ćemo pokazati neke primjere izvođenja samog programa. Na početku je potrebno unijeti broj redaka i stupaca u rešetki. Kao ograničenje valja izdvojiti da se kod svih unosa mora unijeti cijeli broj koji je veći od nule. Nakon toga unosi se jedinstveni identifikator početnog, a potom i konačnog čvora u rešetci. Kada unesemo početni i konačni čvor možemo unijeti čvorove za koje želimo da nisu prohodni, odnosno čvorove koji predstavljaju prepreke na putu od početnog do konačnog čvora. Ukoliko ne želimo imati prepreke, unijet ćemo 1000 kako bismo izašli iz petlje za unos prepreka. Na slici 5.4 možemo vidjeti primjer izvođenja programa kod kojega nisu unesene prepreke. Isto tako, u tome primjeru veličina rešetke je pet redaka i 5 stupaca, dok su jedinstveni identifikatori početnog i konačnog čvora 2 i 24. Isti početni i konačni čvorovi su korišteni u prethodnom primjeru. Iako su početni i konačni čvorovi ostali isti putanja se promijenila u drugome primjeru jer sada postoji mnogo povoljnija mogućnost dijagonalnog kretanja. (Sebastian Lague, 2014, Javidx9, 2017)



Slika 5.4 Prvi primjer izvršavanja treće implementacije A* algoritma

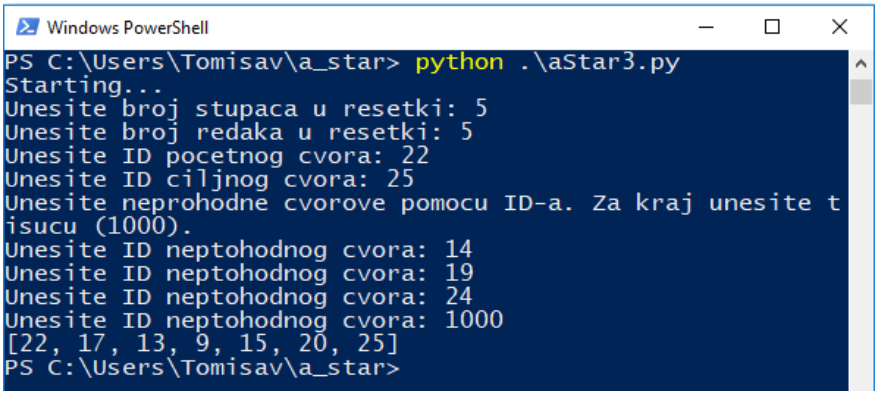
U tablici 5.5 prikazana je rešetka s 5 redaka i 5 stupaca te je na njoj označen put koji smo dobili korištenjem A* algoritma.

Tablica 5.5 Prikaz rešetke iz prvog primjera

5	5 (1,5)	10 (2,5)	15 (3,5)	20 (4,5)	25 (5,5)
4	4 (1,4)	9 (2,4)	14 (3,4)	19 (4,4)	24 (5,4)
3	3 (1,3)	8 (2,3)	13 (3,3)	18 (4,3)	23 (5,3)
2	2 (1,2)	7 (2,2)	12 (3,2)	17 (4,2)	22 (5,2)
1	1 (1,1)	6 (2,1)	11 (3,1)	16 (4,1)	21 (5,1)

Y \ X	1	2	3	4	5

Kod prvoga primjera izvođenja programa nismo koristili prepreke, te ćemo to pokazati na drugome primjeru izvođenja kojega je moguće vidjeti na slici 5.5. U drugome primjeru korištena je veličina rešetke od pet redaka i pet stupaca, ali je jedinstveni identifikator početnog čvora 22, a jedinstveni identifikator konačnog čvora 25, čvorovi s jedinstvenim identifikatorima: 14, 19 i 24 su prepreke te nije moguće proći kroz njih, već ih je potrebno zaobići.



Slika 5.5 Drugi primjer izvršavanja treće implementacije A* algoritma

Kako bismo si ovaj primjer mogli lakše predočiti možemo pogledati tablicu 5.6 na kojoj je označena putanja o početnog do konačnog čvora, ali i čvorovi koji predstavljaju prepreke.

Tablica 5.6 Prikaz rešetke iz drugog primjera

5	5 (1,5)	10 (2,5)	15 (3,5)	20 (4,5)	25 (5,5)
4	4 (1,4)	9 (2,4)	14 (3,4)	19 (4,4)	24 (5,4)
3	3 (1,3)	8 (2,3)	13 (3,3)	18 (4,3)	23 (5,3)
2	2 (1,2)	7 (2,2)	12 (3,2)	17 (4,2)	22 (5,2)
1	1 (1,1)	6 (2,1)	11 (3,1)	16 (4,1)	21 (5,1)
Y \ X	1	2	3	4	5

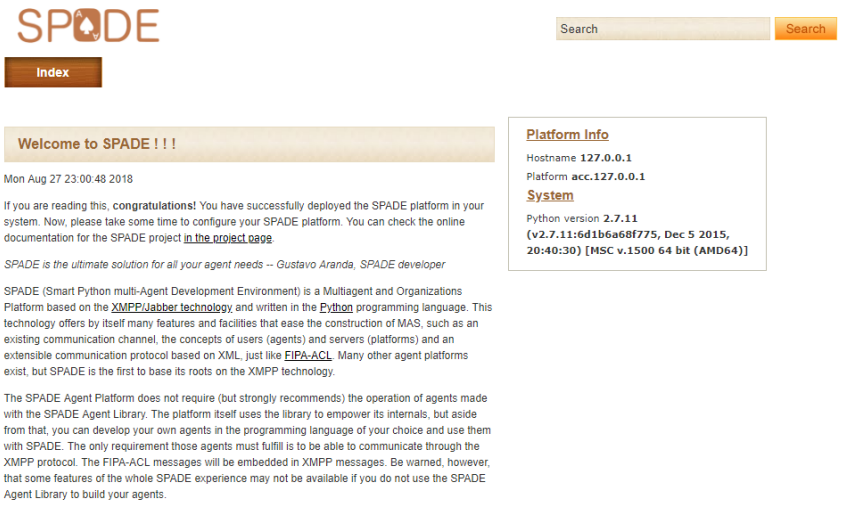
5.2. Implementacija SPADE agenata

Nakon što smo pokazali neke primjere A* algoritma upoznati ćemo se s platformom SPADE. SPADE je akronim te dolazi od engleskih riječi Smart Python Agent Development Enviroment. To je platforma za razvoj višeagentnih sustava koja je napisana u programskom jeziku Python temeljena na brzoj razmjeni poruka (engl. *instant messaging*). Kao posebnu prednost ove platforme ističe se razvoj agenata koji mogu biti u interakciji s drugim agentima, ali i ljudima.

SPADE biblioteka je modul za programski jezik Python za kreiranje SPADE agenata. To je kolekcija klasa, funkcija i alata za kreiranje novih SPADE agenata koji mogu raditi sa SPADE platformom. Prednosti biblioteke, ukoliko ste koristili programski jezik Python, su jednostavnost razumijevanja koncepata i jednostavnost kod korištenja. (Palanca, 2018)

Kako bismo koristili prednosti ove platforme potrebno ju je instalirati. Na GitHub stranici ovoga projekta navode se tri načina za instalaciju: uz pomoć pip-a, uz pomoć easy_installa te preuzimanjem koda s GitHuba te pokretanjem skripte setup.py. Autor je imao problema s instaliranjem SPADE-a korištenjem prva dva načina. Konkretnije, tokom pokušaja instalacije pip je javio pogrešku da nedostaje datoteka *„win32prng.c“*. Identičan problem se dogodio prilikom korištenja easy_installa. Nakon toga je preuzet izvorni kod s GitHub stranice projekta te kopiran unutar direktorija gdje se nalazi korištena verzija jezika Python. Nakon toga je pokrenuta skripta setup.py s parametrom install i instalacija je uspješno započeta. Sama instalacija traje nekoliko trenutaka i uskoro je uspješno završena. Instalirana je verzija platforme SPADE u inačici 2.3.3.

Prije nego se započne s razvojem SPADE agenata, potrebno je konfigurirati i pokrenuti platformu. Platformu ćemo konfigurirati pomoću naredbe „*python configure.py localhost*“, kada se platforma konfigurira dobit ćemo poruku obavijesti. Pomoću ove naredbe smo konfigurirali SPADE da se izvršava na localhostu odnosno na računalu gdje je instalirana platforma. Izvršavanjem te naredbe te pokretanjem skripte kreiraju se konfiguracijske datoteke koje možemo pronaći u etc direktoriju unutar direktorija gdje smo instalirali SPADE. Nakon što smo konfigurirali platformu potrebno ju je pokrenuti. Pokretanje se vrši pomoću naredbe „*python runspade.py*“. Kada se naredba izvrši dobit ćemo poruku da se uspješno izvršila te je moguće vidjeti broj porta koji se koristi za rad platforme. U autorovom slučaju, koristio se port broj 8008. Valja istaknuti kako se konfiguracija i pokretanje radi svaki puta kada želimo koristiti SPADE platformu. Ukoliko otvorimo neki preglednik i upišemo adresu i broj porta vidjet ćemo poruku kao na slici 5.6.



Slika 5.6 Izgled stranice nakon pokretanja platforme SPADE

Sa slike možemo vidjeti osim pozdravne poruke i općih informacija o platformi, podatke vezane uz platformu te podatke o sustavu, primjerice koju inačicu programskog jezika Python koristimo.

Ukoliko želimo, možemo napisati kratku .bat skriptu koju ćemo koristiti kako bismo pokrenuli platformu bez potrebe da se pozicioniramo u direktoriju gdje se nalazi izvorni kod ili upisivanje duge putanje do datoteke. Skripta će se sastojati od dviju linija. Prva linija će se koristiti za promjenu trenutnog direktorija, a druga za izvršavanje naredbe za pokretanje platforme. Primjer skripte se nalazi ispod.

```
cd "C:\Python27\spade\spade-master"
python ./runspade.py
```

Kada smo kreirali ovu skriptu uvelike smo si olakšali posao s pokretanjem platforme jer sada platformu možemo pokrenuti s jednostavnim pritiskom tipke na mišu. Kada smo to napravili imamo sve preduvjete da započnemo koristiti SPADE platformu.

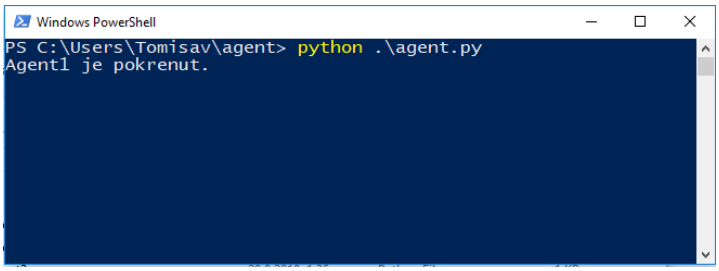
Prije nego što počnemo implementaciju primjera, valja spomenuti kakve vrste agenata postoje unutar platforme. SPADE platforma podržava nekoliko unaprijed definiranih tipova ponašanja agenata poput: cikličkih, periodičkih, agenta koji se izvršava jednom (eng *one-shot*), agenta koji ima istek vremena (engl. *time-out*), konačnog automata (engl. *finite state machine*) i agenta koji reagira na događaje (engl. event behaviour).

U idućem primjeru pokazati ćemo jednostavan primjer agenta koji je kreiran na SPADE platformi.

```
import spade

class FirstAgent(spade.Agent.Agent):
        def _setup(self):
                print "Agent1 je pokrenut."
```

```
if __name__ == "__main__":
    a1 = FirstAgent ("agent1@127.0.0.1", "secret")
    a1.start()
```

U prvoj liniji koda možemo vidjeti poziv biblioteke SPADE nakon čega na raspolaganju imamo klase i metode koje su definirane u toj biblioteci. Nakon toga je definirana klasa FirstAgent koja proizlazi iz klase Agent.Agent koja je općenita klasa za agente u biblioteci. U idućoj liniji definira se _setup metoda koju bi svi agenti trebali nadjačavati jer se ona koristi kod inicijalizacije. U metodi se samo ispisuje tekst na ekran, za što možemo reći da nije složeno ponašanje, ali ovo je tek prvi primjer. U main metodi se kreira objekt a1 klase FirstAgent s dva parametra, od kojih se prvi parametar sastoji od: jedinstvenog identifikatora, znaka @ te lokacije servera na kojemu je pokrenuta platforma, u ovome slučaju je to 127.0.0.1. Nad novokreiranim objektom se poziva metoda start. Na slici 5.7 možemo vidjeti primjer izvršavanja prvog primjera agenta.

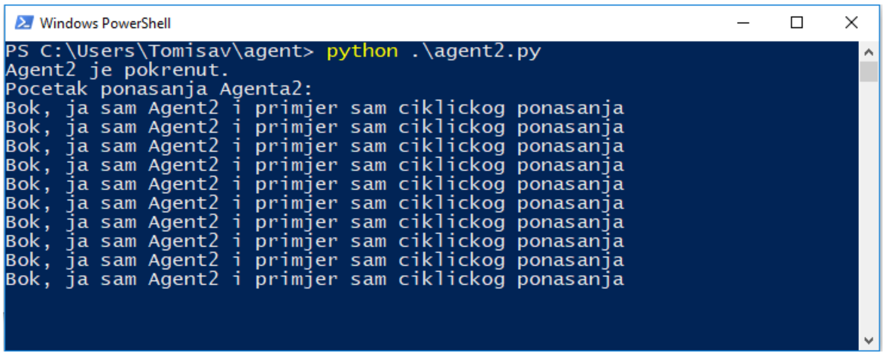


Slika 5.7 Primjer izvršavanja prvog agenta

Prvi primjer agenta nije sofisticiran i nema nikakvo ponašanje te je sam sebi svrha, ali možemo jasno vidjeti kako se kreiraju agenti unutar platforme. U sljedeća dva primjera pokazati ćemo dvije implementacije cikličnog i periodičkog ponašanja agenta.

```
import spade
import time


class SecondAgent(spade.Agent.Agent):
    class Behav(spade.Behaviour.Behaviour):
        def onStart(self):
            print "Pocetak ponasanja Agenta2:"

        def _process(self):
            print "Bok, ja sam Agent2 i primjer sam ciklickog ponasanja"
            time.sleep(1)


    def _setup(self):
            print "Agent2 je pokrenut."
            b = self.Behav()
            self.addBehaviour(b, None)

if __name__ == "__main__":
    a2 = SecondAgent("agent2@127.0.0.1", "secret")
    a2.start()
```

Na početku ovog primjera nalaze se importi dvaju biblioteka: SPADE i time biblioteke. Nakon toga je definirana klasa SecondAgent koja predstavlja agenta. Unutar te klase je definirana klasa Behav koja koristi klasu Behaviour.Behaviour iz biblioteke SPADE. U njoj su definirane dvije metode: start i _process. Metoda start koja će se izvršiti samo jednom te metoda _process koja predstavlja ponašanje agenta. U metodi _proces ispisivati će se tekst i izvršit će se periodički svake sekunde sve dok ne prekinemo agenta u tome jer nije postavljen nikakav uvjet za njegovo zaustavljanje. Unutar SecondAgent klase postoji _setup metoda kao i u prethodnom primjeru te se koristi za ispis poruke, kreiranje objekta tipa Behav te dodavanje kreiranog objekta uz pomoć addBehaviour metode agentu. Nakon toga slijedi main metoda koja je gotovo identična prvom primjeru. Primjer izvršavanja programa možemo vidjeti na slici 5.8



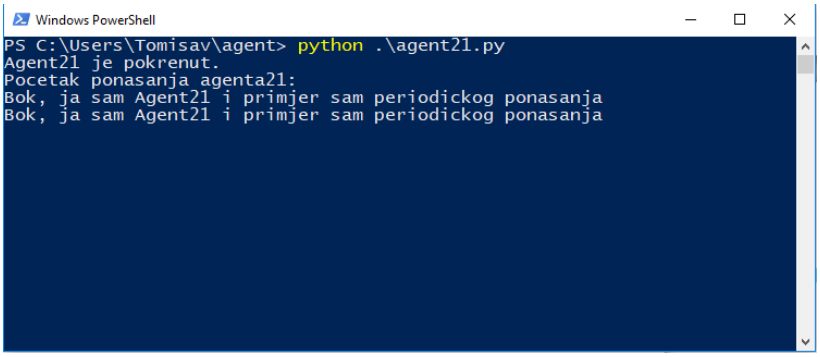
Slika 5.8 Primjer izvršavanja cikličkog ponašanja agenta

Ako usporedimo ovaj primjer s prvim primjerom možemo vidjeti da u drugom primjeru imamo ponašanje agenta. Dakako, vrlo jednostavno ponašanje koje zamjenjuje beskonačnu petlju. U ovome primjeru smo koristili klasu Behaviour iz biblioteke SPADE i ona se koristi za implementaciju raznih oblika ponašanja. Isto tako, umjesto ove klase mogli smo koristiti klasu PeriodicBehaviour koja se koristi specifično za periodičko ponašanje. U sljedećem primjeru koristit ćemo klasu za periodičko ponašanje i uz manje promjene u samome kodu postići ćemo istu funkcionalnost.

```
import spade

class SecondAgent(spade.Agent.Agent):
    class Behav(spade.Behaviour.PeriodicBehaviour):
        def onStart(self):
            print "Pocetak ponasanja agenta21:"

        def _onTick(self):
            print "Bok, ja sam Agent21 i primjer sam periodickog ponasanja"

    def _setup(self):
        print "Agent21 je pokrenut."
        b = self.Behav(5)
        self.addBehaviour(b, None)

if __name__ == "__main__":
```

```
a21 = SecondAgent("agent21@127.0.0.1", "secret")
a21.start()
```

U ovome primjeru koristimo samo biblioteku SPADE za razliku od prethodnog primjera. Prva razlika između ovog i prethodnog primjera je u klasi Behav jer u ovome primjeru koristimo klasu Behaviour.PeriodicBehaviour koja se koristi kod periodičkih agenata. Druga razlika je korištenje klase _onTick umjesto _proces. Iduća razlika je što kod instanciranja klase Behav moramo proslijediti parametar, dok to nismo morali napraviti u prethodnom primjeru. U ovome slučaju taj parametar je broj 5 koji predstavlja broj sekundi koliko će proteći između dva izvođenja metode _onTick. Razlika između ova dva primjera kod izvođenja je što se u drugom primjeru rečenica ispisuje svakih pet sekundi, dok se u prvom primjeru ispis događao svake sekunde. Na slici 5.9 možemo vidjeti primjer izvršavanja ovog programa.



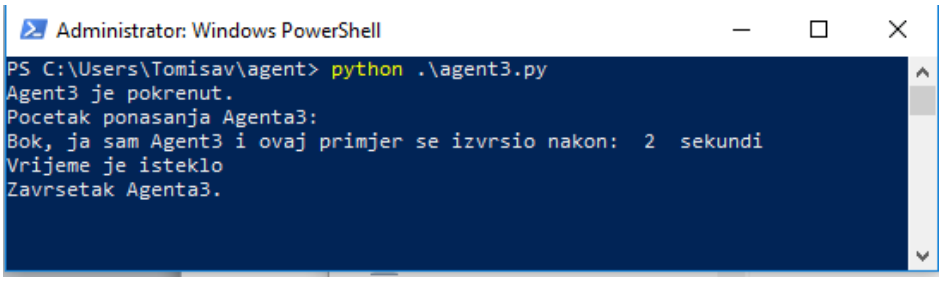
Slika 5.9 Primjer izvršavanja periodičkog ponašanja agenta

U idućem primjeru pokazati ćemo primjer agenta čija radnja se izvršava nakon određenog vremena tj. agenta koji će određenu metodu izvršiti nakon određenog broja sekundi koji mu se proslijedi. Ispod ovoga odlomka nalazi se primjer jednog takvog agenta.

```
import spade

x = 2
class ThirdAgent(spade.Agent.Agent):
        class Behav(spade.Behaviour.TimeOutBehaviour):
                def onStart(self):
                        print "Pocetak ponasanja Agenta3:"

                def timeOut(self):
                        print "Bok, ja sam Agent3 i ovaj primjer se izvrsio
nakon: ", x," sekundi"
                        print "Vrijeme je isteklo"

                def onEnd(self):
                        print "Zavrsetak Agenta3."

        def _setup(self):
                print "Agent3 je pokrenut."
                b = self.Behav(x)
                self.addBehaviour(b, None)
```

```
if __name__ == "__main__":
        a = ThirdAgent("agent3@127.0.0.1", "secret")
        a.start()
```

Na početku uvozimo biblioteku SPADE. Nakon toga, inicijaliziramo vrijednost globalnoj varijabli x koja će nam predstavljati istek vremena, odnosno, broj sekundi nakon kojega će se određena metoda izvršiti. Nakon toga definiramo klasu ThirdAgent i podklasu Behav koja će nam olakšati implementaciju isteka vremena. S metodom onStart smo se upoznali u prošlom primjeru. U ovom primjeru je najvažnija metoda timeOut koja, kao što smo već rekli, se izvršava nakon određenog broja sekundi, konkretno u ovom slučaju to će biti dvije sekunde nakon što se izvrši metoda onStart. Nadalje, u ovome primjeru možemo vidjeti i metodu onEnd koja će se izvršiti nakon metode timeOut. U ostatku programskog koda nije bilo promjena u odnosu na prethodne primjere. Primjer izvršavanja ovog programa možemo vidjeti na slici 5.10.



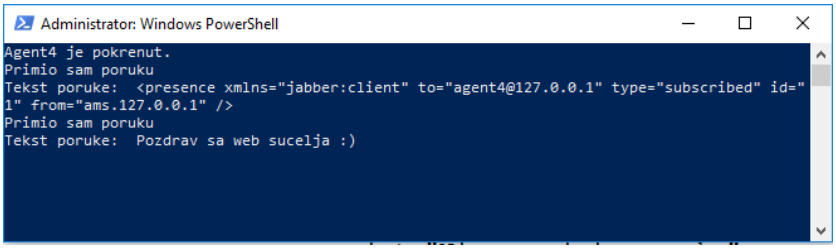
Slika 5.10 Primjer izvršavanja ponašanja agenta nakon određenog vremena

Nakon implementacije jednostavnog agenta te primjera tri vrste ponašanja agenta implementirati ćemo primjere kod kojih se želi pokazati komunikacija između agenata. Kako bi komunikacija bila uspješna moramo znati kome je potrebno poslati poruku. Za ovu zadaću se unutar SPADE platforme koristi jedinstveni identifikator agenta ili AID. U sljedećem primjeru pokazati ćemo primjer agenta koji čeka na poruke, ukoliko dobije poruku ispisuje obavijest da je primio poruku te ispisuje tekst poruke.

```
import spade

class FourthAgent(spade.Agent.Agent):
    class ReceiveBehav(spade.Behaviour.Behaviour):
        def _process(self):
            self.msg = None
            self.msg = self._receive(True, 10)
            if self.msg:
                print "Primio sam poruku"
                print "Tekst poruke: ", self.msg.content
            else:
                print "Nisam primio poruku"

    def _setup(self):
        print "Agent4 je pokrenut."
        rb = self.ReceiveBehav()
        self.setDefaultBehaviour(rb)
```

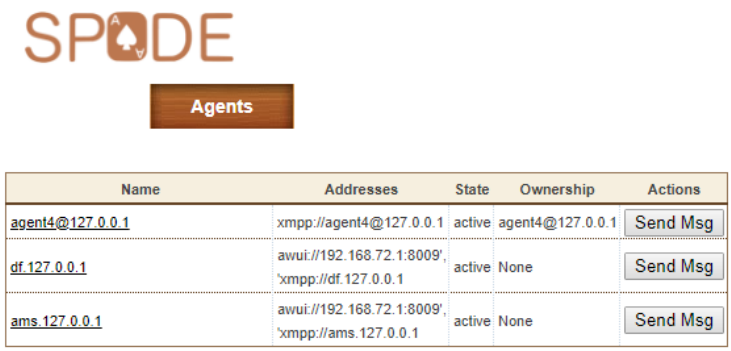
```
if __name__ == "__main__":
    a = FourthAgent("agent4@127.0.0.1", "secret")
    a.start()
```

Iz programskog koda možemo vidjeti da agent ima jednu vrstu ponašanja definiranu u klasi ReceiveBehav. Ta klasa se sastoji od jedne metode koja nosi ime _process i u toj metodi se definira ponašanje agenta nakon što zaprimi poruku. Valja obratiti pozornost da se poruka dohvaća pomoću metode _receive koja se nalazi unutar klase Behaviour.Behaviour platforme SPADE. Kod _setup metode agenta možemo vidjeti još jednu novost, a to je korištenje metode setDefaultBehaviour umjesto addBehaviour. Metoda setDefaultBehaviour se koristi za primanje svih vrsta poruka, pa tako i poruka koje dobije od platforme prilikom kreiranja, primjer jedne takve poruke možemo vidjeti na slici 5.11 u trećoj liniji ispisa konzole. Ukoliko bismo koristili metodu addBehaviour prva poruka ne bi bila ispisana. Odmah nakon pokretanja agenta on ispisuje da je zaprimio poruku, a kasnije prima i poruku koja mu je poslana. Nakon što agent primi poruku, ispisati će obavijest da je zaprimio poruku te će ispisati sadržaj poruke na zaslon.



Slika 5.11 Primjer agenta koji prima poruke

U idućem primjeru implementirati ćemo agenta koji će slati poruku, no u ovom primjeru je poruka agentu bila poslana uz pomoć web sučelja platforme SPADE, pa ćemo to sada pojasniti. Web sučelju možemo pristupiti uz pomoć adrese na kojoj se nalazi server kojemu pristupamo te broja porta. U autorovom slučaju ta putanja je glasila: http://127.0.0.1:8008. 127.0.0.1 predstavlja adresu localhosta, odnosno računala na kojemu je pokrenuta platforma, a nakon nje slijedi broj porta. Kada pristupimo toj adresi otvorit će se početna stranica koju možemo vidjeti na slici 5.6. Elementi navigacijske trake na prvi pogled su sakriveni, ali ukoliko dođemo kursorom desno od kartice Index možemo ih vidjeti. Nas će trenutno zanimati kartica Agents koja se nalazi odmah do kartice Index. Nakon što kliknemo na nju stranica će nas preusmjeriti na stranicu prijave gdje je potrebno upisati lozinku „secret“ kako bismo mogli pristupiti željenoj stranici. Nakon što smo upisali lozinku i pritisnuli gumb Login otvara nam se popis svih agenata koji su trenutno aktivni. Ukoliko smo pokrenuli prethodno navedeni kod trebali bismo vidjeti tablicu poput tablice prikazane na slici 5.12.



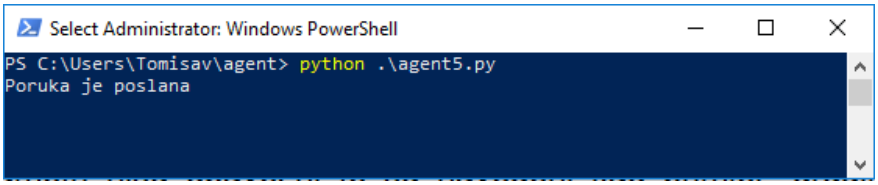
Slika 5.12 Popis svih trenutno aktivnih agenata

Sa slike 5.12 možemo vidjeti da je agent4 aktivan. Ukoliko pritisnemo gumb Send Msg desno od agenta4 otvorit će nam se nova stranica gdje možemo definirati parametre poruke koje želimo poslati agentu. Tekst poruke unosimo unutar odlomka s nazivom Content te pritiskom na gumb Send poruka se šalje agentu. Nakon što smo pokazali kako je moguće poslati agentu poruku uz pomoć web sučelja, implementirati ćemo agenta koji će se koristiti za slanje poruka drugom agentu.

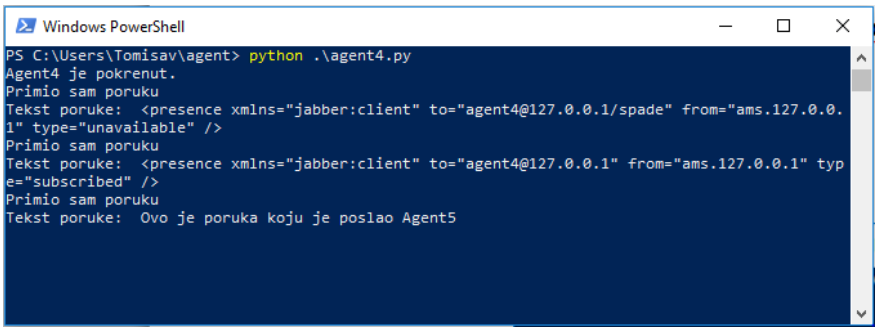
Kao što smo spomenuli kod implementacije četvrtog agenta, metoda _receive nalazi se unutar klase Behaviour.Behaviour. Metoda send, koja će se koristiti za slanje poruka nalazi se unutar klase Agent.Agent. Dakle, za slanje poruke je zadužen agent dok je za primanje poruka odgovorno agentovo ponašanje. Ovo je vrlo bitan koncept i valja ga istaknuti. Ispod ovog odlomka nalazi se programski kod agenta koji je zadužen za slanje poruka.

```
import spade

class FifthAgent(spade.Agent.Agent):
    class Behav(spade.Behaviour.OneShotBehaviour):
        def _process(self):
            receiver = spade.AID.aid(name="agent4@127.0.0.1",
                                     addresses=["xmpp://agent4@127.0.0.1"])
            self.msg = spade.ACLMessage.ACLMessage()
            self.msg.setPerformative("inform")
            self.msg.setOntology("Testing")
            self.msg.setLanguage("OWL-S")
            self.msg.addReceiver(receiver)
            self.msg.setContent("Ovo je poruka koju je poslao Agent5")
            self.myAgent.send(self.msg)
            print "Poruka je poslana"

    def _setup(self):
        b = self.Behav()
        self.addBehaviour(b, None)

if __name__ == "__main__":
    a = FifthAgent("agent5@127.0.0.1", "secret")
    a.start()
```

Unutar klase Behav imamo metogu _process kod koje definiramo primatelja, tijelo poruke te se smanje poruke primatelju. Sada ćemo objasniti liniju po liniju te metode. U prvoj liniji metode _process definiramo objekt receiver koji će nam predstavljati primatelja poruke. Primatelj je objekt klase AID, a kreira se na način da se proslijede parametri name i adresses koji predstavljaju ime i adresu primatelja. Ime se definira na način da koristimo jedinstveni identifikator agenta, nakon čega dolazi znak @, a nakon njega dolazi lokacija servera na kojoj se agent izvršava. U ovome slučaju lokacija je 127.0.0.1. Drugi parametar koji se definira je adresa. Komunikacija u platformi spade se odvija preko protokola XMPP te parametar koji prosljeđujemo treba biti u formatu: xmpp://, nakon čega slijedi jedinstveni identifikator agent, znak @ te lokacija servera na kojoj se agent nalazi. Valja napomenuti kako se parametar adresses treba biti lista, jer ukoliko ne proslijedimo listu, poruka neće biti poslana primatelju. U sljedećoj liniji definiramo objekt msg koji je instanca klase ACLMessage. Klasa ACLMessage temelji se na FIPA-ACL standardu. (Aranda, Palanca, 2005) FIPA je akronim od riječi Foundation for Intelligent Physical Agents. FIPA predstavlja međunarodnu neprofitnu organizaciju koja se zalaže promociju industrije inteligentnih agenata. ACL je akronim od riječi Agent Communication Language i njime se definira oblik poruke koji se koristi za komunikaciju između agenata. (FIPA, 2002) Nakon što smo definirali msg objekt, definirati ćemo neke njegove parametre poput performative (engl. *performative*), ontologije, jezika, primatelja i sadržaja. Performativa će definirati o kakvom tipu komunikacije se radi u poruci, ontologija se postavlja naziv ontologije poruke dok se jezikom postavlja jezik, ovi parametri nisu obavezni te su navedeni kao primjer, drugim riječima moguće je poslati poruku bez njih. Nakon toga, pridružit ćemo objekt primatelja poruci te dodati sadržaj poruke. U pretposljednjoj liniji metode _process poruka se šalje prema agentu4. U toj liniji možemo kako pristupiti agentu koristeći prečac myAgent. Nakon slanja poruke agent5 ispisuje tekst kako bismo mogli provjeriti da li je poruka uspješno poslana. Na slici 5.13 možemo vidjeti primjer izvršavanja agenta5, dok na slici 5.14 možemo vidjeti kako se ponaša agent4, koji je implementiran u prošlom primjeru, nakon primanja poruke.



Slika 5.13 Primjer agenta koji šalje poruke



Slika 5.14 Agent ispisuje poruku primljenu od drugog agenta

Nakon što smo pokazali nekoliko primjera agenata, njihovog ponašanja i primjer komunikacije, u idućem primjeru pokazati ćemo dva agenta koji će na neki način objediniti sve dosadašnje primjere. Implementirati će se primjer koji će se sadržavati dva agenta, tako da možemo reći da se radi o višeagentnom sustavu. Agenti će raditi zajedno na način da će jedan agent obavijestiti drugoga o preprekama koje mu se nalaze na putu do cilja. Cilj u idućem primjeru je napraviti agenta koji će koristit prethodno implementirani A* algoritam za određivanje puta od početne do konačne točke. Ispod ovoga odlomka nalazi se programski kod.

```
import spade
import random

gridx = 5
gridy = 5
numberOfRandomNumbers = 5
startid = 1
endid = 24

class Node(object):
    def __init__(self, x, y, id):
        self.isObsticle = False
        self.gCost = None
        self.hCost = None
        self.x = x
        self.y = y
        self.neighbours = []
        self.parent = None
        self.id = id
    def getFCost(self):
        return self.gCost - self.hCost


class Grid(Node
    def __init__(self, x, y, obsticalList):
        self.gridX = x;
        self.gridY = y;
        self.nodes = []
        self.obsticalList = obsticalList
        i = 0
        j = 0
        id = 1
        while i < x:
            i += 1
```

```
            while j < y:
                j += 1
                n = Node(i, j, id)
                self.nodes.append(n)
                id += 1
            j = 0
        self.addNeighbours(x, y)
        self.setObsticals()


    def returnNodeByID(self, id):
        for node in self.nodes:
            if node.id == id:
                return node

    def returnNodeIDByCoordints(self, x, y):
        for node in self.nodes:
            if node.x == x and node.y == y:
                return node.id

    def addNeighbours(self, gridx, gridY):
        for node in self.nodes:
           for xi in range(-1, 2):
               for yi in range(-1, 2):
                    if xi == 0 and yi == 0:
                       continue
                    else:
                        verifyX = node.x + xi
                        verifyY = node.y + yi
                        if verifyX > 0 and verifyX <= gridx and verifyY > 0
and verifyY <= gridY:

node.neighbours.append(self.returnNodeIDByCoordints(verifyX, verifyY))

    def setObsticals(self):
        if self.obsticalList:
            for obstical in self.obsticalList:
                for node in self.nodes:
                    if obstical == node.id:
                        node.isObsticle = True

    def setGCost(self, end):
        endNode = self.returnNodeByID(end)
        for node in self.nodes:
            if node == endNode:
                node.gCost = 0
            else:
                node.gCost = self.getDistance(node, endNode)

    def getDistance(self, nodeA, nodeB):
        distA = abs(nodeA.x - nodeB.x)
        distB = abs(nodeA.y - nodeB.y)
        if distA > distB:
            return 14 * distB + 10*(distA - distB)
        return 14 * distA + 10*(distB - distA)


class astar_Solver:
    def __init__(self, gridX, gridY, start, goal, obsticalList):
        self.grid = Grid(gridX, gridY, obsticalList)
        self.start = start
```

```
        self.goal = goal
        self.openset = []
        self.closedset = []
        self.finalPath =[]
        self.grid.setGCost(self.goal)
        self.openset.append(self.grid.returnNodeByID(start))

    def solve(self):
         while (self.openset):
            currentNode = self.openset[0]
            for i in range(len(self.openset)-1):

                if (self.openset[i].getFCost() < currentNode.getFCost() or
self.openset[i].getFCost()       ==       currentNode.getFCost()       and
self.openset[i].hCost < currentNode.hCost):
                    currentNode = self.openset[i]
            self.openset.remove(currentNode)
            self.closedset.append(currentNode)

            if currentNode == self.grid.returnNodeByID(self.goal):
                self.finalPath = self.getFinalPath(self.start, self.goal)
                self.printFinalPath()
                return

            for x in currentNode.neighbours:
                neighbour = self.grid.returnNodeByID(x)
                if neighbour in self.closedset or neighbour.isObsticle:
                    continue

                movementCostToNeighbour      =      currentNode.gCost      +
self.getDistance(currentNode, neighbour)
                goalNode = self.grid.returnNodeByID(self.goal)
                if movementCostToNeighbour < neighbour.gCost or neighbour
not in self.openset:
                    neighbour.gCost = movementCostToNeighbour
                    neighbour.hCost = self.getDistance(neighbour, goalNode)
                    neighbour.parent = currentNode

                    if neighbour not in self.openset:
                        self.openset.append(neighbour)

    def getDistance(self, nodeA, nodeB):
        distA = abs(nodeA.x - nodeB.x)
        distB = abs(nodeA.y - nodeB.y)
        if distA > distB:
            return 14 * distB + 10*(distA - distB)
        return 14 * distA + 10*(distB - distA)

    def getFinalPath(self, start, end):
        reversePath = []
        startNode = self.grid.returnNodeByID(start)
        currentNode = self.grid.returnNodeByID(end)
        while currentNode != startNode:
            reversePath.append(currentNode.id)
            currentNode = currentNode.parent
        reversePath.append(startNode.id)
        return reversePath[::-1]

    def printFinalPath(self):
        print self.finalPath
```

```
class SixthAgent(spade.Agent.Agent):
        class Behav(spade.Behaviour.TimeOutBehaviour):
            def _process(self):
                self.msg = None
                self.msg = self._receive(True, 3)
                if self.msg:
                    print "Agent6: Primio sam poruku"
                    print "Agent6: Tekst poruke: ", self.msg.content
                    message = self.msg.content
                    obsticalList = self.convert(message)
                    if startid in obsticalList:
                        obsticalList.remove(startid)
                    if endid in obsticalList:
                        obsticalList.remove(endid)
                    a   =   astar_Solver(gridx,   gridy,   startid,   endid,
obsticalList)
                    print "Agent6: Putanja do cilja je:"
                    a.solve()

                else:
                    print "Agent6: Nisam primio poruku"

            def convert(self, originalString):
                l2 = []
                s1 = originalString.replace("[", "")
                s2 = s1.replace("]", "")
                s3 = s2.replace(",", "")
                l1 = list(s3.split(" "))
                for item in l1:
                    try:
                        x = int(item)
                        l2.append(x)
                    except:
                        print  "Agent6:  Doslo  je  do  pogreske  prilikom
pretvanja stringova u integere"
                return l2

        def _setup(self):
                print "Agent6: Agent6 je pokrenut."
                b = self.Behav(4)
                message_template = spade.Behaviour.ACLTemplate()
                message_template.setOntology("Example")
                mt = spade.Behaviour.MessageTemplate(message_template)
                self.addBehaviour(b, mt)
                #self.setDefaultBehaviour(b, mt)

class SeventhAgent(spade.Agent.Agent):
        class Behav(spade.Behaviour.OneShotBehaviour):
            def _process(self):
                self.receiver = spade.AID.aid(name="agent6@127.0.0.1",
                                        addresses=["xmpp://agent6@127.0.0.1"])
                self.msg = spade.ACLMessage.ACLMessage()
                self.listN = self.defineObsticals()
                self.msg.setPerformative("inform")
                self.msg.setOntology("Example")
                self.msg.setLanguage("OWL-S")
                self.msg.addReceiver(self.receiver)
                self.msg.setContent(self.listN)

            def onEnd(self):
                self.myAgent.send(self.msg)
```

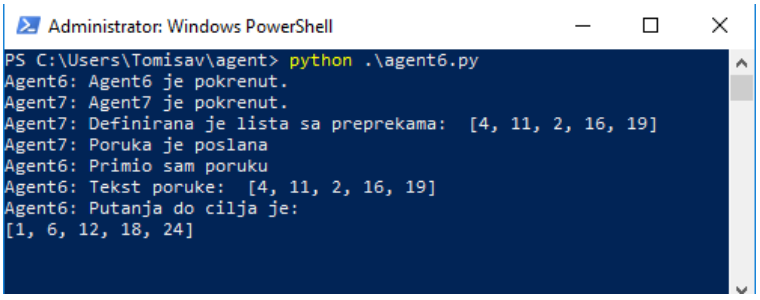
```
                print "Agent7: Poruka je poslana"

            def defineObsticals(self):
                listN = []
                for x in range(numberOfRandomNumbers):
                    listN.append(random.randint(1, gridx*gridy))
                print "Agent7: Definirana je lista s preprekama: ", listN
                return listN

        def _setup(self):
                print "Agent7: Agent7 je pokrenut."
                b = self.Behav()
                self.addBehaviour(b, None)


if __name__ == "__main__":
    a6 = SixthAgent("agent6@127.0.0.1", "secret")
    a6.start()
    a7 = SeventhAgent("agent7@127.0.0.1", "secret")
    a7.start()
```

Na početku programskog koda uključuju se dvije biblioteke koje ćemo koristiti: SPADE i random. Biblioteka random će se koristiti za generiranje pseudo slučajnih brojeva. Nakon toga definira se nekoliko globalnih varijabli. Globalne varijable su gridx, gridy, numberOfRandomNumbers, startid i endid. Gridx varijabla predstavlja broj stupaca u rešetki, dok gridy predstavlja broj redaka koji ćemo imati u rešetci. NumberOfRandomNumbers predstavlja koliko će polja na rešetci biti postavljeno kao prepreka. Startid predstavlja jedinstveni identifikator polja na rešetci koji je uzet kao polazište za kretanje, dok endid predstavlja jedinstveni identifikator odredišnog polja na rešetci. Nakon postavljanja vrijednosti globalnih varijabli slijedi dio koda koji je preuzet iz trećeg primjera A* algoritma iz ovog diplomskog rada. Taj dio koda je pojašnjen i ovdje ga nećemo opet pojašnjavati. Kod s kojim se nismo dosad susreli počinje klasom SixthAgent, a završava main metodom. Ovoga puta prilikom objašnjavanja programskog koda krenuti ćemo od kraja prema početku, tako ćemo opisati main metodu, pa sedmog agenta pa šestog agenta. U main metodi prvo se kreira instnca šestog agenta te se nakon toga pokreće šesti agent, odmah nakon toga kreira se sedmi agent te se on pokreće.

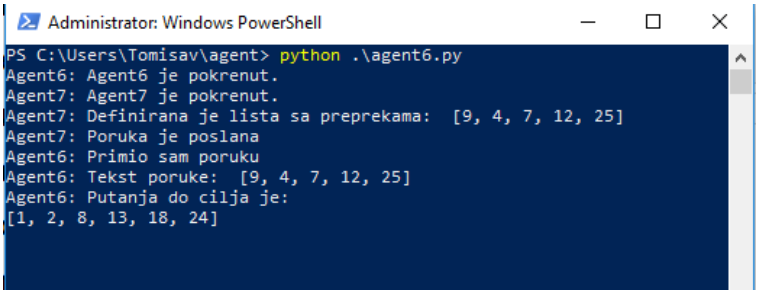
Kod sedmoga agenta tj agenta definiranog u klasi SeventhAgent postoji podklasa Behav koja je odgovorna za ponašanje agenta. Ovaj agent izvodi svoje ponašanje samo jednom (engl. OneShotBehaviour). U _process metodi se definira poruka koja će biti poslana šestom agentu, te se postavljaju određeni parametri poput performative, ontologije, jezika i sl. S definiranjem poruke susreli smo se kod agenta koji se koristio za slanje poruka. Najzanimljivija metoda kod ovog agenta je metoda defineObsticals kod koje se generira lista pseudo slučajnih brojeva. Koliko pseudo-sučajnih brojeva će se generirati ovisi o parametru numberOfRandomNumbers. Lista generiranih brojeva poslati će se šestom agentu koji će ju

koristiti kako bi izbjegao prepreke na rešetci. Pošiljanje poruke šestom agentu implementirano je unutar metode onEnd.

Na poslijetku dolazimo do šestog agenta odnosno agenta koji je implementiran unutar klase SixthAgent. Šesti agent je implementiran na način da samo prihvaća poruke koje koriste ontologiju s imenom Example. Ukoliko stigne takva poruka, agent će ju nastojati obraditi. Ova funkcionalnost je implementirana uz pomoć ACL predloška. Kada agent primi poruku, prvo ju pretvara iz znakovnog niza (engl. *string*) u brojeve, te provjerava da li se u listi nalazi jedinstveni identifikator njegove početne ili konačne pozicije i ukoliko je to slučaj, ukloniti će te brojeve iz liste. Nakon toga poziva A* algoritam uz korištenje metode astar_Solver kojoj će proslijediti parametre poput broja stupaca u rešetci, broja redaka u rešetci, jedinstvenog identifikatora početnog polja, jedinstvenog identifikatora konačnog polja u rešetci te liste u kojoj se nalaze jedinstveni identifikatori prepreka. Kao rezultat ove metode agent će dobiti listu s jedinstvenim identifikatorima polja koja mora posijetiti kako bi došao od početne pozicije do konačne pozicije. Primjer izvršavanja ovog programa možemo vidjeti na slikama 5.15 te 5.16.



Slika 5.15 Prvi primjer izvođenja agenta koji koristi A* algoritam



Slika 5.16 Drugi primjer izvođenja agenta koji koristi A* algoritam

U programu je kao početna pozicija postavljeno polje s jedinstvenim identifikatorom 1, dok je kao cilj postavljeno polje s jedinstvenim identifikatorom 24. Sa slika možemo vidjeti na kojima se lokacijama nalaze prepreke, tj polja na rešetci preko kojih agent ne može preći i možemo vidjeti put koji bi odabrao određeni agent kako bi došao s početne do konačne pozicije na rešetci.

6. Zaključak

Na početku ovoga rada smo se upoznali s definicijama agenata, višeagentnih sustava, namjernog ponašanja te što će slijediti u idućim poglavljima.

Nakon toga smo posvetili jedno poglavlje umjetnoj inteligenciji. Navedeno je nekoliko definicija umjetne inteligencije prema različitim autorima te su naglašene ideje koje stoje iza tih definicija. Isto tako naveden je problem zbog kojega je teško definirati umjetnu inteligenciju. U idućih nekoliko poglavlja bavili smo se modernom poviješću umjetne inteligencije. Godina koja je uzeta kao početna točka je 1943. godina. U tim poglavljima su opisano nekoliko događaja koji su bili važni za razvoj umjetne inteligencije od Turingovog testa, preko konferencije u Dartmouthu pa sve do nedavnih trendova u umjetnoj inteligenciji poput Bayesovih mreža, big data te dubokokog učenja.

U trećem poglavlju detaljnije smo se bavili agentima i višeagentnim sustavima. Navedeni su trendovi koju su obilježili računalstvo te su ukratko objašnjeni. U istom poglavlju, navedene su definicije agenata različitih autora te je opisano koji je autor stavio naglasak na po njemu bitne karakteristike agenata. Nakon toga navode se bitne komponente svakog inteligentnog računalnog agenta. Na kraju ovoga poglavlja nalaze se definicije višeagentnih sustava te kratko obrazloženje samih definicija.

Četvrto poglavlje posvećeno je teoretskoj strani planiranja. Valja ponoviti da se planiranje sastoji od odabira i organiziranja radnji kojima ćemo ispuniti zadani cilj, a kao konačni rezultat kreirati će se plan. Planiranje, sa stajališta domene, možemo podijeliti u dvije velike cjeline: planiranje nezavisno od domene i planiranje specifično za određenu domenu. Kod planiranja specifičnog za domenu posebno se spominju: planiranje pomaka i kretanja, planiranje percepcije, planiranje navigacije te planiranje operacijama. Kod planiranja nezavisnog od dome posebno se spominju: planiranje projekata, raspored i alokacija resurse te sinteza plana. Nakon toga posebno pod poglavlje je posvećeno konceptualnom modelu kod planiranja. Da ponovimo, konceptualni model je teoretski zapis za opisivanje elemenata nekog problema.

U zadnjem poglavlju pobliže smo se upoznali s algoritmom A* koji se može koristit za planiranje puta. Isto tako implementirali smo ovaj algoritam tri puta. Jedamputa se koristio kao neka vrsta algoritma za sortiranje, dok je u iduća dva primjera korišten za pronalazak puta od startne pozicije do odredišne pozicije. Nakon toga, posvetili smo se implementaciji agenata od posve jednostavnog primjera, preko različitih tipova ponašanja kod agenata i slanja i primanja poruka, pa sve do završnog primjera u kojemu agent koristi A* algoritam za pronalazak puta. U zadnjem primjeru pokušalo se objediniti više primjera.

Popis literature

1. Ferber, J., (1999) *Multi-Agent System: An Introduction to Distributed Artificial Intelligence*. Dostupno: 22.5.2017 na: http://jasss.soc.surrey.ac.uk/4/2/reviews/rouchier.html
2. Maess, P., (1995) *Artificial life meets entertainment: Life like autonomous agents*
3. Wooldridge, M., (2002) *An Introduction to Multiagent Systems*
4. Ghallab, M., Nau, D., Traverso, P., (2016) *Automated Planning and Acting*
5. Ghallab, M., Nau, D., Traverso, P., (2004) *Automated Planning: Theory and Practise*
6. Nau, D., (2007) *Current Trends in Automated Planning*
7. Hrvatski jezični portal, *namjera, djelovanje, plan* Dostupno 5.6.2018 na: http://hjp.znanje.hr/index.php?show=search
8. Luger, G. F., (2009) *Artificial intelligence Structures and Strategies for Complex Problem Solving*, sixth edition
9. Buchanan, B. G., (2006) *A (Very) Brief History of Artificial Intelligence,* AI Magazine Volume 26
10. McCarthy, J., Minsky, M. L., Rochester, N., Shannon, C. E. (1955) *A Proposal for the Darthmouth Summer Research Project on Artificial Intelligence* Dostupno 9.7.2018 na: http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html
11. Russell, S., Norvig P., (2010) *Artificial Intelligence A Modern Approach*, Third edition
12. McCorduck, P., (2004) *Machines Who Think*
13. Newell, A., Shaw, C. J., Simon, H. A., (1958) *Report on a General Problem-Solving Program*, Dostupno 25.7.2018 na: http://bitsavers.trailing-edge.com/pdf/rand/ipl/P-1584_Report_On_A_General_Problem-Solving_Program_Feb59.pdf
14. Goodfellow, I., Bengio, J., Courville, A., (2016) *Deep learning*
15. LeCun, Y., Bengio, Y., Hinton, G., (2015) *Deep learning*
16. Wikibooks, (2015) *Python Programming*, Dostupno 1.8. 2018 na: https://upload.wikimedia.org/wikipedia/commons/9/91/Python_Programming.pdf
17. Palanca, J., (2018) *SPADE Documentation, Release 3.0.0* Dostupno 25.8. 2017 na: https://media.readthedocs.org/pdf/spade-mas/feature-3.0/spade-mas.pdf

18. Aranda, G., Palanca, J., (2005) *SPADE User's Manual For SPADE 2.1*, Dostupno 5.9.2018 na: https://pythonhosted.org/SPADE/index.html

19. FIPA, Foundation for Intelligent Physical Agents, (2002) *FIPA ACL Message Structure Specification*, Dostupno 5.9.2018 na: http://www.fipa.org/specs/fipa00061/SC00061G.pdf

20. Stone, P., Veloso, M., (2000) *Multiagent systems: A survey from a machine learning perspective*

21. Trevor Payne, (2013) *Let's Learn Python #20 - A* Algorithm* Dostupno: 5.9.2018 na: https://www.youtube.com/watch?v=ob4faIum4kQ&t=195s

22. Sebastian Lague, (2014) *A* Pathfinding (E01 – E03)* Dostupno 5.9.2018 na: https://www.youtube.com/watch?v=-L-WgKMFuhE

23. Javidx9, (2017) Path Planning - A* (A-Star) Dostupno: 5.9.2018 na: https://www.youtube.com/watch?v=icZj67PTFhc

24. Duchoň, F., Babinec, A., Kajan, M., Beňo, P., Floreka, M., Fico, T., Jurišica, L., (2014) *Path planning with modified A star algorithm for a mobile robot*, Dostupno 12.9.2018 na: https://www.sciencedirect.com/science/article/pii/S187770581403149X

Popis slika

Popis slika treba biti izrađen po uzoru na indeksirani sadržaj, te upućivati na broj stranice na kojoj se slika može pronaći.

Popis tablica

Popis tablica treba biti izrađen po uzoru na indeksirani sadržaj, te upućivati na broj stranice na kojoj se tablica može pronaći.

Prilozi (1, 2, …)