SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

Ivan Marković

TEORIJA IGARA U VIŠEAGENTNIM SUSTAVIMA

DIPLOMSKI RAD

Varaždin, 2014.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ivan Marković

Matični broj: 40565/11–R

Studij: Informacijsko i programsko inženjerstvo

TEORIJA IGARA U VIŠEAGENTNIM SUSTAVIMA

DIPLOMSKI RAD

Mentor:

Doc.dr.sc. Markus Schatten

Varaždin, srpanj 2014.

Sadržaj

1. Uvod

Naziv ovog rada je teorija igara u višeagentnim sustavima. Takav naziv otkriva da će u radu biti obrađene dvije različite discipline. Iako je teorija igara nastala i prije same pojave prvih računalnih sustava, a kamoli i koncepta višeagentnih sustava, ove dvije discipline počivaju na sličnim temeljnim načelima. Pojednostavljeno rečeno, obje discipline opisuju sustave entiteta u međusobnoj interakciji koja utječe na okolinu u kojoj se nalaze. Teorija igara prikazuje matematičkim modelima odnose između tzv. igrača u interakciji koje se naziva igrom i kojom mijenjaju svoju korist u ovisnosti o potezima koje odigravaju. Višeagentni sustavi istu stvar opisuju kao komunikaciju porukama između samostalnih računalnih programa kojom mijenjaju okolinu u kojoj se nalaze.

U ovom radu će se dati temeljni uvodi u obje discipline. U prvom poglavlju bit će riječ o teoriji igara, povijesti i formalnom opisu te vrstama različitih igara. U drugom poglavlju bit će objašnjeno što je to agent, što je višeagentnih sustav i također će biti dan formalni opis. Praktični rad sadržava dvije različite aplikacije, koje odgovaraju dvama različitim implementacijama igre dilema zatvorenika. Prva je implementacija klasične ponavljajuće dileme zatvorenika, dok je druga također implementacija spomenute igre, ali u prostornom okruženju, gdje lokacija igrača, odnosno agenata dolazi do izražaja. Aplikacija je implementirana u SPADE platformi za izradu. Na kraju će biti obrazloženi rezultati testiranja aplikacija s usporedbom između dvaju različitih implementacija iste igre.

2. Teorija igara

Teorija igara je znanstvena disciplina kojoj je cilj pronalaženje matematičkih modela procesa donošenja odluka s više sudionika. Modeli takvih procesa u teoriji igara se nazivaju igrama. Svoju primjenu pronalazi u raznim granama znanosti i društva poput ekonomije, biologije, psihologije, sociologije, vojske i računalstva. Modeliraju se sustavi u kojima više sudionika međusobno utječe na donošenje odluka i posljedice tih odluka.

Svoju primjenu, teorija igara je prvo pronašla u ekonomiji. Razlog tome je prepoznavanje određenih situacija u kojima se pojedinci i organizacije nalaze u sukobu s drugim pojedincima i organizacijama. U takvim situacijama donošenje pojedinih odluka može biti ključno za opstanak organizacije. Postoji još mnogo primjera takvih situacija, poput političkih stranaka u parlamentima demokratskih država, sportskih organizacija, općenito u svim društvenim disciplinama koje sadrže entitete čiji se interesi sukobljavaju s interesima drugih. Slične sukobe pronalazimo i u jednostavnijim interakcijama više sudionika koje nazivamo igrama. Popularne igre poput šaha i „*monopolya*“ su upravo nastale na temelju sukoba iz stvarnog života. Zbog svoje jednostavnosti, igre mogu poslužiti kao odlični modeli za pronalaženje najboljih rješenja spomenutih konfliktnih situacija. U ovom poglavlju će biti dan uvod u teoriju igara, povijest teorije, klasifikaciju igara i formalni opis.

2.1. Povijest teorije igara

„Ocem“ teorije igara mnogi smatraju matematičara i pionira modernog računalstva Johna von Neumanna. Iako je Borel 20-ih godina objavio nekoliko članaka u kojima je opisivao slične probleme, von Neumann je postavio matematičke temelje teoriji u svojim člancima 1928. i 1937. Sve dotadašnje ideje je zajedno s Morgensternom objedinio u knjizi *Theory of games and Economic Behaviour* 1944. i u drugom izdanju 1947. godine. U uvodu u knjigu von Neumann i Morgenstern (1947:1) navode da će u knjizi dati primjenu teorije matematike na „strateške igre“ razvijene od strane njih dvojice 1928. i 1940. – 1941. godine i na koji način one opisuju temeljne ekonomske probleme. Iako su se prvotno razmatrali ekonomski problemi, velik je utjecaj u motivaciji razvoja teorije igara imao i drugi svjetski rat u kojem su se pokušale modelirati ratne situacije. Takve situacije su većinom uključivale dva suprostavljena sudionika.

Vrlo bitna je 1949. Godina kada je John Forbes Nash objavio svoju doktorsku disertaciju nazvanu *Non-Cooperative Games*, gdje je uveden koncept *točke ravnoteže (eng.*

equilibrium), danas nazvane Nashov *ekvilibrij* za koju je kasnije dobio i Nobelovu nagradu. (Thomas, 1986:20). U idućim godinama, teorija igara se počela primjenjivati u sve više disciplina poput politike i pregovaranja u ekonomiji. Knjiga Kuhna i Tuckera *Theory of Games* daje dobar pregled problema koji su se tada razmatrali kroz četiri izdanja 1950., 1953., 1957., 1959.

Još jedan bitan događaj za teoriju igara dogodio se 1984. izlaskom knjige Roberta Axelroda *The Evolution of Cooperation* u kojoj je proučavao teoriju igara s malo drugačije strane nego prije, sa strane suradnje. Pokazao je kako je međusobna kontinuirana suradnja u sustavima sebičnih agenata poželjna i stvara dobrobit svim stranama. Organizirao je turnir igranja iterirane igre dileme zatvorenika u kojem je pokazao kako se isplati surađivati u dugoročnoj interakciji.

2.2. Tipovi igara

Igre u teoriji igara se mogu podjeliti prema nekoliko kriterija. S obzirom na broj igrača mogu biti:

- Igre s jednim igračem – nisu posebno zanimljive teoriji igara
- Igre s dva igrača – najčešći tip igara proučavan u teoriji igara
- Igre s više igrača

S obzirom na ukupnu dobit u igri mogu biti igre nulte sume i one koje to nisu. Igre nulte sume su one igre u kojima je razlika između dobiti pobjednika i gubitka gubitnika nula. Jednostavnije rečeno, pobjednik igre dobiva upravo onoliko koliko gubitnik gubi.

S obzirom na količinu informacija igre mogu biti: igre potpune informacije i igre nepotpune informacije. U igrama potpune informacije funkcija dobiti i dostupne strategije svakog igrača su opće znanje svih sudionika igre. Primjer takve igre je šah gdje je za svaki potez poznata posljedica. Primjer igre nepotpune informacije bi bila aukcija, u kojoj nije poznato koliko pojedini sudionik vrednuje predmet aukcije.

Igre zatim mogu biti iterirane, što znači da isti suparnici igraju više puta istu igru i s obzirom na ishode mogu prilagođavati i svoju strategiju.

Još jedna podjela su simetrične i nesimetrične igre. U simetričnim igrama, transponiranjem matrice dobiti jednog igrača možemo dobiti matricu dobiti drugog igrača.

2.3. Terminologija teorije igara

U teoriji igara postoje termini koji opisuju samu teoriju i koriste se u modeliranju igara. Sam pojam igre u teoriji igara predstavlja model konfliktne situacije između dva ili više sudionika. Kao što sama definicija modela govori, igrom se pokušava sažeti ključne probleme u sukobu interesa (Thomas L.C., 1986:17). Idući termin su igrači. Igrače opisujemo kao sudionicima igara i oni su modeli stvarnih entiteta koji sudjeluju u nekoj konfliktnoj interakciji.

Sljedeći pojam je potez. Svaka igra se sastoji od niza poteza. Potez definiramo kao događaj u igri koji je posljedica neke odluke igrača ili slučajnog događaja. Na primjer, u pokeru je prvi potez slučajni događaj dodjeljivanja karte (Thomas L.C., 1986:18)

Dobit predstavlja vrijednost na kraju igre. Posljedica je prethodno odigranih poteza. Dobit se najčešće izražava brojčano, a u nekim igrama je pojednostavljena, u smislu da ako je igrač pobjedio u igri dobiva +1 bod, ako je bilo nerješeno 0 i ako je izgubio -1. Korisnost predstavlja prioritet u igri. (Thomas L.C., 1986:18) navodi primjer jednostavne igre „nogometna utakmica ili kino“ gdje igrač odlučuje hoće li ići na nogometnu utakmicu ili u kino. Ako se korisnost nogometne utakmice označi s $u(FM)$, a korisnost kina s $u(C)$, tada prioritet nogometne utakmice nad kinom se označuje s $u(FM) > u(C)$. Ako se malo zakomplicira situacija i kaže da: „Prioritet ima nogometna utakmica nad kinom ukoliko ne pada kiša“ tada možemo postaviti ovakav prioritet: $u(FM\ suho) > u(C) > u(FM\ kiša)$.

Sljedeći vrlo bitan termin u teoriji igara je strategija. Strategija predstavlja opis svih odluka koje igrač donosi u bilo kojoj situaciji koja se može dogoditi u igri. Na primjer, u igri „kružić – križić“, strategija predstavlja odluku o tome u koji od preostalih polja staviti kružić ili križić u bilo kojem trenutku igre. Strategija se može shvatiti i kao stablo odlučivanja u kojem je definirana svaka situacija koja se može dogoditi u igri. Prikaz strategije kao stabla odlučivanja može biti pogodno za male igre, no recimo na primjeru šaha, već u prvom potezu igrač ima mogućnost odigravanja 20 različitih scenarija.

2.4. Normalna i ekstenzivna forma igre

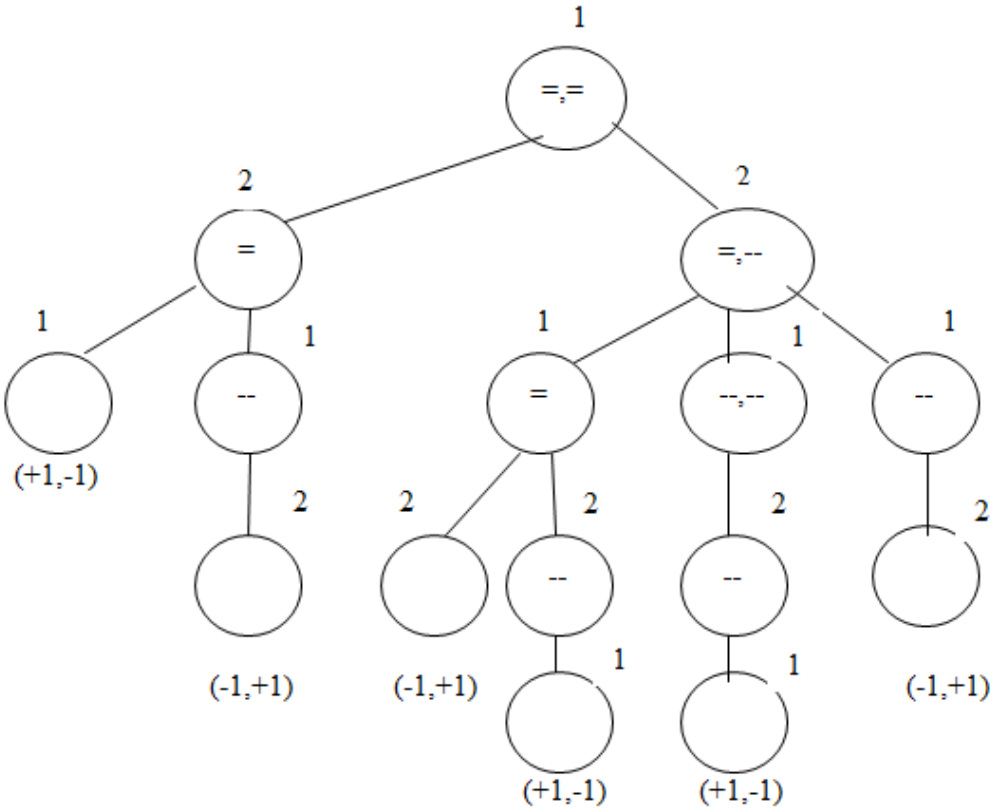
Za opis normalne i ekstenzivne forme igre koristit će se primjer igre: *Izvlačenje šibica.* Opis igre je ovakav:

Određen broj šibica je podijeljen u dvije grupe. Igru igraju dva igrača. Igrači se izmjenjuju u uzimanju šibica. U svakom koraku igrač mora uzeti najmanje jednu šibicu. Može uzeti više pod uvjetom da su sve iz iste grupe. Gubitnik je onaj koji uzme zadnju šibicu. Zbog jednostavnosti primjera će se pretpostaviti igra s 4 šibice podijeljene u grupe po dvije.

Ekstenzivna forma igre služi za detaljno specificiranje igre. Njome se prikazuju sva moguća stanja igre i svi mogući potezi igrača u bilo kojem trenutku igre. (Gibbons R., 1992:115) definira ekstenzivnu formu igre:

Ekstenzivna forma kao reprezentacija igre specificira: (1) igrače u igri, (2a) kada igrač ima potez (2b), što svaki igrač može učiniti u svakoj prilici za potez i (3) dobit svakog igrača za kombinaciju moguću kombinaciju poteza

Prikladan prikaz ekstenzivne forme igre je graf ili stablo. Primjer igre sa šibicama u ekstenzivnoj formi bi izgledao ovako:



Slika 1: Prikaz ekstenzivne forme igre *Uzmi šibicu*

Čvor grafa prikazuje trenutno stanje igre (koliko je šibica preostalo u kojoj grupi) i koji igrač je na potezu (broj iznad čvora). Iz čvorova idu sve kombinacije koje igrač može učiniti u određenom trenutku. Kraj igre predstavljaju prazni čvorovi (uzeta je zadnja šibica) i dobiti koje igrači dobivaju u tom trenutku. Iz danog grafa se lako mogu saznati strategije dostupne igračima.

Igraču 1 su dostupne strategije:

- I_1 – uzmi jednu šibicu u stanju (=,=) i jednu u stanju (=,)
- I_2 – uzmi jednu šibicu u stanju (=,=) i dvije u stanju (=,)
- I_3 – uzmi dvije šibice u stanju (=)

Igraču 2 su dostupne strategije:

- II_1 – ako je stanje (=,) uzmi dvije šibice, a ako je stanje (=, -) uzmi jednu iz manje grupe
- II_2 – ako je stanje (=,) uzmi dvije šibice, a ako je stanje (=, -) uzmi jednu iz veće grupe
- II_3 – ako je stanje (=,) uzmi dvije šibice, a ako je stanje (=, -) uzmi dvije iz veće grupe

- II_4 – ako je stanje (=,) uzmi jednu šibicu, a ako je stanje (=, -) uzmi jednu iz manje grupe
- II_5 – ako je stanje (=,) uzmi jednu šibicu, a ako je stanje (=, -) uzmi jednu iz veće grupe
- II_5 – ako je stanje (=,) uzmi jednu šibicu, a ako je stanje (=, -) uzmi dvije iz veće grupe

Svaka od ovako definiranih strategija pokriva sve događaje u kojima je bitno što igrač odabere. Vidi se da u opisima strategija nisu pokriveni oni događaji koji ne utječu na konačan ishod.

Normalna forma igre služi za jednostavnije analiziranje same igre. Normalna forma bi bila reprezentacija igre u kojoj igrači odabiru strategiju u istom trenutku i kombinacija strategija određuje konačan ishod i dobit igre. Tako normalna forma igre specificira: (1) igrače u igri, (2) strategije dostupne igračima i (3) dobiti koje svaki igrač dobiva s obzirom na moguće kombinacije strategija (Gibbons R., 1992:3.). Prema (Gibbons R., 1992:3.) definicija normalne forme igre glasi:

Normalna forma kao reprezentacija igre s n igrača određuje prostor strategija $S_1, \dots, S_n$ i njihove funkcije korisnosti $u_1, \dots, u_n$. Takvu igru označavamo s $G = \{ S_1, \dots, S_n ; u_1, \dots, u_n \}$.

Prikladan prikaz dobiti u takvoj formi igre je matrica dobiti. U matrici dobiti je prikazan odnos između strategija igrača. Ako se primjeni takva matrica na prethodno opisanu igru izvlačenja šibica i strategije oba igrača dobivamo 3 x 6 matricu:

S	II_1	II_2	II_3	II_4	II_5	II_6
I_1	1	1	−1	1	1	−1
I_2	−1	1	−1	−1	1	−1
I_3	1	1	1	−1	−1	−1

U matrici S se vide dobiti za igrača 1 s obzirom na sve moguće kombinacije strategija. Kako je ovo jednostavna igra, jasno je da je dobit protivnika, odnosno igrača 2 suprotna onoj u polju matrice. Razlika između normalne i ekstenzivne forme igre je u tome što ekstenzivna forma sadrži dinamiku i opisuje igru u svakom njenom trenutku, dok je normalna forma statičan prikaz igre.

2.5. Analiziranje igre

Nakon postavljanja temeljnih termina i oblika igre u teoriji igara slijedi bit teorije igara, a to je analiziranje. Teorija igara ne nudi odgovor na pitanje na koji način igrati određenu igru da bi se pobjedilo nego daje analizu mogućih strategija i pokušava ponuditi *rješenje igre.* U ovom poglavlju će biti objašnjeno na koji način se analizira igra u teoriji igara i koji se pri tome mehanizmi i tehnike koriste.

2.5.1. Pareto optimalnost

Postavlja se pitanje, kako u nekoj igri možemo odrediti koji ishod igre je bolji od drugog. Kao vanjski promatrač, teško je odrediti da je dobit nekog igrača bitnija od dobiti drugoga. Stoga je potrebno pronaći način na koji se mogu uspoređivati ishodi neke igre. Recimo da se kao primjer uzme igra gdje je dobit nekog igrača njegova plaća u određenoj valuti. Problem je u tome što su valute u kojima igrači dobivaju plaću različite, a nemamo podatak o tečaju. Iako je nemoguće odrediti čija dobit je bolja, neki zaključci se mogu izvući. Na primjer, bolji ishod je 10 jedinica valute A i 3 jedinice valute B nego 9 jedinica valute A i 3 jedinice valute B. Iz tog zaključka se može povući sljedeća definicija (Shoham Y., Leyton – Brown K. 2009:61)

Definicija Pareto dominacija. *Strateški profil s Pareto dominira strateški profil s' ako za svaki* $i \in N$, $u_i(s) \geq u_i(s')$, *i postoji neki* $j \in N$ *za koji* $u_j(s) > u_j(s')$.

Pod strateškim profilom se podrazumjeva neka od mogućih kombinacija strategija koje su dostupne igračima. Drugim riječima, u Pareto dominiranom profilu, neki igrač može imati veću dobit, bez da se drugom igraču dobit smanji. Sada se može definirati i Pareto optimalnost.

Definicija Pareto optimalnost. *Strateški profil s je Pareto optimalan(efikasan) ukoliko ne postoji drugi strateški profil* $s' \in S$ *koji dominira s.*

Dakle Pareto optimalan profil je onaj u kojem ne možemo povećati dobit jednog igrača, bez da smanjimo dobit nekog drugog igrača.

2.5.2. Dominirana strategija

U analiziranju neke igre u teoriji igara postavljaju se određene pretpostavke. Jedna od pretpostavki je da igru igraju racionalni igrači. Pod pojmom racionalni se podrazumjeva da neće odigrati strategije koje su dominirane. Definicija dominirane strategije glasi (Gibbons R., 1992:5):

U normalnoj formi igre $G = \{ S_1, \dots , S_n ; u_1, \dots , u_n \}$, *neka su* s_i' *i* s_i'' *dostupne strategije igraču i. Strategija* s_i' *čvrsto je dominirana od strategije* s_i'' *ako za svaku kombinaciju strategija protivnika, dobit igrača i je manja ukoliko igra strategiju* s_i' *umjesto strategije* s_i''.

Ako se zada sljedeća matrica dobiti neke apstraktne igre (Gibbons R., 1992:6):

		Igrač 2		
		Lijevo	Sredina	Desno
Igrač 1	Gore	1,0	1,2	0,1
	Dolje	0,3	0,1	2,0

Iz ovog primjera se mogu eliminirati dominirane strategije. Igrač 1 ima dvije strategije: *gore* i *dolje*, a igrač 2 tri: *lijevo*, *sredina* i *desno*. Za igrača 2, *desno* je dominirana strategija od strategije *sredina* jer je dobit strategije *sredina* bez obzira na odabir strategije igrača 1 veća od strateije desno ($2 > 1$ i $1 > 0$).

		Igrač 2	
		Lijevo	Sredina
Igrač 1	Gore	1,0	1,2
	Dolje	0,3	0,1

Nakon eliminacije strategije *desno*, može se uočiti nova dominirana strategija, ovog puta za igrača 1, a to je strategija *dolje*. Igrači znaju međusobno dominirane strategije i igrač 1 je ostao samo na strategiji *gore*, dok igrač 2 zna da je najbolja strategija protiv strategije *gore* *sredina*. Igranje strategije *sredina* se u ovom apstraktnom slučaju može smatrati rješenjem.

Ovakvo iterirano eliminiranje strategija za uvjet ima da igrači međusobno znaju sve moguće strategije i dobiti svojih protivnika. U mnogo igara iteriranim eliminiranjem dominiranih strategija se ne može doći do rješenja.

2.5.3. *Maximin* kriterij

Općenita pretpostavka u traženju rješenja neke igre je da bi svi igrači trebali biti, u načelu pesimistični, naročito u igrama nulte sume. Budući da je glavni cilj igrača maksimizirati svoju dobit, u igrama nulte sume to znači, izravno i minimiziranje protivnikove dobiti.

Maximin kriterij podrazumjeva maksimiziranje minimalne dobiti. Igrač za svaku od svojih strategija računa minimalnu dobit i odabire najveću. Na taj način je maksimizirao svoju minimalnu dobit. Ta vrijednost se naziva minimalnom vrijedosti igre.

$$max_i min_j(u_{ij})$$

Protivnik također pokušava pronaći istu vrijednost za svoje strategije. Gledajući iz perspektive prvog igrača, protivnik pokušava minimizirati maksimalnu dobit prvog igrača, što je ekvivalentno njegovoj minimalnoj dobiti. Ta vrijednost se naziva gornjom vrijednosti igre.

$$min_j max_i(u_{ij})$$

2.5.4. Nashov ekvilibrij (ravnoteža)

Iako se iteriranim eliminiranjem dominiranih strategija može doći do rješenja igre, puno jači koncept rješenja je Nashov ekvilibrij. Ako se pretpostave strategije koje će igrači igrati u igri, razumni igrači će igrati najbolju strategiju koja im je dostupna protiv strategija protivnika. Ukoliko su dvije protivničke strategije ujedno i najbolji odgovori jedna na drugu onda se one nalaze u Nashovom ekvilibriju. Slijedi definicija prema (Gibbons R., 1992:8):

U normalnoj formi igre s n igrača $G = \{ S_1, \dots , S_n ; u_1, \dots , u_n\}$, *strategije* $\{s_1^*, \dots , s_n^*\}$ *su u Nashovom ekvilibriju ako, za svakog igrača i,* s_i^* *je (najmanje izjednačena) najbolji odgovor igrača i na strategije ostalih n-1 igrača,* $(s_1^*, \dots , s_{i-1}^*, s_{i+1}^*, \dots , s_n^*)$*:*

$$u_i(s_1^*, \dots , s_{i-1}^*, s_i^*, s_{i+1}^*, \dots , s_n^*) \geq u_i(s_1^*, \dots , s_{i-1}^*, s_i, s_{i+1}^*, \dots , s_n^*)$$

za svaku moguću strategiju s_i u S_i; tako da s_i^* rješava

$$\max u_i(s_1^*, \dots, s_{i-1}^*, s_i, s_{i+1}^*, \dots, s_n^*).$$

Ova formalna definicija u prijevodu znači da su strategije u Nashovom ekvilibriju ukoliko ne postoji druga kombinacija strategija koja daje bolju dobit za svaku strategiju u međusobnom sučeljavanju. Na idućem jednostavnom primjeru je objašnjen postupak pronalaska Nashovog ekvilibrija:

		Igrač 2		
		L	C	R
Igrač 1	T	0,4	4,0	5,3
	M	4,0	0,4	5,3
	B	3,5	3,5	6,6

Proces pronalaska Nashovog ekvilibrija u ovom primjeru se sastoji od pronalaska najboljeg odgovora (strategije) na svaku strategiju. Najbolji odgovor strategiji T je strategija L, jer igraču 2 daje najveću dobit (4). Ako gledamo recimo strategiju C igrača 2, najbolji odgovor na nju je strategija T. U donjoj tablici su podcrtani svi najbolji odgovori na strategije:

		Igrač 2		
		L	C	R
Igrač 1	T	0,<u>4</u>	<u>4</u>,0	5,3
	M	<u>4</u>,0	0,<u>4</u>	5,3
	B	3,5	3,5	<u>6,6</u>

Iz tablice se može vidjeti da su strategije (B, R) u Nashovom ekvilibriju jer su najbolji međusobni odgovori jedna na drugu. Nashov ekvilibrij predstavlja najbolji model rješenja neke igre, međutim nema svaka igra strategije koje su u Nashovom ekvilibriju.

2.5.5. Mješovita i čista strategija

U nekoj igri, određenu strategiju možemo smatrati mješovitom ili čistom. Čista strategija je ona u kojoj igrač može biti siguran u sljedeći potez protivnika ukoliko zna koju strategiju protivnik igra. Mješovitu strategiju se može opisati kao kombinaciju čistih strategija. Ukoliko je igrač siguran da protivnik igra određenu mješovitu strategiju, on ipak ne može biti siguran u idući potez protivnika. Primjer bi mogao biti scenarij u pokeru. Čista strategija bi bila ona u kojoj igrač uvijek blefira ili ona u kojoj uvijek odustaje ako ima slabiju kartu. Logičan odabir je povremeno blefiranje koje ulijeva nesigurnost u prepoznavanje strategije koju igrač igra. Takva strategija gdje igrač povremeno naizmjenično blefira i odustaje je mješovita strategija. Za primjer se može iskoristiti igra *Matching pennies*. Normalna forma igre je prikazana u sljedećoj tablici (Gibbons, R. 1992:29).

		Igrač 2	
		Glava	Pismo
Igrač 1	Glava	-1,1	1,-1
	Pismo	1,-1	-1,1

Dva igrača odlučuju hoće li izabrati pismo ili glavu na novčiću. Igrač 2 pobjeđuje ukoliko se oba novčića okrenu na istu stranu, a u suprotnom igrač 1. U ovakvoj igri, igračima je cilj međusobno nadmudrivanje. Postoji nesigurnost u idući potez svakog igrača i u ovom slučaju Nashovog ekvlibrija, kao što je definiran u prethodnom poglavlju nema. Formalno, mješovita strategija je distribucija vjerojatnosti nad prostorom čistih strategija S. Ako se vjerojatnost odabiranja pisma u prethodno opisanoj igri označi s q, onda je vjerojatnost odabiranja glave 1-q, gdje je $0 \leq q \leq 1$. Čista strategija uvijek odabiranja pisma je na taj način prikazana kao (0, 1), a strategija odabiranja glave (1, 0). Slijedi formalna definicija mješovite strategije (Gibbons, R. 1992:31):

U normalnoj formi igre $G = \{ S_1, \dots , S_n ; u_1, \dots , u_n\}$, *neka je* $Si = \{s_{i1}, \dots , s_{iK}\}$. *Mješovita strategija za igrača i je distribucija vjerojatnosti* $p_i = \{p_{i1}, \dots , p_{iK}\}$, *gdje je* $0 \leq p_{ik} \leq 1$ *za svaki* $k = 1,\dots,K$ *i* $p_{i1 + \dots + } p_{ik} = 1$.

Neka je *J* broj čistih strategija u S_1, a *K* u S_2. Pišemo $S_1=\{S_{11}, \dots , S_{1J}\}$ i $S_2=\{S_{21}, \dots, S_{2K}\}$. Ukoliko igrač 1 vjeruje da će igrač 2 igrati strategije $(s_{21}, \dots, s_{2K})$ s vjerojatnostima $(p_{21}, \dots, p_{2K})$, tada je očekivana dobit igrača 1 za strategiju s_{1j}:

$$\sum_{k=1}^{K} p_{2k} u_1(s_{1j,} s_{2k}),$$

dok je, očekivana dobit za igranje mješovite strategije $(p_{11}, \dots, p_{1J})$

$$v_1(p_1, p_2) = \sum_{j=1}^{J} \sum_{k=1}^{K} p_{1j} \cdot p_{2k} u_1(s_{1j,} s_{2k})$$

gdje je $p_{1j} \cdot p_{2k}$ vjerojatnost da igrač 1 odigra $s_{1j,}$ a igrač 2 s_{2k}.

Kako bi mješovita strategija $(p_{11}, \dots, p_{1J})$ bila najbolji odgovor za mještovitu strategiju p_2 igrača 2, mora vrijediti da je $p_{1j} > 0$ samo ako:

$$\sum_{k=1}^{K} p_{2k} u_1(s_{1j,} s_{2k}) \geq \sum_{k=1}^{K} p_{2k} u_1(s_{1j}', s_{2k})$$

za svaki s_{1j}' u S_1. To znači, da bi mješovita strategija bila najbolji odgovor strategiji p_2, mora pridodati pozitivnu vjerojatnost danoj čistoj strategiji, samo ako je čista strategija sama najbolji odgovor. Ukoliko igrač ima više čistih strategija koje su najbolji odgovori strategiji $p_{2,}$ tada bilo koja mješovita strategija koja svu vjerojatnost pridodaje jednoj od tih čistih strategija je najbolji odgovor.

Budući da je moguće odrediti najbolji odgovor na mješovitu strategiju tada se može i proširiti definicija Nashovog ekvilibrija:

U normalnoj formi igre s 2 igrača G = { S_1, S_n **;** u_1, u_2}, *mještovite strategije* (p_1^*, p_2^*) *su u Nashovoj ravnoteži ukolikojedna najbolji odgovor na drugu.*

2.5.6. Nashov ekvilibrij i linearno programiranje

Računanje Nashovog ekvilibrija za igre nulte sume s dva igrača je najlakše. U tom slučaju se problem Nashovog ekvilibrija može izraziti kao linearni program (Shoham Y., Leyton – Brown K. 2009:90,91). Neka je igra nulte sume s dva igrača definirana:

$$G = (\{1,2\}, A_1 \times A_2, (u_1, u_2))$$

Neka je U_i^* očekivana dobit za igrača *i* u ekvilibrijumu. Budući da je igra nulte sume vrijedi $U_1^* = -U_2^*$.

minimiziraj U_1^*

prema $\sum_{k \in A_2} u_1(a_1^j, a_2^k) \cdot s_2^k \leq U_1^* \qquad \forall j \in A_1$

$\sum_{k \in A_2} s_2^k = 1$

$s_2^k \geq 0 \qquad \forall k \in A_2$

U ovakvom linearnom programu, konstante su $u_1(\cdot)$, a varijable su s_2 i U_1^*. Prvo ograničenje programa govori da za svaku čistu strategiju *j* prvog igrača, s obzirom na mješovitu strategiju s_2 drugog igrača, očekivana dobit je najviše U_1^*. Rješavanjem ovog linearnog programa, igrač 2 pronalazi svoju mješovitu strategiju s_2^k koja minimizira očekivanu dobit igrača 1 (U_1^*). Druga dva ograničenja postavljaju mješovitu strategiju s_2^k u interval [0,1] kako bi izražavala vjerojatnost. Naravno, postoji i *dual* ovog linearnog programa, i on je problem maksimuma i on nam daje mješovitu strategiju igrača 1 koja maksimizira njegovu očekivanu dobit, s obzirom na svaku čistu strategiju igrača *k*. Takav program je definiran ovako:

maksimiziraj U_1^*

prema $\sum_{j \in A_1} u_1(a_1^j, a_2^k) \cdot s_1^j \geq U_1^* \qquad \forall k \in A_2$

$\sum_{k \in A_2} s_1^j = 1$

$s_1^j \geq 0 \qquad \forall j \in A_1$

Izračunate mješovite strategije s_1^j i s_2^k se nalaze u Nashovom ekvilibriju.

2.6. Dilema zatvorenika

Igre nulte sume su igre u teoriji igara u kojima je suma između dobiti jednog igrača i gubitka drugog jednaka nuli. Zanimljive su za analizu upravo zbog svoje jednostavnosti i zbog toga što se uzorci takvog ponašanja mogu pronaći u svim društvenim situacijama.

Simetrične igre su igre u kojima transponiranjem matrice dobiti jednog igrača dobijemo matricu dobiti drugog igrača. Dilema zatvorenika je simetrična igra nulte sume.

Popularna priča iza ove igre glasi otprilike ovako:

„*Dva zatvorenika su sudjelovali zajedno u kaznenom djelu. Nakon što su uhvaćeni, odvojeni su i dana im je šansa da izdaju svog partnera. Ukoliko ga izdaju mogu biti pušteni, a ukoliko ga ne izdaju moraju u zatvor. Trik je u tome što im sudbina ne ovisi o njima samima, nego i o potezu partnera. Ukoliko se i jedan i drugi izdaju robijaju 3 godine, ali ukoliko odluče šutiti, robijaju godinu dana, no ukoliko jedan zatvorenik izda partnera, a drugi odluči šutiti, partner koji je izdan robija 5 godina, a drugi biva pušten.*

Dva igrača igraju igru u kojoj odlučuju o medusobnoj suradnji ili sukobu. Ovisno o odabiru dobivaju prikladnu korist. Na slici 2 je prikazana matrica plaćanja. Kao što slika prikazuje, dva igrača imaju 2 moguća poteza: sudjelovati ili ući u sukob sa svojim protivnikom. Ukoliko oba igrača odluče surađivati, taj potez ih nagrađuje u ovom slučaju s 3 boda. Ako oba igrača uđu u sukob svaki dobiva po jedan bod. Suprotni potezi znače i maksimalan broj bodova, u ovom slučaju 5, za igrača koji se odlučio za sukob i 0 bodova za igrača koji se odlučio za suradnju. Ako zamislimo situaciju u kojoj na odluku igraca utjecaj nemaju nikakvi vanjski faktori poput činjenice da se igrači poznaju i sl., logičan potez bi bio sukob, jer sa sukob je igrač siguran da će dobiti minimalno 1 bod i neće postati "gubitnik" no ipak uvijek ostaje saznanje da su s međusobnom suradnjom mogli dobiti 3 boda svaki. Naravno suradnja nosi i opasnost od gubljenja svih bodova.

		Column Player	
		Cooperate	*Defect*
Row Player	Cooperate	$R=3, R=3$ Reward for mutual cooperation	$S=0, T=5$ Sucker's payoff, and temptation to defect
	Defect	$T=5, S=0$ Temptation to defect and sucker's payoff	$P=1, P=1$ Punishment for mutual defection

Slika 2: Matrica plaćanja u igri *Dilema zatvorenika*. (Izvor: Axelrod 1984)

Očigledno je da iz ovog proizlazi i Nashov ekvilibrij. Nashov ekvilibrij u ovoj igri je kombinacija strategija (D, D), odnosno kombinacija u kojoj se oba igrača odlučuju za sukob, jer u toj kombinaciji, nijednom igraču promjena strategije ne donosi profit.

Ako se promatra varijanta igre u kojoj isti igrači znaju da će ponovno igrati s istim protivnikom, na istu problematiku se gleda iz druge perspektive. Ukoliko igrač zna da će igrati s istim protivnikom nedefinirani broj puta, medusobna suradnja postaje logičan odabir. Ukoliko se igra ponavlja nedefinirani broj puta, sukob opet postaje logičan odabir, jer suradnjom u samo jednoj od iteracija se riskira gubljenje bodova ukoliko protivnik odluči u svakoj iteraciji ući u konflikt. Ukoliko isti koncept igre prebacimo u iteriranu igru, stvari postaju zanimljivije

2.7. Axelrodov turnir

Jasno je da je najisplativija strategija igranje sukoba u dilemi zatvorenika no što ako igramo protiv niza protivnika i oni također igraju jedni protiv drugih međusobno, a konačni pobjednik je onaj s najvećim brojem bodova u svim igrama. Axelrod je 1984. organizirao turnir u kojem su računalni programi igrali iteriranu verziju dileme zatvorenika jedni protiv drugih. Svatko je mogao predložiti strategiju za koju je mislio da je najbolja. Nakon sučeljavanja svih strategija svake sa svakom u 200 iteracija igre pobjedu je odnijela strategija

TIT FOR TAT Anatola Rapoporta. Navedena strategija surađuje u prvom koraku, a kasnije ponavlja prethodni protivnikov potez. Činjenica da iskorištava prednosti igre s igračem koji želi surađivati, ali i odgovara na sukob čine ju strategijom koja odnosi najviše bodova kod najraznolikijeg broja protivnika. (Axelrod, 1984) navodi 3 glavna faktora koji utječu na uspješnost TIT FOR TAT strategije:

- Izbjegavanje nepotrebnog sukoba surađujući sve dok protivnik surađuje
- Odgovor na protivnikov neizazvani sukob
- Opraštanje protivniku zbog ulaska u sukob

Ideja iza *„tit for tat"* strategije je skupiti što više bodova suradnjom, ali ne dopustiti gubljenje previše bodova na iskazivanje dobronamjernosti. Također je vrlo bitna treća stavka, koja govori da strategija pruža mogućnost oprosta protivniku zbog ulaska u sukob i samim time nastavak korištenja suradnje za dobivanje što više bodova.

2.8. Prostorne igre

Posebnu zanimljivost izazivaju prostorne igre u teoriji igara. Kažemo da je prostorna igra s n igrača igra koja u normalnoj formi dobit igrača prikazuje kao aritmetičku sredinu igara sa svojim susjedima, gdje su susjedi određeni prostornom strukturom igre.
Formalni opis prostorne igre pomoću teorije grafova (Baron et al. 2002):
Osnovna komponenta za izgradnju prostorne igre je igra s 2 igrača G, nazvana temeljnom igrom, sa skupom poteza X i funkcijom dobiti $\pi : X \times X \to R^2$ koja pridružuje svakom strateškom profilu $x = (x_1, x_2)$ dvojku $\pi(x) = (\pi_1(x), \pi_2(x))$ dobiti. G se označava prostornom igrom

$$G = (I, \Gamma_n, (S_i)_{i \in I}, (u_i)_{i \in I})$$

na sljedeći način. Skup igrača je $I = \{1,..., n\}$. Svaki igrač $i \in I$ ima skup strategija $S_i = X$. Nadalje elementi skupa I čine vrhove težinskog grafa Γ_n reda n tako da se igrač i nalazi na vrhu i. E označava skup bridova grafa Γ_n. Dva vrha ili igrača i i j su susjedi ako je $\{i, j\} \in E$. $N(i)$ označava skup susjeda igrača i. Γ_n je neusmjeren graf, što znači da $j \in N(i)$ ako i samo ako je $i \in N(j)$. Također vrijedi da je N (i) $\neq \emptyset$, za sve i, što znači da ni jedan rub nije izoliran.

Svaki $\{i, j\} \in E$ ima težinu w_{ij} koja mjeri njegovu relativnu važnost. Dobiti u prostornoj igri *G* su izražene

$$u_i\,(s) = \sum_{j\,\in N\,(i)} w_{ij}\pi_i(s_i, s_j)$$

Za svaki $i \in I, s = \left(s_j\right)_{j\,\in I} \in \Pi_{j\in I}\, S_j \equiv S$, što znači da igrač prikuplja zbrojenu ponderiranu dobit igranja sa svakim od svojih susjeda.

Neka je $s_i^* \in S$ strateški profil prostorne igre *G*. Profil s^* je Nashov ekvilibrij za svaki $i \in I$, $s_i \in S_i$,

$$u_i(s^*)\ \geq\ u_i\,(s_i, s_i^*)$$

gdje je $s_i^* = (s_j^*)_{j\neq i}$.

Prostorna igra je dakle igra s više igrača raspoređenih u matricu. Svaki igrač ima svoje koordinate i zauzima jednu ćeliju matrice. Ono što karakterizira prostorne igre je uređenje u kojem su igrači u susjedskim vezama jedni s drugima. Igra se odvija u iteracijama u kojima igrač igra sa svojim susjedima, a dobit se izračunava ponderiranom dobiti pojedinačnih igara sa svakim od susjeda.

Prostorne igre pripadaju grupi evolucijskih igara. Evolucijske igre se najčešće primjenjuju u biologiji i pokušavaju oponašati situacije u kojima određena grupa (populacija) „opstaje“ u okolini zahvaljujući svojim genima. Tako da dobit individualnih jedinki ne može biti izmjerena u izolaciji, nego treba biti mjerena unutar populacije u kojoj jedinka živi (Easley et al. 2010). Evolucijskim igrama se pokušava prikazati dinamika razvoja strategija igrača kroz generacije. U prostornim igrama je ta populacija dodatno definirana „susjedskim“ vezama između igrača i koordinatama igrača. Budući da u prostornim igrama igrači komuniciraju sa svojim susjedima, opstanak u takvom okruženju ovisi o „dobrosusjedskim“ odnosima. Naravno ishod ovisi o vrsti igre. Dakle u prostornim igrama, na opstanak neke strategije kroz generacije, utječe njeno „susjedstvo“. Igrač preuzima najbolju strategiju svog susjeda, ukoliko njegova nije najbolja. U praktičnom radu će se pokušati pokazati kako određene strategije u klasičnoj igri teorije igara imaju drugačiji značaj u prostornoj igri.

3. Višeagentni sustavi

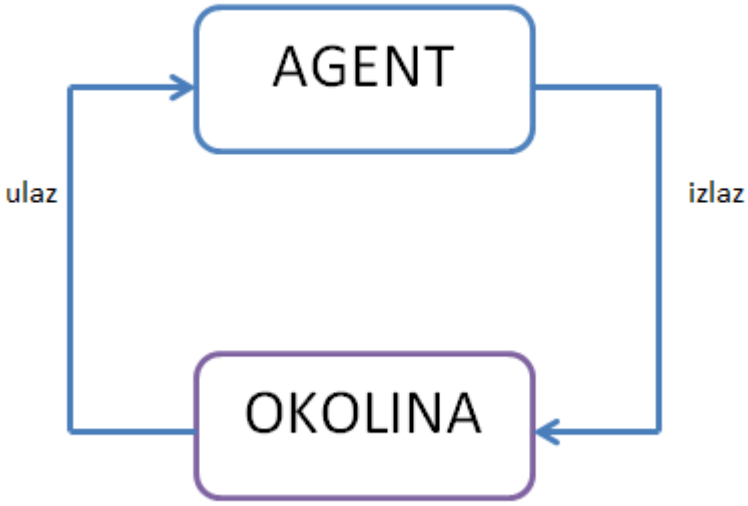
Višeagentni sustavi su sustavi koji se sastoje od više računalnih elemenata (agenata) koji se nalaze u interakciji. Agenti su računalni sustavi koji imaju sposobnost samostalnog odlučivanja svojih akcija kako bi ostvarili definirane ciljeve. Sposobni su međusobno djelovati s drugim agentima poput surađivanja, koordinacije, pregovaranja i sl. (Wooldridge M., 2009.)

Trendovi u svijetu računalstva su se razvijali tokom povijesti. Računala postaju sve jeftinija i snažnija. Računalni programi sve inteligentniji, a informacijski sustavi sve više distribuirani i umreženi. Sljedeći korak u razvoju je apstrakcija računalnih sustava bliža ljudskom razumijevanju svijeta. Svi nabrojani trendovi su utjecali na nastanak višeagentnih sustava. Agent mora biti dovoljno inteligentan da sam donosi odluke, s obzirom na svoje ciljeve. Višeagentni sustav mora biti distribuiran i umrežen; sustav u kojem agenti mogu međusobno komunicirati iako su fizički udaljeni i najvažniji cilj. Agent mora biti sposoban predstavljati korisnika u određenim socijalnim situacijama, kao što je na primjer kupnja preko interneta. Zbog toga što je ideja višeagentnih sustava oponašanje socijalnog sustava, područje višeagentnih sustava je interdisciplinarno. Pod utjecajem je ekonomije, filozofije, teorije igara, logike, ekologije, sociologije i drugih disciplina. Mnogi vide kritiku u toj interdisciplinarnosti poistovjećujući područje višeagentnih sustava s umjetnom inteligencijom, teorijom igara, distibuiranim sustavima, sociologijom, no činjenica je da je područje višeagentnih sustava se, iz zajedničkih karakteristika mnogih disciplina, uspjelo profilirati kao samostalno područje vrijedno proučavanja.

3.1. Agent

Definicija agenta, preuzeta iz Wooldridge (2009) glasi:

Agent je računalni sustav, smješten u određenoj okolini u kojoj je sposoban samostalno djelovati, kako bi ostvario temeljne ciljeve.



Slika 3. Interakcija agenta sa svojom okolinom. Agent prima ulaz iz okoline i producira izlaz koji utječe na okolinu.

Na slici je prikazan apstraktni opis agenta. Agent preuzima uzima iz okoline i producira izlaz koji utječe na okolinu. Vrlo je bitno naglasiti interakciju agenta sa svojom okolinom. Agent ima skup mogućih akcija koje može poduzeti. Svojim akcijama on utječe na okolinu koja je stohastička. Što znači da dvije iste akcije ne moraju dati jednake rezultate. Prema takvom apstraktnom modelu, bilo kakav sustav može biti agent, na primjer termostat. On ispituje temperaturu okoline i po potrebi uključuje ili isključuje grijanje. Takav agent nije inteligentan. Da bi agent bio inteligentan, prema (Wooldridge, M. 2009) mora biti:

- Reaktivan. Inteligentni agenti su u mogućnosti spoznati svoju okolinu i brzo reagirati na promjene okoline kako bi zadovoljili svoje ciljeve.
- Proaktivan. Inteligentni agenti su u mogućnosti manifestirati ciljno-orijentirano ponašanje pokrečući inicijativu kako bi zadovoljili svoje ciljeve.
- Društven. Inteligentni agenti su u mogućnosti komunicirati s drugim agentima (po mogućnosti i ljudima) kako bi zadovoljili svoje ciljeve.

3.1.1. Formalni opis agenta

Nakon apstraktnog opisa agenta, valjalo bi ga i formalizirati. Može se početi od definicije okružja (okoline). Pretpostavimo da je okružje konačan skup *O* diskretnih, instantnih stanja:

$$O = \{o_1, o_2, \dots\}$$

Pretpostavimo da svaki agent ima konačan repertoar akcija koje transformiraju stanje okružja:

$$A_K = \{a_1, a_2, \dots\}$$

Prolaz *p* agenta kroz okružje je naizmjenični niz stanja okružja i akcija agenta:

$$p: o_0 \overset{a_0}{\rightarrow} o_1 \overset{a_1}{\rightarrow} o_2 \overset{a_2}{\rightarrow} \dots \overset{a_{n-1}}{\longrightarrow} o_n$$

Zatim neka je *P* skup svih konačnih prolaza (nad *O* i A_K) i neka je P^{A_K} podskup svih onih prolaza koji završavaju akcijom, a P^O podskup svih onih prolaza koji završavaju stanjem okružja. Funkcija promjene stanja je funkcija koja će za zadano stanje okružja i zadanu akciju odrediti buduće stanje okružja:

$$\tau : P^{A_K} \rightarrow \sigma(\mathrm{E})$$

Okružja su ovisna o povijesti i stohastička, stoga ukoliko je $\tau(p) = \emptyset$ tada ne postoji niti jedno moguće stanje koje slijedi p i kažemo da je sustav završio prolaz p. Formalno je okružje *Okr* uređena trojka $Okr = (O, o_0, \tau)$ u kojoj je *O* skup mogućih stanja okružja, o_0 početno stanje i τ funkcija transformacije stanja.

Nakon formalne definicije okružja, slijedi formalna definicija agenta.

Agent je funkcija koja prolazima pridružuje akcije:

$$Ag : P^0 \rightarrow A_K$$

Na temelju povijesti sustava kroz koji je prošao, agent donosi odluku o sljedećoj akciji. Par koji se sastoji od jednog agenta i jednog okružja se naziva sustavom. Niz prolaza nekog sustava je dvojka *P(Ag, Okr)*.

3.1.2. Vrste agenata

3.1.2.1. Čisto reaktivni agenti i agenti sa stanjem

Čisto reaktivni agenti su agenti koji nemaju stanje. Oni na temelju podražaja (trenutnog stanja okruženja) izvode neku akciju. Primjer takvog agenta je naveden i u prošlom poglavlju (termostat). Takvog agenta se može opisati funkcijama *vidi* i *akcija.*

$$vidi : O \rightarrow Do$$

O predstavlja okolinu i *vidi* okolini pridužuje doživljaj (*Do*).

$$akcija : Do \rightarrow A_K$$

Funkcija *akcija* doživljajima pridužuje akcije (A_K).

Agenti sa stanjem održavaju svoje unutarno stanje koje utječe na odabir akcije. Pod stanjem se smatra neka vrsta podatkovne strukture koja se koristi za bilježenje informacija o stanju okružja i povijesti. Neka je U skup svih unutarnjih stanja agenta. Takav agent ima dodatnu funkciju *sljedeće:*

$$sljedeće : U \times Do \rightarrow U$$

Ona paru unutarnih stanja i doživljaja pridružuje unutarnja stanja, tj. transformira unutarnje stanja agenta na temelju doživljaja. Funkcija *akcija* takvog agenta je

$$akcija : U \rightarrow A_K$$

3.1.2.2. BDI model agenta

Ovaj model agenta oponaša proces odlučivanja kod ljudi. Nazvan je kao skraćenica „*Belief – Desire – Intention"* što u prijevodu znači „*Uvjerenje – Želje – Planovi*. Uvjerenja predstavljaju agentovo znanje o svijetu, želje su poželjna buduća stanja u kojima se agenti može naći, a planovi su načini postizanja tih ciljeva.

Slijedi formalna definicija BDI agenta:

BDI agent je definiran kao uređena trojka (Z, C, P) pri čemu je:

- Z – skup znanja agenta (niz formula u LPR)
- C – skup ciljeva koje agent (niz formula LPR)
- P – skup planova kako postići ciljeve (niz preslikavanja/akcija oblika $I: P \rightarrow Q$ u kojem je *P* preduvjet, a *Q* očekivani rezultat preslikavanja

Preduvjet *P* je formula LPR koja obzirom na *Z* mora vrijediti. Rezultat *Q* je formula koja se dodaje u *Z* ako je preslikavanje uspješno.

Spajanjem koncepata agenata sa stanjem i BDI agenata dobivamo BDI agenta u okružju. Funkcija *sljedeće* postaje ovisna o trenutnim planovima. Funkcija *akcija* postaje sljedeće preslikavanje trenutnog plana. Stanje agenta uključuje njegovo trenutno znanje, a dugoročno planiranje agenta postaje mogućnost.

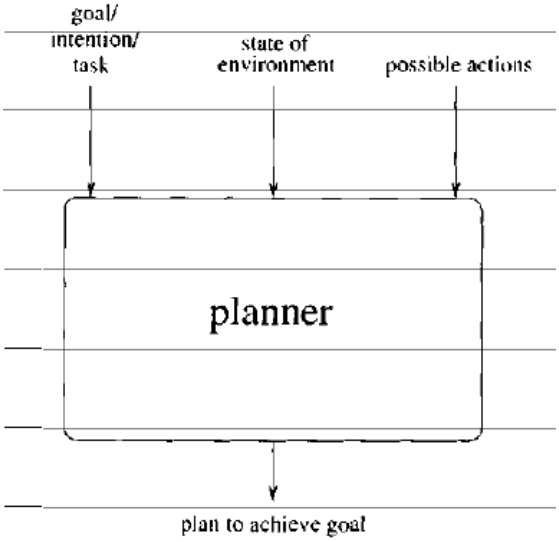
3.1.2.3. Agenti s praktičnim rezoniranjem.

U prošlom poglavlju je bio predstavljen model agenta koji odlučuje o idućim akcijama korištenjem logičkog zaključivanja. Praktično rezoniranje se ne može osloniti na samo korištenje logike. Ono je okrenuto prema akcijama. Bitno ga je razlikovati od teoretskog rezoniranja. Teoretsko rezoniranje ovisi o vlastitim uvjerenjima o svijetu. Na primjer, logički je zaključiti da, ukoliko je Sokrat čovjek, i ukoliko su ljudi smrtni, da je Sokrat smrtan. Takvo rezoniranje je uvjetovano vlastitim uvjerenjima o svijetu. Rezoniranje o tome, treba li putovati autobusom ili vlakom je praktično rezoniranje jer je usmjereno prema akciji (Wooldridge, 2009:66).

U praktičnom rezoniranju krajnji cilj je postići neku nakanu. Može se reći da nakane imaju ova svojstva (Wooldridge, 2009:66):

- Nakane induciraju rezoniranje kako bi se postigao krajnji cilj. Ukoliko postoji nakana, tada agent pokušava ostvariti nakanu, što uključuje i odlučivanje kako ju postići. Ukoliko neki pokušaj ne uspije, agent pokušava drugim pristupom.
- Nakane opstaju. Agent će pokušati ostvariti nakanu dokle god ima dobar razlog smatrati da je ona ostvariva.
- Nakane ograničuju buduće promišljanje. Agent se neće baviti nakanama koje su nekonzistentne s trenutnim nakanama
- Nakane utječu na uvjerenja na čemu je zasnovano buduće praktično rezoniranje. Agenti mogu ostvarivati planove na temelju vjerovanja da će ostvariti trenutne nakane.

Omogućavanje planiranja znači i samostalnije ponašanje agenta. Agentu treba dati reprezentaciju nakana koje treba postići, akcija koje može obavljati i okružja u kojem se nalazi. Na temelju tih podataka on treba izgenerirati plan kako postići te ciljeve.



Slika 4 Ilustracija planiranja (preuzeto iz Wooldridge 2009:71)

Algoritam planiranja, kao izlaz, generira plan. Ukoliko algoritam ispravno generira plan, agent ga izvodi iz stanja koje je specificirano u stanju okružja. Nakon izvođenja plana provodi se nakana. Postavlja se pitanje, kako bi se implementirao takav agent. Slijedi pseudokod kontrolne petlje agenta koji planira.

```
Z ← Z0;
P ← P0;
while istina do
    Prihvati sljedeći doživljaj d;
    Z ← ažuriraj(Z, d);
    C ← opcije(Z, P);
    P ← filtriraj(Z, C, P);
    π ← planiraj(Z, P);
    while ¬(prazan(π) ∨ postignut(P, Z) ∨ nemoguć(P, Z)) do
        a ← glava(π);
        obavi(a);
        π ← tijelo(π);
        Prihvati sljedeći doživljaj d;
        Z ← ažuriraj(Z, d);
        if razmotri(P, Z) then
            C ← opcije(Z, P);
            P ← filtriraj(Z, C, P);
        end
        if ¬smislen(π, P, Z) then
            π ← planiraj(Z, P)
        end
    end
end
```

Slika 5 Pseudokod agenta koji planira (Preuzeto i prilagođeno iz Wooldridge 2009:76)

Agent u gornjem pseudokodu u kontrolnoj petlji ažurira trenutno saznanje o svijetu (*Z*). Zatim generira opcije na temelju trenutnog saznanja i obvezuje se za njih (*C*). Plan generira na temelju filtriranih opcija koje postaju planovi (*P*) i saznanja o svijetu (*Z*). S π je definiran postupak izvođenja plana. Unutarnja petlja omogućuje rezolutno obvezivanje agenta, tj. omogućava agentu da pokušava obaviti plan u slučaju da nije već obavljen, nije postignut ili agent smatra da ga je još uvijek moguće obaviti. U unutarnjoj petlji je implementirano i ponovno planiranje ukoliko se plan ne ostvari (funkcija *smislen*). Prije planiranja agent ažurira svoje znanje o svijetu i ponovno razmatra opcije s *filtriraj*. Može se vidjeti da je razmatranje opcija uvjetovano funkcijom *razmotri* kojom agent odlučuje je li vrijeme za ponovno razmatranje, kako bi se spriječilo razmatranje opcija u svakoj iteraciji petlje i smanjio trošak vremena.

3.2. Teorija igara u terminologiji višeagentnog sustava

Višeagentni sustav se može okarakterizirati kao sustav koji se sastoji od određenog broja agenata koji su međusobno povezani kroz komunikaciju. Agenti su u mogućnosti utjecati na okružje. Mogu imati različite ili preklapajuće „domene utjecaja“, u smislu da svaki agent utječe na neki dio okružja.

Budući da teorija igara modelira mnoge scenarije u kojima se mogu naći sudionici, može se reći da je u uskoj povezanosti s višeagentnim sustavima. Strukturu igre možemo, iz društvenog pogleda, shvatiti kao koordinacijski mehanizam (Johansson 1999:3) . Agenti u tom smislu postaju igrači, a njihovo okružje pravila i sredstva igre. Pretpostavimo da neki sustav čine dva agenta *i* i *j*. Svaki od agenata je sebičan i ima svoje viđenje kako bi svijet trebao izgledati. Neka je $\Omega = \{\omega_1, \omega_2, \dots\}$ skup stanja koja agenti preferiraju. Takav skup se može zamisliti kao skup mogućih ishoda igre koju igraju agenti. Sklonost agenata (igrača) se može iskazati kao funkcija dobiti *u*:

$$u_i: \Omega \to \mathbb{R}$$
$$u_j: \Omega \to \mathbb{R}$$

Stoga se može reći da agent više preferira stanje ω' od stanja ω ukoliko je funkcija korisnosti stanja ω' veća od korisnosti stanja ω ili:

$$\omega \preccurlyeq_i \omega' \text{ znači } u_i(\omega) \leq u_i(\omega')$$
$$\omega \succ_i \omega' \text{ znači } u_i(\omega) > u_i(\omega')$$

Višeagentni sustav je moguće izmodelirati kao igru putem sučeljavanja. Potrebno je uvesti model okružja u kojem agenti djeluju. Ako pretpostavimo da višeagentni sustav izgleda poput jednostavne igre s dva igrača, tada agenti istovremeno biraju akcije koje će poduzeti u okružju, a ishod njihovih akcija bit će neko stanje iz Ω. Pretpostavimo da agenti na raspolaganju imaju dvije moguće akcije „C“ i „D“ (suradnja i konflikt). Tada se njihovo sučeljavanje može prikazati kao ponašanje okružja:

$$\tau: A_C \times A_C \rightarrow \Omega$$

Funkcija τ je transformator stanja okružja koja sučeljenim akcijama agenata pridružuje stanje okružja.

Ako se pretpostavi da imamo dva podskupa Ω koji se mogu označiti kao Ω_1 i Ω_2. Kaže se da Ω_1 dominira Ω_2 za agenta *i*, ukoliko svaki ishod u Ω_1 je poželjniji od svakog ishoda u Ω_2. Formalno, skup ishoda Ω_1 strogo dominira Ω_2 ukoliko vrijedi sljedeće (Wooldridge 2009:112)

$$\forall\, \omega_1 \in \Omega_1, \forall\, \omega_2 \in \Omega_2,\ \omega_1 \succ_i \omega_2$$

Racionalni agent nikad neće igrati strategiju koja je dominirana.

Razni su scenariji u kojima se agenti mogu naći u ovakvoj interakciji. Na primjer, agent u postizanju svog cilja komunicira i dijeli informacije s drugim agentima. U određenom trenutku se može naći u situaciji u kojoj dijeljenjem neke informacije s drugim agentom mogu profitirati i jedan i drugi agent, ali možda ne toliko kao da informaciju zadrži za sebe.

Interakciju agenata možemo prikazati pomoću matrice:

		Agent 2	
		Konflikt	Suradnja
Agent 1	Konflikt	-2,-2	0,-3
	Suradnja	-3,0	-1,-1

Agentu su na raspolaganju dva poteza (strategije), suradnja i konflikt s drugim agentom. Može se primjetiti da je gore prikazana interakcija između agenata normalna forma igre.

4. Praktični rad

Za praktični rad je odabrana igra iterirana dilema zatvorenika. Igra je opisana u poglavlju o teoriji igara, a primjenit će se na dva različita načina. Na klasičan način i kao prostorna igra. U ovom poglavlju će biti opisana implementacija igre i rezultati dobiveni simulacijama, kao i usporedba između dvije varijante iste igre. Na samom početku će ukratko biti opisana platforma za izradu ove aplikacije, a to je SPADE. Za opis SPADE-a je korištena dokumentacija dostupna na webu, navedena u literaturi.

4.1. SPADE platforma

4.1.1. FIPA standard i XMPP protokol

Platforma za razvoj višeagentnog sustava korištena u ovom radu je SPADE (eng. **S**mart **P**ython multi-**A**gent **D**evelopment **E**nvironment). Kao što je otkriveno u samom imenu, platforma je napisana u programskom jeziku Python. U skladu je s FIPA (eng. *Foundation for Intelligent Physical Agents*) standardom. FIPA standard postavlja temelje za izgradnju platforme za razvoj višeagentnih sustava. Temeljne značajke FIPA standarda su sljedeće:

- Definira okružje u kojem agenti „žive“ i obavljaju svoje aktivnosti
- Kanal za komunikaciju – mehanizam pomoću kojeg agenti mogu međusobno komunicirati
- Sustav za upravljanje agentima – mehanizam registriranja i pristupanja agentima
- Sustav za registriranje usluga (svojevrsne „žute stranice“ za agente)
- FIPA komunikacijski jezik – zajednički jezik komunikacije za sve agente

Komunikacija u SPADE-u je implementirana na temelju Jabber/XMPP protokola, također korištenom od strane raznih alata za instant komunikaciju, poput Google Talka, Facebook chata i dr. Protokol je razvijen od strane Jabber *open-source* zajednice 1999. godine kako bi postavio standard za razvoj alata za *instant* komunikaciju (IM). Cilj je bio razviti XML-om inspiriran protokol za asinkronizirano komuniciranje, sadržavanje informacije o dostupnosti kontakata i posjedovanje liste kontakata. Glavne značajke XMPP protokola su: otvorenost koda, asinkronost, decentraliziranost, sigurnost i proširivost.

4.1.2. SPADE biblioteka

Preporučuje se za izgradnju višeagentnih sustava za SPADE platformu koristiti SPADE biblioteka, iako to nije obavezno. Moguće je agente razvijati pomoću drugih biblioteka i programskih jezika. Jedini zahtjev je da agenti mogu komunicirati XMPP protokolom. Biblioteka je skup klasa, funkcija i alata za kreiranje SPADE agenata i modul je za programski jezik Python.

Model agenta u SPADE-u se sastoji od mehanizma komuniciranja prema platformi, mehanizma za slanje poruka i skupa različitih ponašanja. Svaki agent je određen Jabber identifikatorom (**JID**). JID se sastoji od korisničkog imena, specijalnog znaka „@“ i serverske domene (npr. myagent@localhost). Agenti se registriraju sa željenim korisničkim imenom i lozinkom na SPADE platformu. SPADE uključuje Jabber server komponentu koja omogućuje agentima registraciju kao Jabber klijentima. Komunikacija između agenata se odvija putem slanja poruka putem FIPA ACL jezika (eng. *Agent Communication Language*). Svaki agent posjeduje identifikator (AID). AID se sastoji od jedinstvenog imena i skupa adresa koje se mogu koristiti za komunikaciju s agentom. U većini slučajeva agent koristi svoj JID kao identifikator i jednu adresu.

```
Name="agent@myhost.myprovider.com"
Addresses=["xmpp://agent@myhost.myprovider.com"]
```

Za kreiranje i slanje poruka se koristi klasa `spade.ACLMessage.ACLMessage`. Poruka se šalje funkcijom `send` klase `spade.Agent.Agent`. Poruci se mogu postaviti sljedeći parametri

Tablica 1 Parametri ACL poruke

Parametar	**Kategorija**
`performative`	Vrsta komunikacije
`sender`	Sudionik u komunikaciji
`receiver`	Sudionik u komunikaciji
`reply-to`	Sudionik u komunikaciji
`content`	Sadržaj poruke
`language`	Opis sadržaja
`encoding`	Opis sadržaja
`ontology`	Opis sadržaja
`protocol`	Kontrola konverzacije
`conversation-id`	Kontrola konverzacije

`reply-with`	Kontrola konverzacije
`in-reply-to`	Kontrola konverzacije
`reply-by`	Kontrola konverzacije

Jedan od najčešće korištenih parametara je `ontology`. Njime se može odrediti svrha poruke putem koje će primatelj moći odlučiti koju akciju je potrebno poduzeti.

Primanje poruka se obavlja putem metode `_receive` koja je specifična za pojedinu implementaciju ponašanja agenta. Akcije agenta se implementiraju putem takozvanih ponašanja. U SPADE-u je implementirano nekoliko tipova ponašanja predviđenih za korištenje u različitim situacijama:

- Ciklično i periodično ponašanje – za implementiranje ponavljajućih zadataka (`PeriodicBehaviour`)
- Jednokratno i *Time-out* ponašanje – za implementiranje rijetkih zadataka (`OneShotBehaviour, TimeOutBehaviour`)
- Konačni automat – za implementiranje složenijih ponašanja (`FSMBehaviour`)
- Ponašanje uzrokovano događajem – za implementiranje zadataka uzrokovanih percepcijom agenta (`EventBehaviour`)

Sva navedena ponašanja su implementirana u `spade.Behaviour` klasi. Određenom ponašanju se može dodijeliti obrazac poruka (`MessageTemplate`) koje će aktivirati ponašanje.

```
class CookingBehaviour(EventBehaviour):
    ...

t = ACLTemplate()
t.setOntology('cooking')
m = MessageTemplate(t)
self.addBehaviour(self.CookingBehaviour, m)
```

U gornjem primjeru je prikazano ponašanje `CookingBehaviour.` Agentu i spomenutom ponašanju se može pridružiti `MessageTemplate` koji u sebi ima definiranu ontologiju *cooking*. Time se definira da će „okidač“ za ponašanje `CookingBehaviour` biti pristizanje poruke s ontologijom *cooking*.

4.2. Klasična iterirana dilema zatvorenika

4.2.1. Opis implementacije

U donjoj tablici se može vidjeti matrica plaćanja korištena u ovoj implementaciji igre.

		Igrač 2	
		Suradnja	Sukob
Igrač 1	Suradnja	1,1	5,0
	Sukob	0,5	3,3

Kako bi se ostalo u duhu naziva igre, boljim ishodom je odabran manji broj, odnosno može se reći da broj u gornjoj tablici predstavlja godine zatvora. Dakle pobjednik je igrač s najmanjim brojem godina. Igra je organizirana popout Axelrodovog turnira, na način da svaki igrač igra sa svakim točno određeni broj puta (određeno parametrom **broj iteracija**). Krajnji rezultat je zbroj svih bodova.

Implementacija je zamišljena kao višeagentni sustav koji se sastoji od agenta organizatora koji organizira turnir i koji je posrednik u komunikaciji između agenata igrača tijekom igre. Sva komunikacija između igrača se odvija preko organizatora. Za komunikaciju se koristi FIPA ACL protokol koji je opisan u prethodnom poglavlju, a ponašanja agenata su određena dolascima ACL poruka određene ontologije.

Igra je implementirana putem sljedećih agenata:

Organizator

Agent *Organizator* je zaslužan za uspostavljanje igre i kao posrednik u komunikaciji između igrača. Implementirana su mu sljedeća ponašanja:

- Prijava: Pokreće se dolaskom poruke s ontologijom *Prijava*. Prihvaća prijave igrača i dodjeljuje im protivnika
- Igranje: Pokreće se dolaskom poruke s ontologijom *Igranje*. Prihvaća poteze igrača i prosljeđuje ih njihovom protivniku.

Agent organizator je zadužen za pokretanje igre. Prvi se pokreće i zadužen je za početak igre. Organizator prijavljuje igru kao uslugu u SPADE-u, kako bi igrači mogli pretražiti uslugu i dobiti adresu organizatora kojem šalju poruku.

Igrač

U agentu Igrač su implementirana ponašanja i strategije koje oponašaju poteze koje odigrava igrač u igri dileme zatvorenika. Sadrži sljedeća ponašanja:

- Prijava: Izvršava se pri samom pokretanju agenta. Pretražuje usluge s nazivom *AxelrodTurnir* i tom akcijom dobiva adresu organizatora, nakon čega se može prijaviti za igru
- Igranje: Pokreće se dolaskom ACL poruke s ontologijom *Protivnik.* Poruka sadrži adresu protivnika kojem treba poslati idući potez. Igrač na temelju imena protivnika odigrava potez i vraća poruku Organizatoru.
- Rezultat: Pokreće se dolaskom poruke s ontologijom *Rezultat*. Poruka sadrži rezultat igre s protivnikom s kojim je igrač trenutno završio igru.

Agentu igraču su na raspolaganju sljedeće strategije:

- ALLD: *Always defect*: strategija u kojoj igrač uvijek izabire sukob
- ALLC: *Always cooperate:* strategija u kojoj igrač uvijek izabire suradnju
- RAND: Igrač slučajnim odabirom izabire između dvije ponuđene strategije
- TFT: *Tit for tat*: Igrač u prvom potezu odabire suradnju, a zatim ponavlja protivnikov prethodni potez
- TESTER: U prvom potezu igrač odabire sukob. Ako je protivnik surađivao, igra TFT, u suprotnom nasumično surađuje i ulazi u sukob
- JOSS: Identična TFT strategiji, s razlikom da periodično ulazi u sukob (svaku 5. igru)
- PAVLOV: Počinje sa suradnjom, ukoliko je nagrađen za suradnju (protivnik je također surađivao), ili za sukob (protivnik je surađivao), ponavlja prethodni potez, dok u suprotnom odabire suprotni.

4.2.2. Opis korištenja aplikacije

Cijeli program se izvodi u terminalu. Na početku se pokreće agent organizator s nazivom, brojem agenata igrača i brojem iteracija kao argumentima (prikazano na slici 6)

```
ivan@ivan-virtual-machine ~ $ python organizator.py organizator 4 20
```

Slika 6 Pokretanje agenta *Organizator*

Organizator zatim prijavljuje turnir kao SPADE uslugu te čeka za prijavljivanje igrača kako bi mogao započeti turnir. Nakon organizatora se pokreće igrač kao što je prikazano na slici 7.

```
ivan@ivan-virtual-machine ~ $ python igrac.py igrac1 TFT
```

Slika 7 Pokretanje agenta *Igrač*

Igrač prima također naziv agenta koji će mu postati dio adrese i naziv strategije. Nakon pokretanja igrač pretražuje uslugu s nazivom *AxelrodTurnir*.

Na slici br. je prikazan dio ispisa igre i sam rezultat turnira, s nazivom agenta i skupljenim bodovima. Agenti su sortirani od najboljeg (najmanji broj bodova) do najlošijeg (najveći broj bodova).

```
Igrač igrac2@127.0.0.1 je odigrao potez: D
Igrač igrac3@127.0.0.1 je odigrao potez: D
Igrač igrac2@127.0.0.1 je odigrao potez: D
Igrač igrac3@127.0.0.1 je odigrao potez: D
Izbačen: igrac3@127.0.0.1
gotova igra
igrac1@127.0.0.1: 144:  TFT
igrac2@127.0.0.1: 144:  TFT
igrac3@127.0.0.1: 174:  ALLD
igrac@127.0.0.1: 174:  ALLD
```

Slika 8 Ispis rezultata

4.2.3. Opis rezultata

U tablicama su prikazani rezultati testiranja. U svakom testiranju je korišteno 5 agenata igrača. Rezultati sadržavaju prosječni rezultat u 10 testova turnira s 50 iteracija.

Tablica 2 Rezultat 1. testiranja

STRATEGIJA	PROSJEČNI REZULTAT
ALLD	440,4
TFT	467
PAVLOV	529,2
JOSS	537,2
TESTER	605,4

Tablica 3 Rezultat 2. testiranja

STRATEGIJA	PROSJEČNI REZULTAT
TFT	440,4
ALLD	467
JOSS	529,2
JOSS	537,2
PAVLOV	605,4

Tablica 4 Rezultat 3. testiranja

STRATEGIJA	PROSJEČNI REZULTAT
ALLD	440,4
JOSS	467
JOSS	529,2
TFT	537,2
RAND	605,4

U prikazanim testiranjima su korištene različite kombinacije strategija u turniru. U prvom i trećem testiranju pobjedu je odnijela strategija ALLD, dok je u drugom pobjedila TFT. Razlog takvom ishodu leži u broju strategija koje u svom načelu preferiraju suradnju. U prvom testiranju imamo strategije TFT i JOSS koje možemo nazvati takvim strategijama. PAVLOV kreće sa suradnjom, ali sve dok mu se isplati sukob, ustraje na njemu. Strategija TESTER ovisi o prvom potezu hoće li se pretvoriti u TIT FOR TAT ili RAND. U drugom testiranju je izbačen TESTER i ubačen JOSS. To je rezultiralo profitiranjem strategije TFT jer

je JOSS vrlo slična strategija. Može se vidjeti da TFT najviše profitira u situacijama u kojima i druge strategije igraju suradnju. U trećem testiranju je uvrštena RAND strategija i u njemu je pobjedila ALLD.Uvođenje RAND strategije je odigralo ključnu ulogu u ovom rezultatu, jer je TFT izgubila određeni broj bodova koje bi inače dobila suradnjom s PAVLOV. JOSS strategija je sa svojim periodičnim ulaskom u sukob s TFT ostvarila sveukupno bolji rezultat.

4.3. Prostorna iterirana dilema zatvorenika

4.3.1. Općeniti opis

Drugi dio praktičnog rada je implementacija dileme zatvorenika u prostornom okružju. Aplikacija je zamišljena kao višeagentni sustav koji se sastoji od sveukupno 27 agenata i to od:

- 25 agenata "igrača" (Prisoner)
- 1 agenta "kreatora" (AgentCreator)
- 1 agenta zaduženog za proslijeđivanje zahtjeva s weba ka igračima (Handler)

Agenti igrači su smješteni u polje ili tablicu dimenzija 5x5 gdje svaki zauzima jednu ćeliju. Svaki agent ima koordinate (x,y). Svaki agent komunicira s AgentCreator agentom, Handler agentom i svojim susjedima. Susjedi agenta su definirani "Von Neumannovim susjedstvom", što znači da ukoliko su koordinate agenta (x,y), njegovi susjedi imaju koordinate (x+1,y), (x,y+1), (x-1,y), (x,y-1). Svaki agent ima jednu od mogućih 5 inicijalnih strategija (TESTER, PAVLOV, JOSS, ALLD, TFT) koje su opisane u prethodnom poglavlju. Igra funkcionira na način da agent na kraju svakog kruga preuzima najbolju strategiju nekog od svojih susjeda, pod uvjetom da njegova nije bila barem jednako uspješna. Igra završava ukoliko su se ponovila 2 kruga s jednakim strategijama.

AgentCreator

Centralni agent čija je zadaća delegiranje komunikacije između korisnika i agenata i agenata međusobno. Ima više funkcija:

- primanje korisničkih zahtjeva kroz web sučelje
- kreiranje Prisoner agenata i definiranje njihovih karakteristika (preuzimaju se iz xml datoteke)

- kreiranje Handler agenta
- prikupljanje rezultata od agenata i vraćanje korisniku
- delegiranje nove strategije Prisoner agentu na temelju rezultata jednog kruga igre.

Handler

Agent Handler se pokreće kada korisnik pokrene simulaciju, šalje podatke prema Prisoner agentima i završava. Njegova glavna funkcija je slanje odabranih strategija već kreiranim i pokrenutim Prisoner agentima, kako se oni ne bi svaki put prijavljivali kad korisnik pokrene simulaciju.

Prisoner

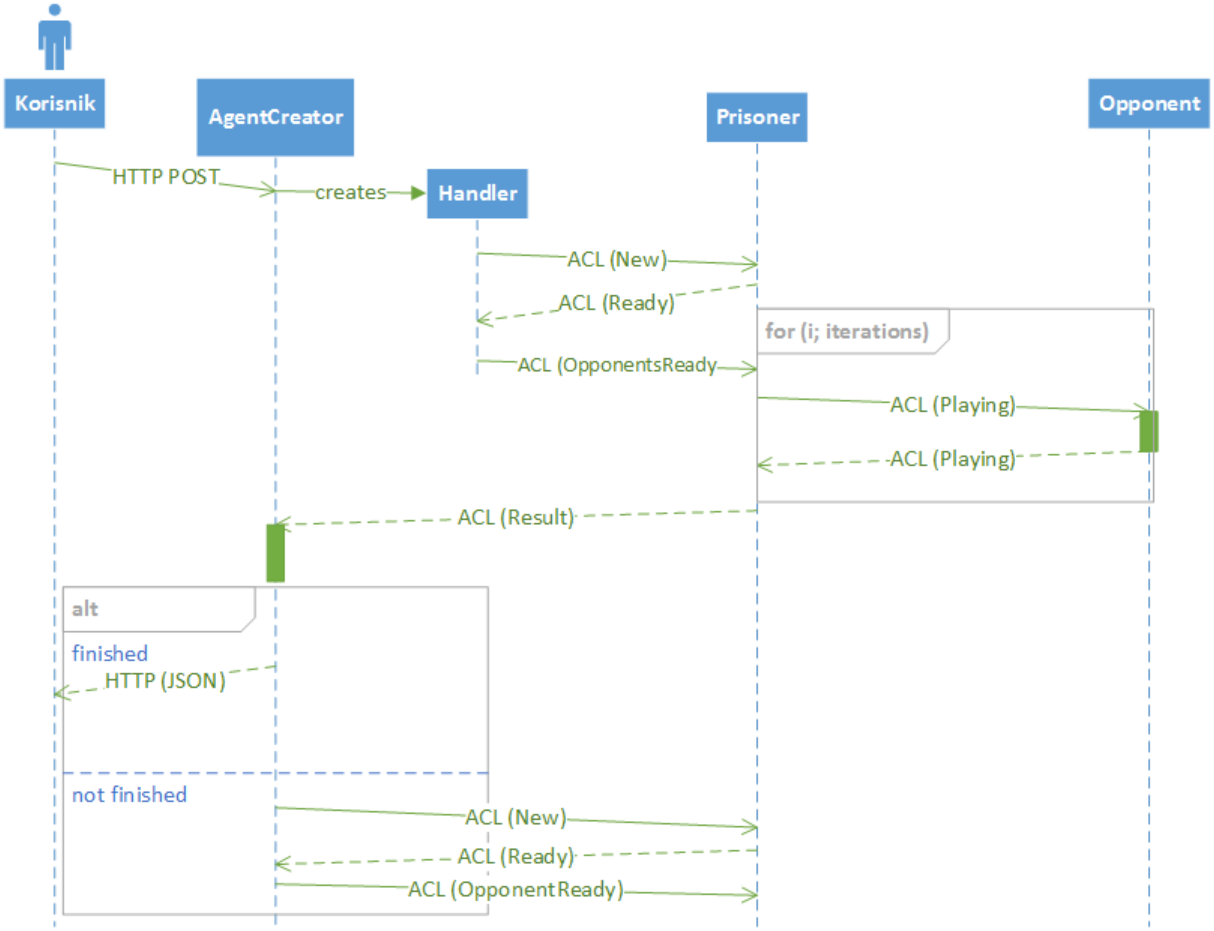
Prisoner agent je agent u kojem je implementirano ponašanje igrača u igri iterirane dileme zatvorenika. On je definiran svojom strategijom i susjedima. Pokreće ga AgentCreator koji ga prijavljuje na server i definira mu ime i adresu i adrese susjeda. Strategiju mu dodjeljuje Handler agent svaki put kad korisnik pokrene simulaciju. Prisoner agent tijekom igre komunicira s drugim Prisoner agentima (svojim susjedima), a rezultate igre šalje AgentCreator agentu. Od AgentCreator agenta u novom krugu dobiva novu strategiju ukoliko igra nije završila.

4.3.2. Opis komunikacije

Komunikacija među agentima se odvija slanje FIPA ACL poruka. Svaka poruka ima određenu ontologiju i na taj način agent zna njenu svrhu. Handler šalje poruku s ontologijom ("New") svim agentima Prisoner. S tom porukom agent Prisoner zna da počinje nova igra i da treba inicijalizirati potrebne podatke. U svrhu sinkronizacije agenata Prisoner, oni šalju poruku s ontologijom ("ReadyResponse"). Ukoliko Handler dobije od svih agenata poruke s ontologijom ("Ready"), šalje poruku s ontologijom ("OpponentReady") Prisoneru kako bi ovaj znao da može započeti igru.

Istu komunikaciju ima i AgentCreator s Prisonerom pri svakom krugu igre. Komunikacija tijekom kruga igre se odvija međusobno između Prisoner agenata. Svaki

Prisoner agent šalje inicijalnu poruku s ontologijom ("Playing") "desnom" i "donjem" susjedu, što automatski znači da prima inicijalnu poruku od "lijevog" i "gornjeg" susjeda. Takav način komunikacije je zamišljen kako bi se spriječile duplicirane poruke. Poruke agenti šalju izravno jedni drugima, bez posrednika. To znači da jedan igrač uvijek dobije potez svog protivnika. Radi optimizacije, kako ne bi sve poruke išle preko posrednika, agent ne zlorabi tu informaciju i u trenutku odigravnja svog poteza „ne zna“ potez svog protivnika. Ukoliko je krug igre dostignuo definiran broj iteracija Prisoner šalje poruku s ontologijom ("Result") AgentCreatoru.



Slika 9 Slijedni dijagram razmjene poruka između agenata i korisnika

4.3.3. Opis ponašanja agenata

Većina ponašanja su definirana putem *EventBehaviour* klase, odnosno ponašanja uzrokovanog događajem (dolaskom poruke određene ontologije).

Prisoner ima sljedeća ponašanja:

- NewGame: ponašanje uzrokovano dolaskom poruke s ontologijom "New". Služi za "resetiranje" agenta uslijed pokretanja novog kruga igre
- Receive: Ponašanje uzrokovano dolaskom poruke s ontologijom "Playing". Temeljno ponašanje tijekom igre, primanje, odigravanje poteza i slanje protivniku

AgentCreator ima sljedeća ponašanja:

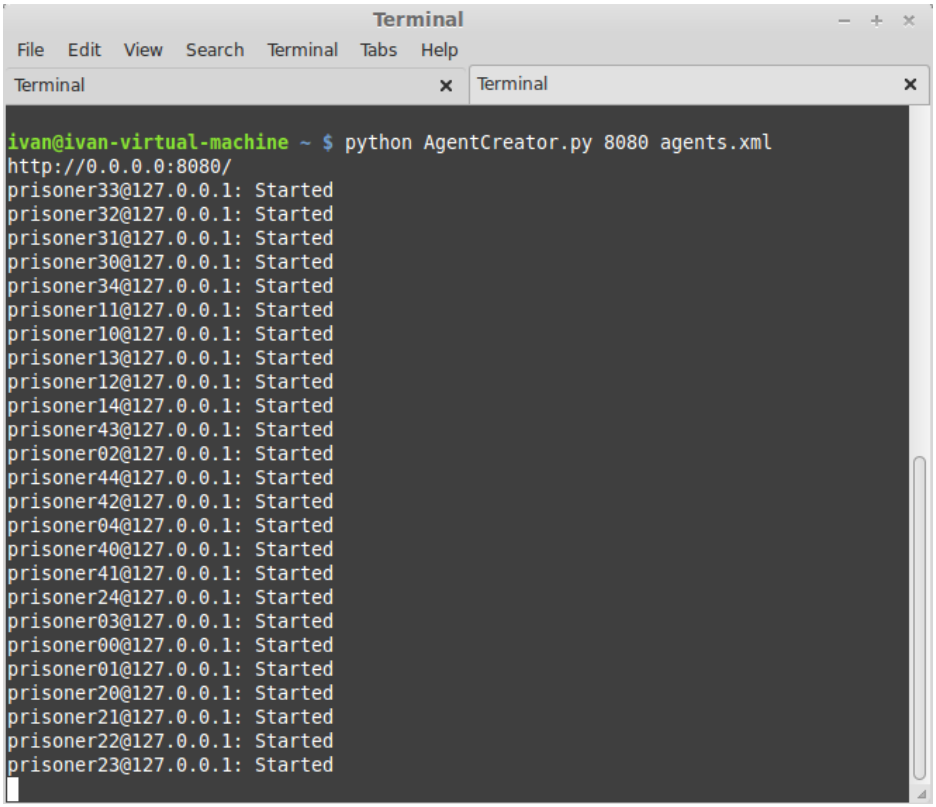
- ReadyResponse: ponašanje uzrokovano dolaskom poruke s ontologijom "ReadyResponse": služi za sinkronizaciju svih Prisoner agenata. Tek nakon što su svima dodijeljene nove strategije, Prisoner agenti šalju poruku AgentCreatoru i on nakon što je dobio "ReadyResponse" poruke od svih agenata, šalje potvrdu da novi krug može početi.
- StrategyEvaluation: ponašanje uzrokovano dolaskom poruke s ontologijom "Result": na kraju svakog kruga, Prisoner agenti šalju svoje rezultate AgentCreatoru koji na temelju rezultata dodjeljuje nove strategije agentima, ukoliko nije ispunjen uvjet za kraj igre.

Handler ima ponašanje ReadyResponse koje je opisano kod AgentCreator agenta.

4.3.4. Opis korištenja aplikacije

Korisnik aplikaciju pokreće putem terminala. Aplikacija je napisana u programskom jeziku python i za nju pokretanje se koristi python interpreter. Kao što je prikazano na slici 10, aplikacija prima 2 argumenta, TCP port za pristupanje aplikaciji preko web sučelja i lokaciju konfiguracijskog xml dokumenta. U dokumentu se nalaze podaci koji služe za konfiguriranje agenata. Struktura xml dokumenta je prikazana na slici 11[1].

[1] Na slici su prikazana 2 agenta. Datoteka sadrži jednako definiciju za preostala 23 agenta.

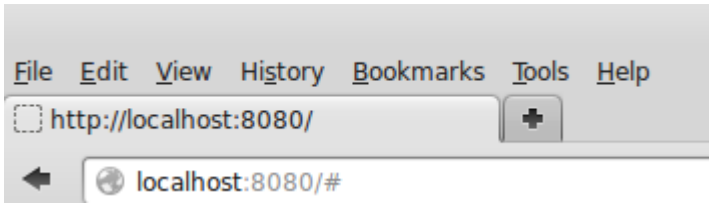


Slika 10 Pokretanje aplikacije iz terminala

Slika 11 Struktura konfiguracijskog XML dokumenta

Nakon pokretanja aplikacije kroz terminal, pokreću se agenti i zatim korisnik može pristupiti aplikaciji kroz web sučelje. Aplikaciji se pristupa kroz web preglednik upisujući adresu (u ovom slučaju lokalnu) i broj porta koji je unešen pri pokretanju aplikacije. Web adresa za pristup je prikazana na slici 12.

Za web sučelje se radi jednostavnosti koristi biblioteka **web.py**[2]**.** Posluživanje web sadržajem je uključeno u *AgentCreator* putem klasa *Start* i *Index*. *Index* sadrži metodu GET, a *Start* metodu POST. Web.py implementacijom tih metoda omogućuje jednostavno posluživanje web sadržaja. *Index* poslužuje statički web sadržaj, odnosno html, javascript i css dokumente prilikom pristupa aplikaciji preko web sučelja. *Start* implementacijom POST metode omogućuje prihvaćanje korisničkog unosa, odnosno početnih strategija za agente, a vraća odgovor u JSON obliku, koji sadrži povijest korištenih strategija za sve agente. Web sučelje je implementirano pomoću javascript biblioteke *jQuery*.



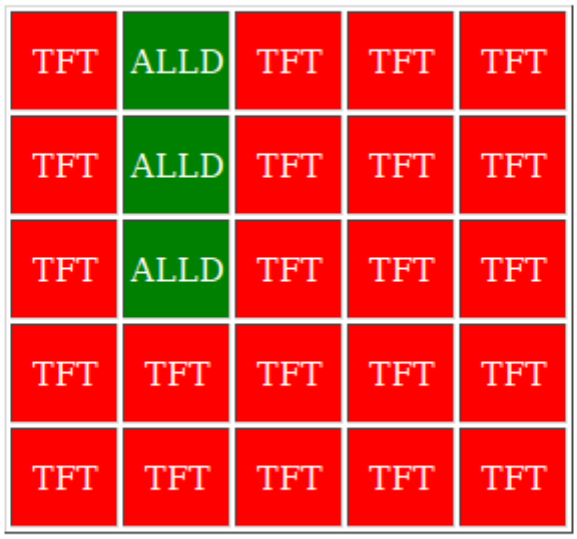
Slika 12 URL adresa za pristupanje web sučelju aplikacije

Na slici 13 je prikazan izgled web sučelja. Prva i druga tablice sadrže 25 ćelija koje svaka odgovaraju jednom od 25 agenata igrača. U prvoj tablici je omogućeno biranje između ponuđenih strategija. Nakon što se odabere strategija za svakog od igrača, klikne se na gumb „Pokreni“. Nakon toga se šalje zahtjev serveru koji prosljeđuje unešene strategije agentima te igra počinje. Nakon određenog vremena rezultat stiže i prikazuje se u donjoj tablici. Korisnik može navigacijom ispod tablice vidjeti dinamiku promjena strategija kroz igru. Svaka strategija je obojana različitom bojom.

[2] Python biblioteka za jednostavno posluživanje python web aplikacija. (http://webpy.org/).

Prostorna iterirana dilema zatvorenika

Aplikacija "Iterativna dilema zatvorenika u prostornom okruženju". Jedna ćelija predstavlja jednog igrača. Strategija se igraču može dodati iz drop-down menija. Nakon završetka igre, rezultat je prikazan u tablici ispod. Navigacijom se može pregledati dinamika strategija kroz igru.



1 2 3

Slika 13 Izgled web sučelja

4.4. Opis testiranja

Kao i u prethodnom testiranju klasične iterirane dileme zatvorenika i u ovom slučaju su za testiranje korištene različite kombinacije strategija. U tablicama su prikazani rezultati, odnosno dinamika izmjena strategija kroz igru. Igra traje sve dok agenti više nemaju razloga za promjenom strategija, što u načelu znači da ili dvije iteracije zaredom igraju suradnju ili sukob. Kombinacija strategija ima jako puno, budući da postoji 25 agenata s mogućnosti 5 različitih strategija. Provedeno je 6 različitih slučajeva i objasnit će se rezultati. Agenti u jednom krugu igraju 5 iteracija igre sa svojim susjedima. Rezultati su prikazani u tablicama ispod tako da su iteracije prikazane s lijeva na desno pa prema dolje.

Test 1

ALLD	ALLD	ALLD	TFT	TESTER
ALLD	ALLD	ALLD	TFT	ALLD
ALLD	ALLD	ALLD	ALLD	JOSS
ALLD	ALLD	TFT	ALLD	ALLD
ALLD	ALLD	TFT	PAVLOV	TFT

TESTER	ALLD	TFT	PAVLOV	TFT
ALLD	ALLD	ALLD	TFT	ALLD
ALLD	ALLD	ALLD	ALLD	ALLD
ALLD	TFT	TFT	PAVLOV	ALLD
TFT	TFT	PAVLOV	PAVLOV	PAVLOV

TFT	TFT	PAVLOV	PAVLOV	PAVLOV
ALLD	ALLD	TFT	PAVLOV	TFT
ALLD	TFT	TFT	ALLD	ALLD
TFT	TFT	PAVLOV	PAVLOV	PAVLOV
TFT	PAVLOV	PAVLOV	PAVLOV	PAVLOV

PAVLOV	PAVLOV	PAVLOV	PAVLOV	PAVLOV
TFT	TFT	PAVLOV	PAVLOV	PAVLOV
TFT	TFT	PAVLOV	TFT	PAVLOV
TFT	TFT	PAVLOV	PAVLOV	PAVLOV
TFT	PAVLOV	PAVLOV	PAVLOV	PAVLOV

Prvi test je imao četiri kruga. U početnim pozicijama najviše agenata je imalo ALLD strategiju, zatim TFT, a po jedan agent je imao PAVLOV, JOSS i TESTER strategiju. U zadnjoj iteraciji se može vidjet da većina agenata ima strategiju PAVLOV, što znači da se kroz evoluciju strategija PAVLOV od samo jednog agenta proširila na čak 17. Na početku agent sa strategijom PAVLOV je okružen s dva agenta sa strategijom TFT i dva s ALLD. Takva pozicija mu omogućuje da profitira od jednog i od drugog, jer nastavlja započetu suradnju s TFT i odgovara sukobom na ALLD strategiju. Na početku 3. kruga većina agenata s PAVLOV strategijom je okružena s drugim agentima s istom strategijom ili s TFT pa mu to omogućuje prednost nad agentima s TFT koji su većinom okruženi s ALLD. U zadnjem krugu se njegova brojnost samo još poveća te je na kraju rezultat 17 agenata s PAVLOV i 8 s TFT strategijom. TESTER i JOSS nestaju odmah u drugoj iteraciji, a ALLD zbog svoje brojnosti ostaje sve do zadnje. Igra se prekida jer PAVLOV i TFT u zadnjoj iteraciji ulaze u suradnju i nastavak igre ne bi utjecao na promjenu rezultata.

Test 2

ALLD	ALLD	ALLD	TFT	TESTER
ALLD	ALLD	ALLD	TFT	ALLD
ALLD	ALLD	ALLD	ALLD	JOSS
ALLD	ALLD	TFT	ALLD	ALLD
ALLD	ALLD	ALLD	PAVLOV	TFT

ALLD	ALLD	TFT	TFT	TFT
ALLD	ALLD	TFT	TFT	ALLD
ALLD	ALLD	ALLD	ALLD	ALLD
ALLD	ALLD	ALLD	ALLD	ALLD
ALLD	ALLD	ALLD	TFT	PAVLOV

TFT	TFT	TFT	TFT	TFT
ALLD	TFT	TFT	TFT	TFT
ALLD	ALLD	TFT	TFT	ALLD
ALLD	ALLD	ALLD	TFT	ALLD
ALLD	ALLD	TFT	TFT	TFT

TFT	TFT	TFT	TFT	TFT
TFT	TFT	TFT	TFT	TFT
ALLD	TFT	TFT	TFT	TFT
ALLD	ALLD	TFT	TFT	TFT
TFT	TFT	TFT	TFT	TFT

TFT	TFT	TFT	TFT	TFT
TFT	TFT	TFT	TFT	TFT
TFT	TFT	TFT	TFT	TFT
TFT	TFT	TFT	TFT	TFT
TFT	TFT	TFT	TFT	TFT

Drugi test je imao 5 krugova. Razlikuje se od prvog testa u samo jednom detalju, a to je da je na poziciji (1,3) strategija iz TFT promijenjena u ALLD. To je uzrokovalo da strategija PAVLOV nestane već u drugom krugu. U prvoj je desni susjed strategije PAVLOV primio tu strategiju, ali nije uspjela ostati na početnom mjestu. Budući da je PAVLOV u ovom slučaju bila okružena s 3 strategije ALLD, nije uspjela profitirati toliko kao u prvom testu jer je s njom konstantno ulazila u sukob i dobivala 3 boda. U 3. krugu situacija je jasna, jedine preostale strategije su TFT i ALLD. TFT je brojčano nadmoćnija, a ALLD prosječno dobiva 3 boda i kroz još 2 kruga TFT postaje jedina strategija u matrici.

Test 3

ALLD	ALLD	ALLD	TFT	TESTER
ALLD	ALLD	ALLD	TFT	ALLD
TFT	JOSS	TFT	JOSS	ALLD
ALLD	TFT	TFT	TFT	ALLD
ALLD	ALLD	ALLD	PAVLOV	TFT

ALLD	ALLD	TFT	PAVLOV	TFT
ALLD	JOSS	TFT	JOSS	TFT
JOSS	JOSS	JOSS	JOSS	JOSS
TFT	JOSS	TFT	JOSS	TFT
ALLD	ALLD	TFT	PAVLOV	PAVLOV

TFT	TFT	PAVLOV	JOSS	PAVLOV
JOSS	JOSS	JOSS	JOSS	JOSS
JOSS	JOSS	JOSS	JOSS	JOSS
TFT	JOSS	JOSS	JOSS	JOSS
PAVLOV	TFT	PAVLOV	JOSS	PAVLOV

PAVLOV	TFT	JOSS	JOSS	JOSS
TFT	TFT	JOSS	JOSS	JOSS
JOSS	JOSS	JOSS	JOSS	JOSS
PAVLOV	JOSS	JOSS	JOSS	JOSS
PAVLOV	PAVLOV	JOSS	JOSS	PAVLOV

PAVLOV	TFT	TFT	JOSS	JOSS
JOSS	TFT	JOSS	JOSS	JOSS
JOSS	JOSS	JOSS	JOSS	JOSS
PAVLOV	JOSS	JOSS	JOSS	JOSS
PAVLOV	PAVLOV	JOSS	JOSS	PAVLOV

TFT	TFT	TFT	JOSS	JOSS
JOSS	TFT	JOSS	JOSS	JOSS
JOSS	JOSS	JOSS	JOSS	JOSS
PAVLOV	JOSS	JOSS	JOSS	JOSS
PAVLOV	TFT	JOSS	JOSS	PAVLOV

U 3. testu je početna situacija slična kao u prva dva, s tim da je strategija JOSS pomaknuta u sredinu i okružena s lijeve i desne strane strategijama TFT. Već se u drugom krugu može vidjeti učinak takve organizacije gdje JOSS dolazi s 2 agenta na čak 9. Razlog se može tražiti u tome što JOSS, iako gotovo identična TFT strategiji, u 5. krugu igra sukob i toj joj donosi prednost nad svim strategijama koje surađuju upravo zbog toga što je broj iteracija u simulaciji određen na 5 te agenti drugih strategija nemaju priliku odgovoriti na njen sukob. U 3. krugu nestaje strategija ALLD i ostaju samo strategije JOSS, PAVLOV i TFT. U zadnja 3 kruga se događa zanimljiva situacija gdje broj agenata sa strategijom JOSS ostaje 17, a mijenja se broj PAVLOV i TFT strategija. U 4. i 5. krugu 5 agenata ima PAVLOV strategiju, a 3 agenta TFT strategiju. Na kraju je situacija obrnuta gdje 5 agenata ima TFT, a 3 PAVLOV. U tom trenutku dolazi do situacije u kojoj su agenti u takvim pozicijama gdje periodični ulazak strategije JOSS u sukob svakih 5 iteracija ne uzrokuje daljnju promjenu.

Test 4

ALLD	ALLD	ALLD	TFT	TESTER
ALLD	ALLD	ALLD	TFT	ALLD
TFT	JOSS	TFT	JOSS	ALLD
ALLD	TFT	TFT	TFT	ALLD
ALLD	ALLD	ALLD	PAVLOV	TFT

ALLD	ALLD	TFT	PAVLOV	TFT
ALLD	JOSS	TFT	TFT	TFT
JOSS	TFT	TFT	TFT	JOSS
TFT	TFT	TFT	TFT	TFT
TFT	TFT	TFT	PAVLOV	PAVLOV

TFT	TFT	PAVLOV	PAVLOV	PAVLOV
TFT	TFT	TFT	TFT	TFT
TFT	TFT	TFT	TFT	TFT
TFT	TFT	TFT	TFT	PAVLOV
PAVLOV	TFT	TFT	PAVLOV	PAVLOV

Početna raspodjela ovog testa je identična prethodnom, jedina promjena broj iteracija koje agenti igraju u određenom krugu s 5 na 8 (promjena u xml datoteci oznake *iterations*). Očigledna je razlika između prethodnog i ovog testa. Ovaj test ima 3 kruga i JOSS strategija nestaje u zadnjem. Razlog tomu leži u činjenici da je broj iteracija promijenjen na 8 pa strategije TFT i PAVLOV mogu odgovoriti strategiji JOSS na sukob u 5. iteraciji. Pobjedu je odnijela strategija TFT zbog brojčane prednosti nad PAVLOV.

Test 5

TFT	ALLD	TESTER	ALLD	TFT
ALLD	TESTER	ALLD	ALLD	ALLD
TFT	TESTER	ALLD	ALLD	TFT
ALLD	TFT	TFT	TFT	ALLD
ALLD	ALLD	TESTER	ALLD	JOSS

ALLD	ALLD	ALLD	TFT	TFT
ALLD	ALLD	ALLD	ALLD	TFT
TESTER	ALLD	TFT	ALLD	ALLD
ALLD	TFT	TFT	TFT	ALLD
ALLD	ALLD	TFT	TESTER	TFT

TFT	ALLD	ALLD	TFT	TFT
ALLD	ALLD	ALLD	ALLD	TFT
ALLD	ALLD	TFT	ALLD	ALLD
ALLD	TFT	TFT	TFT	ALLD
ALLD	ALLD	TFT	TESTER	TFT

Ovo je jedini test u kojem je strategija ALLD ne samo pobjedila, nego uopće ostala u zadnjem krugu. Iako je i u prijašnjim testovima bilo najviše agenata s ALLD strategijom na početku, razmještaj je u ovom testu bio presudan. Može se vidjeti da je ALLD u 2. i 3. stupcu okružen TESTER strategijama koje se većinom pretvaraju u ALLD. Ključno je i to što su se dva agenta sa strategijom TFT pretvorila u TESTER u 2. krugu.

Test 5

TFT	ALLD	TESTER	PAVLOV	JOSS
TFT	ALLD	TESTER	PAVLOV	JOSS
TFT	ALLD	TESTER	PAVLOV	JOSS
TFT	ALLD	TESTER	PAVLOV	JOSS
TFT	ALLD	TESTER	PAVLOV	JOSS

JOSS	TFT	PAVLOV	JOSS	JOSS
JOSS	TFT	PAVLOV	JOSS	JOSS
JOSS	TFT	PAVLOV	JOSS	JOSS
JOSS	TFT	PAVLOV	JOSS	JOSS
JOSS	TFT	PAVLOV	JOSS	JOSS

TFT	TFT	PAVLOV	PAVLOV	JOSS
TFT	TFT	PAVLOV	PAVLOV	JOSS
TFT	TFT	PAVLOV	PAVLOV	JOSS
TFT	TFT	PAVLOV	PAVLOV	JOSS
TFT	TFT	PAVLOV	PAVLOV	JOSS

Ovaj test ima specifičnu raspodjelu. Svaka strategija zauzima jedan stupac u matrici. U drugom krugu se vidi da je strategija JOSS napravila najveći rezultat jer su ju svi agenti susjedi poprimili. TESTER i ADD su poprimili strategije PAVLOV i TFT. No na kraju u trećem krugu strategija JOSS nije najbolje prošla jer je igrala sama sa sobom te sa strategijom PAVLOV i TFT. Pavlov i TFT su međusobno napravili dobar rezultat igrajući suradnju i zauzeli prvi i četvrti stupac, tako da je JOSS u 3. krugu završila tamo gdje je i počela, u zadnjem stupcu.

4.4.1. Zaključak testiranja

Provedeno je 6 testiranja. Cilj je bio pokazati različite odnose među strategijama. Budući da ima velik broj kombinacija koje se mogu testirati, ovih 6 nije velik broj, ali ipak se iz njih mogu izvesti određeni zaključci. Prvi zaključak koji se naočigled vidi je da strategija ALLD nije dobro prošla u testiranju. Ako se usporedi s klasičnim oblikom igre, može se reći da je nagrada za suradnju u prostornoj verziji igre daleko izraženija. Također se vidjelo da u slučaju strategije JOSS, rezultat ovisi o broju iteracija. Kad je bilo 5 iteracija JOSS je bila puno uspješnija nego u slučaju s 8. Naročto je na to utjecalo i to što je JOSS strategiji bilo definirano da svaku 5. iteraciju ulazi u sukob pa strategije koje su s njom dotad surađivale nisu imale odgovora. U 3. testiranju se dogodila zanimljiva situacija gdje se broj JOSS strategije nije mijenjao 3 kruga, nego je strategija samo promijenila lokaciju (agente) i dogodila se situacija da takav raspored više ne utječe na promjenu strategija.

Zanimljivost ovakvog pristupa je u tome da strategija nekog agenta ovisi o svim ostalim agentima. Strategija agenta ovisi o susjednoj strategiji čiji se rezultat izračunava interakcijom s njenim susjedima. Na taj način su svi agenti u matrici povezani i više ili manje ovise jedni o drugima.

5. Zaključak

U radu su opisane dvije samostalne, ali kako se vidjelo i usko povezane discipline. Na početku je bilo riječi o teoriji igara kao znanstvenoj disciplini koja proučava socijalna ponašanja entiteta i pokušava modelirati svakodnevne interakcije kroz matematičke modele. Definirani su termini poput Pareto optimalnosti, dominacije i Nashove ravnoteže. U drugom dijelu je opisana novija disciplina, višeagentni sustavi. Opisano je na koji način računalni programi mogu biti entiteti u sustavu u kojem mogu djelovati samostalno i biti u interakciji s drugim agentima. Spomenuti su različiti tipovi agenata poput BDI agenta i agenta sa sposobnošću praktičnog rezoniranja.

Poveznica između ove dvije discipline je upravo u spomenutoj komunikaciji. Agenti, iako zasebni entiteti, u komunikaciji s drugim agentima u sustavu mijenjaju okolinu u kojoj se nalaze. Takve interakcije upravo proučava teorija igara. Modeliranje scenarija u kojem agenti svojim potezima utječu na izgled višeagentnog sustava je temeljna zadaća teorije igara.

Praktični rad je izrađen u SPADE platformi za razvoj višeagentnih sustava, koja omogućuje jednostavnu implementaciju višeagentnog sustava. U praktičnom dijelu rada je odabrana naizgled jednostavna igra: Dilema zatvorenika. Iako naizgled igra jednostavne definicije, nudi mogućnost analize i implementacije na različite načine. U uvom radu je obrađena na 2 načina, kao klasična iterirana dilema zatvorenika, organizirana poput turnira kakvog je održao i Axelrod 1984. godine. Drugi način implementacije je u obliku vrste evolucijske igre nazvane prostornom igrom. Opet se radilo o iteriranoj verziji igre, ali sada stavljenoj u „prostor“, što je omogućilo praćenje razvoja neke strategije interakcijom s drugim strategijama. Moglo se uočiti i sličnosti i razlika u odnosima strategija između spomenutih verzija igre.

U radu je prikazana jednostavna interakcija agenata kroz jednostavnu igru. Teorija igara se može primjeniti u mnogo kompleksnijim računalnim sustavima. Kao jedna od primjena se mogu navesti aukcije. Aukcije, iako još postoje u svom osnovnom obliku, naveliko su implementirane kao računalni sustavi. Kao primjer se može navesti **e-bay**. Aukcija se također može opisati kao igra između više agenata koji se međusobno natječu u procjeni vrijednosti nekog dobra.

Višeagentni sustavi se često primjenjuju u analiziranju socijalnih sustava. Njima se može modelirati i implementirati ponašanje modela nekih stvarnih ekonomskih, inteligentnih, i socijalnih sustava te iskoristiti podatke za analizu interakcije elemenata.

Iako novu disciplinu višeagentnih sustava mnogi poistovjećuju s različitim disciplinama poput umjetne inteligencije, ekspertnih sustava pa i same teorije igara, ona se ipak profilirala u samostalnu disciplinu koja koristi ostale discipline. Može se zaključiti da teorija igara disciplini višeagentnih sustava nudi mehanizam za opisivanje interakcija među agentima i utjecaju na cijeli višeagentni sustav.

6. Literatura

[1] Von Neumann J., Morgenstern O. (1953) *Games and economic behaviour* Princeton: Princeton university press

[2] Thomas L.C. (1986) *Game Theory and Applications*. Manchester: ELLIS HORWOOD LTD.

[3] Axelrod A. (1984) *The Evolution of Cooperation*, Basic Books, New York. Dostupno na http://www-personal.umich.edu/~axe/Axelrod_Evol_of_Coop_excerpts.pdf (učitano 6.7.2014.)

[4] Gibbons R. (1992) *Game Theory for Applied Economists* New Jersey: Princeton University.

[5] Shoham Y., Leyton – Brown K. (2009) *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge: Cambridge University Press. Dostupno na http://www.masfoundations.org/mas.pdf (učitano 6.7.2014.)

[6] Wooldridge M. (2009) *An Introduction to Multiagent Systems.* John Wiley & Sons, Ltd

[7] Johansson S.J. (1999) *Game Theory and Agents*, Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby. Dostupno na http://www.agent.ai/doc/upload/200403/joha99_1.pdf (učitano 6.7.2014.)

[8] Baron R., Durieu J., Haller, H., Solal P., *Finding a NashEquilibrium in Spatial Games is an NP-Complete Problem* (2002), Department of Economics University of Copenhagen. Dostupno na http://www.econ.ku.dk/research/publications/pink/2002/0219.pdf (učitano 6.7.2014.)

[9] Easley D., Kleinberg J. (2010), *Evolutionary Game Theory*., poglavlje 7 Cambridge University Press. Dostupno na http://www.cs.cornell.edu/home/kleinber/networks-book/networks-book-ch07.pdf (učitano 6.7.2014.)

[10] Aranda G., Palanca J. (2005) *SPADE User's Manual*, dokumentacija SPADE platforme, dostupna na http://pythonhosted.org/SPADE (učitano 6.7.2014.)

7. Prilozi

7.1. PRILOG 1 – Implementacija klasične igre dilema zatvorenika

organizator.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import sys, re, spade
import random
from spade.Agent import Agent
from spade.Behaviour import Behaviour, EventBehaviour, ACLTemplate,
MessageTemplate
from spade.DF import ServiceDescription, DfAgentDescription
from spade.AID import aid
from spade.ACLMessage import ACLMessage
from sets import Set
from random import choice

class Igra:
    def __init__(self):
        self.igraci = []
        self.potezi = [None]*2
        self.ukupni_rezultat = [0,0]
        self.brojac = 0

class Organizator(Agent):
    class Prijava(EventBehaviour):
        def _process(self):
            msg = self._receive()
            print msg.getContent()
            igrac = msg.getSender()
            self.myAgent.myIgraci.append(igrac)
            self.myAgent.strategije[igrac.getName()] = msg.getContent()
            self.myAgent.bodovi[igrac.getName()] = 0
            self.myAgent.igraci_pomocna[igrac.getName()] = igrac

            if len(self.myAgent.igraci_pomocna) ==
self.myAgent.brojSudionika:
                for i in range(0, self.myAgent.brojSudionika-1):

self.myAgent.igraci[self.myAgent.myIgraci[i].getName()] = []
                    for j in range(i+1, self.myAgent.brojSudionika):

self.myAgent.igraci[self.myAgent.myIgraci[i].getName()].append(self.myAg
ent.myIgraci[j])
                keys = self.myAgent.igraci.keys()
                self.myAgent.aktualni_igrac = keys[0]
                self.myAgent.dodjeliProtivnika(False)


    class Igranje(EventBehaviour):
        def _process(self):
            msg = self._receive()
            index_igre = self.myAgent.mapa[msg.getSender().getName()]
```

```
            igra = self.myAgent.turnir[index_igre]
            if(igra.brojac < self.myAgent.brojIteracija):
                print "Igrač "+msg.getSender().getName()+" je odigrao
potez: "+msg.getContent()
                igra.potezi[igra.igraci.index(msg.getSender())] =
msg.getContent()
                if None not in igra.potezi:
                    rezultat =
self.myAgent.matrica_placanja[igra.potezi[0]+igra.potezi[1]]
                    igra.ukupni_rezultat[0] = igra.ukupni_rezultat[0] +
rezultat[0]
                    igra.ukupni_rezultat[1] = igra.ukupni_rezultat[1] +
rezultat[1]
                    self.myAgent.sendMSG(rezultat[0], igra.igraci[0],
"Rezultat")
                    self.myAgent.sendMSG(rezultat[1], igra.igraci[1],
"Rezultat")
                    igra.potezi[0] = None
                    igra.potezi[1] = None
                    igra.brojac = igra.brojac + 1
            else:
                self.myAgent.sendMSG("FIN", msg.getSender(),"Rezultat" )
                if msg.getSender().getName() ==
igra.igraci[0].getName():

self.myAgent.bodovi[igra.igraci[0].getName()]+=igra.ukupni_rezultat[0]

self.myAgent.bodovi[igra.igraci[1].getName()]+=igra.ukupni_rezultat[1]
                    self.myAgent.dodjeliProtivnika(True)

    def izvjesti(self):
        for key, value in sorted(self.bodovi.iteritems(), key=lambda
(k,v):
            (v,k)):
                print "%s: %s:  %s" % (key, value, self.strategije[key])

    def dodjeliProtivnika(self, pop):
        protivnici = self.igraci[self.aktualni_igrac]
        if pop:
            izbacen = protivnici.pop(0)
            print "Izbačen: "+ izbacen.getName()
        if len(protivnici) == 0:
            izbaceni = self.igraci.pop(self.aktualni_igrac)
            if len(self.igraci) == 0:
                print "gotova igra"
                for igrac in self.myIgraci:
                    self.sendMSG("SHUTDOWN", igrac, "Rezultat")
                self.izvjesti()
                self._shutdown()
            keys = self.igraci.keys()
            self.aktualni_igrac = keys[0]
        protivnici = self.igraci[self.aktualni_igrac]
        igrac = self.igraci_pomocna[self.aktualni_igrac]
        igrac2 = protivnici[0]
        igra = Igra()

        igra.igraci.extend([igrac, igrac2])
        print igra.igraci[0].getName() + " vs.
"+igra.igraci[1].getName()
        self.turnir.append(igra)
        self.mapa[igra.igraci[0].getName()] = self.turnir.index(igra)
```

```
        self.mapa[igra.igraci[1].getName()] = self.turnir.index(igra)
        self.sendMSG(igra.igraci[0].getName(), igra.igraci[1],
"Protivnik")
        self.sendMSG(igra.igraci[1].getName(), igra.igraci[0],
"Protivnik")

    def sendMSG(self, content, receiver, ontology):
        msg = ACLMessage()
        msg.addReceiver(receiver)
        msg.setOntology(ontology)
        msg.setContent(content)
        self.send(msg)

    def _setup(self):
        self.opisnik_usl = DfAgentDescription()
        self.opisnik_usl.setAID(self.getAID())
        ou = ServiceDescription()
        ou.setName('Axelrodov turnir')
        ou.setType('AxelrodTurnir')
        self.opisnik_usl.addService(ou)
        self.registerService(self.opisnik_usl)

        self.brojSudionika = int(sys.argv[2])
        self.brojIteracija = int(sys.argv[3])
        self.myIgraci = []
        self.igraci = {}
        self.igraci_pomocna = {}
        self.strategije = {}
        self.bodovi = {}
        self.aktualni_igrac = None
        self.turnir = []
        self.mapa = {}
        self.matrica_placanja =
{'CC':[1,1],'DC':[0,5],'CD':[5,0],'DD':[3,3]}
        self.brojac = 0


        p = ACLTemplate()
        p.setOntology('Prijava')
        m = MessageTemplate(p)
        self.addBehaviour(self.Prijava(), m)

        p1 = ACLTemplate()
        p1.setOntology('Igranje')
        m1 = MessageTemplate(p1)
        self.addBehaviour(self.Igranje(),m1)



if __name__ == '__main__':
    if len(sys.argv) < 3:
        raise ValueError("Nedostaju argumenti: naziv_igraca
broj_sudionika broj iteracija")
    if(re.compile("^[\w\d_-]+$").match(sys.argv[1])):
        naziv = sys.argv[1]
    else:
        raise ValueError("Prvi argument: naziv igraca: mora biti
kombinacija slova ili brojeva bez razmaka")
    if not str.isdigit(sys.argv[2]):
        raise ValueError("Drugi argument: broj sudionika: mora biti
broj")
```

```
    if not str.isdigit(sys.argv[3]):
        raise ValueError("Treci argument: broj iteracija: mora biti
broj")
    ime = naziv + '@127.0.0.1'
    o = Organizator(ime, 'tajna')
    o.start()
```

igrac.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import sys, spade
import re
from spade.Agent import Agent
from spade.DF import ServiceDescription
from spade.Behaviour import OneShotBehaviour, EventBehaviour, ACLTemplate,
MessageTemplate
from spade.ACLMessage import ACLMessage
from random import choice

class Igrac(Agent):
    class Prijava(OneShotBehaviour):
        def _process(self):
            u = ServiceDescription()
            u.setType('AxelrodTurnir')
            res = self.myAgent.searchService(u)

            self.myAgent.organizator = res[0]
            self.myAgent.sendMSG(self.myAgent.strategija,
self.myAgent.organizator.getAID(), 'Prijava')


    class Igranje(EventBehaviour):
        def _process(self):
            msg = self._receive()
            self.myAgent.protivnik = msg.getContent()
            if not self.myAgent.protivnik in self.myAgent.potezi:
                self.myAgent.potezi[self.myAgent.protivnik] = []
            strategija =
self.myAgent.mapa_strategija[self.myAgent.strategija]
            content = strategija.metoda()
            self.myAgent.sendMSG(content,msg.getSender(),"Igranje")

    class Rezultat(EventBehaviour):
        def _process(self):
            msg = self._receive()
            rezultat = msg.getContent()
            self.myAgent.rezultati.append(rezultat)
            print "rezultat: "+rezultat
            if rezultat == "SHUTDOWN":
                self.myAgent._shutdown()
            else:
                mapa = {3:'D',0:'C',5:'D',1:'C'}

self.myAgent.potezi[self.myAgent.protivnik].append(mapa[int(rezultat)])
```

```
                strategija =
self.myAgent.mapa_strategija[self.myAgent.strategija]
                content = strategija.metoda()
                self.myAgent.sendMSG(content, msg.getSender(), "Igranje")


    #slijede implementacije strategija
    class AllC:
        def metoda(self):
            return 'C'

    class AllD:
        def metoda(self):
            return 'D'

    class Rand:
        def metoda(self):
            return choice(['C','D'])

    class Tft:
        def __init__(self, agent):
            self.agent = agent

        def metoda(self):
            if len(self.agent.potezi[self.agent.protivnik]) == 0:
                return 'C'
            return self.agent.potezi[self.agent.protivnik][-1]

    class Tester:
        def __init__(self, agent):
            self.agent = agent

        def metoda(self):
            if len(self.agent.potezi[self.agent.protivnik]) == 0:
                return 'D'
            if self.agent.potezi[self.agent.protivnik][0] == 'C':
                return self.agent.potezi[self.agent.protivnik][-1]
            return choice(['C','D'])

    class Joss:
        def __init__(self, agent):
            self.agent = agent

        def metoda(self):
            if len(self.agent.potezi[self.agent.protivnik]) == 0:
                return 'C'
            if len(self.agent.potezi[self.agent.protivnik]) % 5 == 0:
                return 'D'
            return self.agent.potezi[self.agent.protivnik][-1]

    class Pavlov:
        def __init__(self, agent):
            self.agent = agent

        def metoda(self):
            if len(self.agent.potezi[self.agent.protivnik]) == 0:
                return 'C'
            if self.agent.rezultati[-1] == str(0):
                return 'D'
            if self.agent.rezultati[-1] == str(1):
                return 'C'
```

```
            if self.agent.rezultati[-1] == str(3):
                return 'C'
            if self.agent.rezultati[-1] == str(5):
                return 'D'



    def sendMSG(self, content, receiver, ontology):
        msg = ACLMessage()
        msg.addReceiver(receiver)
        msg.setOntology(ontology)
        msg.setContent(content)
        self.send(msg)

    def _setup(self):
        self.strategija = sys.argv[2]
        self.addBehaviour(self.Prijava())
        self.protivnik = None
        self.potezi = {}
        self.mapa_strategija = {"ALLC":self.AllC(), "ALLD":self.AllD(),
"RAND":self.Rand(), "TFT":self.Tft(self),
        "TESTER":self.Tester(self), "JOSS":self.Joss(self),
"PAVLOV":self.Pavlov(self)}
        self.rezultati = []
        self.potezi = {}

        p = ACLTemplate()
        p.setOntology('Protivnik')
        m = MessageTemplate(p)
        self.addBehaviour(self.Igranje(),m)

        p1 = ACLTemplate()
        p1.setOntology('Rezultat')
        m1 = MessageTemplate(p1)
        self.addBehaviour(self.Rezultat(),m1)

if __name__=='__main__':
    if len(sys.argv) < 3:
        raise ValueError("Nedostaju argumenti: naziv_igraca strategija")
    if re.compile("^[\w\d_-]+$").match(sys.argv[1]):
        naziv = sys.argv[1]
    else:
        raise ValueError("Prvi argument: naziv igraca: mora biti
kombinacija slova ili brojeva bez razmaka")
    if
re.compile("ALLC|ALLD|RAND|TFT|TESTER|JOSS|PAVLOV|").match(sys.argv[1]):
        strategija = sys.argv[2]
    else:
        raise ValueError("Drugi argument: strategija: mora biti nešto od
sljedećeg: ALLD, ALLC, RAND, TFT, JOSS, TESTER, PAVLOV")
    ime = naziv + '@127.0.0.1'
    s = Igrac(ime,'tajna')
    s.start()
```

7.2. PRILOG 2 – Implementacija prostorne igre dilema zatvorenika

Prisoner.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import sys, spade
import re
import threading
from spade.Agent import Agent
from spade.DF import ServiceDescription
from spade.Behaviour import OneShotBehaviour
from spade.Behaviour import EventBehaviour, ACLTemplate, MessageTemplate
from spade.ACLMessage import ACLMessage
from random import choice
from pprint import pprint

class Prisoner(Agent):

    class NewGame(EventBehaviour):
        def _process(self):
            self.myAgent.moves = {}
            self.myAgent.result = 0
            self.myAgent.counter = 0
            msg = self._receive()
            sender = msg.getSender()
            self.myAgent.strategyObject =
self.myAgent.strategies[msg.getContent()]
            msg = ACLMessage()
            msg.setOntology('ReadyResponse')
            msg.setContent("ready")
            msg.addReceiver(sender)
            self.myAgent.send(msg)

    class OpponentReady(EventBehaviour):
        def _process(self):

            msg = self._receive()
            msg = ACLMessage()
            msg.setOntology('Playing')
            move = self.myAgent.strategyObject.play('')
            msg.setContent(move)
            for i in range(0,2):
                name = self.myAgent.neighbours[i] + '@127.0.0.1'
                self.myAgent.moves[name] = {'O':[move], 'P':[]}
                msg.addReceiver(spade.AID.aid(name, ["xmpp://"+name]))
            self.myAgent.send(msg)

    class Receive(EventBehaviour):
        def _process(self):
            agent = self.myAgent
            msg = self._receive()
            content = msg.getContent()
            opponentAid = msg.getSender()
            opponent = msg.getSender().getName()
```

```
            if not opponent in agent.moves:
                agent.moves[opponent] = {'O':[], 'P':[]}

            if len(agent.moves[opponent]['O']) < agent.iterations:
                added = False
                if len(agent.moves[opponent]["O"]) >
len(agent.moves[opponent]["P"]):
                    agent.moves[opponent]["P"].append(content)
                    added = True
                move = self.myAgent.strategyObject.play(opponent)

                msg1 = ACLMessage()
                msg1.setOntology("Playing")
                msg1.setContent(move)
                msg1.addReceiver(opponentAid)
                agent.send(msg1)
                agent.moves[opponent]["O"].append(move)
                if not added:
                    agent.moves[opponent]["P"].append(content)

            else:
                agent.moves[opponent]["P"].append(content)
            if len(agent.moves[opponent]['O']) == agent.iterations and
len(agent.moves[opponent]['P']) == agent.iterations:
                agent.result += self.calculateResult(opponent)
                agent.counter += 1
                if agent.counter == 4:
                    msg = ACLMessage()
                    msg.setOntology("Result")
                    msg.setContent(agent.result)
                    msg.addReceiver(spade.AID.aid("creator@127.0.0.1",
["xmpp://creator@127.0.0.1"]))
                    agent.send(msg)


        def calculateResult(self, name):
            moves = self.myAgent.moves[name]
            payment_map = {'CC':1, 'DC':0,'CD':5,'DD':3}
            result = 0
            for i in range(0,len(moves['O'])):
                o = moves['O'][i]
                p = moves['P'][i]
                result += payment_map[o+p]
            return result

    class Tft:
        def __init__(self, agent):
            self.agent = agent

        def play(self, opponent):
            if opponent not in self.agent.moves:
                return 'C'
            elif len(self.agent.moves[opponent]['P']) == 0:
                return 'C'
            else:
                opponent_move = self.agent.moves[opponent]['P'][-1]
                return opponent_move

    class AllD:
        def play(self, opponent):
            return 'D'
```

```
class AllC:
    def play(self, opponent):
        return 'C'

class Rand:
    def play(self, opponent):
        return choice(['C','D'])

class Tester:
    def __init__(self, agent):
        self.agent = agent

    def play(self, opponent):
        if opponent not in self.agent.moves:
            return 'D'
        elif len(self.agent.moves[opponent]['P']) == 0:
            return 'D'
        elif self.agent.moves[opponent]['P'][0] == 'C':
            return self.agent.moves[opponent]['P'][-1]
        else:
            return choice(['C','D'])

class Joss:
    def __init__(self, agent):
        self.agent = agent

    def play(self, opponent):
        if opponent not in self.agent.moves:
            return 'C'
        elif len(self.agent.moves[opponent]['P']) == 0:
            return 'C'
        elif (len(self.agent.moves[opponent]['O']) + 1) % 5 == 0:
            return 'D'
        else:
            opponent_move = self.agent.moves[opponent]['P'][-1]
            return opponent_move

class Pavlov:
    def __init__(self, agent):
        self.agent = agent

    def play(self, opponent):
        if opponent not in self.agent.moves:
            return 'C'
        elif len(self.agent.moves[opponent]['P']) == 0:
            return 'C'
        else:
            payment_map = {'CC':1, 'DC':0,'CD':5,'DD':3}
            lastMove = self.agent.moves[opponent]['O'][-1]
            lastOpMove = self.agent.moves[opponent]['P'][-1]
            lastResult = payment_map[lastMove+lastOpMove]
            if lastResult == 0 or lastResult == 5:
                return 'D'
            if lastResult == 1 or lastResult == 3:
                return 'C'

def _setup(self):
    print self.getName()+": Started"
    self.moves = {}
    self.strategies = {"ALLD":self.AllD(), "TFT":self.Tft(self),
```

```
        "JOSS":self.Joss(self), "PAVLOV":self.Pavlov(self),
"TESTER":self.Tester(self)}
        self.result = 0
        self.counter = 0
        self.fileName = self.getName()

        template1 = ACLTemplate()
        template1.setOntology('Playing')
        messageTemplate1 = MessageTemplate(template1)
        self.addBehaviour(self.Receive(), messageTemplate1)

        template2 = ACLTemplate()
        template2.setOntology("New")
        messageTemplate2 = MessageTemplate(template2)
        self.addBehaviour(self.NewGame(), messageTemplate2)

        template3 = ACLTemplate()
        template3.setOntology("OpponentReady")
        messageTemplate3 = MessageTemplate(template3)
        self.addBehaviour(self.OpponentReady(), messageTemplate3)
```

AgentCreator.py

```
import sys, pprint
import xml.etree.ElementTree as ET
import sys, spade
import Prisoner
import HandlerAgent
from spade.Agent import Agent
from spade.ACLMessage import ACLMessage
from spade.Behaviour import EventBehaviour, ACLTemplate, MessageTemplate
from pprint import pprint
import web, json
from time import sleep
from copy import deepcopy

def createNeighbourhood(x,y, matrix):
    neighbours = ['' for i in range(4)]
    neighbours[0] = matrix[x][(y+1)%5]
    neighbours[1] = matrix[(x+1)%5][y]
    neighbours[2] = matrix[x][(y+5-1)%5]
    neighbours[3] = matrix[(x+5-1)%5][y]
    return neighbours

render = web.template.render('templates/')
oldStrategies = {}
strategyArchive = []
finished = False

class index:
    def GET(self):
        return render.agents()

class start:
    def POST(self):
        globals()["finished"] = False
        del strategyArchive[0:len(strategyArchive)]
        postData = web.input()
        handler = HandlerAgent.HandlerAgent("handler@127.0.0.1","tajna")
```

```
        handler.agents = postData
        for k in postData.keys():
            strategy = handler.agents[k]
            oldStrategies[k] = strategy
        handler.start()

        while finished == False:
            sleep(0.5)
        return json.dumps(strategyArchive)

class AgentCreator(Agent):

    class ReadyResponse(EventBehaviour):

        def _process(self):
            msg = self._receive()
            self.myAgent.agentCounter += 1

            if self.myAgent.agentCounter == len(self.myAgent.agents):
                msg = ACLMessage()
                msg.setContent("OpponentReady")
                msg.setOntology("OpponentReady")
                for k in self.myAgent.agents.keys():
                    address = k + "@127.0.0.1"
                    msg.addReceiver(spade.AID.aid(address,
["xmpp://"+address]))
                self.myAgent.send(msg)

    class StrategyEvaluation(EventBehaviour):
        def _process(self):
            msg = self._receive()
            print "primio rezultat: "+msg.getContent() + " od:
"+msg.getSender().getName()
            sender = msg.getSender().getName()
            senderName = sender.split('@')[0]

            self.myAgent.results[senderName] = int(msg.getContent())


            if len(self.myAgent.results) == len(self.myAgent.agents):
                for k in self.myAgent.results.keys():
                    minimum = self.myAgent.results[k]
                    minOpponent = k
                    for opponent in self.myAgent.agents[k]['neighbours']:
                        if self.myAgent.results[opponent] < minimum:
                            minimum = self.myAgent.results[opponent]
                            minOpponent = opponent

                    self.myAgent.newStrategies[k] =
oldStrategies[minOpponent]

                for k in oldStrategies.keys():
                    if oldStrategies[k] != self.myAgent.newStrategies[k]:
                        self.myAgent.lastIteration = False
                        break


                if not self.myAgent.lastIteration:
                    self.myAgent.lastIteration = True
                    self.myAgent.results = {}
                    self.myAgent.agentCounter = 0
```

```
                    strategyArchive.append(deepcopy(oldStrategies))
                    for k in self.myAgent.newStrategies.keys():
                        msg = ACLMessage()
                        msg.setOntology("New")
                        msg.setContent(self.myAgent.newStrategies[k])
                        address = k + "@127.0.0.1"
                        msg.addReceiver(spade.AID.aid(address,
["xmpp://"+address]))
                        oldStrategies[k] = self.myAgent.newStrategies[k]
                        self.myAgent.send(msg)
                else:
                    print "GOTOVA IGRA!! "
                    strategyArchive.append(self.myAgent.newStrategies)

                    self.myAgent.results = {}
                    self.myAgent.lastIteration = True
                    self.myAgent.agentCounter = 0
                    globals()["finished"] = True


    def _setup(self):
        template = ACLTemplate()
        template.setOntology('Result')
        messageTemplate = MessageTemplate(template)
        self.addBehaviour(self.StrategyEvaluation(), messageTemplate)

        template = ACLTemplate()
        template.setOntology('ReadyResponse')
        messageTemplate = MessageTemplate(template)
        self.addBehaviour(self.ReadyResponse(), messageTemplate)

        self.results = {}
        self.newStrategies = {}
        self.lastIteration = True
        self.agentCounter = 0


        for k in self.agents.keys():
            agent = agents[k]
            agent['neighbours'] = createNeighbourhood(agent['longitude'],
agent['latitude'], matrix)

            name = agent['name'] + '@127.0.0.1'
            p = Prisoner.Prisoner(name, 'tajna')
            p.neighbours = agent['neighbours']
            p.iterations = agent['iterations']
            p.start()

if __name__ == '__main__':
    fileName = sys.argv[2];

    tree = ET.parse(fileName)
    root = tree.getroot()

    matrix = [['' for i in range(5)] for j in range(5)]
    agents = {}

    game = root.find('game')
    iterations = game.find('iterations').text
    for child in root.findall('agent'):
        agent = {}
```

```
        agent['name'] = child.get('name')
        longitude = int(child.find('coordinates').find('longitude').text)
        latitude = int(child.find('coordinates').find('latitude').text)
        matrix[longitude][latitude] = child.get('name')
        agent['longitude'] = longitude
        agent['latitude'] = latitude
        agent['iterations'] = int(iterations)
        agents[agent["name"]] = agent

    agentCreator = AgentCreator('creator@127.0.0.1', 'secret')
    agentCreator.agents = agents
    agentCreator.start()

    urls = (
    '/', 'index','/start','start'
    )
    app = web.application(urls, globals())
    app.run()
```