SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD

Klasifikatorski sustavi s mogućnošću učenja

Martin Babić, 0035134608

Voditelj: Prof.dr.sc. Marin Golub

Zagreb, rujan 2012.

Sadržaj

1. Uvod

Prošlo je već više od tri desetljeća otkad je John Holland 1976. godine objavio svoj rad na temu **klasifikatorskih sustava s mogućnošću učenja** (eng. ***Learning Classifier Systems***, skraćeno: ***LCS*** – u daljnjem će se tekstu, zbog jednostavnosti, na većini mjesta koristiti ova skraćenica). Ovaj je rad po mnogočemu bio revolucionaran na području računalnih znanosti, a po nekim je pitanjima bio i ispred svog vremena. Ovoj tvrdnji u prilog govori i činjenica da niti danas još znanstvenici s polja računalstva i matematike nisu u potpunosti opisali matematički model ovakvih sustava.

Ovaj rad pokušat će u svom prvom dijelu dati čitatelju uvid u povijesni kontekst u kojem se rodila ideja LCS-a i opisati osnovne koncepte koji tvore jedan takav složeni sustav. Razumijevanje ovih koncepata, potrebno je, kako za shvaćanje rada LCS-a, tako i za razumijevanje ideje koja iza njih stoji, tj. cilja koji se želi postići razvijanjem ovih sustava.

U kasnijim će se poglavljima govoriti o povijesnom razvoju LCS-ova, prikazati će se najvažnije inačice koje su vremenom nastale na temeljnom konceptu te će se analizirati njihove značajke i skupovi problema za rješavanje kojih su pogodni.

2. Genetski algoritmi

2.1. Teorijska podloga: evolucija i genetika

2.1.1. Evolucija

Ideja o evoluciji, to jest ideja o postupnoj promjeni vrsta, stara je gotovo kao i naša civilizacija – poznata nam je još od vremena starih Grka, par stoljeća p. n. e. Ipak, tek su, uvjetno rečeno, nedavna dostignuća u znanosti doprinijela pomnijem shvaćanju evolucijskih mehanizama.

Prvo takvo dostignuće bilo je objavljivanje djela "*O podrijetlu vrsta*" Charlesa Darwina. Ova je knjiga, objavljena 1859. godine, bila prekretnica u poimanju evolucije. Ona je pružila prvi uvjerljiv mehanizam po kojem se odvijala evolucijska promjena – **teoriju prirodnog odabira**.

Teorija prirodnog odabira je proces kojim se u populaciji kroz vrijeme smanjuje ili povećava učestalost neke karakteristike. Ova se promjena događa ovisno o tome utječe li ta karakteristika pozitivno ili negativno na izglede za preživljavanje jedinki populacije u njihovom prirodnom okolišu. Ako karakteristika pridonosi vjerojatnosti duljeg preživljavanja, češće će se javljati kod jedinki i obrnuto – ako karakteristika nepovoljno utječe na preživljavanje, njena će se učestalost pojavljivanja smanjiti.

Darwinova teorija bila je revolucionarna, pa ipak ta teorija nije mogla objasniti nekoliko važnih značajki evolucijskog procesa. Darwin nije mogao objasniti izvor varijacija u osobinama unutar vrste, te nije mogao identificirati mehanizam koji je mogao prenositi osobine vjerno s jedne generacije na drugu [**1**].

2.1.2. Genetika

U to vrijeme, dakle sredinom 19. stoljeća, češki je znanstvenik Gregor Mendel, pronašao odgovor na pitanje o načinu prijenosa osobina sa generacije na generaciju. To mu je pošlo za rukom, eksperimentirajući s nasljednim osobinama biljaka[1]. Rezultate svoga rada Mendel je objavio u obliku znanstvenog članka pod nazivom "*Eksperimenti u hibridizaciji biljaka*", 1866. godine.

Na Mendelov rad u početku nije bilo odjeka i bilo je potrebno nekoliko desetljeća da relevantni znanstveni krugovi postanu svjesni ovog djela. Tek početkom 20. stoljeća, drugi su znanstvenici, baveći se istim problemom nasljeđivanja svojstava, pronašli Mendelov rad. Među njima bio je i engleski biolog William Bateson, koji je diskretne jedinice zaslužne za prijenos svojstava među generacijama prvi nazvao **genima**. Tako je nova znanstvena disciplina dobila i svoje ime – **genetika**.

Genetika se bavi proučavanjem molekularne strukture gena i njihove funkcije. Do danas su se razvili mnogi smjerovi unutar same genetike, što jasno ukazuje kako je materija

[1] Mendel je za svoje pokuse odabrao obični grašak – bilo ga je lako uzgojiti i imao je jasno izražena obilježja koja su se mogla pratiti iz generacije u generaciju.

kojom se bavi složena. Ipak, osnovna je ideja jednostavna i njeno poznavanje može biti od koristi za razumijevanje genetskih algoritama, pa ćemo se u nastavku osvrnuti na nju.

2.1.3. Osnovni biološki pojmovi

Sva su živa bića građena od stanica, a u svakoj se stanici nalazi isti skup **kromosoma** – niti DNA – koji služi kao nacrt organizma. Kromosom se koncepcijski može podijeliti u **gene**, od kojih svaki kodira neki protein. Ugrubo, možemo zamisliti da je u genu kodirana **osobina**, poput boje očiju. Različite moguće "postavke" osobine (npr. plava, zelena, smeđa) zovu se **alele**. Svaki je gen lociran u pojedinom **lokusu** (poziciji) na kromosomu. [**2**].

Kod naprednijih oblika života, pa tako i sisavaca, svaka alela ima dva gena – jedan od majke i jedan od oca. Gen koji prevladava zovemo **dominantnim**, dok gen koji je zatomljen nazivamo **recesivnim**.

Križanje ili **rekombinacija** je proces prilikom diobe stanice mejozom. Prilikom mejoze I, točnije u stadiju pahiten njene profaze[2], u njoj se odvija **razmjena genetičkog materijala između nesestrinskih kromatida homolognih kromosoma** [**3**] (eng. ***crossing-over***). Križanjem se dobivaju nove kombinacije gena, koje su mješavina obilježja obaju roditelja.

Mutacija je proces prilikom kojeg se u novoj generaciji slučajno[3] promijeni neka **nukleotida** (elementarna jedinica informacije) unutar lanca DNA, pa se samim time mijenjaju i svojstva potomaka u odnosu na njihove roditelje.

I križanje i mutacija pomažu raznovrsnosti obilježja među pripadnicima iste vrste.

2.2. Jednostavni genetski algoritam

Po načinu djelovanja ubrajaju se u metode **usmjerenog slučajnog pretraživanja prostora rješenja** (eng. ***guided random search techniques***) u potrazi za globalnim optimumom. [**4**]

Genetski algoritam djeluje nad skupinom, tj. populacijom potencijalnih rješenja, na koje primjenjuje genetske operatore, kako bi iznjedrio novu generaciju. Prilikom odabira jedinki koje će donirati svoj genetski materijal novom naraštaju potencijalnih rješenja, koristi se neki mehanizam selekcije, po uzoru na prirodnu selekciju. To osigurava s vremenom sve *bolju* populaciju potencijalnih rješenja, odnosno populaciju koja sve *bolje aproksimira* ***ciljnu funkciju*** *u njenom optimumu*. Ciljna funkcija, funkcija cilja ili funkcija dobrote (eng. ***fitness function***) je funkcija kojom je određena dobrota pojedinog potencijalnog rješenja. Proces pretraživanja se nastavlja dok se ne zadovolji neki zadani uvjet pretrage prostora rješenja (Slika 2.1). Najbolja jedinka, tj. jedinka koja u posljednjem naraštaju ima najveću dobrotu, smatra se rješenjem pretrage.

[2] Mejoza I se sastoji od sljedećih faza: profaza, metafaza, anafaza, telofaza. Profaza se dalje dijeli na leptoten, zigoten, pahiten, diploten, dijakineza.
[3] Vjerojatnost pojave ovog procesa ovisi o raznim faktorima, većinom uvjetovanih okolišem.

Rješenja se u genetskom algoritmu mogu reprezentirati na mnoge načine, a za jednostavne je primjene dovoljan binarni prikaz.

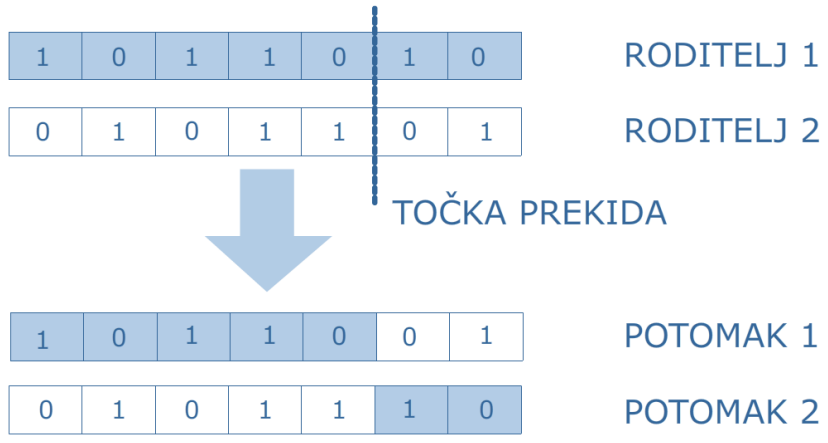
```
genetski_algoritam(n, pₖ, pₘ, M) {
        stvori_inicijalnu_populaciju(n);
        dok(nisu_zadovoljeni_uvjeti_pretrage) {
                odaberi_roditelje();      // selekcija
                križaj_roditelje(pₖ);     // rekombinacija
                mutiraj_djecu(pₘ);        // mutacija
                dodaj_djecu_u_populaciju(M);
        }
}
```

Slika 2.1: Pseudo-kôd genetskog algoritma [5]. Parametri algoritma su redom: n – veličina populacije; p_k – vjerojatnost križanja na pojedinom bitu; p_m – vjerojatnost mutacije; M – broj djece koja se dodaju u populaciju.

2.2.1. Genetski operatori

Genetski algoritam koristi dva različita mehanizma promjene potencijalnih rješenja, tj. dva načina prolaska kroz prostor potencijalnih rješenja: križanje i mutaciju. Kako oni djeluju, pokazat ćemo u nastavku na primjeru rješenja kodiranog binarnim zapisom.

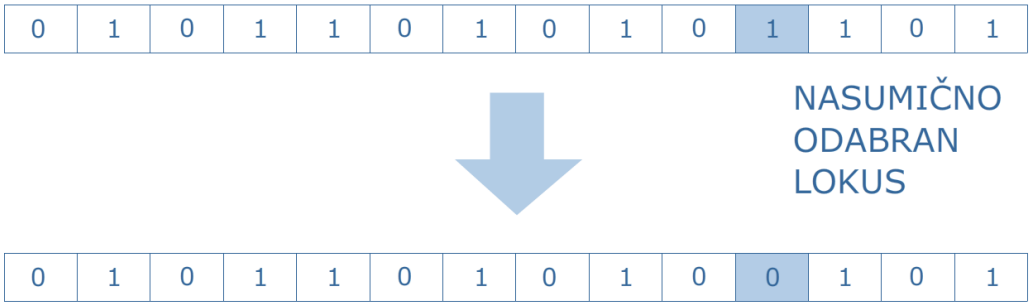
Nakon odabira roditelja koji će sudjelovati u stvaranju nove generacije rješenja, genetski algoritam prelazi na križanje roditelja, odnosno njihovog genetskog materijala. Najjednostavniji primjer ove metode jest križanje sa jednim prekidom. Ovaj se prekid može odabrati slučajno ili može biti predodređen nekom funkcijom ili parametrom. Obično je riječ o parametru koji označavamo s [$\boldsymbol{p_k}$], a predstavljen je brojem bita iza kojeg se radi prekid. Križana će djeca svoje značajke dobiti od jednog roditelja sa jedne strane prekida, a od drugog roditelja sa druge strane prekida (slika 2.2).



Slika 2.2: Križanje ili rekombinacija na primjeru rješenja prezentiranih binarnim brojem.

Nakon što je križanjem stvorena nova generacija potencijalnih rješenja, potrebno je još provesti mutaciju.

Mutacija se može zamisliti kao slučajna promjena jednog bita kod pripadnika nove generacije potencijalnih rješenja. Mutacijom se utječe na raznovrsnost jedinki u novom naraštaju. Parametar koji određuje vjerojatnost mutacije [p_m] jednog bita je ujedno i parametar algoritma. Ako vjerojatnost mutacije teži k jedinici, tada se algoritam pretvara u algoritam slučajne pretrage prostora rješenja. S druge strane, ako vjerojatnost mutacije teži k nuli, postupak će najvjerojatnije već u početku procesa optimiranja stati u nekom lokalnom optimumu [**4**].



Slika 2.3: Mutacija na primjeru rješenja prezentiranih binarnim brojem

Iako je optimiranje, tj. ugađanje parametra genetskog algoritma složen posao, koji zahtjeva puno eksperimentiranja, ovi su se algoritmi pokazali dobrima za pretraživanje velikih prostora rješenja u potrazi za optimumom.

3. Učenje potkrepljenjem

3.1. Teorijska podloga: biheviorizam

Biheviorizam jest psihologijska teorija, koja polazi od pretpostavke da se cjelokupna psihološka struktura pojedinca (objekta promatranja) može opisati kroz njegovo ponašanje.

Jedan od važnijih pojmova, kojima biheviorizam opisuje učenje ponašanja, jest **uvjetovanje**. Klasično uvjetovanje jest način učenja koji je otkrio ruski fiziolog Ivan Pavlov. Riječ je o učenju obrazaca ponašanja na temelju refleksnih podražaja koje asociramo uz određeni kontekst (određene **uvjete**). Pavlov je primijetio da se obrazac ponašanja ponavlja čak i kad nema primarnog podražaja – ali je prisutan dati kontekst. Operantno (ili instrumentalno) je uvjetovanje nadgradnja Pavlovljevog osnovnog koncepta, koju je razradio američki psiholog Burrhus Frederic Skinner. Riječ je o načinu na koji objekt promatranja uči svoje obrasce ponašanja kroz interakciju sa svojim okolišem. Objekt promatranja usvaja određene obrasce ovisno o posljedicama koje ta interakcija s okolišem utječe na njega. Ako se na neku akciju objekta pojavi neka (željena) reakcija od strane okoliša, veća je vjerojatnost da će objekt idući puta u istom okolišu (odnosno, pod istim uvjetima) ponoviti spomenutu akciju (pozitivno potkrepljenje). Također, ako (željena) reakcija izostane, vjerojatnost ponavljanja akcije se smanjuje (negativno potkrepljenje[4]). Ovakav se tip učenja obrazaca ponašanja naziva **učenjem potkrepljenjem** (eng. ***Reinforcement Learning***, skraćeno: ***RL***).

Iako se o bihevioralnoj teoriji godinama pisalo i s odobravanjem i sa skepsom (postoje mnogi istaknuti pojedinci koji se nisu libili kritizirati[5] biheviorizam), ona je nadahnula jedan od uspješnijih oblika učenja ponašanja agenata u računalstvu.

3.2. Učenje potkrepljenjem kao metoda strojnog učenja

Učenje potkrepljenjem kao metoda strojnog učenja, kakvu je danas znamo, formirala se dosta kasno – tek krajem osamdesetih godina prošlog stoljeća, no njeni korijeni sežu u dalju prošlost, do sredine dvadesetog stoljeća, baš u vrijeme kad je biheviorizam bio u punom zamahu. Stoga nije čudno da je ova, u to vrijeme vrlo popularna filozofija, inspirirala znanstvenike da neke od njenih koncepata pokušaju preslikati i na računalnu znanost.

[4] Negativno se potkrepljenje ne bi trebalo miješati sa (pozitivnom) kaznom. Riječ je o dva različita načina uvjetovanja, pa tako i kazna može biti a) pozitivna (kažnjavanje objekta, ako učini neku akciju, čime se vjerojatnost ponavljanja te akcije smanjuje) i b) negativna (kažnjavanje objekta izostaje - povećavamo vjerojatnost da objekt pod istim uvjetima akciju ponovi)

[5] Npr. Anthony Burgess u svom slavnom djelu *'Paklena naranča'* kritizira uvjetovanje kao nemoralnu metodu za utjecanje na ponašanje pojedinca. Također, kritiku upućuje na račun kontroverznog poimanja slobodne volje od strane biheviorista. Naime, biheviorizam, pogotovo *Radikalni biheviorizam* B. F. Skinnera, sve misli i osjećaje tumači kao još jedan oblik ponašanja koje slijedi ista pravila po kojima se ravnaju i ostala, 'vanjska' ponašanja – dakle, gleda ih kao na odgovor na podražaje od strane okoliša, time umanjujući ulogu slobodne volje pojedinca.

3.2.1. Korijeni u ranom računalstvu

Sredinom dvadesetog stoljeća, računalstvo je bilo prilično mlada znanost i računala su bila usko specijalizirani strojevi, s kojima je radila šačica najsposobnijih znanstvenika tog vremena, uglavnom s područja matematike. Govorimo o vremenu kad se tranzistor tek pojavio, kao revolucionarno otkriće, a tehnologija poluvodičkih sklopova još je uvijek bila u povojima. Računala tog vremena imala su sposobnost obraditi od 1,905 (UNIVAC I – 1951.) do 75,000 (AN/FSQ-7 – 1958.) osnovnih informacija u sekundi. Ipak, i takvi, iz današnje perspektive, skromni[6] strojevi, nadahnjivali su ljude da pomoću njih pokušaju riješiti sve složenije probleme.

U takvom okruženju, jedan od problema, kojem se posvetio američki matematičar Richard Bellman, bio je **problem optimalne kontrole**[7]. Ovaj se problem svodi na pronalaženje **strategije odlučivanja** (eng. ***control policy***) takve da se **optimira** (po nekoj osnovi) ponašanje nekog **dinamičkog sustava** u vremenu [**6**].

Dinamički je sustav sustav gibanja nekog entiteta (npr. tijelo, populacija, dionice) u nekom prostoru (npr. fizički prostor, eko-sustav, tržište) kroz vrijeme. Matematički, ovakav se sustav opisuje sustavom diferencijalnih jednadžbi, koji može, ali ne mora, bit analitički rješiv. Ove diferencijalne jednadžbe opisuju promjene sustava iz jednog stanja u neko drugo stanje ovisno o vremenu. Sustav je deterministički, ako iz istog početnog stanja promatranja, za određeno vrijeme, sustav prijeđe u uvijek isto buduće stanje ili je stohastički, ako iz istog početnog stanja promatranja, za određeno vrijeme, sustav može prijeći u više različitih stanja. U ovom drugom slučaju, obično postoji funkcija razdiobe vjerojatnosti prelaska iz sadašnjeg stanja u potencijalna buduća stanja.

Bellman je, proučavajući problem optimalne kontrole, uveo i posebnu klasu diskretnih, stohastičkih dinamičkih sustava, poznatih i kao **Markovljevi procesi odlučivanja** (eng. ***Markov Decision Process***; skraćeno ***MDP***). Pokazalo se da ovakvi dinamički sustavi daju pogodan okvir za rješavanje problema optimalne kontrole pomoću računala.

Bellman je također osmislio sustav računalnih metoda za rješavanje problema optimalne kontrole, koje se zajedno nazivaju **dinamičkim programiranjem**. Dinamičko se programiranje, pojednostavljeno rečeno, svodi na raščlanjivanje problema na sve manje dijelove, kako bi se smanjila njegova složenost. U slučaju problema optimalne kontrole – riječ je o raščlanjivanju problema pronalaska optimalne strategije odlučivanja (optimalni put od početnog stanja s_0 do nekog ciljnog stanja s_n), na manje korake (gleda se odlučivanje za prelazak u svako sljedeće stanje i traži ono optimalno), kako bi se došlo do optimalne strategije odlučivanja.

Iako je dinamičko programiranje danas zasebna disciplina u računalstvu i obuhvaća puno širu primjenu od one koja je ovdje spomenuta, pokazalo se da je to i jedan od osnovnih

[6] Današnja moderna stolna računala imaju sposobnost izvršiti oko 150 x 10^9 operacija po sekundi (npr. Intel Core i7 Extreme Edition i980EE)

[7] Kao i u mnogim drugim znanstvenim disciplinama toga vremena, Amerikanci nisu bili usamljeni u pokušajima rješavanja ovog problema. S druge strane željezne zavjese, u tadašnjem SSSR-u, istim se problemom bavio i ruski matematičar Lav Semjenović Pontrjagin, koji je, podjednako kao i R. Bellman, pridonio njegovu rješavanju.

principa na kojima je kasnije nastala teorija učenja potkrepljenjem, kakvom ju danas poznajemo.

3.2.2. Formalna definicija učenja potkrepljenjem

Kao što je već rečeno, problematiku učenja potkrepljenjem moguće je prikazati Markovljevim procesom odlučivanja. MDP možemo formalno definirati sljedećim elementima:

- S – konačan skup svih mogućih stanja okoliša
- A - konačan skup mogućih akcija agenta
- P – funkcija prijelaza definirana kao:

$$P : S \times A \to \Pi(S) \tag{2.1}$$

  gdje funkcija $\Pi(S)$ označava skup raspodjela vjerojatnosti nad S. Pojedina raspodjela vjerojatnosti $Pr(s_{t+1} \mid s_t, a_t)$ nam kazuje vjerojatnosti da sustav prijeđe u različita moguća stanja s_{t+1}, ako agent izvede akciju a_t u stanju s_t.

- R – funkcija nagrade definirana kao:

$$R : S \times A \to \mathbb{R} \tag{2.2}$$

  koja nam daje nagradu u obliku skalara za svaki par stanje-akcija.

Agent djeluje na okoliš u diskretnim vremenskim razmacima $t = 0, 1, 2, 3, \ldots$. U nekom od tih trenutaka t_n, agent kao ulaznu informaciju prima stanje s_n:

$$s_n \in S \tag{2.3}$$

gdje je S skup svih mogućih stanja okoliša. Na temelju stanja s_n, agent odabire akciju a_n:

$$a_n \in A(s_n) \tag{2.4}$$

gdje je $A(s_n)$ skup akcija primjenjivih u s_n. Radi jednostavnosti, u većini će slučajeva skup akcija primijenjivih u s_n biti jednak za sve $s_n \in S$.

Jedan vremenski korak unaprijed, u vremenu t_{n+1}, agent dobiva nagradu r_{n+1}:

$$r_{n+1} = R(s_n, a_n) \in \mathbb{R} \tag{2.5}$$

i nađe se u novom stanju s_{n+1}. Iako MDP daje dobru podlogu za razumijevanje učenja potkrepljenjem, ova nam formalna definicija ne govori ništa o ponašanju agenta.

Ponašanje agenta definirano je zapisom o vjerojatnosti poduzimanja svake akcije u svakom stanju. Taj se zapis može prikazati funkcijom koja se naziva strategijom odlučivanja ili kraće samo strategijom π. Npr. $\pi(a_x, s_y)$ jest vjerojatnost da agent izvrši

akciju a_x, kao odgovor na stanje okoliša s_y. Ova se funkcija svakim vremenskim korakom nanovo izračunava na osnovu novih nagrada iz okoliša.

Vrijednosna funkcija strategije π, koju označavamo sa $V^{\pi}(s_t)$ jest funkcija vrijednosti nekog stanja s_t, za danu strategiju π. Vrijednost stanja s_t je jednaka količini nagrade očekivane iz tog stanja (uz primjenu strategije π) zbrojenu sa vrijednostima svih budućih stanja kroz koja će agent proći, ako se primijenjuje strategija π:

$$V^{\pi}(s_t) = \max_{s_{t+1}} \{r_{t+1} + \beta \cdot V^{\pi}(s_{t+1})\} \tag{2.6}$$

gdje je r_{t+1} nagrada izravno dobivena prelaskom iz stanja s_t u stanje s_{t+1}, a β je neki faktor kojim se regulira utjecaj nagrada iz budućih stanja na ovo stanje ($0 \leq \beta < 1$). Izraz 2.6 poznat je kao **Bellmanova jednadžba**. Riječ je o rekurzivnoj funkcijskoj jednadžbi. Rješenja ovakve jednadžbe mogu se tražiti iterativnim postupkom ili metodama dinamičkog programiranja, čijem je razvoju, kao što je već prije spomenuto, također pridonio Bellman.

Strategija π je bolja od neke druge strategije π' (pišemo: $\pi > \pi'$) ako i samo ako je:

$$V^{\pi}(s) > V^{\pi'}(s) \tag{2.7}$$

za svaki $s_n \in S$.

Postoji uvijek jedna strategija koja je bolja ili jednaka bilo kojoj drugoj. Takvu strategiju nazivamo **optimalnom strategijom π^*,** a njena vrijednosna funkcija dana je izrazom:

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{2.8}$$

za svaki $s_n \in S$.

Možemo reći da je cilj učenja potkrepljenjem naučiti optimalnu strategiju, tj riješiti rekurzivnu funkcijsku Bellmanovu jednadžbu za V^*.

3.3. Nadgledano učenje

Metode učenja u računalstvu pokrivaju široko područje, u primjeni i u teoriji. Javile su se razne podjele, a jedna od najčešće spominjanih jest podjela algoritama i metoda na **nadgledano** ili **nadzirano učenje** (eng. ***Supervised Learning***; skraćeno: ***SL***) i **nenadgledano** ili **nenadzirano učenje** (eng. ***Un-supervised Learning***; skraćeno: ***UL***).

Razlika između ova dva pristupa leži u načinu na koji se prezentiraju ulazni podaci za učenje. Dok su kod nadgledanog učenja podaci predstavljeni u formi "ulaz:izlaz", podaci kod nenadgledanog učenja dolaze u formi "ulaz". Ovo ima nekoliko posljedica:

- potrebno je uložiti više truda za pripremanje podataka kod nadgledanog učenja, a kako izlaz mora biti definiran, mora se definirati i neki konačan skup prihvatljivih izlaza
- posljedica ovog je ograničen broj klasa koje agent zasnovan na nadgledanom učenju može raspoznati, dok ih agent zasnovan na nenadgledanom učenju može raspoznati beskonačan broj. Dok su agenti zasnovani na nadgledanom učenju pogodni za klasifikacijske probleme s poznatim brojem klasa, agenti zasnovani na nenadgledanom učenju pogodni su za probleme **klasteriranja podataka** (eng. ***data clustering***).
- ako promjenjivost okoliša zamislimo kao vjerojatnost da se u njemu pojave nove klase podataka, može se zaključiti kako su agenti zasnovani na nadgledanom učenju pogodniji za nepromjenjive okoliše, a agenti zasnovani na nenadgledanom učenju pogodniji za promjenjive okoliše.

Danas je podjela na ove dvije skupine pomalo zastarjela, jer su se u međuvremenu pojavili mnogi novi pristupi i koncepti, koji se nalaze negdje između ove dvije krajnosti, kao što je npr. **polu-nadgledano učenje** (eng. ***semi-supervised Learning***).

3.3.1. Odnos učenja potkrepljenjem i nadgledanog učenja

Po iznesenim razlikama između ova dva tipa učenja, jasno je da je učenje potkrepljenjem vrsta nenadgledanog učenja. Ipak, kroz povijest ova razlika nije bila toliko očita, pa se znalo dogoditi da se pomiješaju koncept učenja kroz nagradu i kaznu, što je koncept preuzet od učenja potkrepljenjem, sa konceptom učenja na podacima tipa "ulaz:izlaz" [**6**].

4. Klasifikatorski sustavi s mogućnošću učenja

4.1. Uvod u klasifikatorske sustav s mogućnošću učenja

Klasifikatorske sustave s mogućnošću učenja možemo zamisliti kao **prilagodljive** agente, koji kroz interakciju sa svojim okolišem izvršavaju neki zadatak. Interakcija se odvija u koracima, a svaki od tih koraka možemo inicijalno podijeliti na tri dijela, koji se ciklički izmjenjuju:

- LCS kroz ulaznu jedinicu, **detektor [*d*]**, dobiva informaciju o stanju **okoliša [*O*]** **(**Slika 4.1**)**.
- Ovisno o stanju okoliša, LCS odabire akciju kojom će utjecati na okoliš. Ova se akcija nad okolišem izvršava kroz **efektor [*e*]**. Obično se akcijom postiže promjena stanja okoliša.
- Sustav dobiva rezultat (nagradu) za svoju akciju primijenjenu na stanje okoliša.

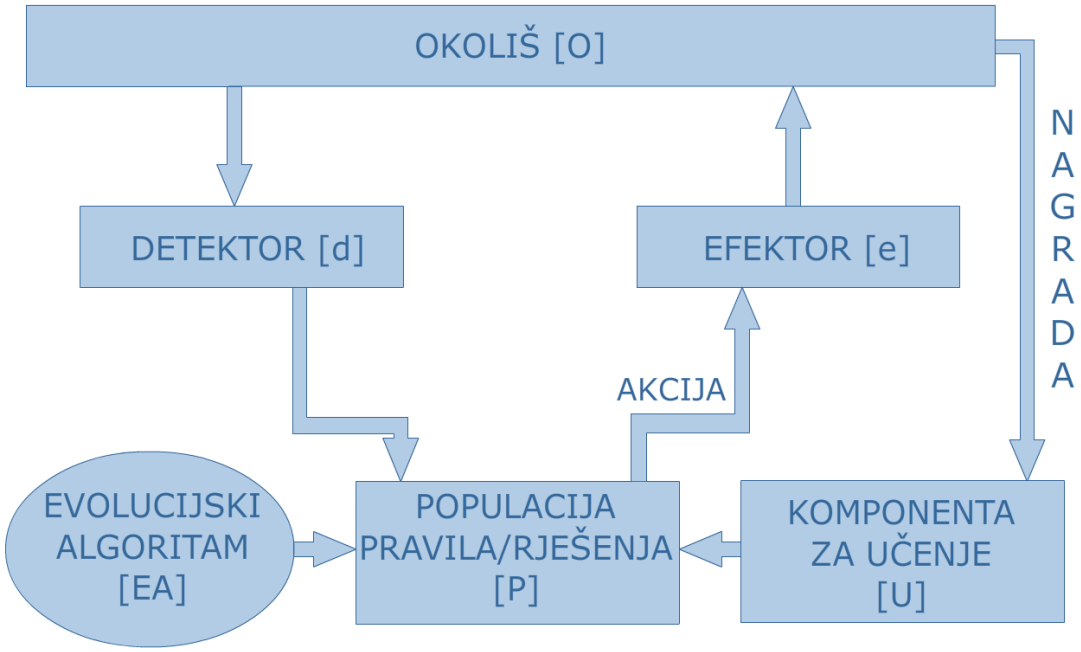
Prilagodljivost možemo definirati kao sposobnost sustava (agenta) da obavlja posao za koji je namijenjen čak i u novim, nepoznatim ili neočekivanim okolnostima. LCS se novim stanjima okoliša prilagođava postupkom pokušaja i pogreške. Naime, kad se sustav nađe u novom stanju, za koje ne zna kojom akcijom da odgovori, odabire se nasumična akcija te se promatraju rezultati. Na temelju rezultata, sustav uči koja je akcija najprimjerenija za dano stanje okoliša.

Znanje o ovoj uzročno-posljedičnoj vezi je unutar LCS-ova reprezentirano **populacijom pravila ili klasifikatora**[8] **[*P*]**. Ova su pravila najčešće reprezentirana kao kombinacija nekog stanja okoliša i neke akcije, koju LCS treba izvršiti kao odgovor na to stanje, a mogu se interpretirati kao "AKO stanje ONDA akcija".

Svakom se pravilu pridjeljuje skalar, koji označava **vrijednost** tog pravila. Vrjednija će se pravila češće koristiti, dok će ona manje vrijedna biti zapostavljena. Pridjeljivanje ovog skalara pravilima obavlja **komponenta za učenje [*U*]**. Komponenta za učenje se odabire ovisno o tome na koji će se način nagrada raspoređivati među pravilima te otkud nagrada uopće dolazi (dolazi li direktno od okoliša ili ju dodjeljuje učitelj sustava).

Na kraju, na populaciju pravila djeluje **evolucijski algoritam [*EA*]**, koji je odgovoran za pretraživanje prostora pravila, u traženju novih, još boljih pravila. Evolucijski algoritam pridonosi prilagodljivosti na način da ubrzava proces prilagođavanja pospješujući pronalazak optimalnog pravila za neko stanje sustava.

[8] U nekim je slučajevima riječ o populaciji skupova pravila, o čemu će više riječi biti u poglavljima koja slijede.



Slika 4.1: Shema općenitog modela LCS-a

4.2. Povijesni razvoj

Povijesni razvoj LCS-ova započeo je 1976. godine. Te je godine John Holland predstavio formalni koncept klasifikatorskih sustava s mogućnošću učenja, koji su kao evolucijsku komponentu koristili genetske algoritme, koje je Holland predstavio godinu ranije. Nekoliko godina kasnije, točnije 1978. godine, u suradnji sa Judith Reitman, predstavio je prvu implementaciju ovakvog sustava. Nakon prve implementacije, Holland je par godina proveo revidirajući svoju zamisao, da bi 1980. godine predstavio potpuni koncept LCS-a. Ovo je inspiriralo eksperimente koji su iznjedrili novi stil LCS-a, no njega ćemo detaljnije opisati u poglavlju o podjelama LCS-ova.

4.2.1. Hollandov LCS

Hollandov LCS prima podatak iz okoliša kroz detektor [*d*]. Detektor je zadužen da stanje okoliša reprezentira nizom binarnih znamenki. Ovaj binarni broj opisuje stanje u kojem se okoliš nalazi, a LCS ga lokalno pohranjuje u internoj radnoj memoriji, koja se može zamisliti kao slijedna **lista poruka [*L*]** (Slika 4.2). Osim stanja okoliša, lista poruka sadržava i neke druge informacije, pa se lista poruka [*L*] može podijeliti na tri elementa:

- memorija za stanje okoliša,
- memorija za interne poruke,
- memorija u kojoj je pohranjena zadnja izvedena akcija.

Baza pravila se u Hollandovom LCS-u sastoji od populacije *N* uzročno-posljedičnih pravila, tj. klasifikatora oblika "AKO stanje ONDA akcija". Ova se pravila sastoje od dvije logičke komponente (stanje i akcija), a kodirana su ternarnim alfabetom {0, 1, #}. Znak #

služi kao znak da određeni bit pravila nije bitan, pa npr. stanje "1#1" odgovara vrijednostima "101" i "111". Također, ako se ovaj znak nalazi na istom lokusu ulaza (stanja) i izlaza (akcije), on dopušta preslikavanje ulazne vrijednosti na izlaznu. Npr. za ulaz "101", pravilo "AKO 1#1 ONDA 0#0" će za izlaz dati vrijednost "000" [**7**]. Također, svakom je pravilu pridijeljen skalar dobrote, koji opisuje korisnost tog pravila u dolasku do nagrade. Ovo se razlikuje u odnosu na Hollandovu prvotnu ideju[9], koja je bila znatno složenija.

Po dolasku vanjske poruke, stanja okoliša reprezentiranog binarnim brojem, u listu poruka, pretražuje se populacija pravila u potrazi za onim pravilima čiji ulazni dio odgovara vanjskoj poruci ili bilo kojoj drugoj poruci u internoj listi poruka. Sva pravila koja zadovoljavaju kriterij, smještaju se u **podudarni skup [*M*]** (eng**. *match set***). Među pravilima iz podudarnog skupa [*M*] **aukcijom** se bira pravilo koje će dati akciju. Akcija se stavlja u memoriju za akciju, a memorija stanja okoliša se oslobađa. Akcija se putem efektora prenosi okolišu i na temelju toga se percipira neki odgovor okoliša u vidu nagrade. Ova se nagrada distribuira pravilu koje je izvršilo akciju i među pravilima koja čine **skup prošlih akcija [*A*]$_{-1}$**. Na taj se način tim pravilima korisnost povećava, iako nisu neposredno zaslužna za dolazak do nagrade – riječ je o jednostavnom **algoritmu lančanog prijenosa** (eng. ***Bucket-Brigade Algorithm***; skraćeno: ***BBA***).

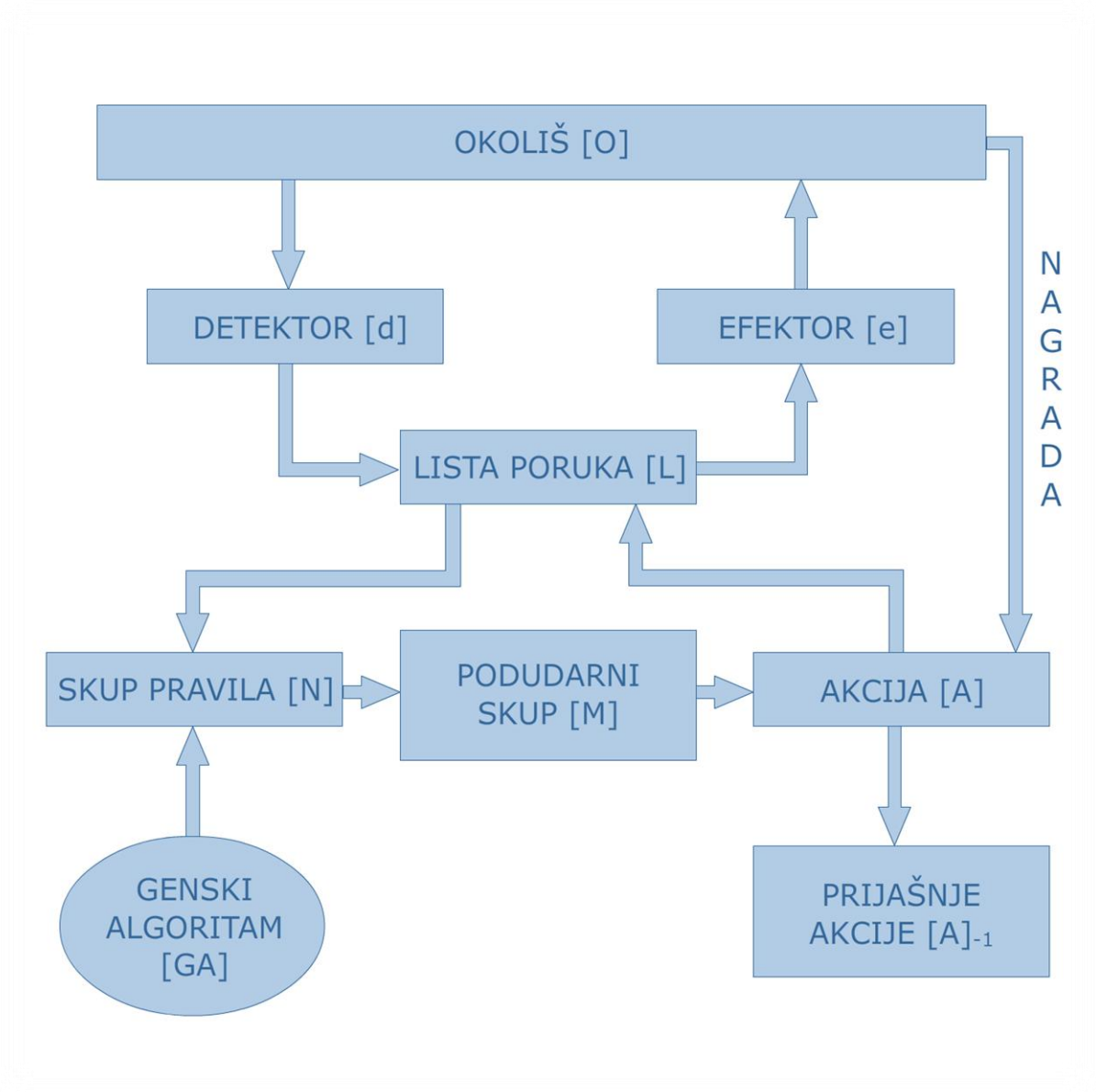
Pravila se na aukciji natječu sa licitacijama sastavljenima od dvije komponente: njihove dobrote i njihove specifičnosti. Specifičnost pravila se izražava kao udio bitova koji nisu predstavljeni znakom *#*. Dodana je još jedna konstanta β, koja je obično značajno manja od 1. Tako možemo reći da se ponuda pravila *C* iz skupa [*M*] u vremenu *t*, može izraziti kao:

$$Licitacija(C,t) = \beta \cdot specifičnost(C) \cdot dobrota(C,t) \qquad (3.1)$$

Nakon ovoga koraka, u sustavu se pokreće genetski algoritam, koji na populaciji pravila obavlja optimizaciju. Najčešće se u tu svrhu koristi eliminacijski genetski algoritam (eng. ***steady-state genetic algorithm***). Genetski algoritam odabire dva roditelja na osnovu njihove dobrote. Roditelji se kopiraju, a na tim se kopijama onda primjenjuju genetski operatori križanja i mutacije. Vjerojatnost za mutaciju po lokusu je reda veličine 10^{-2}. Prilikom mutacije bit mijenja svoju vrijednost u jednu od druge dvije vrijednosti sa jednakom vjerojatnošću. Križanje se odvija sa jednom točkom prekida, koja se odabire nasumično.

Važno je primijetiti da se optimizacijom genetskim algoritmom pokušava doći do skupa kooperativnih pravila, koja zajedno daju optimalno rješenje (ili ponašanje) sustava. Riječ je o različitom pristupu u odnosu na klasične genetičke algoritme, gdje se traži jedan član populacije kao optimalno rješenje.

[9] U originalnoj je implementaciji dobrota pravila bila računata na temelju točnosti pravila u procjeni koliku će nagradu dobiti iz okoliša.



Slika 4.2: Shematski prikaz Hollandovog LCS-a.

4.2.2. Klasifikatorski sustav nulte razine

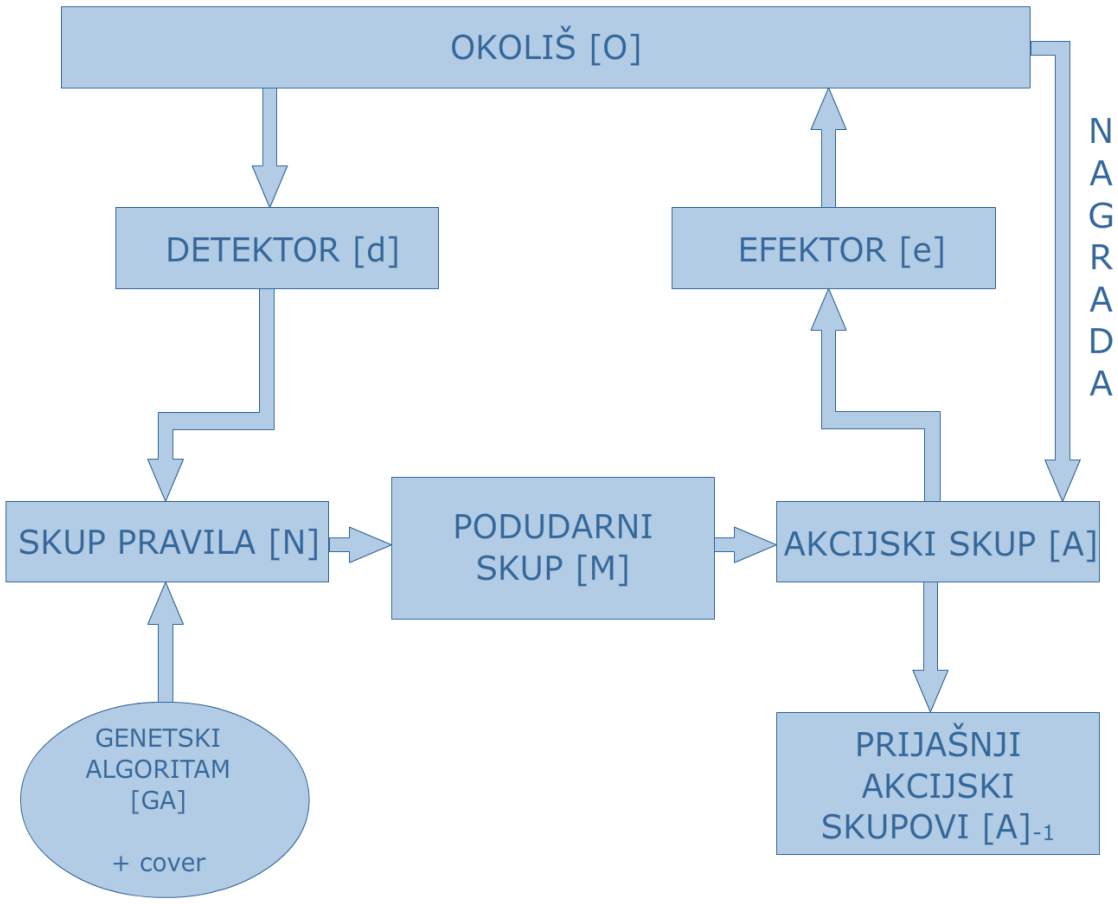
Potpuni koncept LCS-a, kakvog ga je predstavio Holland, bio je presložen i ubrzo je zanimanje za LCS-ima splasnulo. Tek je 1994. godine Stewart W. Wilson predstavio **klasifikatorski sustav nulte razine** (eng. ***"Zeroth-level" Classifier System***; skraćeno: ***ZCS***). Ovaj je koncept zadržao većinu Hollandovih ideja, al je malo pojednostavnio cijeli model, što je pridonijelo razumljivosti ideje i boljim performansama implementiranih rješenja.

Konkretno, Wilson je uklonio listu poruka i licitacije pravila, kao što prikazuje (Slika 4.3). Također, Wilson nije dozvolio da se koristi znak zamjene (*#*) u akcijama. Promijenjen je i način odabira akcije – više se nije biralo jedno pravilo, već **akcijski skup [*A*]** pravila.

Prilikom svog rada, ZCS, kao i Hollandov LCS, prima stanje okoliša [*O*] kroz detektor [*d*], koji ga predaje **skupu pravila [*N*]**. Pravila iz skupa pravila [*N*], čiji se uvjetni dio poklapa

sa stanjem dobivenim iz okoliša, premještaju se u podudarni skup [*M*]. Pravila iz podudarnog skupa [*M*] razvrstavaju se u skupine tako da pravila s istom akcijom pripadaju istoj skupini. Na temelju korisnosti svih pravila pojedine skupine predviđaju se nagrade pojedinih akcija i izabire se akcija. Pravila iz skupine iz koje je odabrana akcija se smještaju u **akcijski skup [*A*]** (eng**. *action set***). Akcija se putem efektora prenosi okolišu i na temelju toga se percipira neki odgovor okoliša u vidu nagrade. Ova se nagrada distribuira među pravilima koja čine skup [*A*] u ovom vremenskom koraku i pravilima koja su činila skup [*A*] u prošlim koracima $[A]_{-1}$. Na taj se način tim pravilima korisnost povećava. Također, pravilima koja su ušla u skup [*M*], a nisu prebačena u skup [*A*], korisnost se smanjuje.

ZCS koristi dva mehanizma pretraživanja. Jedan je mehanizam genetski algoritam, dok je drugi operator prekrivanja (eng. *covering operator*). U svakom ciklusu rada ZCS-a, postoji vjerojatnost *p* da se primijeni optimizacija genetskim algoritmom[10].



Slika 4.3: Shematski prikaz Wilsonovog ZCS-a.

Genetski algoritam odabire, na temelju njihove dobrote, dvije jedinke populacije [*N*]. Dva potomka se zatim stvore primjenom genetskih operatora križanja i mutacije i

[10] Za razliku od Hollandovog LCS-a, gdje se genetski algoritam pokretao u svakoj iteraciji.

svakom od potomaka roditelji doniraju pola svoje dobrote. Tako inicijalizirani potomci zauzimaju svoje mjesto u populaciji, istiskujući dva pravila odabrana po kriteriju manjka dobrote.

Operator prekrivanja je heuristička metoda koja je zadužena za produkciju novog pravila, kojem je uvjetni dio jednak trenutnom stanju okoliša, a akcija je odabrana nasumično. Ovaj se algoritam koristi kad skup [M] ne sadrži dovoljno dobra rješenja ili kad je prazan (ne postoji ni jedno pravilo u populaciji [N], čija se uvjetna komponenta poklapa sa stanjem okoliša).

4.2.3. Prošireni LCS

Wilson nije stao na ZCS-u, već je godinu dana kasnije, 1995., predstavio još jednu verziju klasifikatorskih sustava s mogućnošću učenja, poznatu kao **"prošireni" LCS-ovi** (eng. ***eXtended Classifier System***; skraćeno: ***XCS***). XCS predstavlja značajan korak u razvoju LCS-ova, jer po prvi puta uvodi nov način evaluacije pravila. Pravila se u ovom modelu LCS-a evaluiraju na osnovu preciznosti. U tu se svrhu uvode dva nova parametra:

- p – preciznost predviđanja
- ε – greška pri predviđanju

Tako se sada učenje svodi na ažuriranje triju parametara, što daje preciznije preslikavanje skupa stanja okoline na skup pravila [**5**]. Ovaj proces učenja se može raščlaniti na pet koraka:

1. Pogreška svakog pravila se ažurira formulom:

$$\varepsilon_j = \varepsilon_j + \beta \cdot \left(\left|P - p_j\right| - \varepsilon_j\right) \tag{3.2}$$

   gdje je P nagrada iz okoliša.
2. Preciznost predviđanja svakog pravila se ažurira formulom:

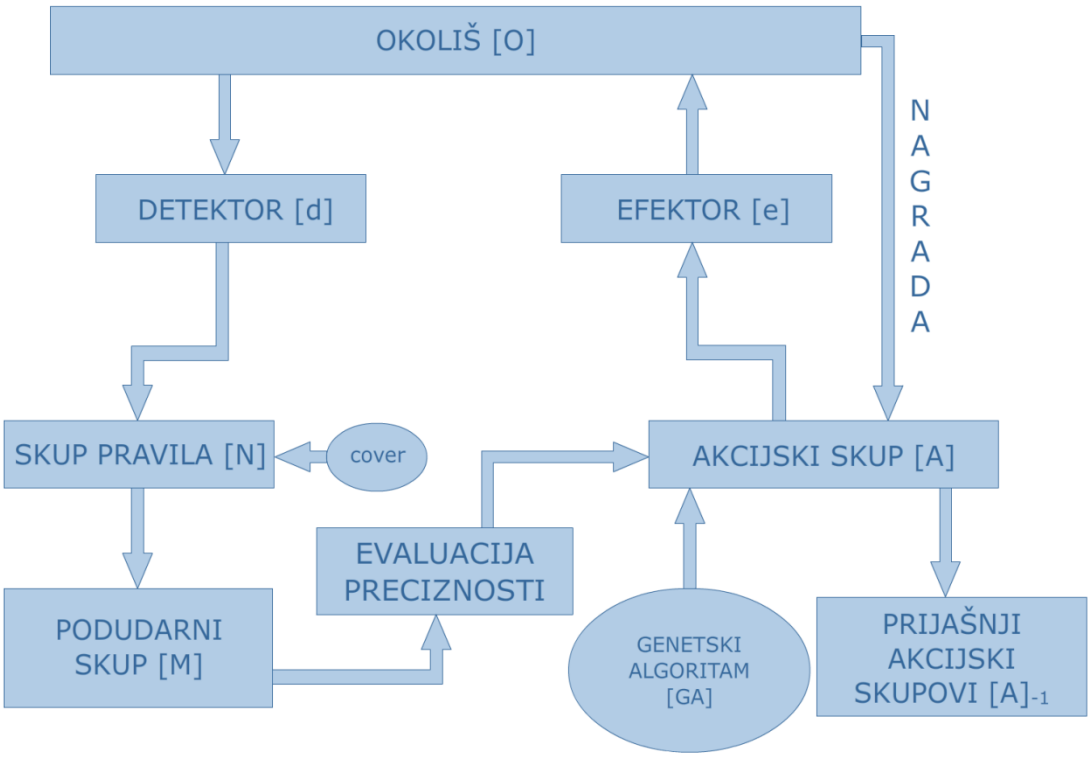
$$p_j = p_j + \beta \cdot \left(P - p_j\right) \tag{3.3}$$

3. Računa se relativna preciznost pravila formulom:

$$\kappa_j = \alpha \cdot \left({\varepsilon_0}/{\varepsilon_j}\right)^{\nu} \; za\; (\varepsilon_j \geq \varepsilon_0), a\; 1\; inače \tag{3.4}$$

   gdje su ν, α, ε_0 konstante koje kontroliraju oblik funkcije preciznosti.
4. Relativna preciznost κ_j' se računa za svako pravilo tako što se podijeli njegova ukupna preciznost sa zbrojem svih ukupnih preciznosti pravila iz akcijskog skupa.
5. Relativna preciznost se koristi kako bi se ažurirala dobrota svakog pravila koristeći **MAM postupak** (eng. ***Moyenne Adaptive Modifee***): Ukoliko smo dobrotu ažurirali $1/\beta$ puta $Q_j = Q_j + \beta\,(\kappa_j' - Q_j)$. U suprotnom, Q_j postaje srednja vrijednost dosadašnjih vrijednosti κ'.

Ova se verzija razlikuje i po mjestu djelovanja genetskog algoritma. Naime, ovdje se genetski algoritam primjenjuje samo na akcijski skup, a ne na čitavu populaciju. Isto tako, za razliku od dotadašnjih LCS-ova, odabir iz podudarnog skupa [*M*] u akcijski skup [*A*], vrši se na osnovu preciznosti pravila.



Slika 4.4: Shematski prikaz XCS-a.

4.2.4. Naknadne zapaženije inačice LCS-ova

Ubrzo nakon pojave XCS-a koji je ponovno popularizirao ove klasifikatorske sustave, javljaju se i druge varijacije LCS-ova. Iz tog razvojnog perioda najpoznatije su **ACS**, **XCSF**[11] (Wilson, 2001.), **UCS** i drugi. Ovdje bi još valjalo spomenuti i **minimalni klasifikatorski sustav** (eng. ***Minimal Classifier System***; skraćeno ***MCS***). Osmislio ga je 1997. godine Larry Bull, u svrhu unaprjeđivanja teorijskog shvaćanja LCS-ova. MCS je napravljen tako da za svoj rad koristi minimalan broj potrebnih komponenti, otkud mu i ime.

Sve kasnije implementacije LCS-ova mogu se svrstati u kategoriju suvremenih ostvarenja.

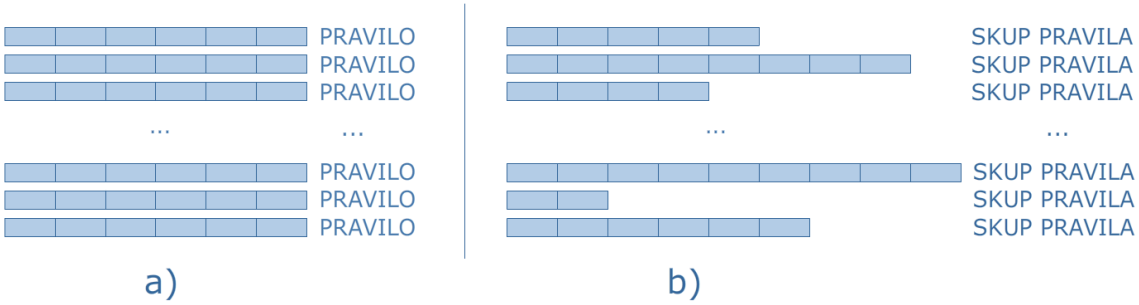
[11] Za XCSF je značajno da je jedan od rijetkih sustava koji podržavaju akcije s kontinuiranim vrijednostima.

4.3. Podjela klasifikatorskih sustava s mogućnošću učenja

4.3.1. Podjela po načinu djelovanja genetskog algoritma

S vremenom razvile su se dvije struje, koje različito tumače ulogu genetskih algoritama u ovim složenim sustavima, a nazvane su po dvjema školama gdje su nastali. Prva škola, kojoj je pripadao i sam Holland, bilo je Sveučilište Michigan, pa se taj pristup naziva **mičigenski stil**. Drugi se stil razvio nešto kasnije na Sveučilištu Pittsburgh[12], pa je tako i dobio naziv **pitsburški stil**.

Ova se dva stila razlikuju po načinu na koji tumače ulogu genetskog algoritma u sustavu. Izvorno, Holland je zamislio da genetski algoritam na jedinstvenoj populaciji pravila za klasifikaciju provodi mehanizme križanja i mutacije, kako bi s vremenom iznjedrio populaciju klasifikatora koja bi sadržavala optimalnu strategiju odlučivanja. Također, genetski je algoritam služio kako bi ubrzao proces prilagodbe agenta na promjene u okolišu.



Slika 4.5: Razlika populacija nad kojima genetski algoritam provodi svoje mehanizme. Mičigenski stil (a) sastoji se od populacije pravila jednake duljine, koja zajedno tvore rješenje. Kod pitsburškog stila (b), genetski algoritam radi nad populacijom skupova pravila, od kojih svaki predstavlja potencijalno rješenje **[**8**]**. Ti skupovi pravila ne moraju biti jednakih duljina.

Za razliku od mičigenskog stila, u pitsburškom se stilu ne koristi jedan skup klasifikatora, već više njih. Genetski algoritam izvodi operacije nad populacijom različitih skupova klasifikatora (a ne nad populacijom pojedinih klasifikatora kao u mičigenskom stilu), kako bi s vremenom pronašao onaj skup klasifikatora koji odražava optimalnu strategiju odlučivanja.

Pitsburški je stil nadahnut problemom raspodjele nagrade među pravilima. To je problem koji proizlazi iz komponente za učenje mičigenskog LCS-a, koja je bazirana na tehnici učenja potkrepljenjem. Kod učenja potkrepljenjem nije samo jedno pravilo zaslužno za dolazak do nagrade, već cijeli lanac pravila koji tvori uzročno-posljedičnu vezu. Stoga bi se nagrada trebala preraspodijeliti među svim članovima ovoga lanca. Ovaj je problem Holland pokušao riješiti uvođenjem algoritma lančanog prijenosa

[12] Literatura ostaje dosta nedefinirana oko pitanja tko je osmislio pitsburški stil. Navodno ga je osmislio S. F. Smith i implementirao u svom projektu u okviru diplomskog rada, pod nazivom LS-1, 1983. godine, dok je samu distinkciju između ova dva stila prvi zamijetio i naglasio Kenneth A. DeJong, 1993. u [**12**].

(BBA)[13]. Kod pitsburškog je stila ovaj problem riješen (tj. zaobiđen) time što se nagrade ne dodjeljuju pojedinim uspješnim pravilima, već cijelom skupu klasifikatora.

Unatoč ovoj početnoj prednosti pitsburškog stila, kad je u pitanju usporedba ovih dvaju stilova, pokazalo se da je zbog manje zahtjevnosti nad hardverskim resursima[14], lakše implementirati mičigenski stil LCS-a. Osim toga, zbog svog načina učenja, mičigenski stil se pokazao responzivnijim u promjenjivoj okolini. Naime, agent koji je implementacija mičigenskog stila, uči u stvarnom okolišu kroz interakciju (RL), dok agenti rađeni po uzoru na pitsburški stil svoje učenje prolaze nadzirano, na skupu primjera za učenje[15] (SL). Iz prije navedenih razlika između ova dva tipa učenja, dolazi se do zaključka da pitsburški stil nije pogodan za promjenjiv okoliš. To su razlozi zbog kojih je mičigenski stil češće proučavan i opisivan u radovima na temu LCS-a.

Ipak, postoje mnoge uspješne implementacije pitsburškog stila i s vremenom je postalo jasno da se i on može dobro primijeniti na rješavanje nekih klasa problema. Problemi, pri rješavanju kojih se pitsburški stil pokazao učinkovitim, uglavnom spadaju u kategoriju **dubinske analize podataka** (eng. ***Data Mining***; skraćeno: ***DM***).

S obzirom da je mičigenski stil zastupljeniji, kako u objavljenim radovima na temu LCS-ova, tako i u implementiranim rješenjima, danas se pod pojmom LCS-a, u većini slučajeva, podrazumijeva ovaj stil klasifikatorskih sustava.

4.3.2. Daljnje podjele mičigenskog stila

Unutar mičigenskog stila, nalazimo daljnje podjele, uglavnom zasnovane na razlikama u načinu evaluacije pravila, odnosno u načinu računanja funkcije dobrote.

LCS zasnovan na snazi

LCS zasnovan na snazi (eng. ***strength-based LCS***) jest najjednostavniji predstavnik mičikenskog stila. Riječ je o LCS-u, koji se pri izračunu funkcije dobrote za neko pravilo s_n oslanja na nagradu koju to pravilo ostvaruje. Možemo reći da se dobrota za akcijski skup pravila ažurira sljedećim izrazom:

$$Q_{[A]} = Q_{[A]} + \alpha \cdot \left[R + \gamma \cdot Q_{[A]_{+1}MAX} - Q_{[A]}\right] \tag{3.5}$$

gdje su α i γ konstante učenja, koje su ujedno i parametri cijelog LCS-a. Primjer predstavnika ove skupine LCS-a jest Hollandov LCS.

LCS zasnovan na preciznosti

Nešto noviji je **LCS zasnovan na preciznosti** (eng. ***accuracy-based LCS***). Kao što je već bilo rečeno u poglavlju o proširenom LCS-u, koji je tipični primjer ove vrste LCS-a,

[13] Algoritam Q-učenja izmišljen je tek kasnije, 1989. godine.
[14] Kod pitsburškog stila potrebna je velika količina memorije, jer populaciju genetskog algoritma čine skupovi skupova pravila. Također, operacije nad tako velikom količinom podataka mogu biti procesorski zahtjevne.
[15] U engleskoj se literaturi ovi različiti pristupi učenju nazivaju '*on-line*' (Michigan, nenadgledano učenje), odnosno '*off-line*' (Pittsburgh, nadgledano učenje) načinom učenja.

dobrota jedinki se u ovakvom algoritmu oslanja najviše na preciznost pravila u pokušaju da pretpostavi svoju nagradu. Detaljan postupak računanja dobrote, dan je ranije, a ovdje ćemo samo formalno izraziti dobrotu nekog akcijskog skupa [A] kao:

$$dob([A]) = dob([A]) + F(\varepsilon_j, p_j) \tag{3.6}$$

gdje je ε_j greška pravila sadržanih u [A], a p_j je njihovo predviđanje nagrade. Primjer predstavnika ove skupine je XCS.

LCS zasnovan na očekivanju

LCS zasnovan na očekivanju (eng. ***anticipatory-based LCS***) dobrotu pravila definira na njegovom očekivanju. Svako pravilo ima definirano očekivanje o tome kako će okoliš izgledati nakon primjene akcije tog pravila. Ovim se postupkom gradi **model prijelaza** (eng. ***model of transitions***). Model prijelaza sustavu omogućava bolje planiranje budućih stanja u ovisnosti o trenutnoj akciji i pospješuje brzinu učenja [**9**].

Očekivanje u svom zapisu može, pored standardnih znakova *0* i *1*, sadržavati i posebne znakove = i *?*. Znak = označava da se atribut stanja okoline ne mijenja, dok znak *?* označava da se atribut stanja ne može predvidjeti. Primjerice, ako na stanje okoline "101" djelujemo akcijom 0, tada pravilo "#01:0" kojem je pridijeljeno očekivanje "=1=" predviđa stanje okoline 111. Znak *?* je sličan znaku # koji se pojavljuje u uvjetnom ili akcijskom dijelu pravila. Ova je arhitektura pogodna za probleme od više koraka, probleme planiranja ili za ubrzavanje učenja [**8**]. Primjer predstavnika ove skupine je ACS.

4.4. Suvremeni dosezi

4.4.1. Struktura pravila

Kroz vrijeme su se pojavili prijedlozi za drugačije predstavljanje znanja u uvjetnom dijelu pravila, dakle prijedlozi za drugačije predstavljanje stanja okoliša. Veliki broj novih prijedloga fokusirao se na problem predstavljanja kontinuiranih atributa okoliša.

Pristupi koji su uspješno zamijenili originalnu strukturu uvjetnog dijela pravila, prikazanu ternarnim alfabetom, su:

- hiperpravokutnici (eng. ***hyperrectangles***),
- hiperelipsoidi (eng. ***hyperellipsoides***),
- neizrazita logika (eng. ***fuzzy logic***),
- stabla odluke (eng. ***decision trees***) i drugi [**10**].

Još jedan novi pristup koji se često javlja posljednjih godina jest pristup **simboličkim izrazima** (eng. ***Symbolic Expressions***; skraćeno***: S-expressions***).

Zabilježeni su slični pokušaji i sa akcijskim dijelom pravila. Akcije u takvim sustavima nisu više diskretne, već se računaju nekom predikcijskom funkcijom.

4.4.2. Evolucija pravila

Evolucijska komponenta služi istraživanju prostora potencijalnih rješenja. U prvim su se LCS-ovima ove komponente oslanjale na jednostavne mehanizme križanja i mutacije. Neka nedavna istraživanja ukazuju da bi takva nasumična primjena ovih mehanizama mogla usporavati čitav proces dolaska do cilja.

Jedan od načina bolje implementacije evolucijske komponente jest **algoritam procjene distribucije** (eng. ***Estimation of Distribution Algorithm***; skraćeno: ***EDA***)[16], koji je nadgradnja klasičnih genetskih algoritama. Ideja je izgraditi model strukture problema i tek tada istraživati prostor potencijalnih rješenja pomoću tog modela [**10**].

Alternativan pristup poboljšanju evolucijske komponente jest integracija dodatnih tehnika lokalne pretrage unutar samog genetskog algoritma. Lokalna se pretraga sastoji od primjenjivanja tehnika poboljšanja na pojedinim kandidatima za rješenje, tj. pripadnicima populacije. Ovakav se novi sustav naziva **Memetičkim algoritmom** (eng. ***Memetic Algorithm***; skraćeno: ***MA***) ili **hibridnim genetskim algoritmom** (eng. ***Hybrid Genetic Algorithm***; skraćeno: ***HGA***).

U praktičnom dijelu rada, biti će ostvaren LCS mičigenskog tipa, kako bi se osigurao optimalan utrošak memorijskih resursa. Radi jednostavnosti implementacije i kasnije analize, ostvareni LCS bit će zasnovan na snazi.

[16] Ponegdje u literaturi se naziva i (eng. ***Probabilistic Model-Building Genetic Algorithms***; skraćeno: ***PMBGA***)

5. Programsko ostvarenje klasifikatorskog sustava s mogućnošću učenja - SokobanLCS

Kako bi se stekao dojam o učinkovitosti LCS-ova, razvijeno je programsko rješenje koje simulira jedan ovakav sustav. S obzirom da različitih inačica LCS-ova ima mnogo, ovdje je naglasak stavljen na mičigenski tip LCS-ova.

5.1. Odabir problematike pogodne za rješavanje putem LCS-a

Već je rečeno da su LCS-ovi pogodni za veliki skup različitih problema u računalnoj znanosti, pa odabir konkretnog problema, na rješavanju kojeg bismo proučavali rad LCS-a, ne bi trebao biti složen posao. Ipak, kako bismo stekli što zorniji prikaz rada ovakvih agenata, programsko rješenje bi trebalo pokušati adresirati problematiku tako da:

a) je moguće jednostavno vizualno prezentirati osnovne entitete koji sudjeluju u učenju koncepata (konkretno – to bi bili agent i okoliš), i
b) sama problematika, korisniku koji se prvi susreće sa konceptom LCS-a, ne bude potpuno strana.

Zbog prvog je zahtjeva za ovu je priliku odabrana forma računalne igre. Ova forma osigurava oblik vizualne komunikacije s korisnikom i prezentacije apstraktnih pojmova, kao što su agent i okoliš, grafičkim simbolima. Iz drugog zahtijeva slijedi da bi ovakva računalna igra trebala biti općepoznata.

Ima mnogo računalnih igara koje zadovoljavaju ova dva zahtjeva, a za ovu priliku je odabrana računalna igra Sokoban.

5.2. Računalna igra Sokoban

Ovu igru osmislio je 1982. godine japanski dizajner računalnih igara Hiroyuki Imabayashi, zaposlen u tvrtki Thinking Rabbit. Igra je ubrzo stekla svjetsku popularnost te je proglašena igrom godine. Ono što je ovu igru činilo tako posebnom bila je njena jednostavnost i lucidnost rješenja pojedinih razina [**11**].

5.2.1. Pravila igre

Sokoban je igra u kojoj je potrebno, upravljajući skladištarom, razmijestiti kutije u skladištu na za to označena mjesta. Niti kutije, niti mjesta, na koja ih je potrebno pomaknuti, nisu označeni, pa je u teoriji moguće igru završiti na više različitih načina. U praksi je ovo rijedak slučaj, jer su pojedine razine igre tako osmišljene da dopuštaju samo jedno rješenje.

Jednu razinu igre predstavlja tlocrt skladišta, sa spomenutim kutijama, ciljevima i skladištarom, kojim upravlja igrač. Skladištar može:

- Utjecati na svoj položaj pomicanjem u jednom od četiri smjera (gore, dolje, lijevo, desno)
- Utjecati na položaj kutija tako da ih pomiče. Pomicanje kutija je ograničeno:
    - Kutije se mogu samo gurati.

- Najviše se može gurati jedna kutija istovremeno. Ako skladištar pokuša gurnuti kutiju iza koje stoji još jedna kutija, to mu neće poć za rukom (stanje okoliša se ne mijenja).

Razina se smatra uspješno savladanom, kad se sve kutije nađu na ciljnim pozicijama.

5.3. Programsko ostvarenje

5.3.1. Odabir programskog jezika i razvojnog okruženja

S obzirom na razinu apstrakcije na kojoj se nalaze LCS sustavi, lako je bilo odlučiti se za jezik neke više razine pri implementaciji ovakvog sustava. Najpogodniji među njima, činili su se jezici iz skupa objektno-orijentiranih jezika, pa je za ovu priliku odabran jedan od najznačajnijih i, u zadnje vrijeme, najzastupljenijih jezika ove vrste: **C#**.

Programski jezik C#

Programski jezik C# osmišljen je od strane Microsofta, a predstavljen je 2001. godine. Uparen sa razvojnim okruženjem Visual Studio, ubrzo je stekao svjetsku slavu. C# je zsamišljen kao multi-paradigmatski programski jezik, koji u objedinjava programerske discipline poput **strogih tipova** (eng. ***strong typing***), **imperativno programiranje** (eng. ***imperative programming***), **deklarativno programiranje** (eng. ***declarative programming***), **funkcijsko programiranje** (eng. ***functional programming***) te **objektno-orijentirano** (eng. ***object-oriented***) i **komponentno-orijentirano programiranje** (eng, ***component-oriented programming***).

Osnovni su ciljevi, prema ECMA[17]-i, kojima se težilo prilikom dizajniranja jezika, bili:

- Jezik treba biti jednostavan, moderan, općenamjenski i objektno-orijentiran.
- Jezik i njegove impelmentacije moraju pružati podršku za principe programskog inžinjerstva, kao što su provjera strogih tipova, detekcija pokušaja korištenja neinicijaliziranih varijabli, automatsko **čišćenje smeća** (eng. ***garbage collection***). Robusnost softvera, njegova izdržljivost te produktivnost programera koji se njime služi su važne.
- Jezikom se moraju moći ostvariti softverske komponente prilagođene distrtibuiranim okruženjima.
- Portabilnost je vrlo bitna.
- Podrška za intenacionalizaciju je vrlo bitna.
- C# je namijenjen da bude prilagođen izradi aplikacija za velike sustave, kao i za ugrađane sustave. Na taj se način pokrivaju potrebe sustava sa različitim sofisticiranim operativnim sustavima, kao i sustavi koji imaju mali skup dediciranih funkcija.
- Iako su aplikacije izrađene u C# programskom jeziku namijenjene da budu ekonomične u pogledu memorije i procesorske snage, jeziku nije namjena da se u pogledu ovih karakteristika natječe sa C jezikom ili strojnim jezicima.

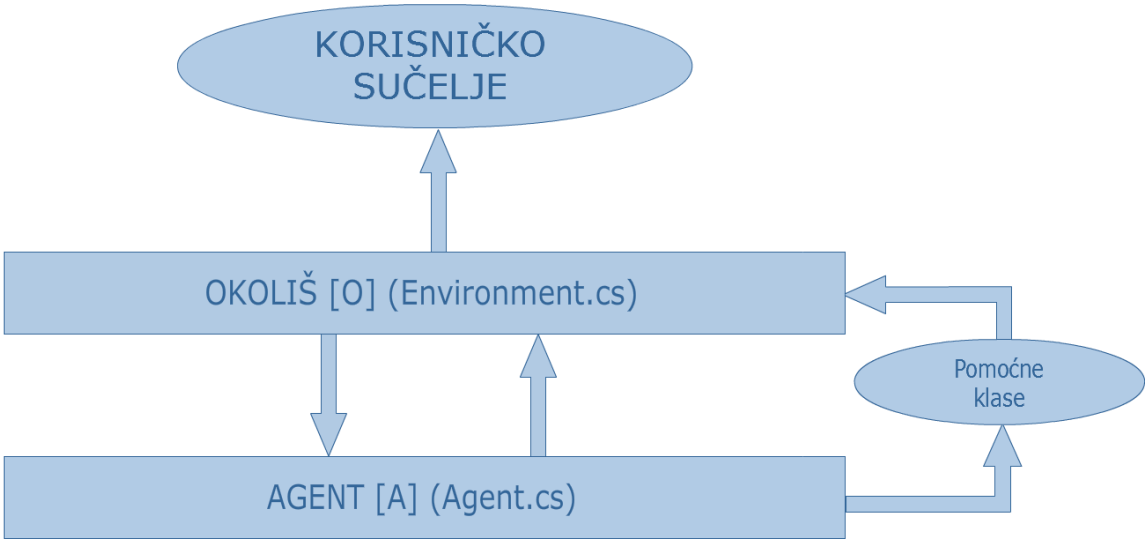
[17] ECMA je skraćenica za *European Computer Manufacturers Association.* Ovo se tijelo bavi standardizacijom informatičkih i komunikacijskih sustava, pa tako i programskih jezika.

Postoje razna razvojna okruženja u kojima je moguće pisati programe u C# programskom jeziku[18], ali je najzastupljeniji i najrazvijeniji Microsoft Visual Studio. Ovo razvojno okruženje dolazi sa nizom naprednih opcija u pogledu pronalaženja i otklanjanja pogrešaka (eng. ***debuging***), kao i u pogledu korisničke podrške (riječ je o vrlo detaljnoj dokumentaciji cijelokupnog C# programskog jezika, kao i ostalih podržanih jezika – MSDN).

Visual Studio, osim C#, podržava i mnoge druge programske jezike, kao što su Visual Basic, J#, Visual C++ i drugi.

5.3.2. Osnovna struktura SokobanLCS sustava

Sustav SokobanLCS, pisan je tako da se prilikom izrade pazilo na utiliziranje, što je više moguće, koncepata objektno-orijentirane paradigme. Zato se u grafičkom prikazu koji slijedi, lako mogu razaznati osnovne klase programa:



Slika 5.1: Osnovni elementi SokobanLCS sustava

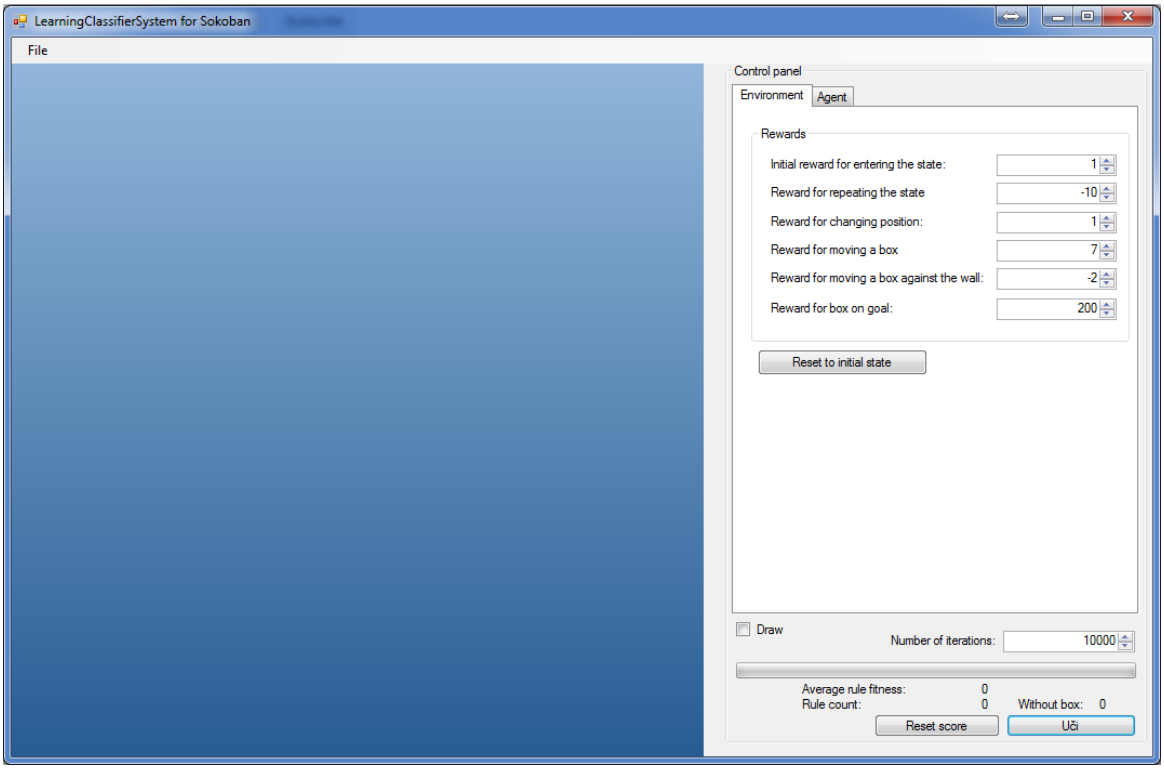
[18] Valjalo bi ovdje navesti Mono I DotGNU

Korisničko sučelje

Korisničko je sučelje sastavljeno od tri dijela. To su:

a) Traka s izbornikom – izbornik je jednostavan i služi manipulaciji datotekama, kao i namještanju nekih postavki (opcija) na razini cijele aplkacije
b) Prikaz okoliša – okoliš ima svoju manifestaciju u sučelju, kako bi se korisniku omogućio što bolji prikaz procesa koji se događaju prilikom učenja, odnosno prikaza rezultata.
c) Kontrolna ploča – Ovdje se mogu mijenjati postavke LCS-a i pratiti rezultati učenja u vidu grafa.

Prikaz sučelja:



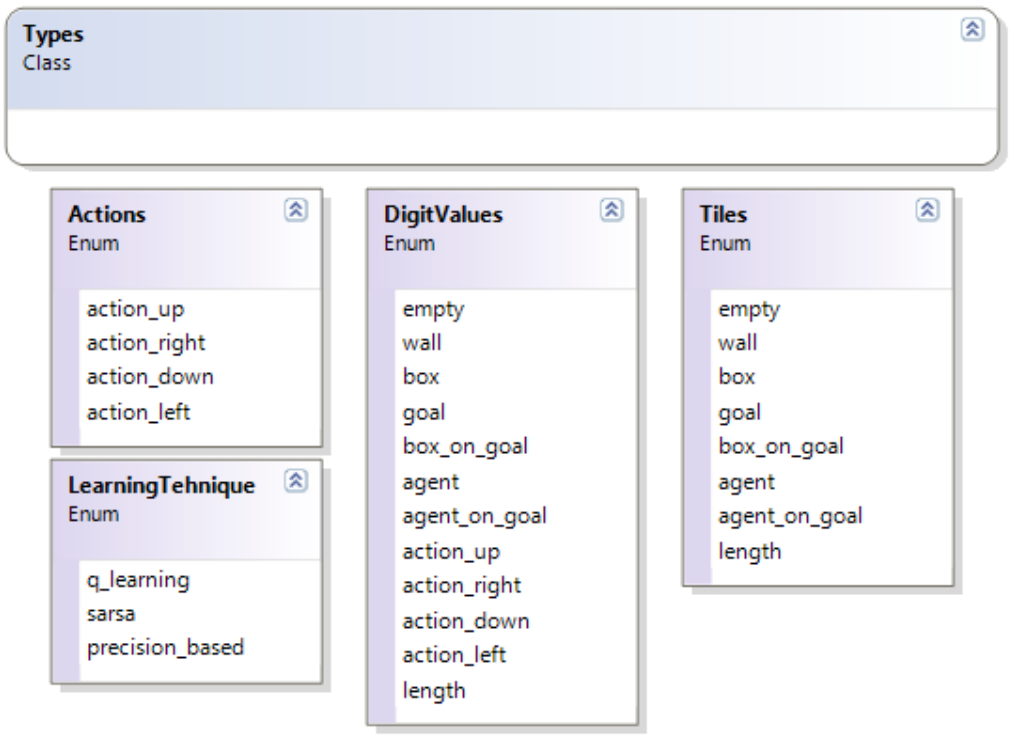
Slika 5.2: Sučelje SokobanLCS-a nakon pokretanja aplikacije

Pomoćne klase

Pomoćne klase definiraju tipove koji su poslužili kao osnovni tipovi podataka u ostvarenju LCS-a. Kao što je već rečeno, razina apstrakcije na kojoj se nalaze LCS-ovi, zahtjevaju od nas da proširimo osnovne tipove koji dolaze sa razvojnim jezikom, te ih prilagodimo potrebama ove teme.

Klasa *Types.cs* sadrži jednostavne tipove podataka, koji su se mogli ostvariti upotrebom enumeratora. Tipovi definirani u ovoj klasi su:

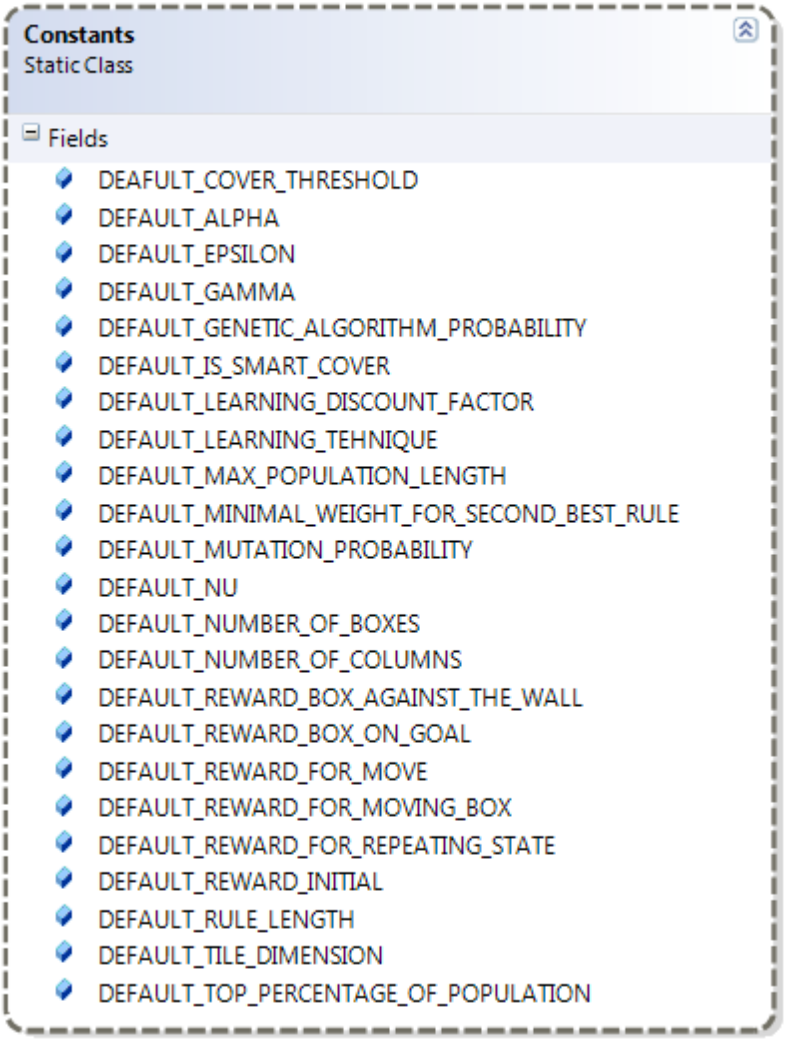
- `DigitValues` – popis svih znamenaka koje se mogu naći u bilo kojem lokusu u prezentaciji pravila
- `Actions` – popis znamenaka koje označavaju akciju (ima ih četiri: gore, dolje, lijevo, desno)
- `Tiles` – Popis znamenaka koje označavaju jedno polje okoliša
- `LearningTehnique` – Popis oznaka za označavanje Q-učenja ili SARSA-e



Slika 5.3: Dijagram klase Types

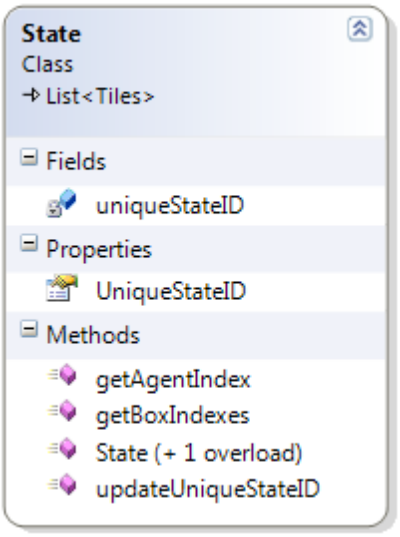
Klasa *Constants.cs* sadržava sve konstante koje se koriste u programu. Ovdje su uglavnom smještene pretpostavljene vrijednosti parametara.

Popis svih konstanti prikazan je na slici 5.4:



Slika 5.4: Dijagram klase Constants

Klasa *State.cs* osnovna je klasa SokobanLCS sustava. Ova klasa nasleđuje klasu `List<Tiles>` i proširuje ju metodama za dohvat indeksa lokusa na kojima se nalaze agent i kutije. Iz ovih se pozicija izračunava jedinstveni broj stanja (`uniqueStateID`).



Slika 5.5: Dijagram klase State

Klasu State kasnije koriste mnoge više klase programa, kao što su Agent.cs ili Environment.cs.

Klasa *Rule.cs* implementira sve značajke i metode vezane uz pojedino pravilo. Interna prezentacija pravila dana je listom:

```
private List<DigitValues> value;
```

Duljina ove liste ovisi o veličini razine igre. Zadnji element liste reprezentira akciju.

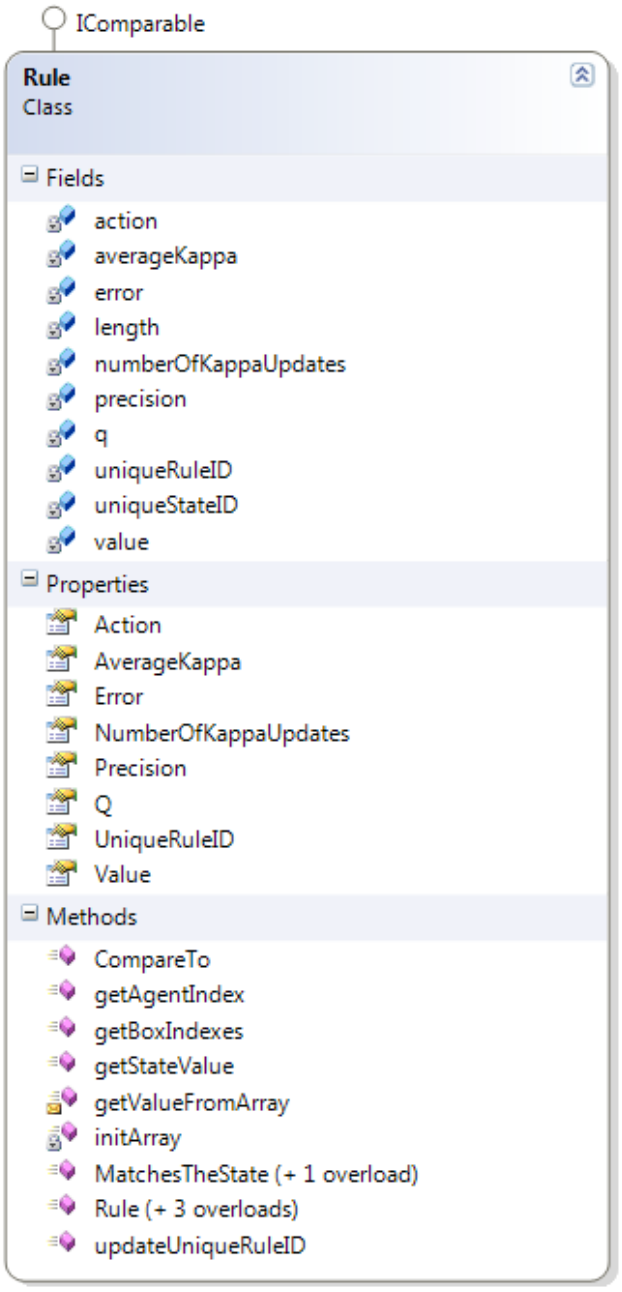
Osim interne reprezentacije pravila, ova klasa sadrži podatke o akciji koju ovo pravilo ima, jedinstveni broj (ključ) pravila i metode za izračunavanje ovog broja[19] (`uniqueRuleID`).

Klasa dodatno sadrži niz varijabli u koje se spremaju svi parametri za izračun dobrote po nekoj osnovi (po nekom od algoritama za učenje) te samu dobrotu kao realni broj (varijabla q).

Klasa Rule implementira sučelje `IComparable`, tj. metodu `CompareTo(Object obj)`. U metodi je implementirana jednostavna logika odlučivanja temeljena na usporedbi dobrote (varijable q) dvaju pravila.

Na ovaj se način omogućilo sortiranje lista koje sadrže elemente tipa Rule.

[19] Ovaj nam je jedinstveni broj potreban kako bismo lakše i brže pretraživali populaciju stanja.



Slika 5.6: Dijagram klase Rule

Klasa Environment.cs

Klasa *Environment*, jedna je od tri najveće klase aplikacije. Ova se klasa brine za prikaz okoliša u sučelju. Iz tog razloga, nasljeđuje baznu klasu *Control*, preko klase *PictureBox*. Osim što se brine o prikazu, ova klasa u svakom trenutku ima zapis stanja u kojem se okoliš nalazi, pomoću klase State.

Nadalje, u klasi *Environment* je smještena i metoda za izračun sljedećeg stanja u ovisnosti o sadašnjem stanju i akciji koja se pojavila na ulazu u okoliš od strane agenta:

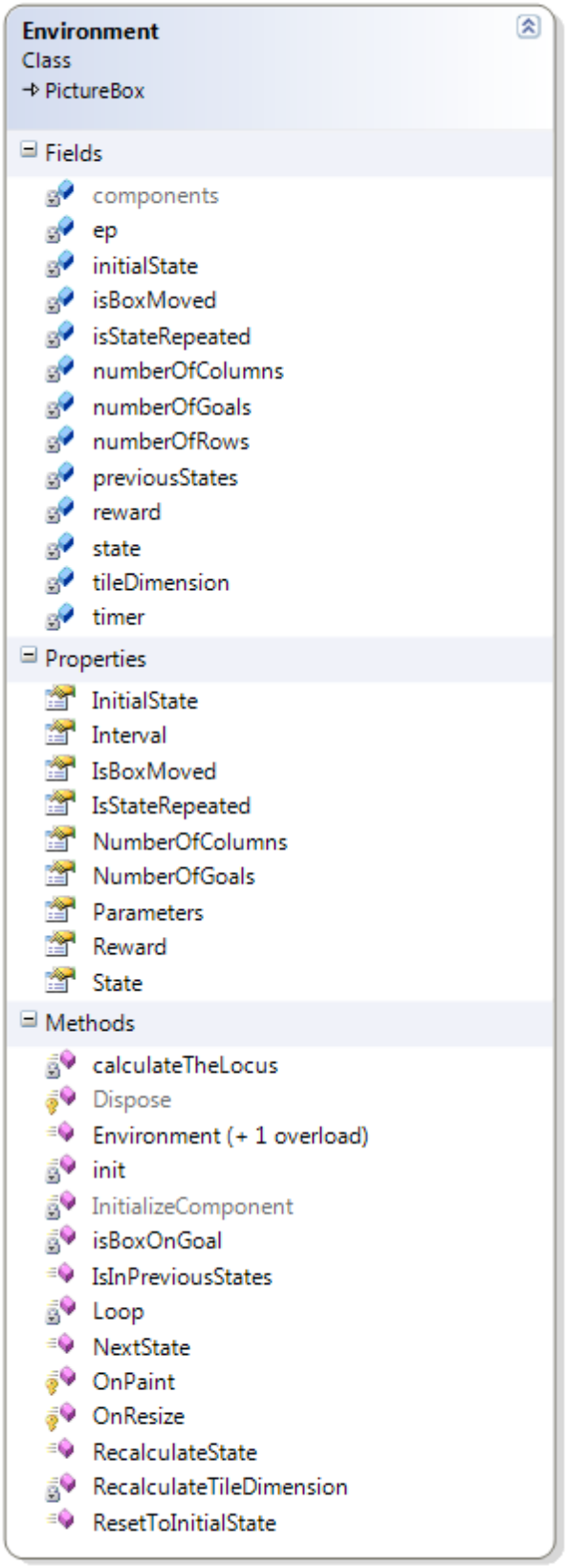
```
public int RecalculateState( List<Tiles> state, Actions a ) { … };
```

Ova je metoda prevelika da bi se ovdje u cijelosti naveo njen kôd, no ukratko, u njoj se obrađuju sve moguće kombinacije interakcije agenta sa njegovim neposrednim okolišem (dakle, jedno do tri polja u smjeru njegova kretanja). Na osnovu ovih konfiguracija okoliša, metoda računa sljedeće stanje u kojem će se okoliš naći. Kako bi se smanjila veličina kôda, koristi se pomoćna metoda:

```
private int calculateTheLocus( Point agentPosition, Actions a, int offset )
```

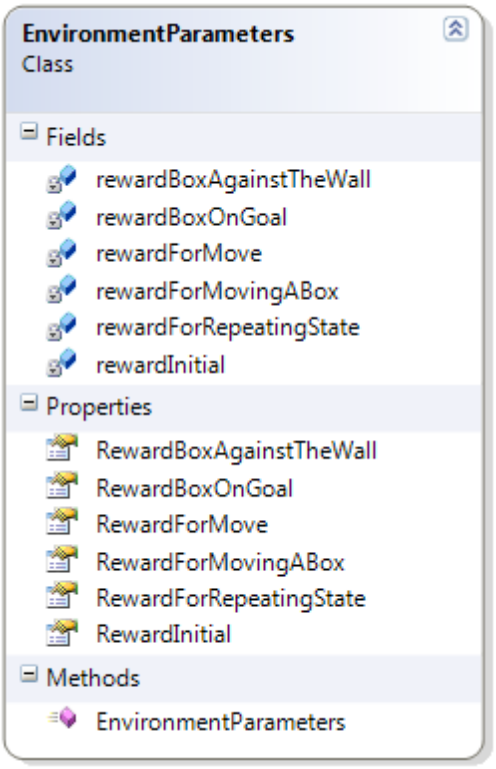
Ova metoda računa lokus, tj. indeks u listi, kojom je interno prikazano stanje okoliša, u ovisnosti o akciji (tj. smjeru kretanja agenta) I odmaku od pozicije agenta. Uporabom ove metode, smanjio se opseg glavne metode *RecalculateState,* jer se izračun za akcije u sva četiri smjera, sveo na jedan izračun, u kojem se gleda relativni, a ne više apsolutni pomak od agenta.

Metoda *RecalculateState* vraća, kao cijelobrojnu vrijednost, nagradu okoliša koju je polučila akcija iz polaznog stanja.



Slika 5.7: Dijagram klase Environment

Klasa Environment sadrži još jednu podklasu u kojoj su smještene postavke vezane uz okoliš. Te se postavke odnose na generiranje nagrade od strane okoliša.



Slika 5.8: Dijagram klase EnvironmentParameters

Klasa Agent.cs

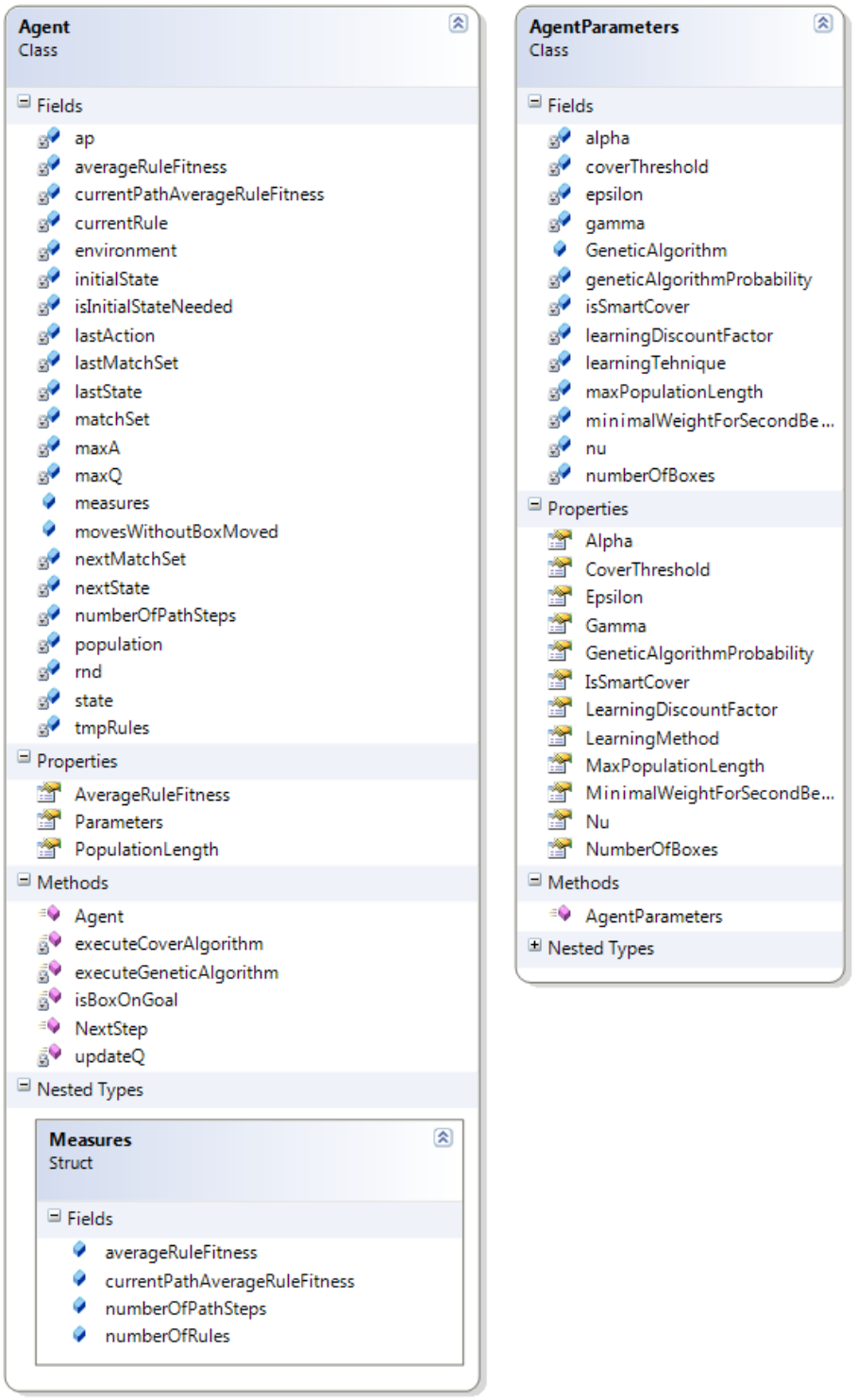
U ovoj je klasi implementirana cjelokupna logika oko izbora sljedeće akcije, kao i logika koja implementira mehanizme učenja. Glavna metoda u kojoj je sadržana većina ove logike dana je sa:

```
public void NextStep() { ... };
```

U ovoj se klasi slijedno izvršavaju sljedeće podcjeline:

- Odabire se skup pravila koji zadovoljavaju ulaz (stanje okoliša),
- Ovisno o ovom skupu, evaluira se postoji li potreba za pokretanjem *Cover* algoritma, i, ako da, ovim se algoritmom generira novo pravilo primjerene dobrote.
- Iz populacije jedinki se brišu prekobrojna pravila. Ova je granica dana parametrom 'Max. rule count'. Brišu se jedinke sa najmanjom dobrotom.
- Odabire se pravilo sa najboljom dobrotom u ovisnosti o parametru ε. Parametar ε utječe na sustav tako da unosi nedeterminizam u odabir pravila i predstavlja vjerojatnost da sustav ne odabere najbolje pravilo, već neko drugo iz skupa podudarnih pravila. Ovaj se proces naziva aukcijom, a specifičnost mu je da pravila s većom dobrotom imaju veću vjerojatnost da budu izabrana.
- Ovisno o metodi učenja (Q-učenje ili SARSA), određuje se nagrada za primjenu odabranog pravila.
- Nad populacijom se provode metode genetskog algoritma u ovisnosti o pripadnim parametrima.
- Ispituju se rubni uvjeti (je li agent rasporedio sve kutije na mjesto, je li agent došao u jedno od nepovratnih stanja, ...) te se ovisno o tome nastavlja sa izvođenjem idućeg koraka u istom okolišu ili se okoliš ponovno postavlja u početno stanje.

Osim ove, glavne, metode, postoji niz pomoćnih metoda koje poziva metoda *NextStep*. Također, u ovoj se klasi nalaze instance klasa *AgentParameters.cs*, koja je zadužena za pohranu parametara vezanih uz agenta, kao i struktura za pohranu podataka o mjerenjima *Measuring.cs*.



Slika 5.9: Dijagram klasa Agent i AgentParameters

6. Djelotvornost SokobanLCS sustava

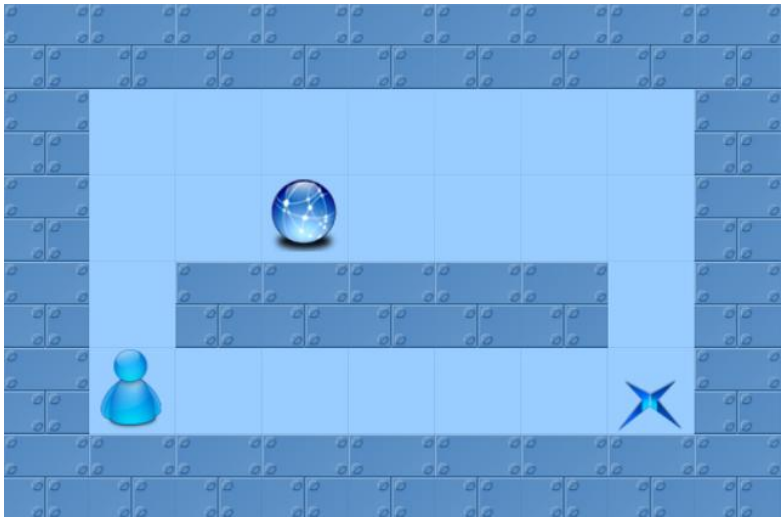
Djelotvornost sustava ispitivana je na pojednostavljenim razinama igre Sokoban. Razina igre je pojednostavljena, da bi se smanjio prostor stanja koji sustav mora pretražiti, kako bi došao do rješenja, tj. do optimalne strategije. Ovime se došlo do velike uštede vremenskih i memorijskih resursa.

Ispitano je nekoliko razina igre, s time da je svaka prezentirala određen koncept koji se pojavljuje kao dio rješenja kod većih primjera.

6.1. Razina 1: primjer sa skretanjem

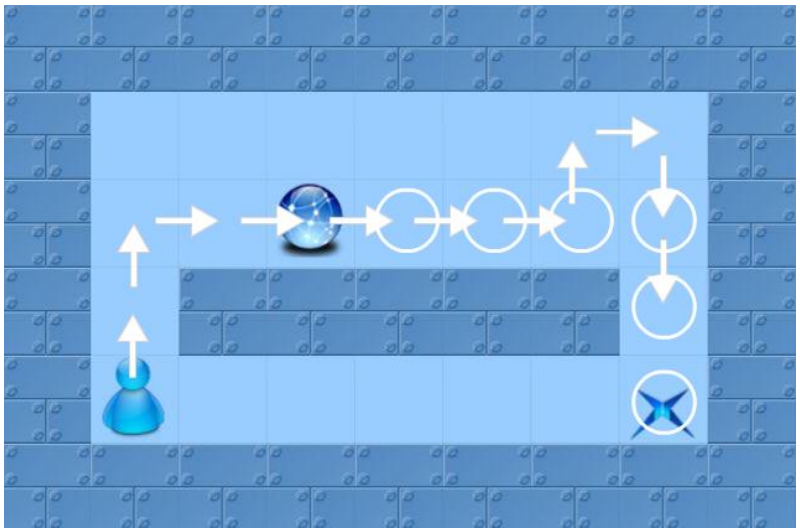
Prva razina služi nam kako bismo ispitali kako se agent snalazi prilikom promjene smjera kretanja. Kako je riječ o relativno maloj razini, ne očekuju se veliki problemi.

Razina korištena u ispitivanju prikazana je na slici 6.1.



Slika 6.1: Pojednostavljena razina igre Sokoban, okoliš razine 1

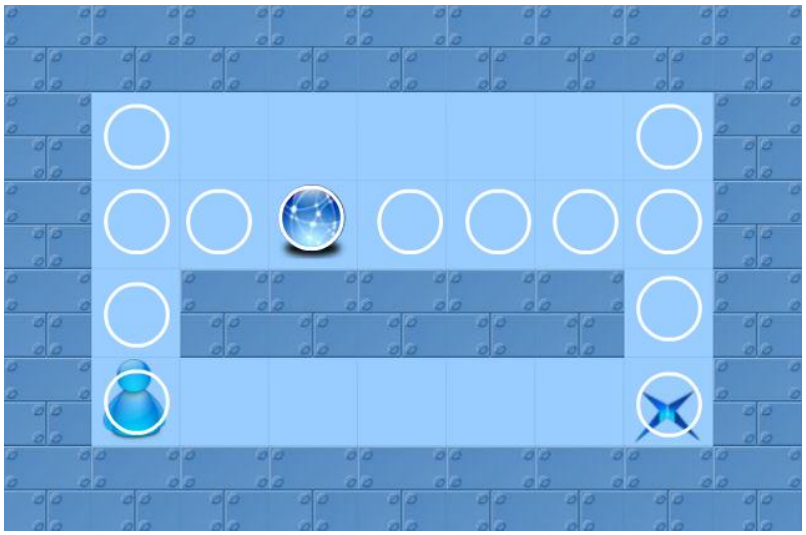
Na slici je prikazan agent (doljnji lijevi ugao) na svojoj početnoj poziciji te kutija (u obliku kugle) i cilj na koji kutiju treba pogurati (oznaka slična slovu x). Optimalna strategija, koju je potrebno naučiti, sastoji se od 11 koraka, a prikazana je na slici 6.2:



Slika 6.2: Optimalna strategija odlučivanja sastoji se od 11 uzastopnih koraka

6.1.1. Očekivani broj pravila

Broj pravila koja se mogu adresirati, možemo jednostavno izračunati. Prvo je potrebno izračunati broj polja na koja može stati agent (N_A). Na ploči za igranje ukupno je 23 polja na koje se može stati, ali je na jednom od njih kutija, pa agentu ostaju 22 polja na koja može stati. Ovaj broj valja pomnožiti sa brojem mjesta na koja možemo gurnuti kutiju. Ovaj broj najviše ovisi o konfiguraciji okoliša, s obzirom da je način micanja kutije definiran (kutiju možemo samo gurati). Broj takvih polja dan je na sljedećoj slici:



Slika 6.3: Polja na koja možemo gurnuti kutiju označena su bijelom kružnicom

Kao što se iz slike vidi, broj polja na koja možemo gurnuti kutiju je 13. S obzirom da igra prestaje u trenutku kad kutija dođe na cilj, broj stanja u kojima se agent može kretati je N_B = 12. Ukupan broj stanja možemo izračunati kao:

$$N_{stanja} = N_A \cdot N_B + 1 \tag{5.1}$$

uvrštavanjem gornjih brojki, dobiva se:

$$N_{stanja} = 22 \cdot 12 + 1 = 265 \tag{5.2}$$

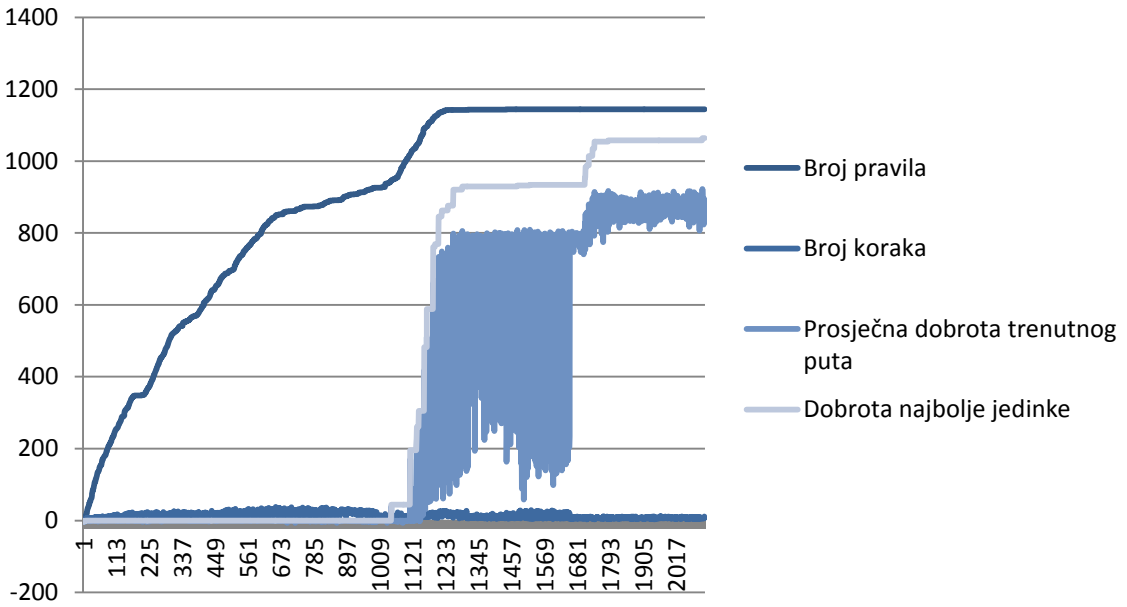
Kako bismo broj stanja pretvorili u broj pravila, potrebno je još sva stanja, osim zadnjeg, u kojem igra prestaje, pomnožiti sa brojem akcija koje agent u tim stanjima može poduzet:

$$N_{pravila} = (N_{stanja} - 1) \cdot N_{akcija} + 1 \tag{5.3}$$

$$N_{pravila} = 1057 \tag{5.4}$$

6.1.2. Djelotvornost algoritma

Broj pravila koje agent treba proći nije jako velik pa je za očekivati da će učenje relativno kratko trajati:



Slika 6.4: Proces učenja za razinu 1

Kao što se iz slike 21 vidi, agentu je bilo potrebno oko 1500 prolazaka kroz okoliš, kako bi prošao kroz sva dostupna stanja. Otprilike mu je toliko bilo potrebno i za pronalaženje optimalne strategije prolaska kroz stanja. Također valja zapaziti da je broj koraka koje je agent izvodio u jednom prolasku kroz okoliši bio veći za vrijeme učenja, nego kasnije, kad je optimalna strategija već bila usvojena.

6.1.3. Ovisnost algoritma o parametrima

Parametri strojnog učenja

U ovom smo se radu pozabavili agentom koji je učio po osnovi Q-učenja. Podsjetimo se formule za prijenos nagrade sa budućih stanja na prethodna (3.5):

$$Q_{[A]} = Q_{[A]} + \alpha \cdot \left[R + \gamma \cdot Q_{[A]_{+1}MAX} - Q_{[A]}\right] \tag{3.5}$$

Iz jednadžbe se jasno vidi da na proces učenja utječu dva parametra tj. konstanti učenja. Osim ova dva parametra, postoji i treći, čije je djelovanje malo drugačije. Parametri su:

- α – faktor kojim se utječe na odnos već naučenog i novonaučenog ($0 \leq \alpha \leq 1$). Za $\alpha = 0$, agent uopće ne uči: nagrada iz okoliša nema utjecaja na dobrotu pravila. Za $\alpha = 1$, ništa od prije naučenog neće ostati pohranjeno u dobroti stanja.
- Υ – faktor zaslužan za prijenos 'odgođene nagrade'. Ovaj faktor određuje utjecaj budućih nagrada na dobrotu sadašnjeg pravila ($0 \leq \Upsilon \leq 1$). Ukoliko je $\Upsilon = 0$, agent je fokusiran na trenutnu nagradu u prostoru. Što je Υ bliža 1, to će agent više učiti iz budućih stanja.
- ε – treći faktor, koji se ne nalazi u jednadžbi kojom ažuriramo dobrotu stanja, već određuje vjerojatnost da se prilikom prolaska kroz stanja okoliša ne prati optimalna strategija koja je do sad naučena. Na ovaj način faktor ε utječe na omjer iskorištavanja naučenog i otkrivanja novog u okolišu ($0 \leq \varepsilon \leq 1$). Za ε = 0, agent će samo prolaziti kroz stanja po naučenoj optimalnoj strategiji, dok će za ε = 1 agent prolaziti kroz stanja okoliša potpuno nasumično.

Parametri koji su korišteni prilikom izrade slike 21 su:

$\alpha = 0{,}5$
$\Upsilon = 0{,}987$
ε = 0,5

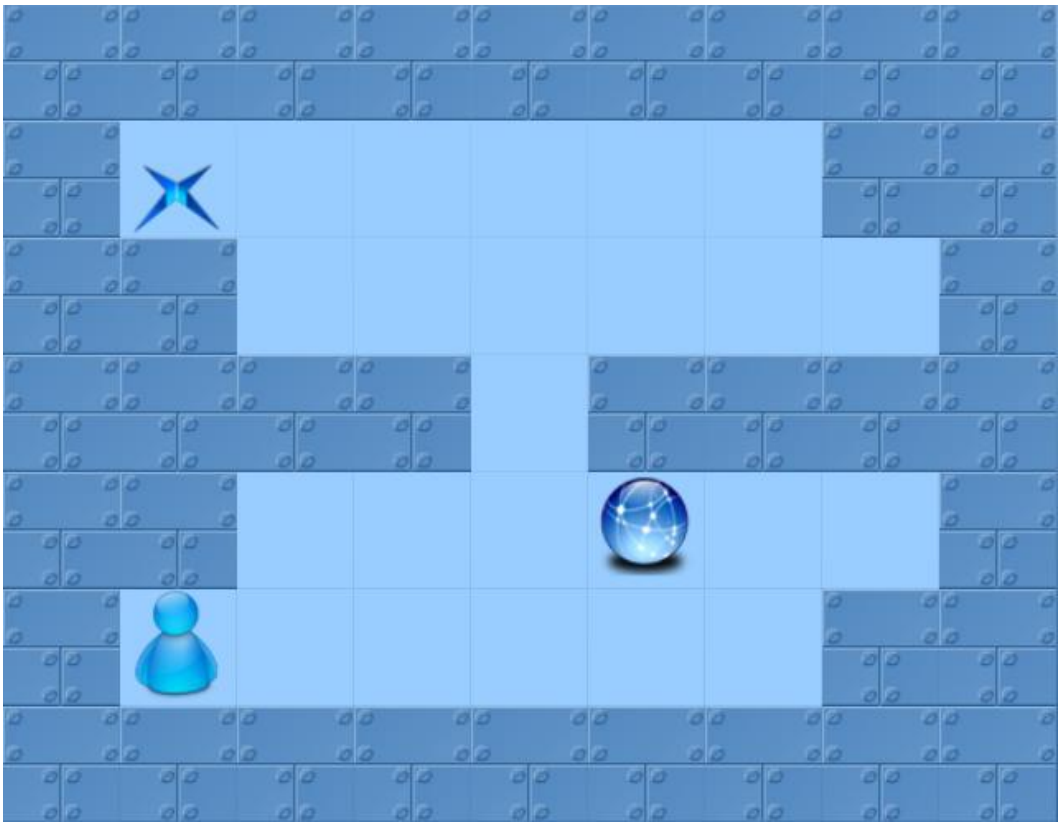
Parametri genetskog algoritma

Na genetski algoritam također utječu tri parametra. To su:

- Faktor pozivanja genetskog algoritma. Ovaj faktor predstavlja vjerojatnost da će se prilikom jednog koraka pozvati genetski algoritam. Vrijednost 0 isključuje genetski algoritam
- Drugi faktor diktira postotak populacije iz koje se odabiru roditelji za križanje. Što je ovaj faktor manji, to će bolja pravila biti izabirana.
- Faktor vjerojatnosti mutacije. Što je faktor veći, veća je i vjerojatnost mutacije. Sa većom razinom mutacije, veća je i vjerojatnost adresiranja nedohvatljivih stanja. U primjeru sa slike 21, za postavku od 0,2, sustav je adresirao 1144 pravila, iako je dohvatljivih bilo, kao što smo i izračunali, svega 1057.

6.2. Razina 2: primjer sa prolaskom kroz 'vrata'

U drugom primjeru, ispitivana je snalažljivost agenta kada je potrebno gurnuti kutiju kroz otvor u zidu da bi se došlo do cilja. Prikaz razine dan je slikom 6.5:



Slika 6.5: Razina 2 - primjer s vratima

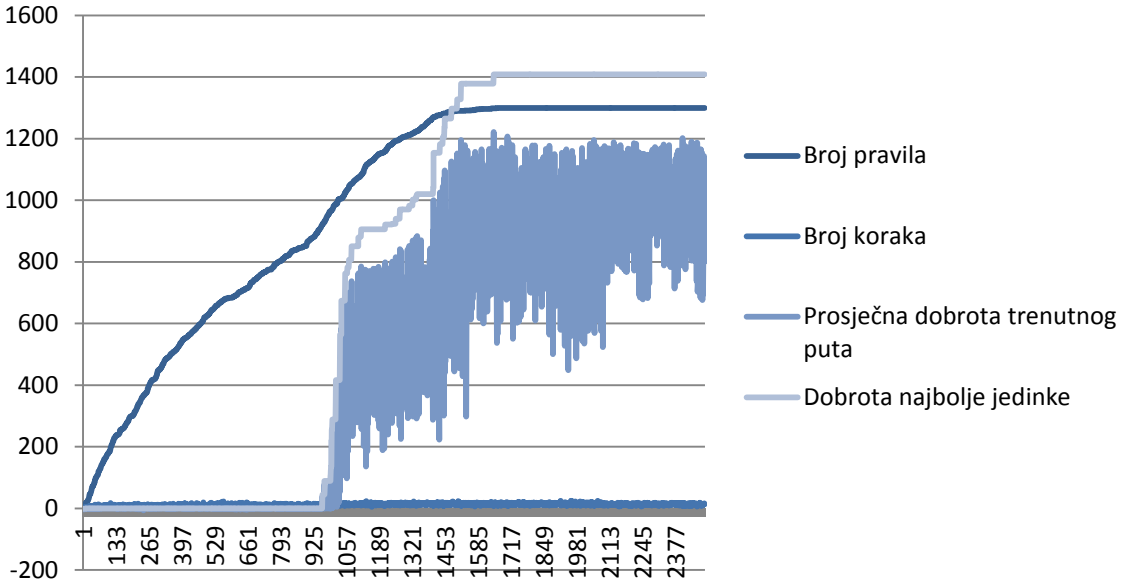
Iz slike se jasno može iščitati konfiguracija razine i izračunati broj dohvatljivih stanja te broj dohvatljivih pravila. Za ovu razinu vrijedi $N_A=25$; $N_B=12$, pa dobivamo:

$$N_{stanja} = 25 \cdot 12 + 1 = 301 \tag{5.6}$$

i

$$N_{pravila} = 1201 \tag{5.7}$$

Proces učenja dan je slikom 6.6:

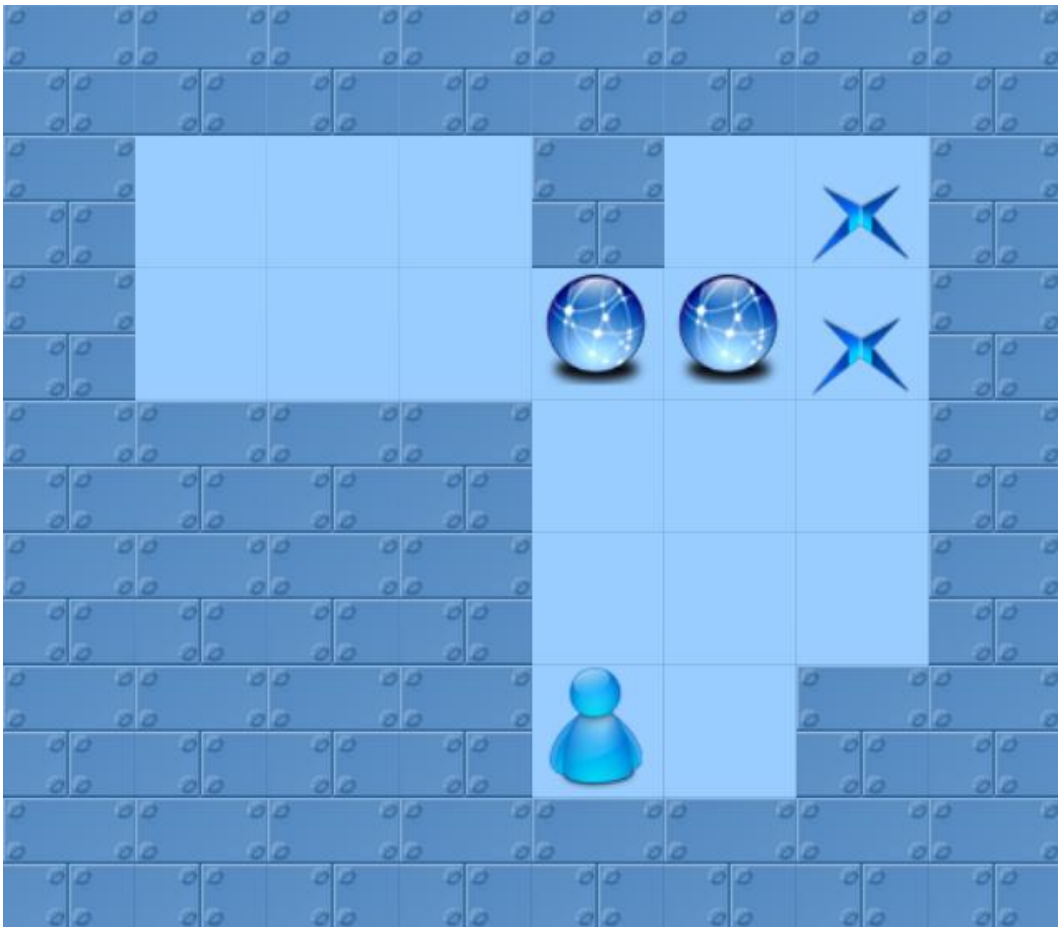


Slika 6.6: Proces učenja za razinu 2

Ova razina ima nešto više stanja, ali je agent svejedno trebao, kao i u prošlom primjeru, oko 1500 prolazaka kroz okoliš da nauči optimalnu strategiju. Adresirano je nešto više pravila od broja dohvatljivih (njih 1299), zahvaljujući operaciji mutacije genetskog algoritma. Veliki broj devijacija kod krivulje koja prikazuje prosječnu dobrotu trenutnog puta, može se objasniti konfiguracijom terena koja dopušta agentu veću slobodu kretanja, pa i veću vjerojatnost odabira pogrešnog smjera, uz $\varepsilon = 0{,}5$. Ove su devijacije sve manje kako vrijeme učenja prolazi, jer agent uči iz više različitih pozicija, tj. iz više različitih stanja, doći do cilja.

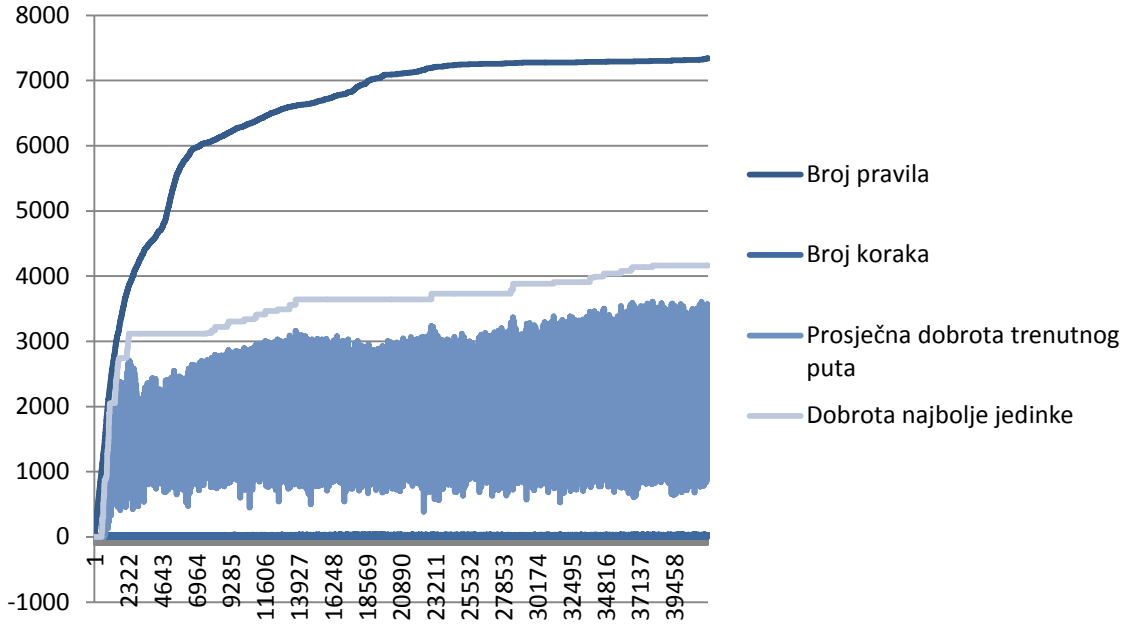
6.3. Razina 3: primjer s dvije kutije

U ovom primjeru ovodimo i drugu kutiju te testiramo ponašanje agenta u situaciji u kojoj je potrebno jednu od kutija gurati u smjeru od cilja, kako bi je se dugoročno uspješno smjestilo na cilj. Izgled okoliša dan je slikom 6.7:



Slika 6.7: Okoliš razine 3

Proces učenja za ovakav okoliš, trajao je znatno dulje:

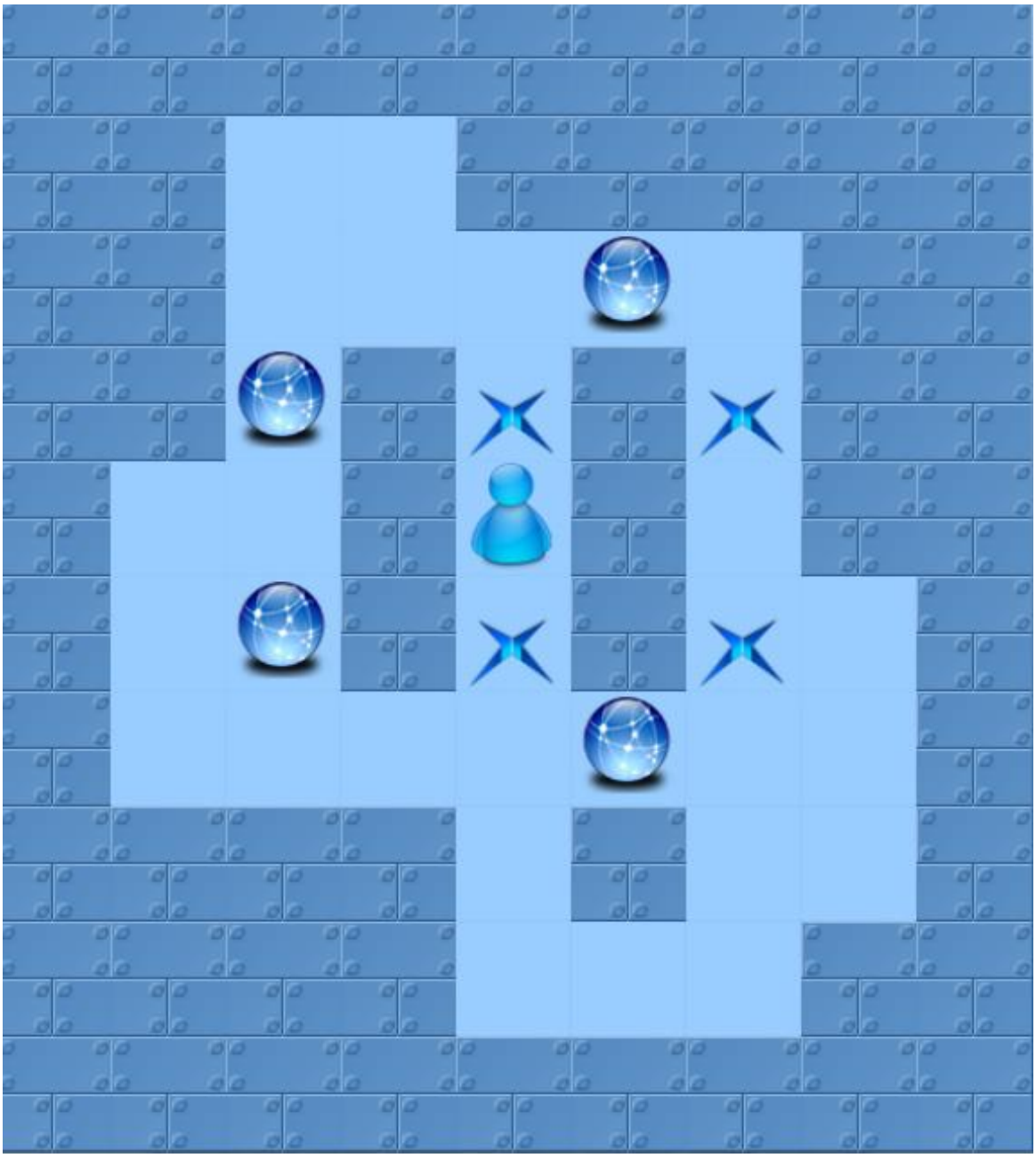


Slika 6.8: Proces učenja za razinu 3

Kao što se iz slike vidi, ni nakon četrdeset tisuća prolazaka kroz okoliš, agent nije adreseirao sva moguća stanja kroz koja može proći s agentom. Ipak, optimalnu strategiju prolaska kroz okoliš usvojio je nakon otprilike 25000 prolazaka.

Iz dobivenih rezultata može se zaključiti da LCS dobro obavlja zadatak pronalaženja optimalne kontrole. No, za primjer igre Sokoban, vjerojatno bi bolje rezultate dala implementacije pitsburškog stila LCS-a. Tome u prilog ide i činjenica da komponenta genetskog algoritma nije imala prevelik utjecaj na pronalaženje optimalnog rješenja. Razlog tome jest relativna ograničenost operatora križanja i mutacije koja proizlazi iz zapisa stanja (većina lokusa je nepromjenjiva – predstavljaju zid, a i oni koji jesu promijenjivi potpadaju pod dodatne restrikcije. npr. u vidu zadanog broja kutija koje dohvatljivo stanje mora sadržavati). Kod pitsburškog stila, genetski bi algoritam imao veću slobodu, jer bi operirao nad jedinkama koje predstavljaju potencijalno rješenje u vidu prolazaka kroz određeni broj stanja. Genetski bi algoritam u tom slučaju zapravo križao (i mutirao) različite puteve do cilja.

Ipak, kako je pitsburški stil memorijski zahtjevniji od mičigenskog, ostaje pitanje koliko bi u praksi ovakav način rješavanja danog problema bio upotrebljiv. Čak i kod ostvarenja prikazanog u ovom radu pojavljivale su se poteškoće kod razmjerno velikih razina igre, kao što je razina prikazana slikom 6.9.



Slika 6.9: Razina kod koje se pojavila OutOfMemory greška

7. Zaključak

Klasifikatorski sustavi s mogućnošću učenja pokazali su se korisnima u rješavanju cijelog niza različitih problema: dubinska analiza podataka, klasična klasifikacija, aproksimacija funkcija, navigacija kroz prostor i drugi. Ti se problemi javljaju u najrazličitijim granama znanosti, od robotike do epidemiologije, stoga ne čudi da se u zadnjih nekoliko godina broj radova i istraživanja na temu ovih sustava jako povećao.

Iako su se LCS-ovi pokazali korisnima u tako širokom spektru primjena, tu njihovo proučavanje ne prestaje i ne može se reći da su oni stigli do kraja svog razvojnog puta. Novije implementacije uzele su u razmatranje nove dosege sa drugih područja računalstva i ovaj se koncept razvija i dalje, sukladno tim dosezima.

U praktičnom je dijelu rada ostvaren razmjerno jednostavan sustav, namijenjen rješavanju problema pronalaska optimalne kontrole kod računalne igre Sokoban. Iako je program bio uspješan pri rješavanju jednostavnijih zadataka, pokazalo se da bi za zahtjevnije zadatke bio potreban drugačiji pristup problemu, tj. pitsburški stil LCS-a.

Može se zaključiti da će klasifikatorski sustavi s mogućnošću učenja još dugo vremena imati svoju primjenu u računalnoj praksi.

Literatura

[1] Bowler P.J., *Evolution: the History of an Idea*.: Berkeley: University of California Press, Jan. 2003.

[2] Melanie Mitchell, *An Introduction To Genetic Algorithms*. Cambridge, Massachusetts, United States of America: MIT Press, 1999.

[3] M. Krsnik-Rasol. (2011, Mar.) Praktikum iz biologije stanice - Mejoza. [Online]. http://hr.wikipedia.org/wiki/Mejoza

[4] Marin Golub. (2004, Sep.) Genetski algoritmi. [Online]. http://www.zemris.fer.hr/~golub/ga/ga_skripta1.pdf

[5] Mario Lučić and Goran Radanović. (2008.) Klasifikatorski sustavi. [Online]. http://www.zemris.fer.hr/~golub/ga/studenti/projekt2007/lcs/PR_Lucic_Radanovic.pdf

[6] Richard S. Sutton and Andrew G. Barto. (1997.) Reinforcement Learning – An Introduction. [Online]. http://webdocs.cs.ualberta.ca/~sutton/book/the-book.html

[7] Larry Bull and Tim Kovacs, *Foundations of Learning Classifier Systems: An Introduction*.: Springer-Verlag Berlin Heidelberg, 2005.

[8] Ryan J. Urbanowicz and Jason H. Moore, "Learning Classifier Systems: A Complete Introduction, Review and Roadmap," *Journal of Artificial Evolution and Applications*, 2009.

[9] Olivier Sigaud and Stewart W. Wilson. (2007) Learning Classifier Systems: A Survey. [Online]. http://pages.isir.upmc.fr/~doncieux/animatlab/public_html/http/papers/sigaud_wilson_lcs_survey.pdf

[10] Jaume Bacardit, Ester Bernado-Mansilla, and Martin V. Butz. (2008) LCS: Looking Glimsing. [Online]. www.cs.nott.ac.uk/~jqb/publications/LCS-Looking-Glimsing.pdf

[11] Anon. Sokoban History. [Online]. http://www.games4brains.de/sokoban_history.htm

[12] Kenneth A. DeJong, William M. Spears, and Diana F. Gordon, "Using Genetic Algorithms for Concept Learning," *Machine Learning*, no. 13, pp. 161-188, 1993.

[13] Joseph Culberson. (1997, Apr.) Sokoban is PSPACE-complete. [Online]. http://webdocs.cs.ualberta.ca/~joe/Preprints/Sokoban/paper.html