Umjetna inteligencija – Labsratorijska vježba 3

UNIZG FER, ak. god. 2017/18.

Zadano: 14.5.2018. Rok predaje: 27.5.2018. do 23.59 sati.

Uvod

U okviru ove laboratorijske vježbe rješavati ćete probleme iz područja potpornog učenja i strojnog učenja. Kod koji ćete koristiti možete skinuti kao zip arhivu na stranicama fakulteta ovdje, ili na github repozitoriju predmeta ovdje.

Laboratorijska vježba sastoji se od dvije implementacijski nepovezane komponente – naivnog Bayesovog klasifikatora i algoritama temeljenih na potpornom učenju. Svaka od komponenata nalazi se u zasebnom direktoriju. Kod za projekt se sastoji od Python datoteka, dio kojih ćete trebati pročitati i razumjeti za implementaciju laboratorijske vježbe, dio koji ćete trebati samostano uređivati te dijela koji ćete moći ignorirati.

Zadatak 1: Naivan Bayesov klasifikator (12 bodova)

Nakon raspakiranja zip arhive te pozicioniranja u poddirektorij za naivnog Bayesa, opis datoteka i direktorija koje ćete vidjeti je u nastavku:

Datoteke koje ćete uređivati:	
naiveBayesClassifier.py	Implementacija naivog Bayesovog klasifikatora
Datoteke koje trebate proučiti:	
pacman.py	Opis logike Pacmanovog svijeta
util.py	Korisne pomoćne strukture i metode
dataLoader.py	Učitavanje podataka te ekstrakcija značajki
classifier.py	Glavna datoteka koja pokreće klasifikator i testove
Pomoćne datoteke koje možete ignorirati:	
layout.py	Opis mapa na kojima se Pacman kreće
game.py	Model igre

Pri rješavanju prvog zadatka nije dostupan autograder, no implementirani su automatski testovi kompatibilnosti. Fiksirani izlazi implementiranog modela su zapisani prvi vrhu datoteke *classifier.py*, te se po završetku implementacije vašeg modela automatski mogu pokrenuti testovi pomoću naredbe `python classifier.py -t 1`, koja će ispisati apsolutno odstupanje od točnih *a posteriori* vjerojatnosti. Naglašavamo da točni rezultati s automatskim testiranjem ne povlače sve bodove na zadatku, budući da se izlazi provjeravaju samo za 5 primjera iz skupova za testiranje i treniranje.

Naivni Bayesov klasifikator je unutar vježbe definiran u stilu popularnog razvojnog okruženja za strojno učenje u pythonu, scikit-learn. Svaki model strojnog učenja prima svoje *hiperparametre* kroz konstruktor, te sadrži metodu *fit*, u kojoj se model nauči na podacima za treniranje, te metodu *predict*, koja za nove i nepoznate podatke predviđa njihove izlazne vrijednosti.

Unutar okvira vježbe, rješavati ćemo problem predviđanja Pacmanovih budućih poteza. U direktoriju *pacmandata* nalaze se komprimirani podaci o trenutnom stanju igre te idućem potezu koji je određena implementacija navigacije Pacmana odabrala. Podaci su podijeljeni u skupove za treniranje i testiranje, te postoje za četiri različita agenta: gladnog Pacmana koji uvijek ide prema najbližoj hrani (prefiks *food*), preplašenog Pacmana koji uvijek stoji na mjestu (prefiks *stop*), hrabrog Pacmana koji uvijek ide prema najbližem duhu (prefiks *suicide*) te Pacmana naučenog potpornim učenjem koji temeljem nekih kriterija relativno uspješno izbjegava duhove te jede hranu (prefiks *contest*).

Kako ćete i vidjeti u drugom dijeli laboratorijske vježbe, učenje ponašanja od nule je prilično mukotrpan proces – treba odigrati iznimno puno igara dok se ponašanje Pacmana ne bi prilagodilo na igru, te on počeo konzistentno pobjeđivati. Ovome možemo stati na kraj učenjem oponašanjem *(tzv. imitation learning)*, te možemo ovisno o ponašanju Pacmana koje smo vidjeli prilagoditi svoje ponašanje kako bi bilo što bliže viđenom, bez da uopće pokušavamo igrati igru.

Glavni problem kod ovakvih modela je što nam je poznata odluka koju je agent kojeg oponašamo donio te stanje svijeta u kojem se agent nalazio, ali ne znamo kriterije na temelju kojih je ta odluka donešena. No, nadamo se da će uz dovoljno promatranja (primjera za učenje) ovaj problem biti premostiv. Prvi korak implementacije učenja oponašanjem je već implementiran za vas, te su se iz podatka o trenutnom stanju igre izvadile određene značajke za svaki potencijalni idući potez.

U direktoriju *classifier_data* nalaze se izvučeni podaci u tab-separated values *('.tsv')* formatu koji je kompatibilan s klasifikatorom. Prvi red, tzv. header sadrži imena svake značajke koja se smatra bitnom za odluku, dok zadnji stupac sadrži ciljne varijable – odluke koju je agent kojeg oponašamo donio za navedene ulaze. Samo su dvije značajke izvučene za svaki mogući potez – jesmo li s tim potezom pojeli hranu i jesmo li s tim potezom smanjili udaljenost do najbližeg duha.

Primjer podataka za značajke koje se nalaze u datoteci *'contest_training.tsv'* nalazi se u nastavku:

foodEaten_East	foodEaten_North	...	ghostCloser_Stop	ghostCloser_West	target
True	False	...	False	True	West
False	False	...	False	False	West

Problem 1.1: Implementacija klasifikatora (12 bodova)

U datoteci *naiveBayesClassifier.py* implementiran je kostur klase naivnog Bayesovog klasifikatora, kojemu fale implementacije nekih metoda. Vaš zadatak u prvom dijelu laboratorijske vježbe je dovršiti implementacije tih funkcija i dobiti potpuno funkcionalan klasifikator.

Podsjetimo se Bayesovog pravila, i predpostavimo da imamo neki skup podataka koji smo opazili u Pacmanovom svijetu, te tražimo najizgledniji potez koji je agent kojeg oponašamo napravio:

$$P(h_i|podaci) = \frac{P(podaci|h_i)P(h_i)}{P(podaci)}$$

Pri čemu nam je h_i hipoteza, tj. svaki mogući potez i, $P(h_i)$ **a priori** vjerojatnost tog poteza, $P(h_1|podaci)$ **a posteriori** vjerojatnost poteza **za ulazne podatke**, te

$P(podaci|h_1)$ **izglednost** pojavljivanja tih podataka uz taj potez. $P(podaci)$ je vjerojatnost pojavljivanja samih tih podataka, te služi kao normalizacijska konstanta.

U našem primjeru, hipoteze su **potezi** koje možemo napraviti (West, North, ...) a podaci su **opažanja** o svijetu (tzv. značajke) – foodEaten_East, foodEaten_North, ... na temelju kojih donosimo odluku. Naivna pretpostavka Bayesovog klasifikatora je ta da su same značajke unutar jednog opažanja međusobno nezavisne, što nam omogućava zapis ukupne vjerojatnosti za sve značajke $P(podaci|h_i)$ kao umnožak vjerojatnosti za pojedine značajke!

Naivni Bayesov klasifikator kao ispravnu hipotezu bira onu koja ima najveću **a posteriori** vjerojatnost *(Maxium A Posteriori, MAP hipoteza)*.

Ukratko, naivni Bayesov klasifikator može se formulirati na idući način:

$$h_{MAP} = \arg\max_{h_i \in H} P(h_i|f) = \arg\max_{h_i \in H} \frac{P(f|h_i)P(h_i)}{P(f)} = \arg\max_{h_i \in H} P(f|h_i)P(h_i)$$

$$h_{MAP} = \arg\max_{h_i \in H} P(h_i) \prod_j P(f_j|h_i)$$

Pri čemu su f_j značajke, a h_i hipoteze (ciljne varijable). Eliminacija normalizacijske konstante je dopuštena budući da ona ne utječe na maksimizaciju, te je zadnja navedena formula upravo klasifikator koji je potrebno naučiti.

`NB.fit(self, trainingData, trainingLabels):`

Unutar metode *fit* našeg klasifikatora, potrebno je upravo naučiti izglednosti (uvjetne vjerojatnosti svake značajke) $P(f_j|h_i)$ te a priori vjerojatnosti za svaku hipotezu $P(h_i)$ na temelju ulaznih podataka. Naučene vrijednosti potrebno je spremiti u varijable klase `self.prior` i `self.conditionalProb`. Također, potrebno je implementirati Laplaceovo zaglađivanje s faktorom k koji je dan kao hiperparametar algoritma. Za $k = 0$ nema zaglađivanja.

Ulazni podaci u metodu fit su dvije liste – *trainingData* i *trainingLabels* koje sadrže niz opažanja u skupu za treniranje. Pod opažanjima smatramo uređene parove (značajke, oznaka), gdje za skup značajki znamo točno koja je ciljna oznaka (odluka).

TrainingData se sastoji od liste riječnika *(engl. dictionary)*, u kojemu su ključevi nazivi značajki (npr. foodEaten_East), dok su vrijednosti upravo vrijednosti tih značajki za to specifično opažanje (npr. False). TrainingLabels se sastoji od niza stringova koji reprezentiraju donešene odluke (npr. West).

Hint: Primjetite da je nužno pratiti koliko se puta pojavila koja vrijednost svake značajke (koliko puta se pojavio foodEaten_East=True za ciljnu odluku West).

Hint: unutar varijable klase `self.featureValues` nalaziti će se sve moguće vrijednosti značajki za svaku značajku. Dakle, featureValues je riječnik kojemu su ključevi nazivi značajki, a vrijednosti setovi svih mogućih vrijednosti te značajke.

`NB.calculateJointProbabilities(self, instance):`

Unutar metode *calculateJointProbabilities* potrebno je izračunati **a posteriori** vjerojatnosti za svaku moguću odluku (hipotezu). Navedenu metodu ulančano poziva metoda *predict*, u kojoj se na temelju a posteriori vjerojatnosti za svaku moguću odluku odabire točna, a vjerojatnosti spremaju radi testiranja.

Ulaz u metodu je jedno testno opažanje *instance* u obliku riječnika, kojemu su ključevi imena značajki, a vrijednosti vrijednosti tih značajki. Izlaz iz metode treba biti riječnik (podklasa util.Counter), koji za ključ svake moguće hipoteze (`self.legalLabels`) sadrži njenu a posteriori vjerojatnost.

`NB.calculateLogJointProbabilities(self, instance):`

Metoda *calculateLogJointProbabilities* obavlja istu funkcionalnost računanja a posteriori vjerojatnosti za svaku odluku, no radi to s logaritamskim trikom radi numeričke stabilnosti. Vaš zadatak je implementirati računanje a posteriori vjerojatnosti s logaritamskim trikom, te provjeriti možete li zaključiti koji hardkodirani rezultati u razredu *classifier.py* pripadaju modelu s log transformacijom, a koji pripadaju onom bez nje. Razmislite i argumentirajte (prvo sebi, a potom i nama na predaji vježbe) zašto je logaritamska transformacija ovdje dobar izbor.

Ulazi i izlazi su jednaki kao u metodi *calculateJointProbabilities*.

Nakon ispravne implementacije svih metoda, vaš klasifikator bi trebao prolaziti sve testove, te dobivati iduće performanse na skupu podataka *contest*:

```
Performance on train set for k=0.000000, log=False: (78.8%)
Performance on test set for k=0.000000, log=False: (82.2%)
Performance on train set for k=0.000000, log=True: (78.8%)
Performance on test set for k=0.000000, log=True: (82.2%)
Performance on train set for k=1.000000, log=True: (78.7%)
Performance on test set for k=1.000000, log=True: (82.0%)
```

Testovi, kao i provjera performanse vašeg modela na nekom skupu podataka pokreću se kroz glavnu metodu koja se nalazi u datoteci `classifier.py`, za koju su upute dostupne u nastavku:

```
USAGE:      python classifier.py <options>
EXAMPLES: >> python classifier.py -t 1
              // runs the implementation tests on the 'contest' dataset
          >> python classifier.py --train contest_training --test contest_test -s 1 -l 1
              // run the naive Bayes classifier on the
                contest dataset with laplace smoothing=1 and log scale transformation
          >> python classifier.py -h
              // display the help docstring

Options:
  -h, --help            show this help message and exit
  --train=TRAIN_DATA    the TRAINING DATA for the model [Default:
                        stop_training]
  --test=TEST_DATA      the TEST DATA for the model [Default: stop_test]
  -s SMOOTHING, --smoothing=SMOOTHING
                        Laplace smoothing [Default: 0]
  -l LOGTRANSFORM, --logtransform=LOGTRANSFORM
                        Compute a log transformation of the joint probability
                        [Default: 0]
  -t RUNTESTS, --test_implementation=RUNTESTS
                        Disregard all previous arguments and check if the
                        predicted values match the gold ones
```

Zadatak 2: Potporno Učenje (12 bodova)

Nakon raspakiranja zip arhive te pozicioniranja u poddirektorij za potporno učenje, opis datoteka i direktorija koje ćete vidjeti je u nastavku:

Datoteke koje ćete uređivati:	
valueIterationAgents.py	Logika algoritma iteracije vrijednosti
qlearningAgents.py	Logika algoritama baziranih na Q-učenju
Datoteke koje trebate proučiti:	
mdp.py	Opis općenitog Markovljevog procesa odlučivanja
learningAgents.py	Logika koja opisuje agente
util.py	Korisne pomoćne strukture i metode
gridworld.py	Opis GridWorlda
featureExtractors.py	Datoteka koja ekstrahira značajke za Q-stanja
Pomoćne datoteke koje možete ignorirati:	
environment.py	"Apstraktna klasa" za probleme potpornog učenja
graphicsUtils.py	Pomoćne funkcije za grafičko sučelje
graphicsGridworldDisplay.py	Grafički prikaz GridWorlda
crawler.py	Logika "puzača"
graphicsCrawlerDisplay.py	Grafički prikaz puzača
autograder.py	Pomoćna datoteka koja pokreće testove

Kod drugog zadatka vježbe je preuzet s predmeta "Uvod u umjetnu inteligenciju" na Berkeleyu, koji je velikodušno ustupio funkcionalno okruženje drugim fakultetima na korištenje u njihovim predmetima. Kod je napisan u Pythonu 2.7, koji za pokretanje laboratorijske vježbe trebate instalirati.

Pri rješavanju drugog dijela laboratorijske vježbe imate dostupan *autograder* - pokretanjem naredbe `python autograder.py` možete okvirno provjeriti ispravnost vaših rješenja. Ukoliko želite pokrenuti autograder za neko specifično pitanje, možete to napraviti pomoću argumenta -q, kao u idućem primjeru: `python autograder.py -q q1`.

Nakon skidanja i raspakiravanja koda te pozicioniranja konzolom u direktorij u kojemu ste raspakirali sadržaj arhive, te u poddirektorij za potporno učenje – možete isprobati igru s ručnim upravljanjem pomoću strelica upisujući:

```
python gridworld.py -m
```

GridWorld je svijet u kojemu vam je kretanje otežano na način da imate šansu od 80% za napraviti akciju koju ste odabrali, te šansu od 20% za nasumičnu akciju. Vjerojatnost nasumičnog kretanja (šum, engl. noise) možete podesiti pomoću parametra '-n (noise)', na primjer:

```
python gridworld.py -m -n 0.5
```

S čime će šansa nasumičnog kretanja biti 50%. Ovo, kao i sve druge parametre možete vidjeti pozivom parametra '-h (help)'. Ispis je u nastavku:

```
Options:
  -h, --help            show this help message and exit
  -d DISCOUNT, --discount=DISCOUNT
                        Discount on future (default 0.9)
```

```
  -r R, --livingReward=R
                        Reward for living for a time step (default 0.0)
  -n P, --noise=P       How often action results in unintended direction
                        (default 0.2)
  -e E, --epsilon=E     Chance of taking a random action in q-learning
                         (default 0.3)
  -l P, --learningRate=P
                        TD learning rate (default 0.5)
  -i K, --iterations=K  Number of rounds of value iteration (default 10)
  -k K, --episodes=K    Number of epsiodes of the MDP to run (default 1)
  -g G, --grid=G        Grid to use (case sensitive; options are BookGrid,
                        BridgeGrid, CliffGrid, MazeGrid, default BookGrid)
  -w X, --windowSize=X  Request a window width of X pixels *per grid cell*
                        (default 150)
  -a A, --agent=A       Agent type (options are 'random', 'value' and 'q',
                        default random)
  -t, --text            Use text-only ASCII display
  -p, --pause           Pause GUI after each time step when running the MDP
  -q, --quiet           Skip display of any learning episodes
  -s S, --speed=S       Speed of animation, S > 1.0 is faster, 0.0 < S < 1.0
                        is slower (default 1.0)
  -m, --manual          Manually control agent
  -v, --valueSteps      Display each step of value iteration
```

Kao što smo obradili na predavanjima, problem koji rješavamo u GridWorldu i sličnim problemima je optimizacija ukupne korisnosti (engl. utility) kroz kretanje po mapi. Ono što vam otežava kretanje po mapi je šum pri kretanju - akcije koje odaberete vas ne moraju nužno voditi na ista mjesta. Ovo se u okviru GridWorlda manifestira na način da imate fiksnu šansu $(1 - n)$ za kretati se u smjeru koji ste odabrali, te šansu n za kretati se nasumično.

Kao i inače, pozicije su definirane pomoću koordinata (x,y), dok su prijelazi između pozicija definirani pomoću strana svijeta (south, east, north, west).

Laboratorijska vježba se sastoji od četiri podzadatka, od kojih svaki nosi po pet bodova. Ukupna suma od 20 konačnih bodova će se potom skalirati na 6.25.

Problem 2.1: Iteracija vrijednosti (3 boda)

U datoteci valueIterationAgents.py nadopunite kod za iteraciju vrijednosti. Podsjetimo se, iteracija vrijednosti je algoritam koji pokušava izračunati vrijednosti svakog stanja na mapi pomoću rekurzivne formule za vrijednost stanja:

$$V_0(s) = 0$$
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')]$$

Kao parametar u konstruktoru vašeg *ValueIterationAgent*-a dobivate argument *mdp*, tj. definiciju markovljevog procesa odlučivanja (u slučaju GridWorlda, to je implementacija apstraktne klase *mdp.MarkovDecisionProcess* se nalazi u *gridworld.py*. Metode koje opisuju markovljev proces odlučivanja su:

`mdp.getStates()`:

Vraća listu svih mogućih stanja (list(tuple(int, int))) u problemu

`mdp.getPossibleActions(state):`

Vraća listu svih mogućih akcija (list(str)) za stanje state

`mdp.getTransitionStatesAndProbs(state, action):`

Vraća listu parova (tuplea) koji se sastoje od elemenata (idućeStanje : tuple(int, int), vjerojatnostPrijelaza: float)

`mdp.getReward(state, action, nextState):`

Vraća float preciznost vrijednost funkcije nagrade za prijelaz iz stanja state uz akciju a u stanje nextState

`mdp.isTerminal(state):`

Provjerava je li neko stanje konačno (vraća True ili False). Konačna stanja nemaju prijelaza (dakle lista prijelaza može biti prazna)!

Metode koje trebate dovršiti su: *__init__*, *computeQValueFromValues*, *computeActionFromValues*. Kako imate preddefinirane vrijednosti prijelaza i nagrada, većina računanja bi trebala biti izvršena u inicijalizaciji. Pri rješavanju koristite prethodno inicijalizirani razred *util.Counter*, koji je zapravo implementacija Pythonovog riječnika (engl. dictionarya), s početnim vrijednostima postavljenim na nulu - dakle, ne morate inicijalizirati vrijednosti za svako stanje.

Opaska 1.1 Pazite na to da pri ažuriranju vrijednosti koristite pomoćnu strukturu u koju ćete spremati međurezultat - u koraku $k+1$ iteracije vrijednosti V_{k+1} ovise samo o vrijednostima iz koraka V_k - pazite da ne koristite vrijednosti koje ste ažurirali u tom koraku.

Kako biste testirali implementaciju, rezultat pokretanja iduće naredbe:

```
python gridworld.py -a value -i 5
```

bi vam trebao izgledati kao na idućoj slici:

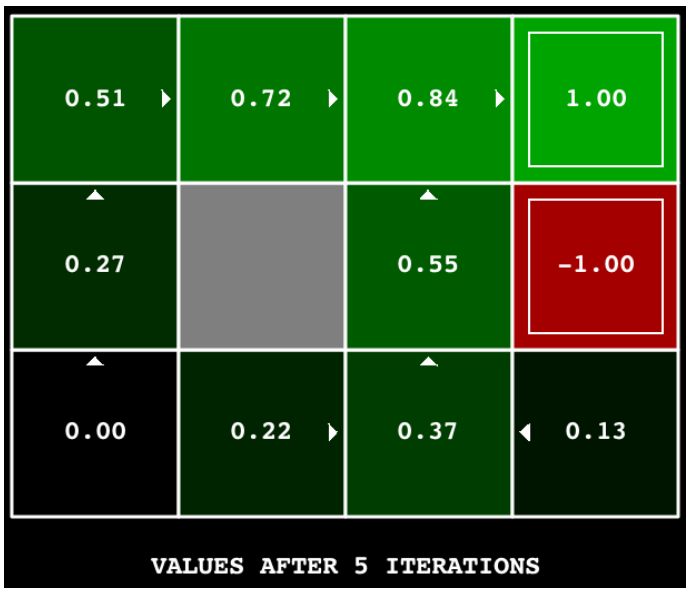
Problem 2.2: Q-učenje (3 boda)

U datoteci qlearningAgents.py nadopunite kod za algoritam Q-učenja. Podsjetimo se, algoritam Q-učenja radi na principu pokretnog prosjeka (engl. running average), te formula za ažuriranje izgleda kao u nastavku:

$$Q_0(s,a) = 0$$
$$Q_{k+1}(s,a) \leftarrow Q_k(s,a) + (\alpha)[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') - Q_k(s,a)]$$

Pri čemu je α stopa učenja (engl. learning rate), dok je γ faktor propadanja (engl. decaying factor).

Metode koje trebate nadopuniti u razredu QLearningAgent su: *__init__* , *getQValue*, *computeValueFromQValues*, *computeActionFromQValues*, *getAction* i *update*. Razred QLearningAgent ima varijable koje nasljeđuje od ReinforcementAgenta, koje eksplicitno



Slika 1: Primjer izvođenja iteracije vrijednosti

ne vidite iako postoje. Varijable koje su nasljeđene, a trebaju vam su: *self.epsilon* - vjerojatnost za istraživanje u epsilon-greedy pristupu (Tek u zadatku 3.), *self.alpha* - stopa učenja, te *self.discount* - stopa propadanja. Sve vrijednosti su u float preciznosti.

Funkcija koja je dostupna kroz nasljeđivanje, iako ju eksplicitno ne vidite je `self.getLegalActions(state)`, koja vam za predano stanje vraća listu mogućih akcija.

Opaska 2.1 Pripazite na činjenicu da ako u nekom stanju Q-stanje koje ste jedino istražili ima negativnu vrijednost, optimalan potez je jedan od onih koji još niste istražili! Ti potezi ne moraju nužno biti inicijalizirani u vašem riječniku (Counteru). Ukoliko imate više Q-stanja s istim vrijednostima, za optimalan potez odaberite bilo koje od njih s uniformnom vjerojatnošću. Za ovo je korisna funkcija `random.choice()`, koja za argument prima niz elemenata i vraća jedan od njih s uniformnom vjerojatnošću.

Opaska 2.2 Pripazite na to da kad god pristupate Q-vrijednostima, koristite metodu `getQValue` budući da će vam to olakšati rješavanje približnog Q-učenja.

Opaska 2.3 Kao ključeve vašeg riječnika (dictionary / Counter) možete koristiti i tupleove (parove stanje, akcija).

Problem 2.3: ϵ-pohlepan pristup (3 boda)

Modificirajte vaš algoritam Q-učenja pomoću implementacije ϵ-pohlepnog pristupa (engl. ϵ-greedy). Podsjetimo se, ϵ-pohlepnost znači da s malom vjerojatnosti ϵ odabirete nasumičnu akciju, koja može biti bilo koja od legalnih akcija, dok s vjerojatnošću $1 - \epsilon$ odabirete trenutno optimalnu akciju.

Vrijednost ϵ vam je dostupna kao varijabla razreda *self.epsilon*, dok bi vam se korisnom trebala pokazati metoda util.flipCoin(p), koja vraća True s vjerojatnošću p, te False s vjerojatnošću $1 - p$.

Bez ikakvih dodatnih izmjena u kodu nakon dodavanja ϵ-pohlepnog pristupa, provjerite kako vam funkcionira puzač (engl. crawler) pokretanjem idućeg programa:

```
python crawler.py
```

Problem 2.4: Približno Q-učenje (3 boda)

Implementirajte približno Q-učenje u datoteci qlearningAgents.py u razredu *ApproximateAgent*.

Prisjetimo se, približna Q-funkcija je u obliku:

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a)w_i$$

Pri čemu je W vektor težina, dok je $f(s,a)$ vektor značajki za Q-stanje (s,a), a svaka težina w_i iz vektora se mapira na jednu značajku. Funkcije koje će računati značajke su već definirane i implementirane u datoteci *featureExtractors.py*, te varijabla razreda *self.featExtractor* koja vam je dostupna na poziv metode *getFeatures(state, action)* vraća Counter značajki za to Q-stanje.

U početku postupka težine možete inicijalizirati na 0 (kako je ovo najlakše riješiti? hint:Counter), te ih potom ažurirajte po jednadžbi s predavanja:

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a)$$
$$difference = (r + \gamma \max_{a'} Q(s',a')) - Q(s,a)$$

Ukoliko vaša implementacija radi, bez problema bi trebali pobjeđivati iduću mapu:

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor
-x 50 -n 60 -l mediumGrid
```

Dok bi s (moguće negdje do minutu treniranja), vaš agent trebao bez problema pobjeđivati i nešto veću mapu:

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumC
```