Umjetna inteligencija – Laboratorijska vježba 1

UNIZG FER, ak. god. 2016/17.

Zadano: 13.3.2016. Rok predaje: 26.3.2017. do 23.59 sati.

Uvod

U ovoj laboratorijskoj vježbi pomoći ćete agentu Pacmanu pronaći put do cilja kroz njegov svijet labirinta. Rješavat ćete dva tipa problema s kojima će se Pacman susresti na svojem putu, jedan od kojih je pronalazak puta do ciljne lokacije, a drugi efikasno skupljanje hrane. Implementirat ćete algoritme pretraživanja te heuristike i primjeniti ih na scenarije iz Pacmanovog svijeta.

Kod koji opisuje ponašanje Pacmanovog svijeta već je napisan i funkcionalan, te je na vama samo zadatak implementirati algoritme koji će kontrolirati Pacmanovo kretanje. Kod koji ćete koristiti možete skinuti kao zip-arhivu na stranicama fakulteta ovdje, ili na github repozitoriju predmeta ovdje.

Kod za projekt sastoji se 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. Nakon raspakiranja zip arhive, opis datoteka i direktorija koje ćete vidjeti je u nastavku:

Datoteke koje ćete uređivati:	
search.py	Datoteka koja sadrži algoritme pretraživanja
searchAgents.py	Datoteka koja sadrži agente bazirane na algoritmima pretraživanja
Datoteke koje trebate proučiti:	
pacman.py	Glavna datoteka koja pokreće igru
game.py	Logika iza Pacmanovog svijeta
util.py	Korisne strukture podataka za implementaciju algoritama pretraživanja
Pomoćne datoteke koje možete ignorirati:	
graphicsDisplay.py	Pacmanovo grafičko sučelje
graphicsUtils.py	Pomoćne funkcije za grafičko sučelje
textDisplay.py	ASCII grafike za Pacmana
ghostAgents.py	Agenti koji kontroliraju duhove
keyboardAgents.py	Sučelje za kontroliranje Pacmana pomoću tipkovnice
autograder.py	Pomoćna datoteka koja pokreće testova
testParser.py	Parsiranje testova
testClasses.py	Automatsko ocjenjivanje testova (Berkeley)
test_cases/	Direktorij koji sadrži primjere testova
searchTestClasses.py	Testovi vezani za prvu laboratorijsku vježbu (Berkeley)

Kod laboratorijskih vježbi 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.

Nakon skidanja i raspakiravanja koda te pozicioniranja konzolom u direktorij u kojemu ste raspakirali sadržaj arhive, možete isprobati igru upisujući:

```
python pacman.py
```

No, umjetna inteligencija se ne može oslanjati na ljudsku kontrolu - Pacman mora biti sposoban samostalno i efikasno se boriti sa svim problemima koji mu se nađu na putu do hrane! Naivan primjer umjetne inteligencije je kretanje uvijek u istom smjeru neovisno o izgledu mape, što možete provjeriti pokretanjem iduće naredbe:

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

U ovom slučaju, kao argumente smo proslijedili mapu *testMaze* i logiku koja Pacmana stalno tjera u smjeru zapada zvanu *GoWestAgent*. No, kao što možete i pretpostaviti, ovaj pristup naiđe na probleme kad je potrebno bilo kakvo skretanje, što je evidentno iz idućeg primjera:

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

Kroz ovu laboratorijsku vježbu, omogućit ćete Pacmanu da navigira ne samo kroz prethodna dva labirinta, već i svakim drugim s kojim se suoči. Igra ima mnogo opcija, te vam preporučamo da ih proučite pomoću:

```
python pacman.py -h
```

Sve naredbe koje će se referencirati kroz upute za laboratorijsku vježbu će biti prisutne u tekstualnoj datoteci *commands.txt*. Ukoliko ste na operacijskom sustavu zasnovanom na UNIX-u, sve naredbe je moguće pokrenuti pomoću *bash commands.txt*.

Zbog jedne poznate greške, koja je uspješno reproducirana samo na prijenosnim računalima s operacijskim sustavu Ubuntu 14.04., animacije tijekom igre su deaktivirane. Greška se manifestira na način da se nakon pokretanja igre trenutačni korisnik odjavi, te svi aktivni programi zagase. Ukoliko želite isprobati radi li Pacman na vašem računalu s punim animacijama, savjetujemo vam da spremite sve što trenutno radite na računalu u slučaju eventualne greške u radu sustava. Pokretanje radite na vlastitu odgovornost.

Za aktiviranje animacija, u datoteci *graphicsUtils.py* potrebno je maknuti komentar (#) ispred linije u 218. retku, čiji je sadržaj *edit(id, ('start', e[0]), ('extent', e[1] - e[0]))*. Ukoliko nakon te promjene uspijete pokrenuti i odigrati osnovnu verziju Pacmana, problem se neće reproducirati na vašem računalu. Ukoliko vam se problem reproducira, molimo vas da se javite mailom kako bismo spremili konfiguraciju vašeg računala za buduće godine.

Laboratorijska vježba se sastoji od osam podzadataka, te implementacija svakog od njih nosi jednaku težinu. Kako bi olakšali ocjenjivanje, svaki od podzadataka nosi po tri boda, s ukupnom sumom od 24. Vaši konačni ostvareni bodovi će biti skalirani, te je stvarni maksimum bodova za prvu laboratorijsku vježbu 6.25.

Problem 1: Pronalazak hrane pomoću pretraživanja u dubinu

U datoteci searchAgents.py pronaći ćete potpunu implementaciju agenta za pretraživanje (*SearchAgent*), koji prvo isplanira put kroz Pacmanov svijet, i tek onda provede taj put korak po korak. Algoritmi pretraživanja koji formuliraju plan nisu implementirani i to je vaš zadatak. Prvo, provjerite radi li agent za pretraživanje kako spada pomoću iduće naredbe:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

Naredba agentu za pretraživanje predaje algoritam pretraživanja *tinyMazeSearch*, čija se implementacija nalazi u search.py. Pregledom metode ćete primjetiti da je ona uspješna samo za specifičnu zadanu mapu – sad je vrijeme da napišete kod koji će pronalaziti rješenje neovisno o mapi!

Pseudokod za algoritme pretraživanja možete naći u predavanjima - držite na pameti da osim pronalaska puta do ciljnog stanja trebate biti sposobni i reproducirati taj put kad ga pronađete. Čvor pretraživanja shodno tome treba sadržavati ne samo informacije o stanju, već i o putu do tog stanja. Također, pri implementaciji koristite strukturu s kojom ćete pamtiti posjećena stanja kako bismo osigurali potpunost.

Opaska 1. Sve vaše metode moraju vraćati niz valjanih poteza koji vode Pacmana od početka do cilja. Valjanost u ovom slučaju znači da Pacman ne može ići do cilja kroz zid.

Opaska 2. Pri implementaciji algoritama pretraživanja koristite razrede *Stack*, *Queue* te *PriorityQueue* s obzirom na to da se na njih oslanja automatsko testiranje.

Uputa 1. Laboratorijska vježba se može riješiti ispunjavanjem samo označenih metoda u kodu - no takvim načinom rješavanja ćete imati mnogo duplikacije koda. Kako su algoritmi DFS, BFS, UCS i A* različiti samo u načinu odabira idućeg čvora za proširenje, razmislite možete li implementirati samo jednu općenitu metodu koja se može prilagoditi za rad sa svim algoritmima. Vaša implementacija **ne mora** biti u ovom formatu kako biste dobili maksimalne bodove!

Implementirajte algoritam pretraživanja u dubinu (DFS) u metodi *depthFirstSearch* u datoteci *search.py*. Vaš kod bi trebao bez problema riješiti iduće labirinte:

```
python pacman.py -l tinyMaze -p SearchAgent

python pacman.py -l mediumMaze -p SearchAgent

python pacman.py -l bigMaze -z .5 -p SearchAgent
```

Pacmanov labirint će prikazivati u kojem trenutku ste istražili koju poziciju - jača crvena boja označava ranije istraživanje, dok su neistražena stanja crna. Je li redoslijed istraživanja stanja kakvim ste ga i očekivali? Prolazi li Pacman kroz sva istražena stanja na putu do cilja? Može li broj istraženih stanja u vašem rješenju varirati i ako da, o čemu taj broj ovisi?

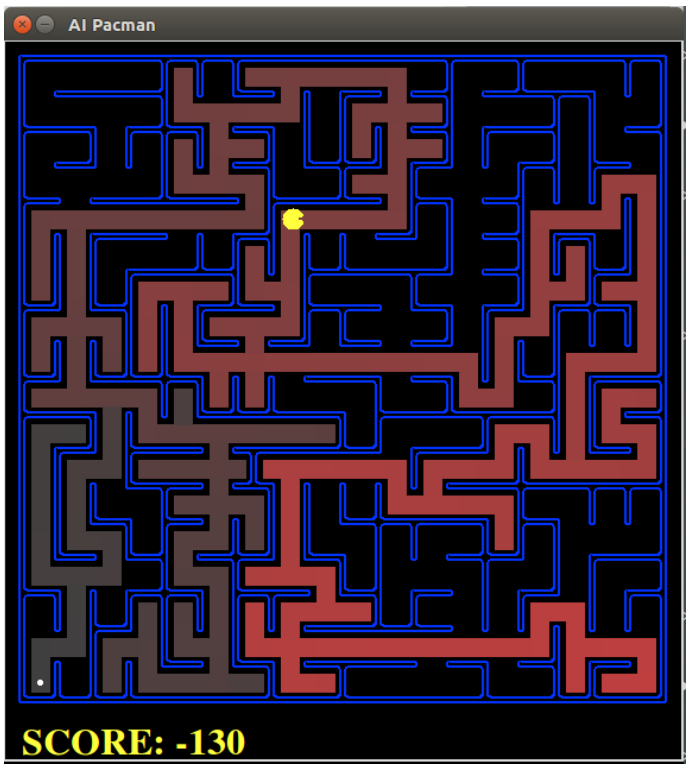
Primjer istraživanja za labirint *bigMaze* je na slici 1.

Problem 2: Pretraživanje u širinu

Implementirajte algoritam pretraživanja u širinu (BFS) u metodi *breadthFirstSearch* u datoteci *search.py*. Ponovno obratite pozornost na to da pratite posjećena stanja kako ih ne biste ponovno posjećivali. Isprobajte kod na isti način kao i DFS, pomoću idućih naredbi:

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```



Slika 1: Primjer pretraživanja u dubinu

Pronalazi li BFS rješenje s najmanjim putem? Ako ne, moguće je da imate grešku u vašem rješenju.

Ako se Pacman kreće presporo za vaš ukus, tu je Pacmanov alter ego PacFlash za kojeg trebate uključiti opciju *–frameTime 0*, kao u idućem primjeru:

```
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5 --frameTime 0
```

Ako ste riješili problem na dovoljno općenit način, vaš algoritam za pretraživanje u širinu bi trebao rješavati i problem slagalice (8-puzzle).

```
python eightpuzzle.py
```

Problem 3: Pretraživanje s jednolikom cijenom

Iako će BFS pronaći optimalan put do cilja, ponekad želimo pronaći puteve koji su najbolji u nekom drugom smislu - primjeri ovog su *mediumDottedMaze* i *mediumScaryMaze*.

Primjerice, cijena straha od prolaza kroz područja kojima vrve duhovi je sigurno veća od mirnih područja, dok su područja bogata hranom sigurno bolja od hladnih i mračnih zidova labirinata, te svaki racionalni Pacman prilagodi svoj put tim situacijama.

Implementirajte algoritam za pretraživanje s jednolikom cijenom (UCS) u metodi *uniformCostSearch* u *search.py*. Trebali biste bez problema riješiti svaki od tri iduća primjera, koji se razlikuju samo u funkciji cijene koju koriste. Funkcije cijene su predefinirane i u sva tri labirinta se koristi UCS.

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
```

```
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

Veoma niska i visoka cijena puta su očekivane za zadnja dva primjera zbog eksponencijalnih funkcija cijene. Ako vas zanima više, kod se nalazi u *searchAgents.py*.

Problem 4: Algoritam A*

Implementirajte algoritam A* u metodi *aStarSearch* u datoteci *search.py*. A* kao argument prima heuristiku, dok heuristike primaju dva argumenta - stanje u problemu pretraživanja (glavni argument), te problem (za referenciranje). Heuristika *nullHeuristic* u datoteci *search.py* je trivijalan primjer.

Možete isprobati vašu implementaciju algoritma A* na originalnom problemu pronalaska puta kroz labirint pomoću heuristike Manhattan udaljenosti (engl. city block distance) koja je već implementirana kao *manhattanheuristic* unutar datoteke *searchAgents.py*.

```
python pacman.py -l bigMaze -z .5 -p SearchAgent
                                    -a fn=astar,heuristic=manhattanHeuristic
```

(Prepišite naredbu u jednom retku.)

Rješenje pomoću algoritma A* bi trebalo biti marginalno brže od pretraživanje s jednolikom cijenom (UCS), pri čemu algoritam A* pronalazi rješenje s istraživanjem 549 čvorova, dok UCS istraži 620. Brojevi mogu varirati ovisno o načinu razrješavanja jednakih cijena. Isprobajte implementacije vaših algoritama pretraživanja na labirintu *openMaze*. Što možete zaključiti iz rezultata?

Problem 5: Prolazak kroz sve uglove labirinta

Ovaj problem se nadograđuje na problem 2

Prava moć algoritma A* možda nije evidentna iz jednostavnih problema - zato ćemo formulirati novi problem i dizajnirati adekvatnu heuristiku za njega. U *uglatim* labirintima postoje četiri hranjive točke, po jedna u svakom uglu labirinta. Naš novi problem je pronalazak najkraćeg puta kroz labirint koji uključuje sva četiri ugla. Držite na umu da za neke labirinte poput *tinyCorners* najkraći put ne prolazi uvijek prvo kroz najbližu hranjivu točku! (Za provjeru, duljina najkraćeg puta za *tinyCorners* je 28 koraka.)

Za početak trebamo formulirati problem uglatih labirinata. Implementirajte *CornersProblem* problem pretraživanja u *searchAgents.py*. Potrebno je odabrati reprezentaciju stanja koje sadrži sve potrebne informacije kako bismo mogli znati da li su sva četiri ugla bila posjećena. Vaš ispravan agent bi trebao bez problema rješavati iduće probleme:

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

Vaša implementacija stanja ne smije uključivati nebitne informacije poput lokacija duhova, dodatne hrane itd. Ni u kojem slučaju nemojte koristiti razred *GameState* kao stanje jer će vaš kod biti iznimno spor (i pogrešan).

Uputa 2. jedini dijelovi stanja igre koji su vam potrebni za implementaciju su početna Pacmanova lokacija i lokacije sva četiri ugla.

Uz ispravnu implementaciju, BFS rješava problem s istraživanjem manje od 2000 čvorova na problemu *mediumCorners*. No, heurističko pretraživanje može dodatno smanjiti broj istraženih čvorova.

Problem 6: Heuristika za problem uglatih labirinta

Ovaj problem se nadograđuje na problem 4

Implementirajte netrivijalnu, konzistentnu heuristiku za *CornersProblem* u metodi *cornersHeuristic*. Isprobajte vašu implementaciju s:

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

AstarCornersAgent je kratica za -p SearchAgent -a fn=aStarSearch, prob=CornersProblem, heuristic=cornersHeuristic.

Prisjetite se svojstava koje heuristike moraju zadovoljavati da bi bile dobre. Obratite pozornost da vaša heuristika mora biti jednaka nuli na svakom ciljnom stanju i nikad ne smije vraćati negativnu vrijednost. Efikasnost heuristike se može procijeniti ovisno o broju istraženih čvorova.

U ovom slučaju, vaša heuristika bi trebala biti bolja od pretraživanja u širinu. Vaši bodovi na ovome zadatku će ovisiti o broju istraženih čvorova i njihov maksimum koji možete ostvariti prije ispitivanja je kao u sljedećoj tablici:

Prošireni čvorovi	Max. bodovi
> 2000	0/3
≤ 2000	1/3
≤ 1600	2/3
≤ 1200	3/3

Problem 7: Nezasitni lijeni Pacman

Ovaj problem se nadograđuje na problem 4

U ovom problemu ćemo malo otežati stvari i riješiti kompliciran problem pretraživanja - Pacman treba pojesti svu hranu u što manje koraka. Za ovo, potrebna će biti nova definicija problema pretraživanja koji formalizira problem jedenja svih hranjivih točaka - razred *FoodSearchProblem* u datoteci *searchAgents.py*. Ovaj razred je već implementiran za vas, a rješenje je put koji skuplja svu hranu u Pacmanovom labirintu. Za trenutnu vježbu, problemi ne uključuju duhove niti kuglice snage, već samo ovise o zidovima, hrani i Pacmanu. Ukoliko ste ispravno implementirali A* pretraživanje, ono bi trebalo naći optimalno rješenje s cijenom 7 koristeći *nullHeuristic* na idućem problemu:

```
python pacman.py -l testSearch -p AStarFoodSearchAgent
```

AStarFoodSearchAgent je kratica za -p SearchAgent -a fn=astar, prob=FoodSearchProblem, heuristic=foodHeuristic.

Iako smo prethodni problem riješili brzo i jednostavno, stvari će postati kompliciranije već s malim labirintima poput *tinySearch*. Za pronalazak rješenja s cijenom 27 na idućoj stazi pomoću algoritma UCS bilo je potrebno 2.5 sekundi!

```
python pacman.py -l tinySearch -p SearchAgent -a fn=ucs,prob=FoodSearchProblem
```

Vaš zadatak je ispuniti heuristiku *foodHeuristic* u datoteci *searchAgents.py* s konzizstentnom heuristikom za *FoodSearchProblem*. Isprobajte vašu implementaciju na labirintu *trickySearch*:

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

UCS algoritam pronalazi optimalan put za 13 sekundi uz istraživanje preko 16000 čvorova.

Sve netrivijalne strogo pozitivne heuristike vam osiguravaju 1 bod. Ovisno o uspješnosti vaše heuristike, dobivat ćete bodove kao u sljedećoj tablici:

Prošireni čvorovi	Max. bodovi
> 15000	1/3
≤ 15000	2/3
≤ 12000	3/3
≤ 7000	4/3

Obratite pozornost na to da vaša heuristika **mora** biti konzistentna, inače nećete dobiti bodove na ovome dijelu vježbe!

Problem 8: Suboptimalno pretraživanje

Ponekad, čak i uz A* i dobru heuristiku, pronalazak optimalnog puta kroz svu hranu nije jednostavan. U nekim slučajevima, zadovoljni smo čak i s dovoljno dobrim putem koji nađemo brzo. U ovom problemu, implementirat ćete Pacmana koji će uvijek pohlepno pojesti trenutno najbližu hranu. Razred *ClosestDotSearchAgent* je već implementiran u *searchAgents.py*, no nedostaje metoda koja pronalazi put do najbliže hrane.

Implementirajte metodu *findPathToClosestDot* u datoteci searchAgents.py. Vaše rješenje bi trebalo riješiti problem *bigSearch* u manje od sekundu s ukupnom cijenom puta 350.

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

Uputa 3. Najbrži način za dovršavanje metode *findPathToClosestDot* je nadopunjavanjem koda u razredu *AnyFoodSearchProblem*, kojemu nedostaje provjera cilja. Potom, riješite taj problem s odgovarajućim algoritmom pretraživanja. Vaše rješenje bi trebalo zahtjevati jako malo dodanih linija koda.