Документација за напреден дел по предметот „Вештачка интелигенција”

(Изработил:Драгана Трифунова [223044])

```
 *** DOBREDOJDOVTE VO IGRATA DAMA ***
Ova e implementacijata na studentot Dragana Trifunova [223044] za igrata dama i ovde e prikazana amerikanskata verzija od igrata
Vo Amerikanskata verzija postojat 12 pioncinja na sekoi od protivnicite i vo sekoi red ima po osum kelii

PRAVILA NA IGRANJE
1. Vie igrate so pionceto oznaceno kako '#men'
2. Dokolku sakate da zapocnete so igranje napisete 'start'
3. Dokolku sakate da prekinete so igranje vo bilo koe vreme pritisnete 'Enter'
4. Dokolku sakate da se predadete napisete 'surrender'
5. Koordinatite 'i' i 'j' gi vnesuvate so zapirka pomegu niv bez prazno mesto
start
       0       1       2       3       4       5       6       7

0    ----00  comp01  ----02  comp03  ----04  comp05  ----06  comp07

1    comp10  ----11  comp12  ----13  comp14  ----15  comp16  ----17

2    ----20  comp21  ----22  comp23  ----24  comp25  ----26  comp27

3    ----30  ----31  ----32  ----33  ----34  ----35  ----36  ----37

4    ----40  ----41  ----42  ----43  ----44  ----45  ----46  ----47

5    #men50  ----51  #men52  ----53  #men54  ----55  #men56  ----57

6    ----60  #men61  ----62  #men63  ----64  #men65  ----66  #men67

7    #men70  ----71  #men72  ----73  #men74  ----75  #men76  ----77
```

Слика 1: Почетен интерфејс на играта

Во мојот напреден дел јас имплементирав веќе постоечка игра која се вика Дама. Оваа игра се базира врз основа на minimax алгоритмот. И овде морам да нагласам дека во овој алгоритам иако најчесто се зема дека компјутерот е максимален играч а, човекот минимален овде, во мојата игра компјутерот е минимален играч. Причината за ова е што при истражувањето на интернет за оваа проектна задача се послужив со такви имплементации каде компјутерот беше минимален играч.

Судејќи според ова, верувам дека веќе претпоставувате дека првиот потег го прави човекот, вториот компјутерот и така најизменично. Нормално на ова евристиката е направена во корист на човекот односно кога човекот има добра положба евристиката враќа повеќе поени а, кога човекот има лоша положба евристиката враќа помалку поени. Целта на компјутерот е да ја минимизира својата штета односно да ја избере положбата во која човекот ќе се чувствува најлошо. Човекот зема максимум поени од евристиката а, компјутерот зема минимум.

Пред да почнеме со објаснувањето на секој метод поединечно сакам да ве наведам во тоа дека човекот игра со пионите означени како „#men“ а, компјутерот со пионите означени како „comp“.

Во оваа имплементација на играта користени се две класи:
- NewGameState ова е класа која ја означува секоја нова положба на таблата. Се користи за да компјутерот открие кој потег треба да го преземе. Со помош на оваа класа и методот во неа „get_children’’ компјутерот знае и ги тестира кои се можните потези на него и на човекот. Тој креира нова табла (матрица) за сите положби на него или на човекот и потоа со помош на хевристиката во minimax алгоритамот им задава поени на тие табли.

- GameCheckers од оваа класа се креира само една истанца со помош на која се игра играта.
Оваа класа содржи: матрица односно таблата на играта, содржи булова променлива за тоа дали е на ред човекот или компјутерот иницијално означена како True затоа што прв започнува човекот, има две променливи една за поените на компјутерот а, другата за поени на играчот, има листа за можни поместувања на човекот или компјутерот. Понатаму, оваа класа содржи број на пиони на човекот и на компјутерот иницијализирани на 12 затоа што ја играме американската верзија.
Во последниот ред на конструкторот на GameCheckers ја иницијализираме матрицата односно ја трансформираме да изгледа како почетната состојба на играта Дама.

main()

Програмата започнува од main(). Таму правиме единствената истанца од класата GameCheckers и на таа истанца ја повикуваме функцијата startGame().

startGame()

Од оваа функција започнува извршувањето на играта. Напочетокот во неа се печати упатство за играње, потоа програмата чека влез од корисникот. Овде се повикува посебна функција getMensInput() која има зе цел да го земе влезот од корисникот. Откако ќе се изврши таа функција (корисникот го поместил пиончето) доаѓа ред на компјутерот. Се повикува функцијата artificialIntelligence() која ќе ја разгледаме подоцна. По завршувањето на artificialIntelligence() се проверува дали на компјутерот и на човекот им останале пиони. Ако барем еден од нив има нула пиони, се печати соодветна порака и играта завршува.

getMensInput()

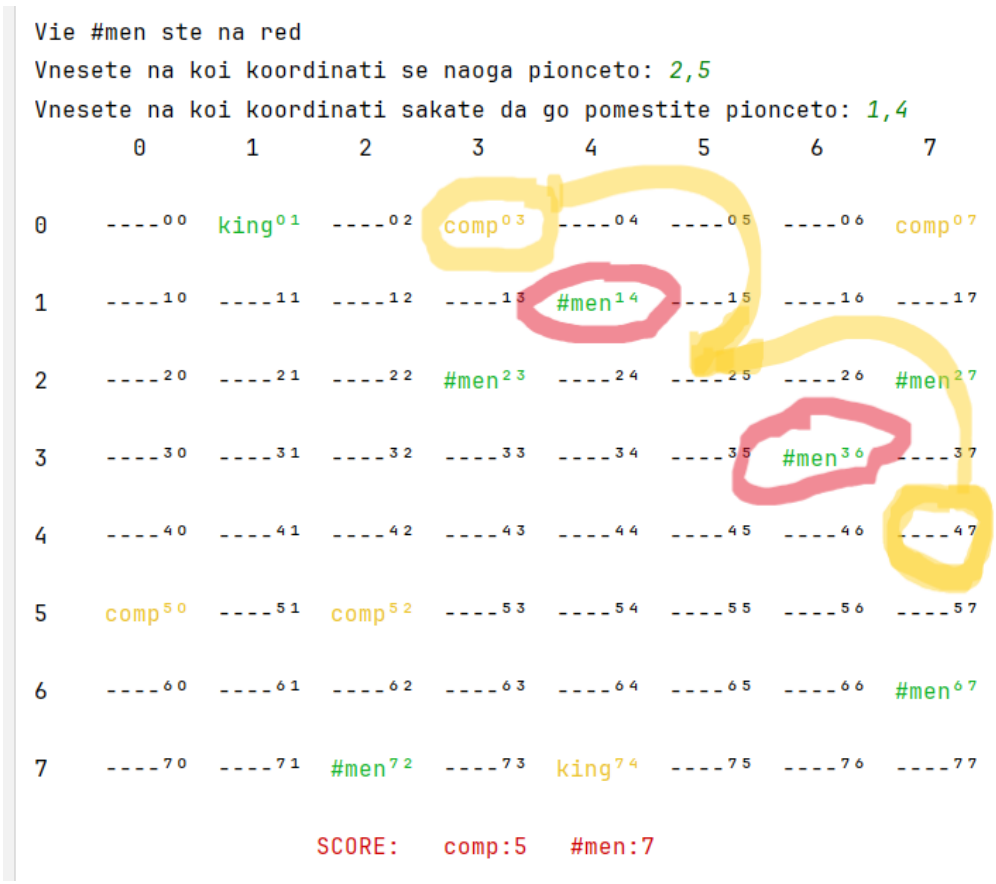
Најпрвин повикува друга функција со која ги проверува легалните потези на човекот. За легални потези се сметаат оние со кои човекот нема да излезе од таблата, нема да го помести пиончето на недозволена позиција или пак нема да ги поместува пионите на компјутерот. Потоа, се внесуваат индексите на пиончето кое корисникот сака да го помести, па се внесуваат индексите на полето на кое треба да се помести. Потоа проверувам колкава е разликата меѓу old_I, new_I (старта и новата редица на пиончето). Ако разликата е еден се работи за обично поместување и во листата ги внесувам:

```
move = [old_I, old_J, new_I, new_J, 100, 100, 100, 100]
```

Каде што четирите стотки ми претставуваат координати на противникот. Во овој случај затоа што човекот не скока противник или противници координатите се 100.
Ако разликата е два се работи за скокање на еден противник и во листата ги внесувам:

```
move = [old_I, old_J, new_I, new_J, (old_I + new_I)//2, (old_J + new_J)//2, 100, 100]
```

Каде што последните 2 стотки ми претставуваат координати на вториот противник. Во превод вториот противник не постои.
Ако разликата е четири се работи за скокање на 2 противници. И овде е малку покомплицирано да ги најдеме кординатите на тие двајца противници. Затоа ја користиме функцијата defineAllPosibleOpponents() за да ни ги дефинира 16-те можни комбинации на координати на противниците. Доколку корисникот навистина внел легален влез, само една од овие 16 комбинации ќе биде точна. Ако корисникот внел нелегален влез тогаш функцијата ќе врати None.
Доколку движењето е дозволено, програмата го прави тоа движење и се повикува функција за пресметување на поените на играчите.



Слика2: Скокање преку две противнички полиња (горна слика)
Слика 3: Изглед на таблата после скокањето (долна слика)

Во оваа имплементација се дозволени поместувања на пиончето за едно поле на лево или на десно, скокање преку едно противничко поле на лево или на десно и скокање преку две противнички полиња на лево или на десно. Нормално пионите на човекот и компјутерот се движат во спротивна насока пред да станат кралеви што значи дека дозволена насока за движење на човекот е:
- од повисок индекс на редицата кон понизок индекс на редицата
а, дозволена насока за движење на компјутерот е:
- од понизок индекс на редицата кон повисок индекс на редицата

Редицата во која човекот станува крал е 0 а, редицата во која компјутерот станува крал е 7. (Слика 2 и 3)

artificialIntelligence()

Најпрвин прави длабока копија од моменталната табла на која се игра играта. Зошто е тоа така? За да креира објект од класата NewGameState со помош на таа табла. Класата NewGameState има метод get_children().
Ќе ги испробува сите можни комбинации на таблата во функцијата get_children(). Ќе го земе секој можен потег на секое пионче на компјутерот и ќе направи нова табла. Затоа при правењето на таа нова табла многу е важно да не ја уништиме оригиналната па, правиме длабока копија. Доколку не направиме длабока копија туку при креирање на објект од NewGameState ја ставиме оригиналната табла тогаш ќе ја измениме неа многу пати без да се консултираме воопшто со алгоритмот minimax. Во превод компјутерот ќе прави рандом потези без да размисли.

Го повикуваме методот get_children() од објектот кој го направивме. Сега ги имаме сите можни потези што може да ги направат сите пиони/кралеви во една листа. Листата содржи одбекти од типот NewGameState.

Најпрвин проверуваме дали оваа листа е празна односно дали компјутерот има услови да направи барем едно двежење. Доколку нема, компјутерот признава пораз.

Потоа во циклус интерираме низ сите објекти од тип NewGameState во листата. Го повикуваме алгоритмот minimax кој враќа број. Тој број го ставаме во речник како клуч а, вредноста на клучот е објектот од типот NewGameState. Во превод правиме проценка на секое движење. Го проценуваме секое движење дали е добро за компјутерот или не.

```
for i in range(len(first_computer_moves)):
    child = first_computer_moves[i]
    value = GameCheckers.minimax(child.get_board(), depth: 1, -math.inf, math.inf, maximizing_player: False)
    dict[value] = child
```

На крајот го бараме клучот кој е **најголем** и ја земаме неговата вредност. Нормално вредноста е од тип NewGameState па од тој објект ја земаме таблата. Оваа табла ја ставаме да биде оригиналната тековна табла на класата.
На крајот повикуваме функција за пресметување на поените.

minimax()

Доколку речам дека „Целиов овој проект го правам за да има смисла“ – тогаш смислата ја дава оваа функција. Благодетите на оваа функција овозможуваат компјутерот да научи како да размислува за да ја искористи својата брзина.

Како што веќе споменав напочетокот алгоритамот minimax работи на тој начин што компјутерот ја минимизира својата штета а, човекот ја максимизира. На крај компјутерот ја избира најмалата несакана ситуација. Позитивното во овој алгоритам е тоа што компјутерот претпоставува дека човекот ќе игра исто толку добро како него. Овој алгоритам смета дека и компјутерот ќе даде сè за да победи и човекот ќе даде сè за да победи. Доколку човекот не игра максимално тоа би било уште подобро ра компјутерот. Оваа функција се повикува онолку пати во artificialIntelligence() колку што има и можни легални поместувања на своите пиони/кралеви а, потоа сама се повикува рекурзивно. Сегогаш кога во играта ќе дојде ред на компјутерот artificialIntelligence() ја повикува со булова променлива како параметар кој покажува дека компјутерот е минимизирачки играч. Само artificialIntelligence() може да одлучи за длабочината на minimax алгоритмот. Таа длабочина исто така се задава при првите повици на алгоритмот кога компјутерот е на ред (во artificialIntelligence()).

Како работи овој алгоритам?

```
def minimax(board, depth, alpha, beta, maximizing_player)
```

При повикување на оваа функција се задаваат 5 параметри. Првиот параметар е матрицата. Овој параметар не е оригиналната матрица. Зошто?

На пример, кога artificialIntelligence() го повикува алгоритамот го повикува за сите можни поместувања на пионите. Првиот пат го поместува првиот пион на лево ако тоа е можно. Вториот пат го поместува првиот пион на десно ако тоа е можно, третиот пот го поместува вториот пион на лево или на десно...
Таблата е во таков формат што цело време еден пион/крал е поместен на друга позиција.

Вториот параметар е длабочина. Неможе на овој алгоритам да му биде зададена било која длабочина. Зошто?

На пример, доколку имаме многу голема имплементација односно има многу пиони, кралеви, премногу можности за поместување и ставам голем број на длабочина на пример 10 програмата нема да издржи и ќе падне. Доколку длабочината е многу мала пример 0 тоа значи дека само хевристичката функција одлучува за следниот потег. Во превод компјутерот ќе размислува само за најдобриот наредeн потег но, нема да

ги размислува потезите по него. Можеби нешто што е на прв поглед добро подоцна може да излезе како лошо.
Моја најпрепорачана длабочина за оваа игра е 1 или 2.

Третиот и четвртиот параметри се многу важни за овој алгоритам. Алфа и бета овозможуваат да не се разгранува гранки од дрвото доколку нема потреба. Тие го поткаструва дрвото. Ја намалуваат сложеноста на алгоритамот. Како?

Напочетокот кога се повикува алгоритамот алфа е иницијализирана на -∞ а, бета на ∞. Затоа што за алфа бараме максимална вредност а за бета минимална. Со текот на извршувањето на алгоритамот алфа и бета ги менуваат своите вредности. Доколку во еден момент алфа е поголема или еднаква на бета тоа значи дека бадиала ќе ги разложуваме следните јазли затоа што во нив ќе најдеме само полошо решение односно подобриот потег ние веќе го имаме најдено, не треба да бараме дополнително. Ова помага да се поткастри дрвото односно алгоритамот да работи за пократко време.

Петиот параметар е maximizing_player. Можеби досега не ја сретнавме оваа променлива директно но, споменивме дека постои булова променлива која нагласува дека компјутерот е минимален играч а, човекот максимален. Кога се повикува алгоритамот рекурзивно вредноста на оваа променлива се менува.

На пример, доколку на ред бил компјутерт во ова изминување на алгоритамот тогаш следен на потег е човекот и кога ќе го повикаме алгоритамот рекурзивно треба да кажеме дали човекот е максимален играч а, не компјутерот.

Како работи алгоритамот?
Најпрвин проверува дали се работи за лист. Доколку се работи за лист ја враќа хевристиката.

Хевристиката се прави во корист на човекот а, на штета на компјутерот. Затоа што човекот ја зема максималната вредност од евристиката а, компјутерот минималната. Компјутерот гледа како да собере што помалку поени.
- Во хевристиката доколку компјутерот има пион хевристиката му одзема поени.
- Доколку пионот се наоѓа до sид тоа значи дека тој пион никој не може да го земе па, евристиката му одзема поени.
- Доколку човекот може да му го земе пионот на компјутерот, хевристиката му дава поени

Хевристиката работи против компјутерот а, компјутерот ја зема најминималната вредност од неа.

Потоа, прави NewGameState објект од таблата (матрицата) која ја проследува како аргумент. Проверува дали на ред е компјутерот или човекот.
Доколку е човекот на ред максималната вредност се иницијализира на -∞ и се проверуваат сите можни потези, се зема максималната вредност од сите можни потези без разлика што функцијата е рекурзивна и потоа се проверува дали можеме нешто да поткастриме со алфа и бета. Слично е сценариото и кога компјутерот е на ред.

Заклучок:
Алгоритамот minimax работи на тој начин што најпрвин ја гледа таблата која првиот потег на човекот ја поместил, потоа ги генерира сите можни потези за него, па гледа како човекот би реагирал на тоа доколку тој ги направи секој од потезите, па како тој да реагира на човекот... Алгоритамот гледа многу далеку. Колку е поголема длабочината на дрвото толку подалеку гледа алгоритамот. Компјутерот е многу побрз од човекот па, тој однапред знае многу повеќе за играта од тоа што самиот човек претпоставува.

MakeAMoveForMen() и MakeAMoveForComputer()

Овие методи сами по себе се многу слични. Едниот се користи за движење на човекот а, другиот на компјутерот. Како аргументи и двата методи ги примаат таблите, координатите на пиончето/кралот што треба да се помести, координатите на кој треба да се помнести, и 4 координати по две за секој од противниците. Доколку пиончето/кралот скока само еден противник тогаш, последните две координати ќе бидат 100. Доколку пиончето/кралот прави обично движење без да скока противници тогаш последните 4 координати ќе бидат 100.

Како работат овие функции?
Најпрвин мора да кажеме дека секое движење што се прави во овие функции е легално двиќење. Зошто? Претходно пред да пристигнат сите овие аргументи на функцијата до самата функција има проверка на конзистентноста. Проверката се прави во две функции getMensAvailableMoves() и getCompAvailableMoves() соодветно. Таму се проверуваат движењата што ги имав споменато и погоре:

„Во оваа имплементација се дозволени поместувања на пиончето за едно поле на лево или на десно, скокање преку едно противничко поле на лево или на десно и скокање преку две противнички полиња на лево или на десно.“

Прво во овие функции се проверува дали се работи за крал или за пион. Затоа што доколку се работи за пион можеби во ова движење пионот треба да стане крал. (Нормално доколку со поместувањето се наоѓа во соодветната редница)
Потоа се проверува дали прави обично движење, скокање преку еден или преку два. Врши промена на таблата соодветно.

CheckPlayerMoves(), checkComputerMoves(), checkPlayerJump(), checkComputerJump(), checkPlayerDoubleJump() и checkComputerDoubleJump()

Овие методи генерално се методи за конзистентност на движењата. Прават проверка за апсолутно секое движење дали е дозволено или не. Дел од нив се користат за проверка на движењата на компјутерот а, дел на човекот.

NewGameState: get_children()

Оваа функција служи за генерирање на сите можни состојби на таблата. Најпрво се повикува во artificialIntelligence() а, потоа ја повикува самиот minimax алгоритам. Кога artificialIntelligence() ја повикува оваа функција ја повикува за следниот потег. Односно на ред дошол компјутерот и се разгледуваат сите можни движења на тековните пиони и кралеви. Додека кога ја повикува minimax алгоритамот ја повикува длабочински. Доколку бил компјутерот на ред кој пиони може да ги помести, па кога ќе ги помести тие пиони кои пиони човекот може да ги помести, па кога ќе ги помести човекот неговите пиони повторно, кои пиони компјутерот да ги помести... Во minimax оваа функција се повикува за да гледаме длабочински додека во artificialIntelligence() ја повикуваме затоа што компјутерот е на ред и сакаме да видиме кои пиони тековно можат да се поместат сега моментално, без да размислуваме што понатаму.

Зошто некој методи се статички а, некој не?

При истражување на овој проблем на интернет посебно внимание ми остави тоа што во различни имплементации различни методи беа статички. Една од причините за тоа е што некои методи мора да се повикуваат надвор од класата GameCheckers. Во тој случај имаме 2 опции:
- Да креираме истанци од оваа класа
- Да ги направиме тие методи статички

Ако направиме нови истанци од GameCheckers тоа може да биде фатално. Затоа што ние немаме повеќе игри „Дама“. Ние во еден момент играме само една игра „Дама“. Ако направиме истанци ќе постојат многу повеќе такви и всушност сите поместувања на пионите/кралевите ќе бидат на многу табли наместо на една. Затоа мора да користиме статички методи.

Заклучок:
Во изминатите месеци изучувајки го предметот „Вештачка интелигенција“ постојано се прашував како ова што го учиме е вештачка интелигенција, како ова е поврзано со неа. Неможев да ја составам слагалката...

Оваа проектна задача за спротивставено пребарување ме натера да ја почувствувам моќта на вештачката интелигенција и конечно да ги поврзам работите. Сега научив дека компјутерот незнае да размислува но, има многу поголема брзина од човекот. Приспособувајќи го компјутерот да размислува, давајќи му знаење како да го направи тоа, тој станува многу моќно суштество.

Компјутерот има огромна моќ не затоа што има многу памет туку затоа што има многу брзина. Додека јас се трудам да направам проценка на неколку потези однапред (можеби 3 или 4) тој има направено проценка на над 100 потези однапред и мои и негови. Затоа вештачката интелигенција станува голема сила и се надевам дека ќе бидам еден од луѓето кој ќе ја направат уште поголема.