УНИВЕРЗИТЕТ „Св. Кирил И Методиј“ – Скопје
Факултет за Информатилки науки и компјутерско инженерство

-ПРОЕКТ-
по предметот
ВЕШТАЧКА ИНТЕЛИГЕНЦИЈА

Тема
Анализа и споредба на алгоритми за пребарување во простор и донесување одлука во игра со противници под влијание на временски лимит

Ментор:
Проф. Д-р Андреа Кулаков

Изработиле:
Александра Настоска(216068)
Сара Манасиева (216054)

Скопје, октомври 2023

Содржина

Апстракт
Една од главните области на фокус на Вештачката Интелигенција секако се игрите со противници. Низ развојот на ВИ, се развиле многу алгоритми за пребарување, од кои вреди да се издвојат minimax и алфа-бета поткастрување. Токму овие алгоритми се тема на истражување на проектот и истите се анализираат под влијание на временски лимит за донесување на одлука. Подлабоко е анализирана играта Othello или Reversi.
Клучни зборови – minimax, алфа-бета поткастрување, Othello, временски лимит

Вовед
Широкиот спектар на области на примена на Вештачката интелигенција не би бил целосен доколку не ги земеме во обзир игрите, особено оние каде ВИ треба да се натпреварува со човек или други ВИ. Суштинска улога во ова сценарио имаат алгоритмите за пребарување и донесување одлуки.

Овој проект се фокусира на анализата и споредбата на алгоритмите за пребарување и донесување одлуки во игри каде што противникот е самиот компјутер, односно ВИ. Посебен акцент се става на алгоритмите „minimax" и „алфа-бета поткастрување" кои се познати по својата ефикасност во анализата на игрите и донесувањето на оптимални одлуки.
Во рамки на истражувањето, се анализираат резултатите од овие алгоритми во контекст на временски лимит. Ова подразбира дека ВИ мора да донесе одлука во одредено временско ограничување. Анализата се спроведува врз конкретна игра Othello или Reversi. Во продолжение ќе биде разработен код кој може да се извршува на модерни компјутери, со што е овозможено практично тестирање на концептите и алгоритмите кои ќе бидат проучени.

Теорија на игри
Една од првите примени на ВИ била во игрите. Првите игри кои добиле ВИ играчи се X-O, шах, Go, Reversi. Решени се оние игри за кои важи дека при секој потег може со сигурност да се одреди исходот на играта ако двата играчи играат совршено. Од гледна точка на ВИ, игрите се од голем интерес бидејќи се многу тешки за решавање. Просечниот фактор на гранење во шах е 35, а просечен број потези на еден играч во една партија е 40. Тоа значи дека бројот на листови во комплетното дрво на играта е 35^{80}. Просечниот фактор на гранење во Go е 250.
Теорија на игри е дел од применетата математика кој се занимава со проучување на однесувањето на единка (која носи некоја одлука) во интеракција со други единки. Нејзиниот успех не зависи само од нејзините одлуки, туку, се разбира влијаат и одлуките кои ги носат другите единки. Оваа гранка се применува и во економија, политика.

Основни концепти

Важен момент во програмирањето игри и улогата на ВИ е труд објавен во 1950 од страна на Claude Shannon наречен *Programming a digital computer for playing Chess.* Тука се дадени две стратегии за избор на потег во игра:

A. Со Минимакс се пребарува дрвото на играта и преку оценување на легалните потези се бира оној со најдобра оцена
B. Потегот се бира врз основа на моменталната позиција во играта и однапред подготвена табела

Од наш интерес е првиот концепт. Доколку со оваа стратегија, дрвото се пребарува до крајните состојби на играта навистина би бил избран најдобриот легален потег. Меѓутоа, за повеќето игри ова не е можно. Дури и пребарување на дрвото на длабочина од само неколку потези во игри со просечно 10-20 легални потези ќе резултира со испитувањена милиони различни позиции. Поради тоа, ефикасната примена на оваа стратегија се заснова на пребарување до релативно мали длабочини со квалитетно избрани хевристики, наместо со наједноставен минимакс алгоритам.

Алгоритам минимакс

Овој алгоритам е во основата на скоро сите алгоритми за избор на потег. Тој со пребарување на дрвото на игра за играчот кој е на потег го одредува најдобриот потег во дадена ситуација. Пребарувањето на целото дрво гарантира наоѓање на најдобриот потег, но поради недостаток на хевристика, тоа може да биде многу долго. Минимакс во основа е рекурзивен алгоритам кој се заснова на DFS.

Алфа-бета поткастрување

Овој алгоритам се заснова на поткастрување на дрвото на игра и во основа е забрзан минимакс. Вака се намалува бројот на јазли кои треба да се испитаат, односно се отстрануваат гранките кои се неоптимални. Алфа се однесува на играчот кој е на потег, додека бета е на противникот. Ако алгоритмот најде гранка каде бета е помало или еднакво на алфа, тогаш е јасно дека противникот нема да го избере овој потег, па гранката може да се скрати. Како што се движиме низ дрвото, алфа и бета добиваат нови вредности, со што алгоритмот ги наоѓа гранките кои може да се отстранат. Вака во голема мера се намалува бројот на јазли, со што пребарувањето е поефикасно. Псевдокодот за овој алгоритам е даден во продолжение

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Сл.1 Алфа-бета псевдокод

Временски ограничувања во дадените алгоритми

Интегрирањето на временски ограничувања во алгоритмите за пребарување е особен предизвик, особено доколку станува збор за алгоритми како минимакс или алфа-бета поткастрување. Како што видовме погоре, минимакс се карактеризира со огромни дрва на одлука, па токму затоа, доколку алгоритмот содржи и временски лимит, времето за пребарување е значително ограничено. Токму затоа, како добра стратегија се издвојува итеративно пребарување по нивоа во длабочина или *iterative deepening.* Вака достигнуваме баланс меѓу длабочината до која пребаруваме и времето кое го имаме на располагање, со што алгоритмот може да ја донесе најдобрата можна одлука до тој момент.
Потребно е да имаме некој избран потег доколку дојде до прекин во играта ако истекло времето за одлука. Важно е тој потег да биде легален и што е можно подобар. Прекин е предизвикан од акција на корисникот или поради временско ограничување по потег. Алгоритмите кои ги разгледавме воглавно имаат избрано некој потег, дури и да истече времето. Сепак, кога алгоритмот не се извршил комплетно, тој потег може да биде многу лош. Затоа, во продолжение ќе разгледаме модификација на алфа-бета алгоритмот чиј резултат често е солиден, задоволувачки потег.

Оптимизација на алфа-бета

Идеја која вреди да се разгледа за оптимизирање на алфа-бета алгоритмот е да се избере добра хевристика. Овој алгоритам дава добри резултати ако кај секој јазол се избира најдобриот потег. Нормално, не е можно однапред да се знае кој потег е најповолен, но со добри проценки се постигнуваат добри резултати.

На пример, да избереме хевристика која индицира на тоа дека при пребарување на дрвото, на длабочина d ($d \geq 1$) W е најдобриот пронајден потег. За секој друг јазол на таа длабочина, испитувањето на птоезите го започнуваме со W. Доколку најдеме некој подобар потег, тој станува W.

Ако пребарувањето се врши до длабочина d_{max}, оваа хевристика се применува за сите длабочини т.ш. $1 \leq d \leq d_{max}$ - 1. Со примена на ваква хевристика не се менува резултатот на алгоритмот (за иста длабочина) т.е. се добива потег со иста оценка, но најчесто со значително помал број на испитани јазли. На пример во шах, ако играчот е под закана за шах-мат и со ниеден потег не може да се спречи матирање, се открива јазолот кој е потегот кој носи крај на играта. Затоа, кога ќе почне пребарувањето низ дрвото, овој потег е на длабочина 1 и кога ќе стигнеме до него, сите останати потези се поткаструваат и времето за носење одлука е значително намалено.

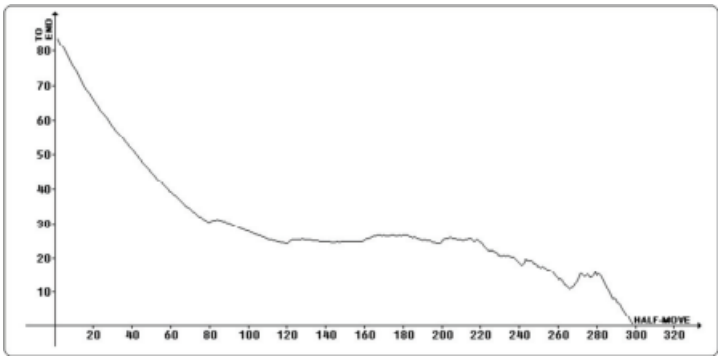
За длабочина d_{max}, најпрво се пребарува до длабочина 1 и најдобриот потег станува W за ниво 0. Потоа за длабочини d ($2 \leq d \leq d_{max}$) се применува истиот алгоритам и добиениот потег станува W за почетниот јазол. Најдобар потег е оној кој се добива со крајната примена на алгоритмот за длабочина d_{max}. Важна особина на овој алгоритам е што во случај на предвремено завршување на пребарувањето (поради истечен временски лимит), во секој момент веќе имаме некој избран „најдобар потег“.

Модел со временско ограничување во шах

Во продолжение ќе разгледаме модел на играта шах со временско ограничување. На почетокот на играта, имаме вкупно време T и просечен број потези N, па оттука, земаме параметар τ што претставува време кое ќе го потрошиме за еден потег. Земаме дека $N=85$. Недостаток на овој модел сметаме дека е тоа што ако играта заврши во помалку од N чекори, останува неискористено време, и соодветно, доколку имаме повеќе потези од предвиденото, времето ќе истече.

За да го подобриме моделот, сега земаме дека N не е просечен број на потези, туку ова ќе биде $N(n)$ – функција од очекувањето траење на играта кога знаеме дека н потези биле направени. Ако го користиме овој модел, проценката на време за еден потег мора да биде направена пред и воопшто да почнеме да размислуваме за потегот. Сега, $\tau = T/(N(n)-n)$. Овој модел е едноставен, троши малку време и практично ја елиминира можноста за пораз поради недостаток на време. Ако програмата сфати дека има можност да направи само еден потег или лимитиран број на потези, одбира потег во истиот миг со што се избегнува непотребно трошење време за размислување, односно времето кое преостанало за одлука

за моменталниот потег, се пренесува на наредните.
Од анализа на очекуваниот број на потези до крај на играта, од множество со податоци од одиграни шаховски игри, може да се види дека тој број линеарно се намалува.

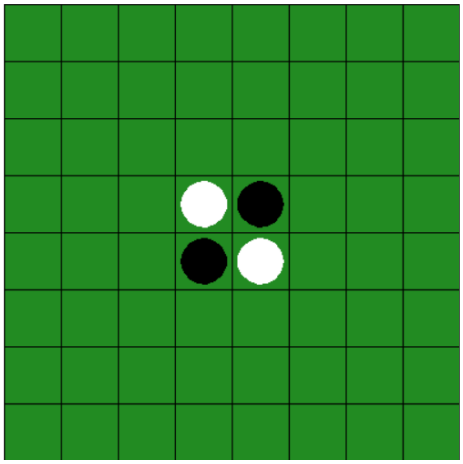


Сл. 2: Очекуван број на потези до крај на играта

Овие резултати можат да се применат во играта, така што програмата ќе го третира τ само како сугестија. Ако сфати дека веројатноста да направи „добар“ потег е *v*, ќе го преземе тој чекор пред τ да истече. Спротивно, ако времето за тој потег истекува, може да биде доделено додатно време *g*τ каде *g* е тежински пропорционално до *v*. Ако експериментално се подеси *g* во разумни граници, програмата ќе зачувува време во јасни позиции и рационално ќе позајмува време од идните потези.

Othello

Othello (Reversi) е игра позната по својата едноставност, но и бројните интригантности во нејзината дигитална имплементација. Се игра на 8x8 табла, со двајца противници, често претставени како ‘X’ и ‘O’. Целта на играта е да се соберат колку што е можно повеќе фигури од противникот, со стратешко пласирање на сопствените фигури на таблата. Играта започнува така што на таблата има 4 фигури, по две за секој противник. На сл.3 е дадена почетната позиција за играта. Целта на секој играч е фигура на противникот да ја стави меѓу две свои, со што тие ја менуваат бојата и му припаѓаат нему.
Дигиталните имплементации на играта се потпираат на напредни алгоритми кои одредуваат оптимални потези. Минимакс е најчесто искористуван за да се најде потенцијален потег и да се предвидат последиците од истиот. Алгоритмот се обидува да ја максимизира добивката на играчот кој е на ред, а истовремено да ја минизимизира добивката на противникот. Алфа-бета поткаструвањето, како што видовме и погоре, ја зголемува ефикасноста на минимакс. Понатаму, со *iterative deepening* алгоритмот ги почитува наложените временски лимити, со што го адаптира пребарувањето по длабочина динамички. Функцијата на евалуација укажува на тоа колку е поволна одредена состојба во игата, преку анализа на мобилност, број на фигури и контрола на таблата.



Сл.3 Почетна позиција на Othello

Othello co time limit

```
def play(self, x, y):
    # USER'S TURN
    if self.has_legal_move():
        self.get_coord(x, y)
        if self.is_legal_move(self.move):
            turtle.onscreenclick(None)
            self.make_move()
        else:
            return

    # COMPUTER'S TURN
    while True:
        self.current_player = 1
        if self.has_legal_move():
            print('Computer\'s turn.')
            start_time = time.time()
            self.make_computer_move()
            end_time = time.time()
            time_taken = end_time - start_time
            print(f"Computer took {time_taken:.2f} seconds to make a decision.")
            if time_taken > time_limit:
                print("LIMIT EXCEEDED")
            self.current_player = 0
            if self.has_legal_move():
                break
        else:
            break
```

Сл.4: Имплементација на играта во код

Овој код е дел од имплементација на играта. Самата игра е управувана од класата *Othello* која е потклаcа на класата *Board* во која се состојат методи за управување со таблата. Класата *Othello* во себе има повеќе методи со кои се иницијализира таблата, се проверува легалноста на еден потег и се менаџира текот на играта.

```
def make_computer_move(self):
    start_time = time.time()
    moves = self.get_legal_moves()
    if moves:
        self.move = random.choice(moves)
        self.make_move()
    end_time = time.time()
    time_taken = end_time - start_time
    if time_taken > time_limit:
        print("Time limit exceeded")
```

Сл.5: Функција make_computer_move

Ова е веројатно најважната функција и тука се одредува потегот на компјутерот. Се прави проверка и се земаат сите легални потези (доколку ги има) и потоа компјутерот се одлучува за еден потег. Се мери времето потребно да се направи еден потег и истиот се прави доколку лимитот не истекол, во спротивно се печати порака „Time limit exceeded“ и компјутерот не прави никаков потег, со што се почитува временскиот лимит.

Анализа на перформанси

По одиграни 20 игри, резултатите се следни:

Игра	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
User	19	47	22	24	46	39	14	44	26	10	29	36	47	44	10	29	30	38	24	45
Computer	45	17	42	40	18	25	50	20	38	54	35	28	17	20	54	35	34	26	40	19

Од податоците може да се извлече заклучок дека:

- Компјутерот остварил победа во 11/20 случаи, односно процентот на успешност е 55%
- Просечниот број фигури кои ги освоил компјутерот се 32,85, наспроти 31,25 на корисникот

Дополнително, кодот има функционалност да го измери времето потребно за секој потег. Оттаму, пресметано е дека **просечното време на компјутерот по потег е 0,73 секунди,** а

најдолгото време за еден потег е 2,57 секунди. Интересен заклучок кој дополнително беше извлечен е тоа дека во колку полоша моментална ситуација е компјутерот, толку повеќе време по потег му е потребно.

Заклучок

Проектот се состои од вовед во теоријата на игри и нејзините фундаментални концепти што потоа неизбежно водеше до алгоритмот минимакс, следен од алфа-бета поткаструваѓето и начин за негова оптимизација. Беше разгледан и модел на шах, како и имплементација на играта Othello. Може да заклучиме дека ВИ лесно може да биде надмоќен над човекот со доволно добар алгоритам, дури и под влијание на временски лимит. Би можеле да се добијат фасцинантни, интригирачки резултати доколку се направи обид овој алгоритам уште повеќе да се оптимизира и подобри.
Во време кога присуството на ВИ е во огромен подем во секојдневниот живот, јасна е потребата од одлично справување со проблемот на носење одлуки под временско ограничување, за да продолжиме да се движиме на патот кон навистина интелигентни ВИ системи.

Референци

1. V. Vuckovic and R.Solak, “Time management procedure in computer chess”, *Facta Universitatis*
2. P.Janicic and M.Nikolic “Vestacka inteligencija”, *Matematicki fakultet u Beogradu*
3. Z. Zhang and Y.Chen, “Searching Algorithms in Playing Othello”, *Oregon State University*
4. A.Aljubran, “Othello”, *Indiana State Univesity*
5. E. Boyarski, A.Felner, D. Harabor, P.Stuckey, L.Cohen, J.Li, S.Koenig, “Iterative-deepening Conflict-based search”