Софийски университет „Св. Кл. Охридски”

Факултет по математика и информатика

Курсов Проект

на тема: Quixo Bot

Студент: Добрин Цветанов Цветков, 81265

Курс: 4, Учебна година: 2018/19

Преподаватели: **проф. Иван Койчев**

==========

Декларация за липса плагиатство:

- Плагиатство е да използваш, идеи, мнение или работа на друг, като претендираш, че са твои. Това е форма на преписване.
- Тази курсова работа е моя, като всички изречения, илюстрации и програми от други хора са изрично цитирани.
- Тази курсова работа или нейна версия не са представени в друг университет или друга учебна институция.
- Разбирам, че ако се установи плагиатство в работата ми ще получа оценка “Слаб”.

3.2.19 г. Подпис на студента:

Съдържание

1. Мотивация и задача за курсова работа

Ще започна като кажа, че курсът по Изкуствен интелект за мен беше изключително интересен и полезен. Разбира се, бях се сблъсквал и преди с някои алгоритми като *DFS* и *BFS*, все пак имаме специален курс за това в учебният план. Текущият курс обаче ми помогна да надградя знанията си и ми показа как да използвам основните идеи на базови алгоритми, за да получа нещо по-сложно и в крайна сметка да напиша нещо, което ще е интересно и полезно за самият мен.

И всъщност мотивацията за този проект дойде от самият курс и по-специално от домашно номер пет – Морски шах с алгоритъм Мин-макс и с алфа-бета отсичане. Винаги съм бил фен на настолните игри и това домашно ми беше особено интересно. След като приключих с него, реших че трябва да направя нещо по-сложно, но в основата си да надгражда над домашното. Така се роди идеята да направя бот за игра, която прилича на морски шах, но с по-сложни правила – *Quixo*.

Правилата на *Quixo* са следните:

- Имаме дъска от 25 кубчета, подредени в 5 реда и 5 колони
- Всяко кубче има една страна с X, една с O и 4 празни страни
- Играта започва като всички кубчета са с празна страна нагоре
- Участникът избира и взема от периферията на дъската едно кубче с празна лицева страна или такова с неговия символ.
- Независимо от това дали кубчето, което е взето, е празно или със символа на играча, то трябва винаги да бъде върнато със символа на играча на лицевата страна.
- При вземането на кубче на дъската остават незавършени редици. Играчът избира една от тях и избутва с кубчето, което е взел, така че да запълни дупката. Не може да играе кубчето обратно на позицията, от която го е взел.
- Победител е този играч, който направи хоризонтал, вертикал или диагонал от 5 кубчета със своя символ.

2. Кратък обзор

Работата по проекта започна с проучване – правен ли е подобен проект преди и какви методи са използвани. Успях да открия няколко варианта на решения:

2.1 Минимаксен подход с алфа-бета отсичане [1]

През 1996 година студентите към университета в Аляска получават за курсов проект по Изкуствен интелект да напишат компютърна програма, която да може да играе *Quixo*. В условието не е казан конкретен метод, който да се използва, но на страницата има линк към стар сорс код, писан на *C* и имплементиращ минимаксен подход с алфа-бета отсичане.

2.2 *Greedy HillClimb* [2]

След това успях да открия хранилище в *Github*, в което имаше линк към същото това условие на университетът в Аляска, но този път реализацията беше на *Java* и използваше методът *HillClimb*.

2.3 Невронни мрежи[3]

Успях да намеря и още едно *Github* хранилище. То обаче нямаше описание и беше написано на *Go*, но след кратък преглед на кода установих, че се използват невронни мрежи.

3. Моето решение и програмна реализация

Моето решение използва езикът *C++* и е основно фокусирано около метода Мин-макс с алфа-бета отсичане. Най-напред написах клас *Board* за представяне на дъската и абстрактен клас *Player* с метод *playMove*. По този начин се възползвам от полиморфизъм и предоставям интерфейс при стартиране на програмата потребителят да може да избере дали иска играчите да са ботове или да играе самият потребител. За игра на самият потребител написах клас *Human*, който наследява *Player* и реализира *playMove* чрез четене на ход от терминала.

След като вече имах основната функционалност за изиграване на игра, следващата стъпка беше да премина към реализирането на самия бот. За целта написах клас *BotMinMax*, наследяващ *Player*, който имплементира *playMove* чрез вземане на минимаксно решение – обхожда се дървото на състоянията определен брой ходове надолу, терминалните състояния се оценяват в зависимост от това кой печели и на каква дълбочина са, а нетерминалните листа биват оценени чрез чисто виртуалната функция *eval*. По този начин отново получих абстрактен клас и вече можех да се възползвам от наследяване, за да реализирам и сравня различни евристични функции. Това постигнах чрез класовете *BotType1*, *BotType2* и *BotType3*.

BotType1 разполага с вектор *cellWorth*, който съдържа 25 числа – стойността на всяка една клетка от дъската. При извикване на *eval* се преглежда какво съдържа всяка клетка и ако е със символа на бота, то добавяме към оценката на дъската стойността на клетката, а ако е с противниковия символ се изважда стойността на клетката.

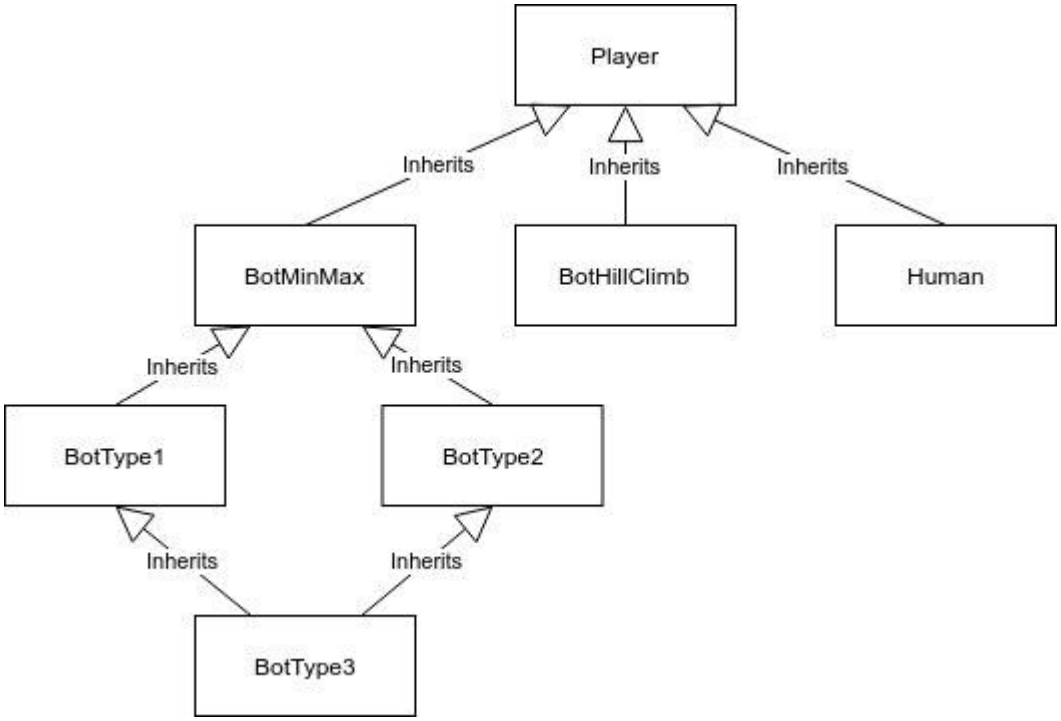
Фиг1. Стойността на всяка клетка при *BotType1*

BotType2 оценява дъската като обхожда двата диагонала, редовете и колоните и за всяка поредица от три собствени символа добавя 1 към оценката на дъската, за всяка поредица от четири собствени символа добавя още 3, а за поредици от символа на противника се изваждат същите стойности.

BotType3 всъщност е комбинация от горните два подхода. Този клас наследява и *BotType1*, и *BotType2*. При извикване на функцията *eval* той взима резултатите от *eval* на двамата си родители и връща сумата от тях.

Накрая реших, че искам да сравня тези евристики не само една срещу друга, а и срещу друг алгоритъм. Затова имплементирах *BotHillClimb*, който да наследява *Player*. Той реализира функцията *playMove* като преглежда всички възможни ходове, оценява дъската след изиграване на хода по същият начин, както и *BotType1* оценява листата, и избира този ход, който има най-висока оценка. Получената логика всъщност е същата като *BotType1* при зададена дълбочина на обхождане 1.

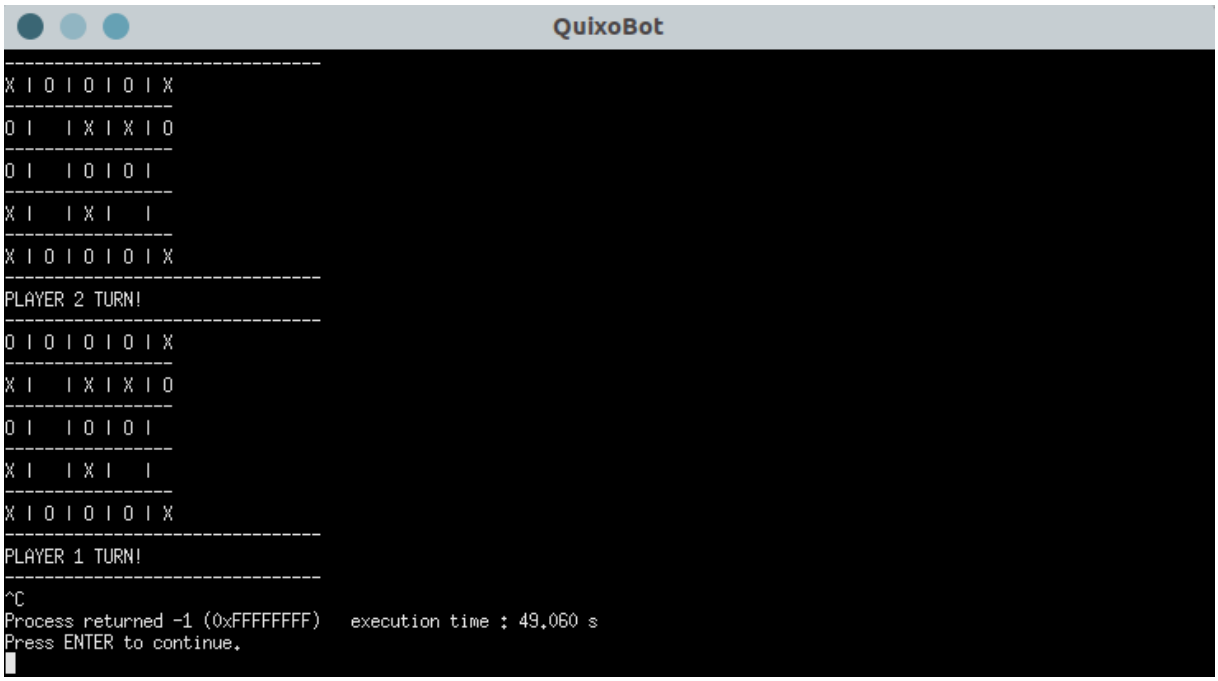
Цялата йерархия на играчите изглежда по следния начин:



Фиг3. Йерархия на играчите

4. Резултати от експерименти

Първият експеримент беше да сравня как се представят евристиките, когато играят една срещу друга. Преди да представя резултата, редно е да отбележа, че според правилата е напълно легално игра да се играе до безкрайност. Понякога ботовете толкова се стремят да не загубят, че се получава нещо подобно на следната ситуация:



Фиг3. Безкрайн игра

На горната принтирана дъска играч 2 е на ред. Той взима кръгчето от клетка с координати (2, 1) и го поставя на позиция (1, 1). Резултата е показан на долната принтирана дъска. На следващият ход играч 1 взима своето хиксче от (2, 1) и го поставя обратно в клетка (1, 1). След което този ход се повтаря безкрайно.

В следващата таблица са показани резултатите от игрите между Минимакс ботовете с различни евристики, като името на реда указва кой играе с ‚X', а колоната указва кой играе с ‚O'. Стойността на клетката е или *Inf* при безкрайна игра, или символ на победителя и продължителност на играта в ходове. Всичките резултатите са при обхождане на дървото до дълбочина 5.

	BotType1	BotType2	BotType3
BotType1	Inf	X - 27	Inf
BotType2	Inf	Inf	Inf
BotType3	X - 37	X - 33	O - 28

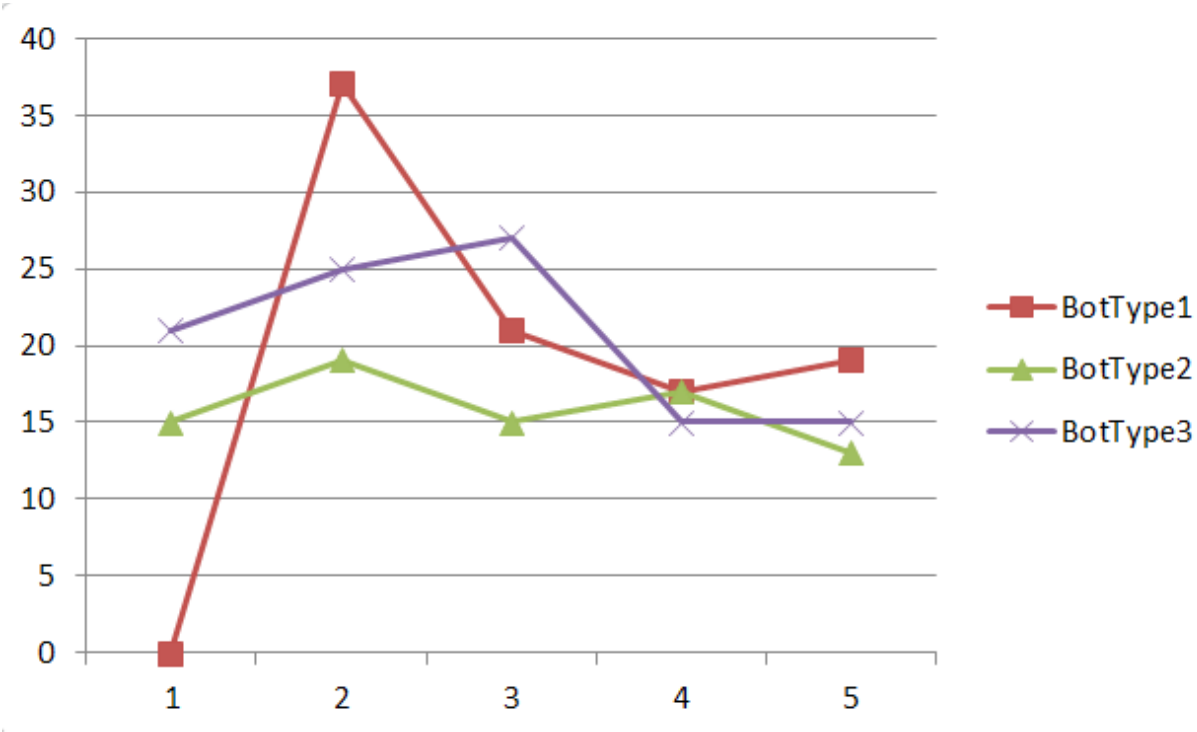
Фиг4. Сравнение на евристиките при минимксен подход

BotType1 успя да победи единствено *BotType2*. *BotType2* не успя да вземе победа в нито една от изиграните игри. *BotType3* постигна най-добри резултати, като единствената му допусната загуба е срещу друг *BotType3*. Интересното тук е, че това е и единственият случай, при който играчът с ‚O' печели.

Вторият експеримент беше да сравня как се представят минимаксните подходи срещу *HillClimb* методът в зависимост от това до каква дълбочина обхождат дървото на състоянията. При всичките експерименти минимаксният бот играе с 'X', а *HillClimb* ботът с 'O'.

Heuristic\Max Depth	1	2	3	4	5
BotType1	O - 30*	X - 37	X - 21	X - 17	X - 19
BotType2	X - 15	X - 19	X - 15	X - 17	X - 13
BotType3	X - 21	X - 25	X - 27	X - 15	X - 15

Фиг5. Сравнение на минимаксен подход срещу *Hillclimb* в таблица



Фиг6. Сравнение на минимаксен подход срещу *Hillclimb* като графика

BotType1 допуска загуба при дълбочина 1, но това не трябва да ни изненадва – все пак както отбелязах *HillClimb* реализацията всъщност е аналогична на *BotType1* с дълбочина 1. От там нататък *BotType1* само печели и постепенно смъква необходимите ходове, като само накрая малко се покачват.

BotType2 винаги печели срещу *BotHillClimb*. Забелязва се, че когато играе срещу по-слаб съперник, средният резултат на *BotType2* е най-бърз от трите евристики. Освен това виждаме, че когато дълбочината е нечетна, той успява да се справи за по-малко ходове. Причината за това е, че при четна дълбочина последният изигран ход при дървото на състоянията е на противника и в такива ситуации ботът е по-песимистичен за крайния резултат.

BotType3 съчетава предпазливостта на *BotType1* и скоростта на *BotType2*. Виждаме, че при достатъчна дълбочина на обхождане, *BotType3* рязко смъква необходимите ходове.

5. Заключение

В този проект успях да реализирам четири различни варианта на бот за играта *Quixo*. Резултатите категорично показват, че алгоритъмът Минимакс с алфа-бета отсичане се представя по-добре от *HillCimb* подхода.

Освен това, при сравнение на различните евристики за минимаксния подход, *BotType3*, който е комбинация между другите два метода, постигна най-много победи и даже не допусна загуба от другите евристики.

В бъдеще бих могъл да проуча по-подробно как може да се реализира бот чрез Невронни мрежи и да го сравня с Минимаксният подход.

6. Ресурси

[1] Quixo – 1996 Ai Project, University of Alaska:
http://www.math.uaa.alaska.edu/~afkjm/ai_games/quixo/quixo.html

[2] Quixo HillClimb Project, Harry Kwon 2015: https://github.com/Harry-Kwon/Quixo

[3] Quixo Neural Networks Project, jjm3x3 2017:
https://github.com/jjm3x3/quixo