Универзитет „Св. Кирил и Методиј“ – Скопје

Факултет за информатички науки и компјутерсо инженерство

Прв циклус студии

Извештај по предметот:

Вештачка интелигенција

На тема:

Спротиставено пребарување состојби во рамките на играта Ultimate Tic-Tac-Toe

Ментор:

Проф. д-р Соња Гиевска

Автори:

Илина Дураковска, 221017
Бојан Котев, 226143

Скопје, Јануари, 2025

Содржина

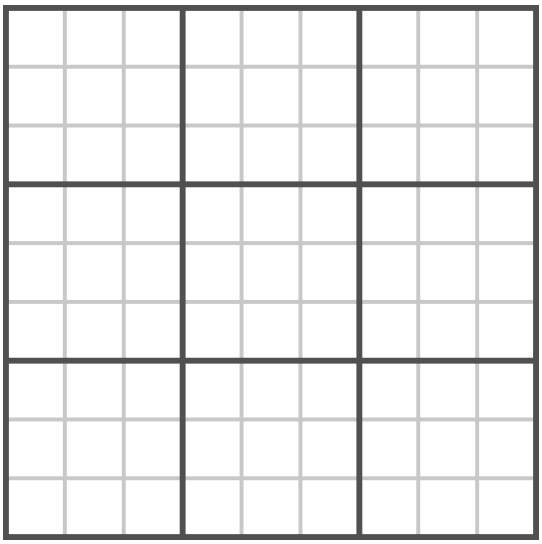
Апстракт

Во овој извештај се дискутираат имплементации на алгоритми за информано пребарување на состојби во рамките на играта „Ultimate Tic-Tac-Toe“, различни хевристички оцени на состојбите и стратегии за победа на играта.

Дополнително се споредуваат постигнатите резултати на секој од алгоритмите, како и нивните перформанси во однос на временска и мемориска комплексност.

1 Вовед

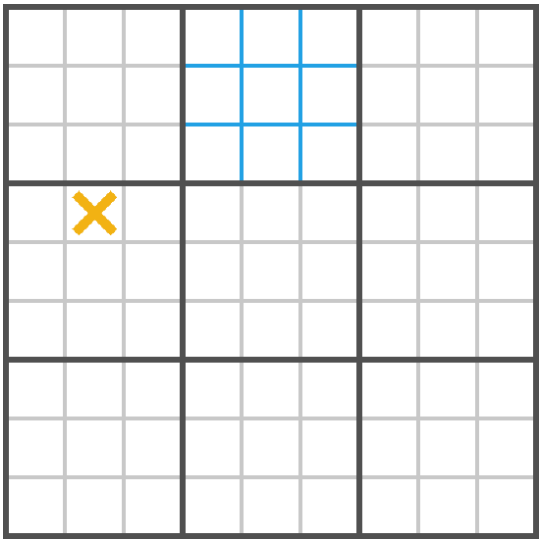
Ultimate Tic-Tac-Toe (UTTT) е покомплексна верзија на познатата игра Tic-Tac-Toe (Икс-Нула). Разликата е во тоа што во рамките на стандардната 3×3 табла (голема табла), секое од деветте полиња содржи посебна 3×3 табла (мала табла) – како што е прикажано на Слика 1.



Слика 1: Табла на UTTT.

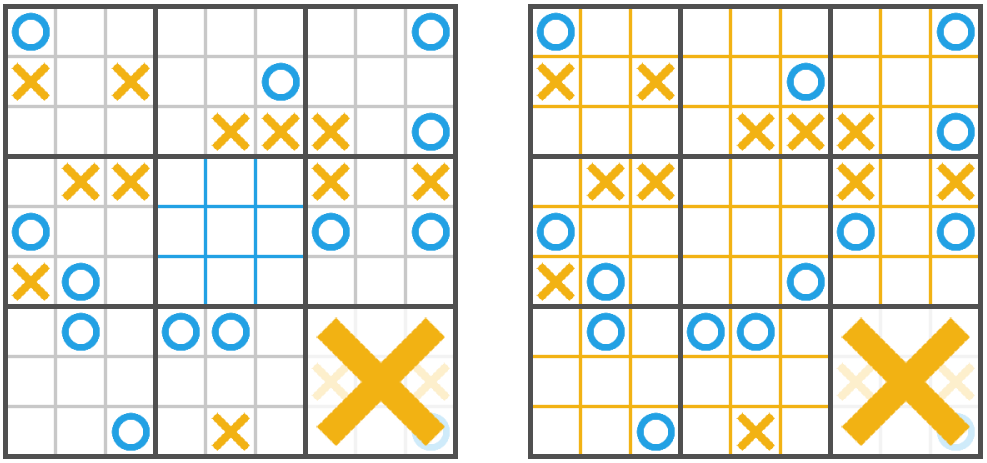
Играта започнува со тоа што првиот играч поставува знак X на било кое поле, потоа на ред е вториот играч со знакот O. Се игра наизменично во рамките на помалите табли така што потегот на едниот играч го носи следниот играч на истата релативна позиција на големата табла.

Нека првиот играч поставил X на полето како што е прикажано на Слика 2. Бидејќи во рамките на малата табла игра горе-средина, следниот играч сега мора да игра на некое од деветте полиња кои се горе во средина на големата табла. Ова е означено на истата слика така што рамката на малата табла чии потези се дозволени ја добива бојата на играчот кој што е на ред (во овој случај O).



Слика 2: Почеток на играта.

Играта продолжува вака со тоа што кога на мала табла ќе победи еден играч, таа табла ја добива ознаката на тој играч (игра потег на големата табла). Доколку едниот играч го однесе другиот на табла која што е веќе победена или нерешена, следниот играч може да игра на било кое од слободните полиња на таблата, прикажано на Слика 3.



Слика 3: О му дава слободен потег на X откако игра доле-десно.

Една табла е победена доколку на неа има 3 исти знаци во ист ред, колона или дијагонала, а е нерешена доколку е целосно пополнета и не е победена. Целта е да се победи на големата табла преку победување на локалните помали табли.

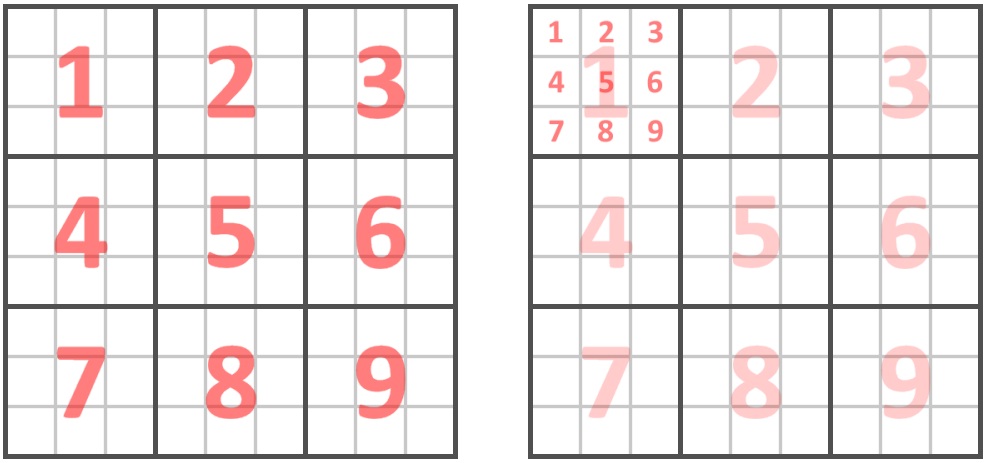
2 Имплементација

2.1 Имплементација на играта

2.1.1 Индексирање

Во рамките на имплементацијата на играта користевме три различни типови на индексирање за одредување на позициите на таблата.

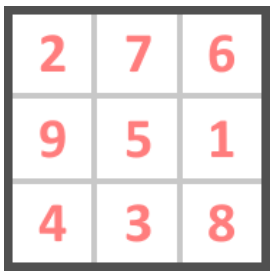
a. **Big-Small:** Секое поле на 9×9 таблата е одредено со торка од два индекси $(\text{big\_index},\ \text{small\_index})$ каде што big_index претставува табла, а small_index претставува поле на истата. Редоследот на броевите е ист за таблите и за полињата, како што е прикажано на Слика 4.



Слика 4: Big-Small индексирање на табли и полиња.

b. **Row-Column:** Целата табла е индексирана како 9×9 матрица, почнувајќи од горното-лево поле $(0,\ 0)$ до долното-десно поле $(8,\ 8)$. Првиот индекс го одредува редот, а вториот колоната.

c. **Magic Square:** Во рамките на таблите за одредени пресметки, како на пример наоѓање на победа, употребивме индексирање по magic square [1]. Тоа е $\mathrm{N} \times \mathrm{N}$ квадрат за кој е задоволено својството дека збирот на сите редови, колони и дијагонали е еднаков. За 3×3 квадрат, збирот е 15.



Слика 5: Magic Square индексирање на полиња.

2.1.2 Состојби на играта

За репрезентација на состојбите на играта користиме торка од 10 речници (dictionaries) од кои првиот се однесува на големата табла, а останатите 9 на малите табли по big_index нотација. Секој речник е во следниот формат:

```
{'X': (), 'O': (), 'display': ('/', '-', '-', '-', '-', '-', '-', '-', '-', '-')}
```

Торките на клучевите X и O ги чуваат magic square индексите на играните потези на соодветната табла. Клучот display се користи за визуелна репрезентација на играните потези по small_index нотација. Дополнително, коса црта (/) се користи како offset, а хоризонтална црта (–) означува празно поле кое може да се замени со “X” или “O”. Кога се зборува за нултата табла хоризонталната црта може да се замени и со “T” за нерешено (tie).

2.1.3 Модели на играчи

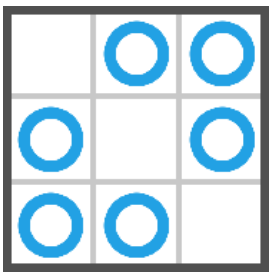
За лесна имплементација и менување на различни типови играчи, управувани од компјутер или човек, воведовме апстрактна класа Player од која наследуваат сите останати играчи. Ова овозможува комбинирање на било кои два играчи при инстанцирање на играта.

2.1.4 Помошни класи

Со цел да се избегне повторување на кодот, често користените функции ги одвоивме во помошни класи:

a. **State Updater:** Менување на состојбата на играта по направен потег.
b. **State Checker:** Проверки за победа и завршени табли.
c. **State Evaluator:** Хевристичка евалуација на состојбите.

Во помошната класа State Checker користиме поразличен пристап од интуитивното решение за наоѓање на победа. Со итерација на елементите од торките на клучевите X или O, се бара барем една тројка на magic square индекси чиј збир е 15. Тоа се прави со прикажаниот код во Фигура 1.



Слика 6: Максимален број потези без победа.

```
positions = sorted(positions)
length = len(positions)
for i in range(length - 2):
    left = i + 1
    right = length - 1
    while left != right:
        score = positions[i] \
                + positions[left] \
                + positions[right]
        if score < 15:
            left += 1
        elif score > 15:
            right -= 1
        else:
            return sign
return False
```

Фигура 1: Код за наоѓање победа.

Во Фигура 1, променливата positions се однесува на торката позиции на моменталниот знак (sign), индексирани по magic square. Теоретската комплексност на овој метод е $O(n^2)$, но во пракса максималниот број на полиња со ист знак за кои нема победа е 6, како што е прикажано на Слика 6. Тоа значи дека во најлош случај може да има победа со 7 позиции и со тоа n ќе биде $7 - 2 = 5$, а бројот на циклуси никогаш нема да надмине $5 * (5 - 1) = 20$ итерации.

Доказ: За една табла да има победа, доволно е барем еден ред, колона или дијагонала да припаѓа на еден ист знак. За да нема победа на 3×3 табла треба да е задоволено следното:

1. Секој ред мора да има барем едно празно поле.
2. Секоја колона мора да има барем едно празно поле.
3. Секоја дијагонала мора да има барем едно празно поле.

Бидејќи редовите меѓусебно не делат полиња, за да се задоволи (1) мора да има барем онолку празни полиња колку што има редови, т.е. 3. Доколку празните полиња се постават долж дијагоналата како на Слика 6, се задоволуваат сите 3 услови за да нема победа. Бројот на празни полиња не може да се намали без да се прекрши услов (1), а од тоа следи дека максималниот број на окупирани полиња од ист знак на 3×3 табла е $9 - 3 = 6$.

Слична имплементација се користи и во помошната класа State Evaluator за пронаоѓање на потенцијални победи (two-in-a-row) и спречување победи на противникот (blocking-moves). Дополнително, со користење на magic square индексирањето е олеснето барањето на позиции каде има повеќе од една потенцијална победа (forks).

2.2 Користени алгоритми

Компјутерот како противник може да игра според два алгоритми кои служат за бирање на оптимални потези: MiniMax и ExpectiMax.

2.2.1 MiniMax

MiniMax е алгоритам за правење одлуки во случаи каде што однесувањето на другиот играч е познато. Односно, алгоритмот очекува секој потег на противникот да е оптимален. Тој работи со хевристичка евалуација на состојбите на играта, креирање на дрво за пребарување низ состојби според легалните потези, максимизирање или минимизирање на оцените добиени од хевристичката функција и избирање на потег кој е најпогоден за моменталната состојба.

Идеално, пред да направи потег, алгоритмот би ги испитал сите идни состојби, но за оваа игра тоа не може да се постигне во реално време. Причината позади ова е факторот на разгранување, кој се менува низ текот на играта, но во најлош случај е 81. MiniMax има комплексност од $O(b^d)$ каде b е факторот на разгранување, а d е длабочината на дрвото на состојби.

Со оглед на тоа што комплексноста на алгоритмот е степенска, имплементиравме неколку оптимизации со цел да се намали времето на чекање за одлука.

1. Alpha-Beta Pruning

Ова е позната техника за забрзување на MiniMax алгоритмот [2]. Целта е да се намали бројот на јазли што треба да се евалуираат во дрвото на состојби. Ова се постигнува со поткастрување (pruning) на гранки од дрвото кои не влијаат на конечната одлука. Односно, ако при истражување на гранка се најде вредност полоша од предходно најдената, таа гранка може да се поткастри. Со ова, комплексноста на алгоритмот потенцијално може да се намали на $O(b^{d/2})$, но во најлош случај останува иста.

2. Ограничена длабочина

Бидејќи просторот на пребарување на играта е доста голем, дури и со поткастрување, за пребарување низ дрвото до целна состојба треба многу време. Првата идеја за справување со ова е да се воведе ограничена длабочина на дрвото. Ова значително ја подобрува брзината на алгоритмот, но губи во перформанси. *Преку тестови увидовме дека со длабочина 3 алгоритмот прави одлука речиси инстантно, додека со длабочина 5 времето за одлука е сеуште прифатливо со задоволителни перформанси.

* Мора да се потенцира дека овие резултати се добиени по имплементацијата на кеширање на состојби, која се дискутира во поголеми детали подолу.

3. Динамичка длабочина

На почетокот на играта можните патеки низ дрвото се најбројни, што значи дека времето за одлука е големо. При крај на играта кога бројот на легални потези е помал, разгранувањето на дрвото се намалува и побрзо се стига до целните состојби. Во тој момент зголемувањето на длабочината не влијае многу врз времето за одлука. Ова може да се искористи со имплементација на динамичка длабочина, која се одредува според следниот код:

```
INIT_DEPTH = 5
INIT_COUNTER = 0
THRESHOLD = 18
STEP = 3
BASE = 2

if moves_made > THRESHOLD and moves_made % STEP == 0:
     depth = depth + BASE ** counter
     counter = counter + 1
```

Секој пат кога алгоритмот ќе направи потег, се зголемува вредноста на $\mathrm{moves\_made}$ на тој играч. После 18-тиот потег, на секои 3 потези длабочината се зголемува за 2^{counter}.

4. **Предефинирани потези**

Во фазата на тестирање приметивме дека има потези кои често се прават на почеток на играта од некои веќе постоечки имплементации [3]. Пример за ова е првиот потег да се игра во центарот на таблата* или кога се прави прв потег на празна мала табла, да се игра така што противникот ќе мора да игра на истата табла. Дополнително, со ова се избегнува пребарување низ дрвото кога длабочината е сеуште мала.

* Додадовме опција за првиот потег да не биде секогаш ист, така што се бира случајно од сите можни први потези. Поради начинот на кој што работи MiniMax кога игра против самиот себе, првиот потег го дефинира текот на целата игра, па за да има разновидност во симулациите понатаму, ја воведовме оваа опција.

2.2.2 ExpectiMax

ExpectiMax е алгоритам за правење одлуки во сценарија каде исходите се несигурни или непредвидливи. Истиот работи на принцип како MiniMax, но се разликува во тоа што воведува случајни јазли кои претставуваат случајни настани со познати веројатности.

Нашата имплементација на алгоритмот претпоставува дека веројатноста противникот да одбере еден потег е еднаква со останатите. Алгоритмот ги користи истите оптимизации околу изборот на иницијалните потези и длабочината на дрвото, со клучната разлика дека не може да се воведе поткастрување. Ова го прави алгоритмот поспор, а комплексноста е иста со онаа на MiniMax пред alpha-beta pruning.

2.3 Дискусија на хевристички оцени

Во рамките на алгоритмите според кои компјутерот игра мора да има начин за евалуација на различни потези, од кои се избира најдобриот. Ова се постигнува со помош на хевристичка функција.

2.3.1 Евалуација на состојба

Кога размислувавме за начин на евалуација на состојбите, прва идеа ни беше да најдеме начин да ги оцениме малите табли [4][5]. Евалуацијата работи со тоа што наградува одредени потези или формации на таблите. Нашата хевристичка функција се фокусира на следните формации со соодветните оцени кои се движат во ранг $[-1000,\ +1000]$:

SCORE_WIN: Најпрвин се проверува дали на малата табла која се разгледува има победа. Доколку има, хевристичката функција враќа оцена **1000** и не проверува за други формации. Оваа вредност ја земавме произволно за понатамошни пресметки, со тоа што единственото ограничување е наредните оцени да се избрани така што нема да има формација која ќе врати оцена поголема од таа за победа.

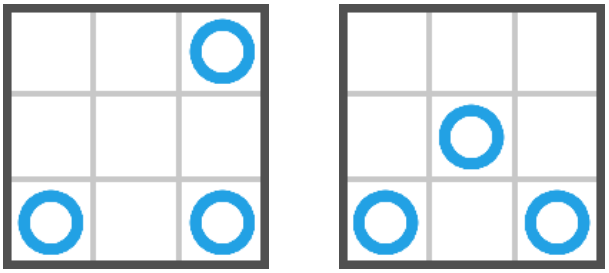
SCORE_TIE: Доколку таблата е пополнета и нема победа, се оценува со **0**.

SCORE_CENTER: Потези во центарот на таблата вредат **40**, бидејќи се дел од најголем број линии за победа (вкупно 4).

SCORE_CORNER: Потези во ќошевите на таблата вредат **16**, бидејќи овозможуваат 3 линии за победа и се поволни за формирање на forks.

SCORE_TWO_IN_ROW: Ова е формација каде што се пополнети две од три полиња со ист знак на линија за победа, притоа третото поле е празно. Се оценува со **56**. Овие формации се бараат на сличен начин како функцијата за пронаоѓање на победа.

SCORE_FORK: Forks се формации кои се состојат од 3 полиња со ист знак во една од шемите прикажани на Слика 7. Во случај да има барем еден fork на таблата, се додава **80** на вкупната оцена.



Слика 7: Шеми на forks.

SCORE_BLOCKING: Оцената за блокирање на two-in-a-row потези на противникот се вреднува со **40**.

Вредностите за секоја од дискутираните формации се земени произволно и преку симулирање на голем број игри, увидовме дека не постои табла која се оценува со вредност еднаква или повисока од вредноста за победа.

Евалуацијата на мала табла се прави за двата знаци, со тоа што за секоја од формациите, X додава на вкупната оцена, а O одзема. Ова овозможува инверзни табли во однос на знаци да имаат иста оцена со спротивен знак.

За оцената на целата состојба, по евалуација на сите мали табли, добиените вредности се скалираат според тоа каде се наоѓаат на големата табла. Таблата во средина добива најголем удел во оцената – 20%, потоа ќошевите – 15%, и преостанатите полиња – 5%. Се одлучивме за овој пристап бидејќи играчот реално ретко има избор каде може да игра на големата табла. Со скалирањето, повторно се дава тежина на поповолните позиции и се избегнува непотребно евалуирање на големата табла.

Дополнително пробавме да додадеме и казна за потези кои му дозволуваат на следниот играч да игра на било кое поле на големата табла (**FREE_MOVE_PENALTY**), така што се одзема **100** по апсолутна вредност од

скалираната вкупна оцена на таблата, но во пракса ова ги влоши перформансите на играчите и се решивме да го отстраниме. Сметаме дека ова се должи на тоа што алгоритамот станува многу одбивен на играње на големата табла, така што почнува да ги избегнува потезите кои реално водат до победа. Дополнително, по повеќе размислување, заклучивме дека FREE_MOVE_PENALTY всушност не е потребно, бидејќи самиот алгоритам кога пребарува низ дрвото може да процени дали дозволување на противникот да игра каде било е поволен потег.

2.3.2 Евалуација на игра

Моменталната имплементација за евалуирање на состојба не зема предвид колку е блиску играта до завршување. Сакавме доколку сите наредни потези водат до загарантирана победа на еден од играчите, тоа да се увиди во оцената за играта (таа да биде ± 1000 соодветно).

За таа цел, решивме да имплементираме уште еден евалуатор со кој ќе имаме увид кој играч има поголеми шанси за победа. Тоа се постигнува користејќи посебна инстанца од MiniMax и земање на просек од оцените на сите наредни можни потези пресметани на одредена длабочина.

Овој евалуатор го користиме и за приказ на evaluation bar.

2.4 Предизвици и решенија

За време на имплементација на различните делови на проектот, се соочивме со различни предизвици. Во главно, не бевме задоволни со времето на извршување на алгоритмите кои прават информирани одлуки.

За да ги откриеме тесните грла ја употребивме библиотеката cProfile, која послужи за мерење на времето на извршување на различните функции во кодот. Така увидовме дека голем дел од времето го одзема функцијата за евалуација на состојби. Со ова се справивме така што воведовме кеширање на малите табли и нивните оцени [6]. Дополнително, кога ќе се одреди оцена на една мала табла, се кешира и нејзината инверзна табла. Оваа промена значително го забрза времето на извршување со тоа што се избегнуваат голем број евалуации на табли кои веќе еднаш биле оценети. Бидејќи во дадено време се игра само на една мала табла, останатите мали табли се непроменети, па се појавуваат често во рамките на дрвото за пребарување. Тука кеширањето е најмоќно со кое се избегнуваат огромен број на реевалуации.

Симулиравме 200 игри помеѓу два MiniMax играчи со исти параметри, од кои половина имаа кеширање, а останатите не. По споредување на перформансите меѓу двете симулации, добивме забрзување од 90.85% .

Време на извршување на 100 игри без кеширање: 118.04 секунди.
Време на извршување на 100 игри со кеширање: 10.79 секунди.

3 Резултати

Додека работевме на проектот, потребен ни беше начин да видиме колку добро играат различните имплементации на алгоритмите. За таа цел, воведовме уште една помошна класа за симулирање на игри и мерење на перформанси.

Клучните влезни аргументи на оваа класа се бројот на игри кои ќе бидат симулирани и инстанци на два играчи. По секоја симулација се зачувува резултатот од играта и на крај се прикажува целосен преглед на перформансите за двата играчи.

Одбравме неколку конкретни имплементации со различни параметри и симулиравме по 100 игри за секоја комбинација. Резултатите од овие симулации се прикажани во Табела 1.

× \ O	MiniMax Dynamic 5	MiniMax Dynamic 3	MiniMax Dynamic 5 + P	MiniMax Static 1	ExpectiMax	Random
MiniMax Dynamic 5	56 / 18 / 26	63 / 15 / 22	51 / 22 / 27	93 / 2 / 5	81 / 3 / 16	99 / 0 / 1
MiniMax Dynamic 3	21 / 11 / 68	48 / 15 / 37	20 / 7 / 73	93 / 5 / 2	53 / 7 / 40	100 / 0 / 0
MiniMax Dynamic 5 + P	51 / 8 / 41	77 / 11 / 12	51 / 14 / 35	99 / 1 / 0	75 / 4 / 21	99 / 1 / 0
MiniMax Static 1	3 / 7 / 90	15 / 2 / 83	9 / 5 / 86	50 / 6 / 44	3 / 5 / 92	96 / 3 / 1
ExpectiMax	29 / 2 / 69	44 / 6 / 50	27 / 1 / 72	97 / 1 / 2	51 / 0 / 49	100 / 0 / 0
Random	0 / 0 / 100	0 / 2 / 98	0 / 0 / 100	0 / 2 / 98	0 / 0 / 100	40 / 25 / 35

Табела 1: Перформанси на различни имплементации.

За различни MiniMax играчи во табелта, бројчаните вредности ја означуваат иницијалната длабочина на пребарувачкото дрво, Dynamic и Static се однесуваат на тоа дали длабочината ќе се менува во текот на играта, и +P означува дали хевристичката функција користи FREE_MOVE_PENALTY. Дополнително, ExpectiMax игра со динамичка длабочина и почетна вредност 3. Причината за ова е тоа што алгоритмот не користи поткастрување и е доста поспор од останатите. Исто така, за побрзо извршување на симулациите, направивме мала измена во параметрите на динамичката длабочина, STEP = 4.

Во рамките на редовите се играчите кои го прават првиот потег, односно играат со X, а во рамките на колоните се играчите кои играат со O. Секоја келија содржи 3 вредности: бројот на победи на X (од лево), бројот на победи на O (од десно), и бројот на нерешени игри (во средина).

Од добиените резултати може да се воочи дека убедливо најдобри перформанси има MiniMax со динамичка длабочина и иницијална вредност 5.

Освен преку симулации, алгоритмите и хевристичките функции ги тестиравме и против веќе постоечки имплементации на codingames.com. Codingames [7] е веб страна каде корисниците можат да се натпреваруваат во разни игри со нивни AI модели, кои потоа се рангирани меѓусебно според нивните перформанси. За време на пишување на овој извештај, нашиот MiniMax модел се рангираше меѓу првите 619 од 9331 модели, односно во најдобрите 6.6% на страната за играта UTTT. Овие резултати беа постигнати со статичка длабочина со вредност 3. Причината за ова е временскиот лимит поставен на страната, кој не дозволува повеќе од 0.1 секунди за извршување на секој потег. Мислиме дека доколку можешевме да користиме поголема длабочина, резултатите би биле уште подобри.

4 Заклучок

Во рамките на овој извештај презентиравме сеопфатна имплементација на вештачка интелигенција за играта Ultimate Tic-Tac-Toe, користејќи напредни алгоритми за информирано пребарување и хевристичка евалуација. Клучните научени и постигнати резултати можат да се сумираат во следните точки:

- Развивме софистицирана AI имплементација која користи MiniMax и ExpectiMax алгоритми со повеќе оптимизации, вклучително алфа-бета поткастрување, динамичка длабочина на пребарување и предефинирани почетни потези.

- Креиравме иновативен пристап за евалуација на состојби користејќи magic square индексирање и повеќестепенска хевристичка функција која зема предвид локални и глобални аспекти на играта.

- Постигнавме значително подобрување на перформансите преку воведување на кеширање на состојби, со кое времето на извршување беше забрзано за 90.85% во споредба со базичната имплементација.

- Резултатите од симулациите и рангирањето на Codingames платформата потврдија висока ефективност на нашиот пристап, со пласман во најдобрите 6.6% од достапните модели.

Идни правци за истражување би можеле да вклучат: експериментирање со други алгоритми за пребарување како Monte Carlo Tree Search и проширување на техниките за оптимизација со цел да се намали пресметковната комплексност.

Овој труд демонстрира дека со внимателен дизајн на алгоритми и софистицирани техники за евалуација, може да се создаде високо-ефективен AI играч, дури и за комплексни игри со големи простори на пребарување.

Референци

[1] *freeCodeCamp.org – Дискусија за користење на magic square и MiniMax.*

[2] *GeeksForGeeks.org – Код за MiniMax со алфа-бета потkастрување.*

[3] *CodinGames – Ранг листа со најдобрите играчи.*

[4] *StackExchange (Board Games) – Дискусија за хевристика.*

[5] *Github Repository – Дискусија за хевристика.*

[6] *LarsWaecher – Оптимизации за MiniMax и хевристичка функција.*

[7] *CodinGames – Ultimate Tic-Tac-Toe.*

[8] *YouTube – Различни AI модели за UTTT.*

[9] *YouTube – MiniMax модел за UTTT и различни хевристички оцени.*

Линкови до кодот:

[*] zerofyy/Ultimate-Tic-Tac-Toe-AI

[*] ilina-d/Ultimate-Tic-Tac-Toe-AI