

**SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR,
INFORMATIKA SZAK**



SAPIENTIA
ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM

Monte Carlo algoritmus használata a Battleship játékban

DIPLOMADOLGOZAT

Témavezető:
Dr. Horobeţ Emil
Egyetemi docens

Végzős hallgató:
Hegyí Benjámin

2023

UNIVERSITATEA SAPIENTIA DIN CLUJ-NAPOCA
FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE,
SPECIALIZAREA INFORMATICĂ



UNIVERSITATEA
SAPIENTIA

Utilizarea algoritmului Monte Carlo în jocul Battleship

LUCRARE DE DIPLOMĂ

Coordonator științific:
Dr. Horobeț Emil
Conferențiar universitar

Absolvent:
Hegyi Benjámín

2023

**SAPIENTIA HUNGARIAN UNIVERSITY OF
TRANSYLVANIA
FACULTY OF TECHNICAL AND HUMAN SCIENCES
INFORMATION SCIENCE SPECIALIZATION**



SAPIENTIA
HUNGARIAN UNIVERSITY
OF TRANSYLVANIA


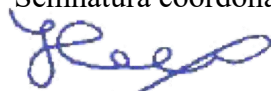
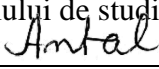
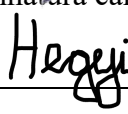
Using the Monte Carlo algorithm in the game Battleship

BACHELOR THESIS

Scientific advisor:
Dr. Horobet Emil
Associate professor

Student:
Hegyí Benjámín

2023

| | | |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|------------------|
| UNIVERSITATEA „SAPIENTIA” din CLUJ-NAPOCA Facultatea de Științe Tehnice și Umaniste din Târgu Mureș Programul de studii: Informatică | | Viza facultății: |
| LUCRARE DE DIPLOMĂ | | |
| Coordonator științific: dr. Horobeț Emil | Candidat: Hegyi Benjámín Anul absolvirii: 2023 | |
| a) Tema lucrării de licență: Utilizarea algoritmului Monte Carlo în jocul Battleship | | |
| b) Problemele principale tratate: Algoritmul Monte Carlo este o tehnică de calcul populară utilizată în multe domenii, cum ar fi fizica, finanțele și informatica. Una dintre aplicațiile proeminente ale algoritmului Monte Carlo este în inteligența artificială a jocurilor. Algoritmii Monte Carlo sunt utili în inteligența artificială a jocurilor, deoarece pot simula și evalua diferite mișcări, permițând inteligenței artificiale să selecteze cea mai bună mișcare pe baza rezultatelor simulării. În această proiect se investighează utilizarea algoritmului Monte Carlo în jocul Battleship, precum și punctele forte și punctele slabe ale acestuia. | | |
| c) Desene obligatorii:- - Schema bloc a aplicației - Diagrame de proiectare pentru aplicația software realizată. | | |
| d) Softuri obligatorii:- Implementarea respectivelor algoritmi în Python pentru a le compara eficiența lor. | | |
| e) Bibliografia recomandată: [1] Metropolis, Nicholas, et al. "Equation of state calculations by fast computing machines." The journal of chemical physics 21.6 (1953): 1087-1092. [2] S. Brown. How to play the battleship board game. 2020-11-12. [Online]. Available: https://www.thesprucecrafts.com/the-basic-rules-of-battleship-411069 [3] M. Kalos and P. Whitlock. Monte carlo methods. 2013-03-01. [Online]. Available: https://phyusdb.files.wordpress.com/2013/03/monte-carlo-methods-second-revised-and-enlarged-edition.pdf | | |
| f) Termene obligatorii de consultații: Minimum o dată din două în două săptămâni. | | |
| g) Locul și durata practicii: Universitatea „Sapientia” din Cluj-Napoca, Facultatea de Științe Tehnice și Umaniste din Târgu Mureș, sala / laboratorul - Primit tema la data de: 01.10.2022 Termen de predare: 30.06.2023 | | |
| Semnătura Director Departament  | Semnătura coordonatorului  | |
| Semnătura responsabilului programului de studiu  | Semnătura candidatului  | |

Declarație

Subsemnatul/a HEGYI BENJAMIN, absolvent(ă) al/a specializării
INFORMATICA, promoția 2023, cunoscând
prevederile Legii Educației Naționale 1/2011 și a Codului de etică și deontologie profesională a
Universității Sapienția cu privire la furt intelectual declar pe propria răspundere că prezenta
lucrare de licență/proiect de diplomă/disertație se bazează pe activitatea personală,
cercetarea/proiectarea este efectuată de mine, informațiile și datele preluate din literatura de
specialitate sunt citate în mod corespunzător.

Localitatea, TÂRGU MUREȘ
Data: 2023.06.15

Absolvent

Semnătura... Hegy

Kivonat

A Monte Carlo algoritmus egy népszerű számítási technika, amelyet számos területen, például a fizikában, a pénzügyekben és az informatikában használnak. A Monte Carlo algoritmus egyik kiemelkedő alkalmazása a játékokat játszó mesterséges intelligencia. A Monte Carlo algoritmusok azért hasznosak a mesterséges intelligencia játékokban, mert képesek szimulálni és értékelni a különböző lépéseket, lehetővé téve az AI számára, hogy a szimuláció eredményei alapján kiválassza a legjobb lépést. Ebben a tanulmányban a Monte Carlo algoritmus használatát vizsgáljuk a Battleship játékban, valamint annak erősségeit és gyengeségeit.

Kulcsszavak: Monte Carlo algoritmus, Minimax algoritmus, Alpha-Beta metszés, Monte Carlo Tree Search

Rezumat

Algoritmul Monte Carlo este o tehnică de calcul populară utilizată în multe domenii, cum ar fi fizica, finanțele și informatica. Una dintre aplicațiile proeminente ale algoritmului Monte Carlo este în inteligența artificială a jocurilor. Algoritmii Monte Carlo sunt utili în jocurile de inteligență artificială, deoarece pot simula și evalua diferite mișcări, permițând inteligenței artificiale să selecteze cea mai bună mișcare pe baza rezultatelor simulării. În această lucrare, investigăm utilizarea algoritmului Monte Carlo în jocul Battleship, precum și punctele forte și punctele slabe ale acestuia.

Cuvinte de cheie: Algoritmul Monte Carlo, Algoritmul Minimax, Intersecția Alpha-Beta, Monte Carlo Tree Search

Abstract

The Monte Carlo algorithm is a popular computational technique used in many fields such as physics, finance and computer science. One of the prominent applications of the Monte Carlo algorithm is in game-playing artificial intelligence. Monte Carlo algorithms are useful in AI games because they can simulate and evaluate different moves, allowing the AI to select the best move based on the results of the simulation. In this paper, we investigate the use of the Monte Carlo algorithm in the game Battleship and its strengths and weaknesses.

Keywords: Monte Carlo algorithm, Minimax algorithm, Alpha-Beta pruning, Monte Carlo Tree Search

Tartalomjegyzék

| | |
|-----------------------------------------------------------------------|-----------|
| 1. Bevezető | 11 |
| 1.1. Háttér | 11 |
| 1.2. A problémafelvetés | 11 |
| 1.3. Célkitűzések | 12 |
| 1.4. Hatókör és korlátozások | 12 |
| 1.5. Módszertan | 12 |
| 2. Irodalmi áttekintés | 13 |
| 2.1. A Battleship játék áttekintése | 13 |
| 2.2. A Monte Carlo algoritmus bevezetése | 14 |
| 2.3. A Monte Carlo Tree search algoritmus bevezetése | 14 |
| 2.4. A Monte Carlo Tree search korábbi alkalmazásai játékokban | 16 |
| 2.5. A Minimax algoritmus bevezetése | 17 |
| 2.6. Az Alpha-Beta metszés bevezetése | 17 |
| 2.7. Az Alpha-Beta metszés korábbi alkalmazásai játékokban | 18 |
| 3. Fejezet: Elméleti alapok | 20 |
| 3.1. Programozás és python technológiák | 20 |
| 3.2. A Monte Carlo Tree Search algoritmus | 21 |
| 3.2.1. Kiválasztás | 21 |
| 3.2.2. Bővítés | 22 |
| 3.2.3. Szimuláció | 22 |
| 3.2.4. Visszaterjesztés | 22 |
| 3.3. Monte Carlo Tree Search algoritmus megvalósítása a Battleshipben | 23 |
| 3.4. A Minimax algoritmus | 25 |
| 3.5. A Minimax algoritmus megvalósítása a Battleshipben | 26 |
| 3.6. Az Alpha-Beta metszés algoritmus | 28 |
| 3.7. Az Alpha-Beta metszés algoritmus megvalósítása a Battleshipben | 29 |
| 3.8. A játékmechanika és a döntéshozatal elemzése | 30 |
| 4. A Monte Carlo algoritmus erősségei | 36 |
| 4.1. Komplex és nem determinisztikus játékok kezelése | 36 |
| 4.2. Skálázhatóság | 37 |
| 4.3. Egyszerűség | 38 |
| 5. A Monte Carlo algoritmus gyengeségei | 40 |
| 5.1. Nehézségek bizonyos játékhelyzetek kezelésében | 40 |

| | |
|-----------------------------------------------------------------------------|-----------|
| 5.2. A pontosság hiánya és szimulációs eredmények változékonysága | 41 |
| 6. Technikák összehasonlítása | 42 |
| 6.1. Minimax | 42 |
| 6.2. Alpha-beta metszés | 44 |
| Összefoglaló | 46 |
| Köszönetnyilvánítás | 47 |
| Ábrák jegyzéke | 48 |
| Táblázatok jegyzéke | 49 |
| Irodalomjegyzék | 51 |

1. fejezet

Bevezető

A Mesterséges Intelligencia (MI) forradalmat hozott sok iparágban, többek között a játékiparban is. A játékokban használt MI algoritmusok tanulnak a múltbeli tapasztalatokból és bizonyos játékokban jobban teljesítenek, mint az emberi játékosok. A Monte Carlo algoritmus, egy szimuláció alapú technika, egy népszerű MI algoritmus, amit játékokban is használnak. Kísérletek elvégzésével, adatgyűjtéssel és egy Monte Carlo-alapú Battleship AI megvalósításával a dolgozat célja, hogy értékelje az algoritmus teljesítményét, és tárgyalja annak korlátait. A tanulmány eredményei betekintést nyújtanak a Monte Carlo algoritmus stratégiai játékokban való alkalmazásába

1.1. Háttér

A Monte Carlo algoritmus egy szimuláció alapú algoritmus, ami statisztikai módszereket használ a komplex rendszerek közelítésére. A nevét a Monacói Monte Carlo kaszinóról kapta, amely ismert a szerencsejátékokról. Az algoritmust eredetileg John von Neumann és Stanislaw Ulam fejlesztették ki az 1940-es években a neutronok viselkedésének tanulmányozására egy nukleáris reaktorban [1]. Azóta különböző területeken is alkalmazzák, beleértve a játékokat is.

A Battleship egy népszerű stratégiai játék, amelyet évtizedek óta minden korosztály élvez. Két játékos próbálja elsüllyeszteni egymás hajóit úgy, hogy stratégiailag kitalálják azok helyét egy rácson. Az évek során a Battleship a mesterséges intelligencia (AI) érdeklődésének középpontjába került, a kutatók különböző algoritmusokat fejlesztettek ki a számítógépes játékosok teljesítményének javítására. Ezeknek az algoritmusoknak az a célja, hogy javítsák a számítógépes játékosok döntéshozatali képességeit és stratégiai gondolkodását, és ezáltal nagyobb kihívást jelentsenek az emberi játékosok számára.

1.2. A problémafelvetés

A Battleshipben vizsgált számos algoritmus közül a Monte Carlo algoritmus elismerést szerzett más játéktérületeken elért hatékonysága miatt. A Monte Carlo algoritmus véletlenszerű mintavételt és statisztikai elemzést használ a lehetséges eredmények szimulálására és megalapozott döntések meghozatalára. A Monte Carlo algoritmus más játékokban is elért sikereket ezért is célja a dolgozatnak a Battleshipben való alkalmazásának kutatása. Ez felveti a kérdést, hogy a Monte Carlo algoritmus javíthatja-e a számítógépes

játékosok teljesítményét a Battleshipben, és hozzájárulhat-e a játékelmény érdekesebbé és kihívást jelentőbbé tételéhez.

1.3. Célkitűzések

A tanulmány célja, a következő pontokban leírható:

- A Battleship játék és a Monte Carlo algoritmus áttekintése, beleértve a legfontosabb fogalmakat és mechanikát.
- A Monte Carlo algoritmus hatékonyságának értékelése a számítógépes játékosok teljesítményének növelésében a Battleshipben.
- Egy Monte Carlo-alapú Battleship mesterséges intelligencia megtervezése és megvalósítása, amely beépíti az algoritmust a számítógépes játékos döntéshozatali folyamatába.
- A Monte Carlo-alapú Battleship mesterséges intelligencia teljesítményének értékelése kísérletezéssel és összehasonlítása más létező stratégiákkal.

1.4. Hatókör és korlátozások

Ez a dolgozat kifejezetten a Monte Carlo algoritmus alkalmazását vizsgálja a Battleshipben, és összehasonlítást végez más mesterséges intelligencia algoritmusokkal, pontosabban a Minimax és az Alfa-Béta metszéssel. Ugyanakkor nem vizsgál további mesterséges intelligencia algoritmusokat vagy technikákat ezen a három algoritmuson túl. Az elvégzett kísérletek és az összegyűjtött adatok egy adott adathalmazra korlátozódnak. Fontos megjegyezni, hogy az algoritmusok megvalósítása egy adott felépítéshez igazodik, és nem biztos, hogy közvetlenül alkalmazható más rendszerekre. Összességében a tanulmány átfogó áttekintést fog adni a Monte Carlo algoritmusról és annak alkalmazásairól a játék alapú mesterséges intelligenciában.

1.5. Módszertan

A tanulmány rendszerezett áttekintést készít a Monte Carlo algoritmusról és annak alkalmazásairól a játékokban használt MI-ben. A tanulmány tárgyalja a Monte Carlo algoritmus használatát konkrétan a Battleship játéknál. A Minimax algoritmust és annak optimalizálási technikáját, az Alpha-Beta metszést alkalmazzuk, hogy megvizsgáljuk hatékonyságukat a számítógépes játékosok teljesítményének növelésében a Battleship játékban.

2. fejezet

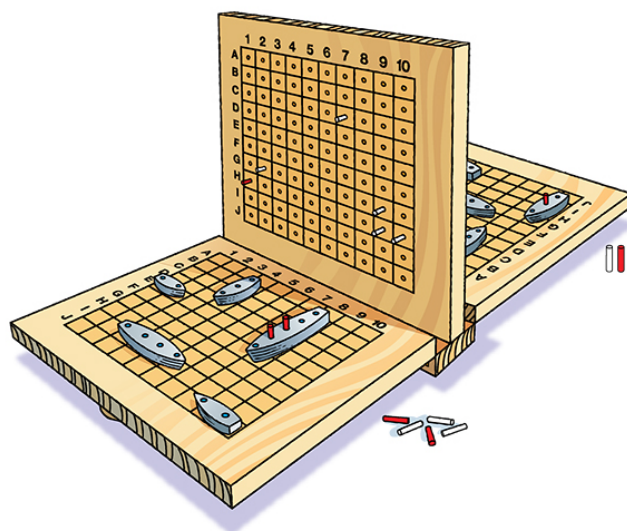
Irodalmi áttekintés

Ebben a fejezetben egy általános leírást kaphatunk a dolgozatban megemlített játékokról illetve algoritmusokról. Megismerhetjük a dolgozat különböző elemeit és elkezdhetünk gondolkodni a közöttük levő kapcsolatokról és megismerhetünk más játékokat is amelyekben felhasználták még az adott algoritmusokat.

2.1. A Battleship játék áttekintése

A Battleship egy klasszikus stratégiai játék, amely a 20. század elejére nyúlik vissza. A játék célja, hogy elsüllyeszd az ellenfél összes hajóját, mielőtt ők süllyesztenék el a tiédet. A játékot általában egy rácsos táblán játsszák, és minden játékosnak saját táblája van. Az alábbiakban áttekintjük, hogyan játsszák a játékot jellemzően:

- A tábla felállítása: Minden játékosnak van egy rácsos táblája, amelyen elhelyezi a hajóflottáját. A játék alapváltozatában öt hajó szerepel: a hordozó (5 mező hosszú), a csatahajó (4 mező hosszú), a cirkáló (3 mező hosszú), a tengeralattjáró (3 mező hosszú) és a romboló (2 mező hosszú). A játékosok felváltva helyezik el hajóikat a táblán, és az ellenfél elől rejtve tartják elhelyezésüket.
- Játékmenet: Miután a táblákat felállították, a játékosok felváltva próbálják kitalálni az ellenfél hajóinak helyét. A táblát általában betűs sorokra és számozott oszlopokra osztják (pl. A-J és 1-10). A soron lévő játékos egy koordinátát, például "B5", kiált, hogy jelezze az ellenfél táblájának egy adott pozícióját, amelyet megcéloz.
- Találatok és hibák: Miután egy játékos tippel, az ellenfele elárulja, hogy az találat vagy melléfogás volt-e. Ha a tipp olyan helynek felel meg, ahol az ellenfél hajója jelen van, az találat. A játékos megjelöli a találatot a nyomkövető rácsán, gyakran egy piros pálcikával vagy jelölővel. Ha a találat nem talál el egy hajót, az nem talált, és a játékos ezt egy fehér cövekkal vagy jelölővel jelöli.
- Nyomon követése a lövések: A játékosok saját nyomon követik rácsukat, használva a találataik és az eredményeit. Ez lehetővé teszi számukra, hogy az elért találatok és elhibázott találatok alapján stratégiát alakítsanak ki.



2.1. ábra. Példa egy valós Battleship táblára.

- A játék megnyerése: A játék addig folytatódik, amíg az egyik játékos sikeresen el nem süllyeszti az ellenfél összes hajóját. Az a játékos, aki először süllyeszti el az összes hajót, megnyeri a játékot. [2]

2.2. A Monte Carlo algoritmus bevezetése

A Monte Carlo (MC) algoritmus egy erőteljes statisztikai technika, amely véletlenszerű mintavételezést használ numerikus eredmények becslésére. Az algoritmust először Metropolis és Ulam vezette be az 1940-es években az atomreaktorok neutron-transzportjának megoldására. Azóta széles körben használják különböző területeken, mint például a fizika, az építőmérnöki tervezés, a pénzügy és a játékprogramokban.[3]

2.3. A Monte Carlo Tree search algoritmus bevezetése

A Monte Carlo Tree Search (MCTS) egy heurisztikus keresési szabályrendszer, amely nagy figyelmet és hírnevet szerzett a mesterséges intelligencia területén, különösen a döntéshozatal és a játékok területén. Ismert arról, hogy képes hatékonyan kezelni az összetett és stratégiai videojátékokat, amelyek hatalmas keresési terekkel rendelkeznek, ahol a hagyományos algoritmusok nehézségekbe ütközhetnek a végrehajtható műveletek vagy akciók puszta száma miatt.

A MCTS egyesíti a Monte Carlo-stratégiák szabványait, amelyek a véletlenszerű mintavételre és a statisztikai kiértékelésre támaszkodnak, a faalapú keresési technikákkal. A hagyományos keresőalgoritmusokkal ellentétben, amelyek a teljes keresési terület kimerítő feltárására támaszkodnak, a MCTS a mintavételezésre és a keresési terület csak ígéretes területeinek feltárására specializálódott.

A MCTS hátterében álló központi ötlet az, hogy a keresési fát fokozatosan építi fel azáltal, hogy több véletlenszerű előadást (rendszeresen rollout vagy ployout néven ismert) szimulál az aktuális rekreációs nemzetből. Ezeket a szimulációkat addig végzik,

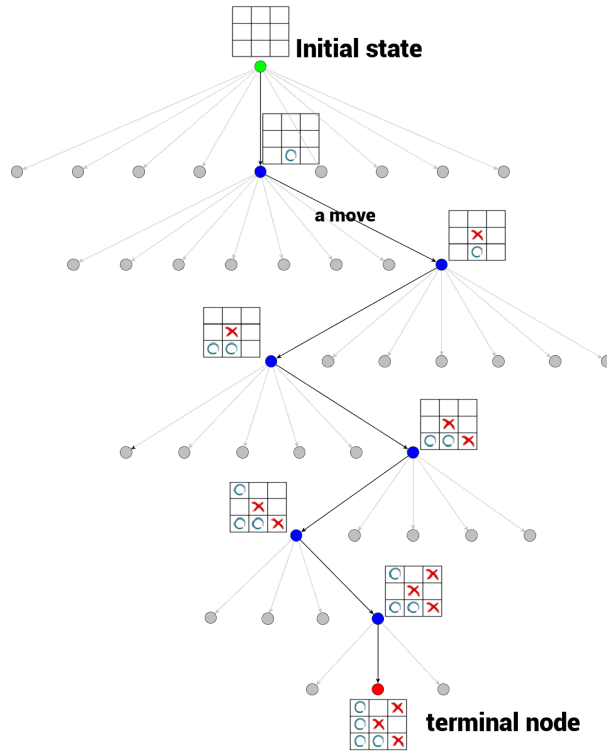
amíg egy végállapotot vagy egy előre meghatározott intenzitást el nem érnek. Ezeknek a szimulációknak az eredményei ezután visszafelé terjednek felfelé a fán, frissítve a játék bizonyos szakaszaiban meglátogatott csomópontok rekordjait, amelyek magukban foglalják a látogatások széles skáláját és a nyerési arányokat.

A keresés előrehaladtával a MCTS dinamikusan egyensúlyba hozza a felfedezést és a kihasználást. A lépéseket úgy választja ki, hogy figyelembe veszi mind a magas nyerési aránnyal rendelkező, különösen ígéretes lépések kihasználását, mind a még fel nem tárt vagy kevésbé feltárt lépések feltárását. Ez a kiegyensúlyozás egy felső bizalmi biztos (UCB) komponensek felhasználásával történik, amely magában foglalja a Felső bizalmi határokat a fákhhoz (UCT), annak eldöntésére, hogy mely mozgásokat vagy csomópontokat kell meglátogatni a vadászat időtartama alatt.

A MCTS-t számos területen hatékonyan alkalmazták, többek között társasjátékokban (pl. Go, sakk és shogi), kártyás videojátékokban (pl. póker) és videojátékokban. Nagyszerű általános teljesítményt nyújtott számos kihívást jelentő szabadidős-játékos forogatókönyvben, gyakran felülmúlva az emberi megértést. A MCTS-t meghosszabbították és testre szabták különböző problémás területek kezelésére is, beleértve a tervkészítést, az ütemezést és az optimalizálást.

A MCTS egyik nagyszerű előnye, hogy ismeretlen vagy tökéletlen adatokkal rendelkező videojátékokat is képes kezelni, mivel a játék állapotának teljes ismerete helyett statisztikai mintavételre támaszkodik. Ezenkívül a MCTS skálázható és hatékonyan párhuzamosítható, így alkalmas a szétszórt számítástechnikai és többmagos architektúrákhoz.

A Monte Carlo Tree Search (MCTS) egy keresési technika a mesterséges intelligencia (AI) területén. Ez egy valószínűségi és heurisztikus vezérelt keresési algoritmus, amely a klasszikus fakesés implementációkat a megerősítéses tanulás gépi tanulási elveivel kombinálja. A fakesésnél mindig fennáll annak a lehetősége, hogy az aktuálisan legjobb akció valójában nem a legoptimálisabb akció. Ilyen esetekben a MCTS algoritmus hasznosnak bizonyul, mivel a tanulási fázisban a jelenlegi optimálisnak vélt stratégia helyett időszakosan más alternatívákat is értékel, és azokat hajtja végre. Ezt nevezik "feltárás-kihasználás kompromisszumnak". Az algoritmus kihasználja az eddig legjobbnak ítélt cselekvéseket és stratégiákat, de továbbra is fel kell tárnia az alternatív döntések lokális terét, és meg kell találnia, hogy azok helyettesíthetik-e a jelenlegi legjobbat. A felfedezés segít a fa feltárásában és felfedezésében a még fel nem fedezett részek feltárásában, ami optimálisabb útvonal megtalálását eredményezheti. Más szóval azt mondhatjuk, hogy a felfedezés jobban bővíti a fa szélességét, mint mélységét. A felfedezés hasznos lehet annak biztosítására, hogy a MCTS ne hagyjon figyelmen kívül potenciálisan jobb utakat. Nagyszámú lépést vagy ismétlést tartalmazó helyzetekben azonban gyorsan hatástalanná válik. Ennek elkerülése érdekében ezt ellensúlyozza a kiaknázás. A kiaknázás egyetlen olyan útvonalhoz ragaszkodik, amelynek becsült értéke a legnagyobb. Ez egy mohó megközelítés, és ez a fa mélységét jobban megnöveli, mint a szélességét. Egyszerűen fogalmazva, a fákra alkalmazott UCB formula segít kiegyensúlyozni a feltárás-kihasználás kompromisszumot azáltal, hogy rendszeresen feltárja a fa viszonylag feltáratlan csomópontjait, és potenciálisan több optimális útvonalat fedez fel, mint amit éppen kihasznál. E tulajdonsága miatt a MCTS különösen hasznos lesz a mesterséges intelligencia (AI) problémákban az optimális döntések meghozatalában.[4][5]



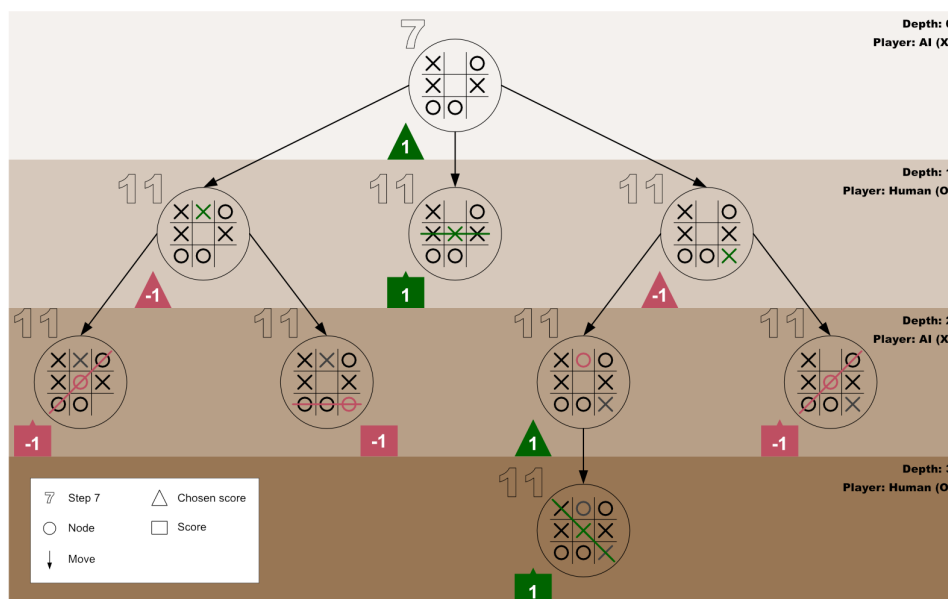
2.2. ábra. Példa monte carlo tree searchre a tic-tac-toe-ban.

2.4. A Monte Carlo Tree search korábbi alkalmazásai játékokban

A Monte Carlo-t a játékokban a Monte Carlo tree search (MCTS) algoritmuson keresztül használták, amely egy olyan algoritmus, amelyet a játék legígéretesebb lépéseinek elemzésére használnak. A MCTS sok végigjátszáson, más néven roll-outon alapul, ahol a játékot véletlenszerű lépések kiválasztásával végigjátsszák. Az egyes végigjátszások végeredményét ezután a játékfa csomópontjainak súlyozására használják, hogy a jövőbeli kijátszások során nagyobb valószínűséggel válasszanak jobb csomópontokat.

A MCTS négy lépésből áll: kiválasztás, bővítés, szimuláció és visszaterjesztés. A szimulációs lépés során egy vagy több szimulációt hajtanak végre, és minden egyes szimulációért jutalmat halmoznak fel. A kimeneti szabály általában egyszerű vagy akár tisztán véletlenszerű, így gyorsan megvalósítható. Például egy győzelem +1 jutalmat, egy döntetlen 0 jutalmat, egy vereség pedig -1 jutalmat eredményezhet.

A Monte Carlo roll-out egy játékban egy pozíció értékének becslésére szolgál, az adott pozíciótól a végéig történő véletlenszerű játszmák lejátszásával. A Monte Carlo roll-out a szimuláció egy olyan típusa, amelyet arra használnak, hogy megbecsüljék egy pozíció értékét egy játszmában azáltal, hogy véletlenszerű játszmákat játszanak le az adott pozíciótól a végéig vagy egy megadott mélységig. A Monte Carlo roll-out sok kijátszáson, más néven roll-outon alapul, ahol a játékot a végéig játsszák ki a lépések véletlenszerű kiválasztásával. Az egyes kijátszások végeredményét ezután a játékfa csomópontjainak súlyozására használják, hogy a jövőbeli kijátszások során nagyobb valószínűséggel választanak ki jobb csomópontokat.[4][5]



2.3. ábra. Példa minimax algoritmusra tic-tac-toe-ban.

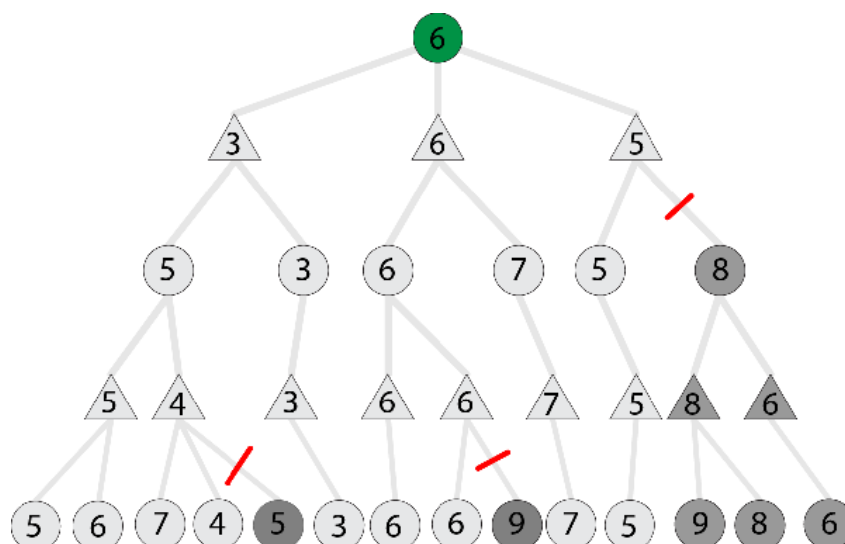
2.5. A Minimax algoritmus bevezetése

A minimax algoritmus nevét a minimax-tételből szerezte melyet John von Neumann bizonyított 1928-ban. A minmax a mesterséges intelligenciában, a döntéelméletben, a játékelméletben, a statisztikában és a filozófiában használt döntési szabály, amely a lehetséges veszteség minimalizálására szolgál a legrosszabb eset (maximális veszteség) forgatókönyve esetén. Ha nyereséggel foglalkozunk, akkor "maximin" néven említik - a minimális nyereség maximalizálása. Eredetileg a többszereplős zéróösszegű játékelméletre fogalmazódott meg, lefedve mind azokat az eseteket, amikor a játékosok váltakozó lépéseket tesznek, mind azokat, amikor egyidejűleg lépnek, de kiterjesztették bonyolultabb játékokra és a bizonytalanság jelenlétében történő általános döntéshozatalra is. A zéróösszegű játék a játékelméletben és a közgazdaságtanban egy olyan helyzet matematikai ábrázolása, amelyben két fél vesz részt, és amelynek eredménye az egyik fél számára előny, a másik számára pedig ezzel egyenértékű veszteség. Más szóval, az egyes játékos nyeresége egyenértékű a kettes játékos veszteségével, aminek következtében a játék nettó hasznának javulása nulla. [6][7]

2.6. Az Alpha-Beta metszés bevezetése

A "metszés" szó az ágak és levelek levágását jelenti. Az adattudományban a metszés egy sokat használt kifejezés, amely a döntési fák és a véletlen erdő utáni és előtti metszésre utal. Az alfa-béta metszés nem más, mint a haszontalan ágak metszése a döntési fákban. Ezt az alfa-béta metszési algoritmust a kutatók az 1900-as években önállóan fedezték fel.

Az alfa-béta metszés a minimax algoritmus optimalizálási technikája, amelyet az előző szakaszban tárgyaltunk. A metszés szükségessége abból a tényből adódott, hogy bizonyos esetekben a döntési fák nagyon összetetté válnak. Ebben a fában néhány haszontalan ág növeli a modell bonyolultságát. Ennek elkerülése érdekében jön tehát az



2.4. ábra. Példa alpha-beta metszésre.

alfa-béta metszés, hogy a számítógépnek ne kelljen a teljes fát átnéznie. Ezek a szokatlan csomópontok lassítják az algoritmust. Ezért ezen csomópontok eltávolításával az algoritmus gyors lesz

Allen Newell és Herbert A. Simon, akik 1958-ban azt használták, amit John McCarthy "közelítésnek" nevez, azt írták, hogy az alfa-bétát "úgy tűnik, hogy többször is újra feltalálták". Arthur Samuelnek volt egy korai változata egy dámaszimulációhoz. Richards, Timothy Hart, Michael Levin és/vagy Daniel Edwards szintén önállóan találta fel az alfa-bétát az Egyesült Államokban. McCarthy hasonló ötleteket javasolt 1956-ban a dartmouthi workshopon, és 1961-ben az MIT-n tanítványai egy csoportjának, köztük Alan Kotoknak is javasolta. Alexander Brudno önállóan fogalmazta meg az alfa-béta algoritmust, és 1963-ban publikálta eredményeit. Donald Knuth és Ronald W. Moore 1975-ben finomította az algoritmust. Judea Pearl két tanulmányban bizonyította optimális voltát a véletlenszerűen kijelölt levélértékekkel rendelkező fák várható futási idejére vonatkozóan. Az alfa-béta véletlenszerű változatának optimalitását Michael Saks és Avi Wigderson mutatta ki 1986-ban.[8] [9]

2.7. Az Alpha-Beta metszés korábbi alkalmazásai játékokban

Íme néhány korábbi játék, amelyben az alfa-béta metszést alkalmazták:

- Sakk
- Tic-tac-toe
- Connect 4
- Dáma
- Reversi

- Go
- Egyéb kétjátékos zéróösszegű játékok

3. fejezet

Fejezet: Elméleti alapok

3.1. Programozás és python technológiák

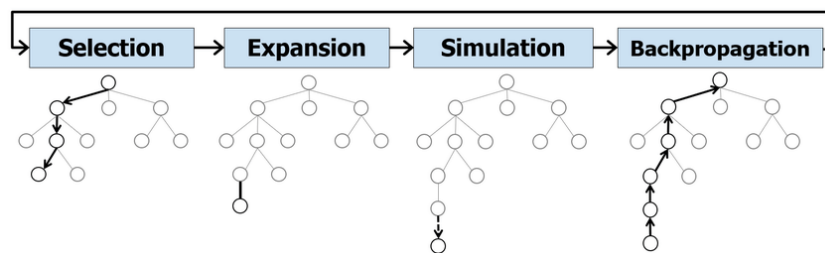
A Python egy általános célú, magas szintű programozási nyelv, amelyet 1989-ben Guido Van Rossum hozott létre. Ez egy interpretált nyelv, ami azt jelenti, hogy a kódot közvetlenül a forrásból tudja futtatni, anélkül, hogy előbb le kellene fordítani. A Pythont széles körben használják különböző módszertanokban, és bármilyen összetett függvényeket tartalmazó alkalmazások bármilyen módjának fejlesztésére használható. Ez az egyik legnépszerűbb programozási nyelv, és vezető szerepet játszik ebben a modern digitális korszakban.

Azért választottam a Pythont mint programozási nyelv mivel Python egyszerűsége, sokoldalúsága és hatékonysága miatt népszerű programozási nyelv komplex döntéshozatali problémák megoldására.

A Pythont különféle területeken használják, például webes alkalmazások, szkriptelés, szoftverfejlesztés, mesterséges intelligencia, gépi tanulás, adattudomány és még sok más területen. A Python különböző platformokat támogat, például Windows, Linux, Mac és olyan mikrokontrollereket, mint a Raspberry Pi.

A Python egyszerűségéről, olvashatóságáról és rugalmasságáról ismert, így kiváló választás a kezdők és a tapasztalt programozók számára egyaránt. Hangsúlyt fektet a kód olvashatóságára és az egyszerű szintaxisra, így a létrehozása és megértése egyszerű és elérhető a felhasználók számára. A Pythont folyamatosan a világ egyik legnépszerűbb programozási nyelveként tartják számon. Olyan vállalatok használják, mint a Google, a Facebook, a Cisco, a Netflix és a NASA, valamint számos nyílt forráskódú projekt.

A Pythont eredetileg a kód olvashatóságának hangsúlyozására tervezték, és szintaxisa lehetővé teszi a programozók számára, hogy kevesebb sornyi kódban fejezzenek ki fogalmakat. Az a programozási nyelv, amelyben a Python állítólag sikert aratott, az ABC programozási nyelv volt, amely az Amoeba operációs rendszerhez kapcsolódott, és rendelkezett a kivételkezelés funkciójával. A Python számos változáson ment keresztül élete során, ami nem meglepő, tekintve, hogy a nyelv Van Rossum hobbiprojektjekjeként indult, és a világ egyik legsokoldalúbb programozási nyelvévé vált.[10][11]



3.1. ábra. MCTS fázisai ábrázolva

3.2. A Monte Carlo Tree Search algoritmus

A Monte Carlo Tree Search (MCTS) egy heurisztikus keresési algoritmus, amely a Monte Carlo algoritmust egy fa struktúrával kombinálja a játékfa hatékony kereséséhez. A MCTS algoritmus négy fázisból áll: kiválasztás, bővítés, szimuláció és visszaterjesztés. Az algoritmus egy játékállapottal indul, amelyet a keresési fa gyökeresomópontja reprezentál. Az algoritmus minden egyes iterációnál kiválaszt egy ígéretes csomópontot a bővítéshez, a kiválasztott csomópontból egy vagy több gyermekcsomópontot generál, szimulál egy kijátszást az újonnan hozzáadott gyermekcsomópontból, és frissíti a gyökértől az újonnan hozzáadott csomópontig vezető út mentén lévő csomópontok statisztikáit. A kiválasztási fázis a MCTS algoritmus lényege, ahol az algoritmus kiválasztja a bővítendő ígéretes csomópontot. A kiválasztást egy olyan kiválasztási eljárás irányítja, amely egyensúlyt teremt a keresési fa feltárása és kihasználása között. Az egyik legnépszerűbb kiválasztási eljárás az Upper Confidence Bound (UCT) algoritmus, amely a látogatások számát és az egyes csomópontok átlagos jutalmát használja a felfedezés-kihasználás kompromisszum kiszámításához. A bővítési fázis egy vagy több gyermekcsomópontot ad hozzá a kiválasztott csomóponthoz. A szimulációs fázis a játék végéig vagy egy előre meghatározott mélység eléréséig szimulálja az újonnan hozzáadott gyermekcsomópontból történő kijátszást. A visszaterjedési fázis a szimuláció eredménye alapján frissíti a csomópontok statisztikáit a gyökértől az újonnan hozzáadott csomópontig tartó út mentén.

3.2.1. Kiválasztás

A kiválasztási fázis a Monte Carlo fakereső algoritmus első lépése. Célja a keresési fa végigjárása a gyökeresomóponttól egy levélcsoomópontig. A levélcsoomópont kiválasztása a kiválasztási eljárás alapján történik, amely egyensúlyt teremt a feltárás és a kiaknázás között. A kiválasztási eljárás olyan gyermekcsomópontot választ, amely maximalizálja az értékére vonatkozó felső megbízhatósági korlátot (UCB), ahol az UCB a gyermekcsomópont várható értékének és az értékbecslés bizonytalanságát mérő kifejezésnek az összegeként van meghatározva. A kiválasztási folyamat addig ismétlődik, amíg egy levélcsoomópontot el nem érünk. Ha a levélcsoomópont nem végcsomópont, a keresés a bővítési fázisba lép. Ha a levélcsoomópont terminális csomópont, akkor a szimulációs fázis kerül végrehajtásra. A szimuláció eredményét ezután a fán felfelé haladva frissíti a kiválasztási fázisban meglátogatott csomópontok statisztikáit.

3.2.2. Bővítés

A bővítési fázis a Monte Carlo Tree Search (MCTS) algoritmus második lépése. Ebben a fázisban egy csomópontot választunk ki a fából a kiválasztási fázisban meghatározott kiválasztási eljárás szerint, és egy új gyermekcsomópontot adunk hozzá a fához. Az új csomópont egy olyan lehetséges jövőbeli játékállapotot képvisel, amelyet korábban még nem vizsgáltunk meg. Az új gyermekcsomópontot a játéktól függően nulla látogatási számmal és nulla vagy egy heurisztikus értékkel inicializáljuk. Az új csomópont ezután a kiválasztott csomópont gyermekeként kerül hozzáadásra, és az új csomópontához tartozó játékállapotot a fa tárolja. A bővítési fázis kritikus fontosságú a MCTS-ben, mert lehetővé teszi az algoritmus számára, hogy korábban fel nem fedezett játékállapotokat fedezzen fel. Egy új csomópont hozzáadásával a fához a MCTS egy új ágat hoz létre, amelyet az algoritmus későbbi iterációi során fel lehet fedezni. A bővítési fázis segít a MCTS-nek elkerülni a lokális optimumokban való megrekedést, és a játékfa nagyobb részét feltárni. A bővítési fázis azonban számításigényes lehet, különösen a nagy elágazási faktorokkal rendelkező játékok esetében. Ilyen esetekben gondos egyensúlyt kell fenntartani a feltárás és a kiaknázás között, hogy az algoritmus ésszerű időn belül jó megoldáshoz konvergáljon. A bővítési fázis változatai javasoltak ennek a problémának a megoldására, például a progresszív bővítés, amely dinamikusan módosítja egy csomópont gyermekeinek számát a látogatások száma alapján.

3.2.3. Szimuláció

A szimulációs fázis a Monte Carlo Tree Search algoritmus döntő fontosságú lépése, amikor a fa egy adott csomópontjától kezdve a játék gördülése vagy lejátszása történik. Ennek a fázisnak a célja a csomópont minőségének értékelése a játék hátralévő részének kijátszásával az aktuális állapottól egy végállapotig. A játék lejátszása teljesen véletlenszerű módon történik, ami függetleníti a lépésválasztástól, és biztosítja, hogy a szimuláció torzítatlan maradjon. A szimulációs fázis során az algoritmusnak egyensúlyt kell teremtenie az új utak feltárása és a már feltárt utak kihasználása között. A kiépítési politikát jellemzően úgy választják meg, hogy egyensúlyt teremtsen e célok között, és a játék sajátos követelményeitől függően különböző politikákat lehet alkalmazni. Az egyik gyakori politika a véletlenszerű politika, ahol a lépéseket egyenletesen véletlenszerűen választják ki. A szimulációs fázis minősége döntő fontosságú a MCTS algoritmus általános teljesítménye szempontjából. Fontos megjegyezni, hogy a szimulációs fázis számításigényes lehet, és különböző technikák, például heurisztikák használhatók a keresési tér csökkentésére és a szimuláció sebességének javítására. Emellett olyan technikák, mint a progresszív ki szélesítés, használhatók a szimulációs fázis során a felderítés-kihasználás egyensúlyának beállítására.

3.2.4. Visszaterjesztés

A szimulációs fázis után a szimuláció eredményeit vissza kell terjeszteni a fán. A visszaterjedési fázis frissíti az összes csomópont statisztikáját az újonnan hozzáadott levélsomóponttól a fa gyökeréig tartó útvonalon. A statisztikák tartalmazzák a csomópontot meglátogatók teljes számát és a csomóponton áthaladó szimulációk során kapott jutalmak összegét. A visszaterjedési fázis célja a kiválasztási és bővítési fázisban részt

vevő csomópontok statisztikáinak frissítése, valamint a lehetséges akciók értékének értékelése a szülő csomópont szempontjából. A statisztikák frissítése a visszaterjedési fázisban a következő egyenlet szerint történik:

$$Q(n) = \frac{W(n)}{N(n)} + c \sqrt{\frac{\ln(N(p))}{N(n)}} \quad (3.1)$$

ahol:

- $Q(n)$ a csomópont becsült értéke
- $W(n)$ a csomóponton áthaladó összes szimulációban kapott értékek összege
- $N(n)$ a csomópont meglátogatásának száma
- $N(p)$ a szülői csomópont meglátogatásának száma
- c egy konstans, amely egyensúlyt teremt az eddig megtalált legjobb akció kihasználása és a még fel nem tárt akciók feltárása között

A visszaterjedési fázis a gyökércsomópont eléréséig folytatódik. A visszaterjedési fázis után a fa készen áll a MCTS algoritmus következő iterációjára. A kiválasztási fázis ismét a gyökércsomópontból indul, és az algoritmus addig folytatódik, amíg egy megállási kritérium nem teljesül. A visszaterjedési fázis a MCTS algoritmus kritikus eleme, mivel a szimulációk eredményei alapján frissíti a csomópontok statisztikáit. Ez lehetővé teszi, hogy az algoritmus tanuljon a korábbi tapasztalatokból, és idővel javítsa döntéshozatali képességeit. A visszaterjedési fázis minősége azonban a szimulációs fázis minőségétől függ, mivel a csomópontok statisztikái a szimulációk eredményei alapján frissülnek. Ezért a szimulációs fázis pontossága és hatékonysága döntő fontosságú a MCTS algoritmus hatékonysága szempontjából.[12] [5]

3.3. Monte Carlo Tree Search algoritmus megvalósítása a Battleshipben

```
def monte_carlo_tree_search(ship_positions, ai_guesses, depth, steps):
    available_positions = [(x, y) for x in range(board_size) for y in
                           range(board_size) if (x, y) not in ai_guesses]
    best_score = float('-inf')
    best_guess = None
    for _ in range(steps):
        guess = random.choice(available_positions)
        prediction, score = simulate_guess(ship_positions, guess, depth)
        if score > best_score:
            best_score = score
            best_guess = prediction
    return best_guess
```

A fenti a `monte_carlo_tree_search` függvény amely a fő függvénye a monte carlo tree searchnek. Ezen belül a `ship_positions`, a `ai_guesses`, a `depth` és a `steps` paramétereket veszi fel. A Monte Carlo Tree Search algoritmus alapján a következő lépés legjobb tippjét adja vissza. Itt fontos megjegyezni, hogy a `ship_positins` az ellenfél hajóinak pozíciói, ezt fontos megadnunk annak érdekében, hogy ezt össze tudjuk majd hasonlítani a véletlenszerűen generált tippünkkel.

A függvény először létrehozza az `available_positions` listáját a `board_size` tartományon való iterációval, és kizárja azokat a pozíciókat, amelyek már szerepelnek az `ai_guesses` listában. Ezután a `best_score`-t negatív végtelenre, a `best_guess`-t pedig `None`-ra inicializálja.

A függvény ezután a `steps` számú ciklus lefutását végzi. Minden egyes iterációban véletlenszerűen kiválaszt egy pozíciót az `available_positions` listából a `random.choice()` függvény segítségével. Ezután meghívja a `simulate_guess` függvényt a találgatás szimulálásához, és visszaadja a `prediction` és az `score` értéket. Ha a `score` nagyobb, mint a `best_score`, akkor a `best_score`-t frissíti az új `score`-ra, és a `best_guess`-t a `prediction`-re állítja.

Végül a függvény visszaadja a `best_guess` értéket.

```
def simulate_guess(ship_positions, guess, depth):
    ship_positions_copy=[position[:] for position in ship_positions]
    score=0
    if guess in ship_positions:
        ship_positions_copy.remove(guess)
        score = score + 1
    if depth>1:
        guesses, newscore = simulate_random_guesses(ship_positions_copy, depth)
        score += newscore
        if guesses:
            guess=guesses.pop()
    return guess, score
```

A `simulate_guess` függvény a `ship_positions` listát, egy `guess` és egy `depth` paramétert vesz fel. Visszaad egy tuple-t, amely tartalmazza a "guess"-t és a "score"-t, amely a `guess` által eltalált hajók számán alapul.

A függvény először létrehozza a `ship_positions` lista másolatát egy listamegértés segítségével. Ezután a `score`-t 0-ra inicializálja.

Ha a tipp szerepel a `ship_positions` listában, a függvény eltávolítja a tippet a `ship_positions_copy` listából, és az `score`-t 1-gyel növeli.

Ha a mélység nagyobb, mint 1, a függvény meghívja a `simulate_random_guesses` függvényt, hogy véletlenszerű találgatásokat szimuláljon a `ship_positions_copy` listán. A `simulate_random_guesses` függvény visszaadja a `guesses` listát és a `newscore` értéket a találatok által eltalált hajók száma alapján. A függvény ezután a `score`-t a `newscore`-val növeli.

Ha a `guesses` lista nem üres, a függvény az utolsó tippet veszi ki a listából, és a `guess` értéket erre az értékre állítja.

Végül a függvény egy tuple-t ad vissza, amely tartalmazza a `guess` és a `score` értéket.

```
def simulate_random_guesses(ship_positions, depth):
```



```

guesses = []
hit_guesses = []
i=0
while (ship_positions and i!=depth-1):
    i = i + 1
    guess = random.choice([(x, y) for x in range(board_size) for y in
                           range(board_size) if (x, y) not in guesses])
    guesses.append(guess)
    if guess in ship_positions:
        ship_positions.remove(guess)
        hit_guesses.append(guess)
return hit_guesses, len(hit_guesses)

```

A megadott `simulate_random_guesses(ship_positions, depth)` függvény véletlenszerű találgatásokat szimulál egy battleship játéktáblán. A függvény két argumentumot vesz fel: A `ship_positions` egy listát, amely a hajók pozícióit mutatja a táblán, és a `depth` egy egész számot, amely a maximálisan kitalálható tippek számát adja meg. A függvény a helyes találatokat és hajókra adott találatok számát adja vissza.

A függvény inicializál egy üres `guesses` listát a kitalált pozíciók tárolására, és a `hit_guesses` listát a helyes találatok tárolására illetve, hogy nyomon kövesse a találatok számát. Az `i` változót 0-ra inicializáljuk, hogy nyomon kövessük a kitalált hajók számát.

A függvény ezután egy `while` ciklusba lép, amely addig folytatódik, amíg vagy az összes hajót eltaláljuk, vagy a maximális találatokat(mélységet) el nem érjük. A ciklus minden egyes iterációjában az `i` változót 1-el növeljük. A függvény a `random.choice()` függvény segítségével generál egy véletlenszerű találgatást a `random` modulból. A találgatás egy, a tábla egy pozícióját jelképező, az összes olyan pozíció közül kiválasztott tuple, amelyet még nem találtak ki. A tippet hozzáadjuk a `guesses` listához.

Ha a tipp a hajók egyik pozíciója, akkor a `ship_positions` listából eltávolítjuk, és a `hit_guesses` listához hozzáadjuk.

Végül a függvény visszaadja az eltalált hajók pozícióját és hajókon elért találatok számát.

Összességében a `monte_carlo_tree_search()` függvény véletlenszerű mintavételezést és szimulációt használ az értékek becslésének gyűjtésére, hogy a keresési fában a leghasznosabb lépések felé irányítson. Több figyelmet fordít az ígéretesebb csomópontokra, így elkerülhető, hogy minden lehetőséget nyers erővel kelljen megragadni, ami nem praktikus.

3.4. A Minimax algoritmus

A minimax algoritmus alapötlete az, hogy a játékfában minden lehetséges lépéshez értéket rendelünk, majd ezeket az értékeket a fán felfelé haladva, a játékosok maximalizálása és minimalizálása között váltakozva haladunk felfelé. A maximalizáló játékos célja a saját eredményének maximalizálása, míg a minimalizáló játékos célja a maximalizáló játékos eredményének minimalizálása. Ez a folyamat addig folytatódik, amíg el nem érjük a végállapotot, ahol a játék véget ér, vagy amíg egy bizonyos mélységhatárt el nem érünk.

A minimax algoritmus alkalmazásához általában minden egyes végállapothoz egy pontszámot vagy hasznossági értéket rendelünk. Sakkban például a maximalizáló játékos

győzelméhez +1 értéket, a minimalizáló játékos győzelméhez -1 értéket, a döntetlenhez pedig 0 értéket rendelhetünk. A dolgozat kódjában is hasonlóan járunk el. Ezután a rekurzív kiértékelés során a maximalizáló játékos a legmagasabb értékkel rendelkező lépést választja, feltételezve, hogy a minimalizáló játékos a legjobb ellenlépést teszi, míg a minimalizáló játékos a legkisebb értékkel rendelkező lépést választja, feltételezve, hogy a maximalizáló játékos a legjobb ellenlépést teszi.

A minimax algoritmus a következő játékelméleti egyenlet segítségével ábrázolható:

$$V(s) = \max(\min(V(s')))$$
 (3.2)

ahol:

- $V(s)$ a játékfában az "s" állapot értékét jelenti.
- A max az érték maximalizálására törekvő játékos fordulóját jelöli.
- min a minimalizáló játékos fordulóját jelöli, amelynek célja az érték minimalizálása.
- $V(s')$ az ellenfél "s" állapotának az aktuális játékos által választott lépésből eredő értékét jelöli.

A játékfát ezen egyenlet segítségével kiértékelve a minimax algoritmus végül meghatározza az aktuális játékos számára legjobb lépést, feltételezve, hogy mindkét játékos optimalisan játszik, és az ellenfél célja a játékos eredményének minimalizálása.[13][7][6]

3.5. A Minimax algoritmus megvalósítása a Battleshipben

```
def minimax(my_positions, their_positions, ai_guesses, player_guesses, depth, is_ai):  
    if depth == 0:  
        return 0, None
```

Ez a minimax függvény definíciója, amely több paramétert kap: A paraméterek: my_positions, their_positions, ai_guesses, player_guesses, depth és is_ai. A my_positions és their_positions az AI játékos, illetve az ellenfél játékos által elfoglalt pozíciókat jelöli. A ai_guesses és a player_guesses az AI és az ellenfél által választott pozíciókat tárolja. A depth paraméter a keresési fa mélységét jelzi, az is_ai pedig egy bólusjelző, amely jelzi, hogy az AI játékos van-e soron.

Az első feltétel azt ellenőrzi, hogy a mélység nulla-e, ami azt jelzi, hogy a keresés elérte a kívánt mélységet. Ebben az esetben a függvény egy (0, None) tuple-t ad vissza, ami azt jelenti, hogy nincs ehhez a csomópontához tartozó pontszám.

```
    if is_ai:  
        available_positions = [(x, y) for x in range(board_size) for y in  
                                range(board_size) if (x, y) not in ai_guesses]  
    else:  
        available_positions = [(x, y) for x in range(board_size) for y in  
                                range(board_size) if (x, y) not in player_guesses]
```

Ezek a sorok határozzák meg a játéktábla elérhető pozícióit, attól függően, hogy ki van soron. Ha az AI játékos van soron (az `is_ai` igaz), akkor az `ai_guesses`-ben nem szereplő pozíciókat ellenőrzi. Ellenkező esetben a `player_guesses`-ben nem szereplő pozíciókat ellenőrzi.

```
if is_ai:
    best_score = float('-inf')
    for pos in available_positions:
        score = 0
        if pos in their_positions:
            score = 1
        maximizing, _ = minimax(their_positions, my_positions, ai_guesses +
                                [pos], player_guesses, depth - 1, not is_ai)
        score -= maximizing
        if score > best_score:
            best_score = score
            best_pos = pos
    return best_score, best_pos
```

Ha az AI játékos van soron, az algoritmus belép ebbe a kódblokkba. A `best_score`-t negatív végtelennel inicializálja, jelezve, hogy a mesterséges intelligencia játékos maximalizálni akarja a pontszámát. Ezután végigmegy az `available_positions` listán, és ellenőrzi, hogy az aktuális pozíció `pos` szerepel-e az `their_positions` listán. Ha igen, akkor a `score` értékét 1-re állítja.

Az algoritmus az előző esethez hasonlóan rekurzív módon hívja meg a `minimax` algoritmust frissített pozíciókkal és paraméterekkel. A rekurzív hívás visszaadott értékét a `maximizing` mezőben tárolja, amely az ellenfél játékosának legjobb lehetséges pontszámát jelenti. A `score` ezután a `maximizing` levonásával frissül. Ha a frissített `score` nagyobb, mint az aktuális `best_score`, a `best_score` és a `best_pos` az új értékekkel frissül.

Miután a függvény az összes rendelkezésre álló pozíciót végigjárta, visszaadja a `best_score` és a `best_pos` értékeket, amelyek a legjobb pontszámot és a mesterséges intelligencia által választott megfelelő pozíciót jelölik.

```
else:
    best_score = float('inf')
    for pos in available_positions:
        score = 0
        if pos in my_positions:
            score = -1
        minimizing, _ = minimax(my_positions, their_positions, ai_guesses,
                                player_guesses + [pos], depth - 1, not is_ai)
        score += minimizing
        if score < best_score:
            best_score = score
            best_pos = pos
    return best_score, best_pos
```

Ha az ellenfél következik, az algoritmus ebbe a kódblokkba lép. A `best_score`-t pozitív végtelennel inicializálja, jelezve, hogy az ellenfél játékos a saját pontszámát akarja mini-

malizálni. Ezután végigmegy az `available_positions` listán, és ellenőrzi, hogy az aktuális pozíció `pos` szerepel-e a `my_positions` listán. Ha igen, akkor a `score` értékét -1-re állítja.

Az algoritmus az előző esethez hasonlóan rekurzívan hívja a `minimax`-ot frissített pozíciókkal és paraméterekkel. A rekurzív hívás visszaadott értékét a `minimizing` mezőben tárolja, ami a mesterséges intelligencia játékos lehető legjobb pontszámát jelenti. A `score` ezután a `minimizing` hozzáadásával frissül. Ha a frissített `score` kisebb, mint az aktuális `best_score`, a `best_score` és a `best_pos` az új értékekkel frissül.

Miután a függvény az összes elérhető pozíciót végigjárta, a függvény visszaadja a `best_score` és a `best_pos` értékeket, amelyek a legjobb pontszámot és az ellenfél által választott megfelelő pozíciót jelölik.

Összességében ez a kód rekurzív módon alkalmazza a `minimax` algoritmust a mesterséges intelligencia játékos legjobb lépésének meghatározására a játékfa adott mélységig történő kiértékelésével. Az összes lehetséges lépést és azok következményeit figyelembe veszi, feltételezve az ellenfél optimális játékát.[9]

3.6. Az Alpha-Beta metszés algoritmus

Az `alpha-beta` algoritmus egy olyan keresési algoritmus, amely a `minimax` algoritmust optimalizálja a játékfa ágainak szelektív feltárásával. Általánosan használják a forduló-alapú játékokban.

Az algoritmus minden egyes lépésnél kiértékeli a játék aktuális állapotát, és eldönti, hogy folytatja-e a keresést vagy megrövidíti a keresési fát. Két értéket tart számon: az `alpha`, amely a maximalizáló játékos számára az eddig talált legjobb értéket, és a `beta`, amely a minimalizáló játékos számára az eddig talált legjobb értéket jelenti.

A kezdeti állapotból kiindulva az algoritmus minden lehetséges lépést figyelembe vesz, amelyet az aktuális játékos megtehet. Minden egyes lépésnél rekurzív módon hívja magát, frissített játékállapottal, az ellenfél játékosával, valamint az aktuális `alpha`- és `beta`-értékekkel.

Ha az aktuális játékos a maximalizáló játékos, az algoritmus frissíti az `alpha`-értéket, ha a rekurzív hívás kiértékelési eredménye nagyobb, mint az aktuális `alpha`. Ez tükrözi a maximalizáló játékos eddig talált legjobb eredményét.

Ha az aktuális játékos a minimalizáló játékos, az algoritmus frissíti a `beta` értéket, ha a rekurzív hívás kiértékelési eredménye kisebb, mint az aktuális `beta`. Ez a minimalizáló játékos eddig talált legjobb eredményét jelenti.

A keresés során, ha az `alpha`-érték bármelyik ponton nagyobb vagy egyenlő lesz a `beta`-értékkel, az algoritmus lemetshi a keresési fát. Ez azért történik, mert az ellenfél játékos nem ezt az ágot választaná, mivel már van egy jobb alternatíva.

Az algoritmus folytatja a különböző lépések vizsgálatát és értékelését, és a játékfán való előrehaladás során iteratív módon frissíti az `alpha`- és `beta`-értékeket. Az algoritmus vagy egy meghatározott mélység elérésekor, vagy a játék befejezésekor fejeződik be.

Végül az algoritmus visszaadja az `alpha`-értéket, ha az aktuális játékos a maximalizáló játékos, vagy a `beta`-értéket, ha az aktuális játékos a minimalizáló játékos. Ezek az értékek az egyes játékosok számára a játékfa feltárt ágai alapján a lehető legjobb eredményt jelentik.

Az alpha- és beta-értékek alapján történő szelektív ágkutatással és a keresési fa metszésével az alpha-beta algoritmus javítja a keresési folyamat hatékonyságát, lehetővé téve a gyorsabb döntéshozatalt a játékokban.[9][8]

3.7. Az Alpha-Beta metszés algoritmus megvalósítása a Battleshipben

```
def alpha_beta(my_positions, their_positions, ai_guesses, player_guesses,
               depth, is_ai):
```

Ez a sor deklarálja az alpha_beta függvényt, és megadja a paramétereit: my_positions, their_positions, ai_guesses, player_guesses, depth és is_ai. Ezek a paraméterek a játék aktuális állapotát és az algoritmus konfigurációját jelentik.

```
    if is_ai:
        available_positions = [(x, y) for x in range(board_size) for y in
                                range(board_size) if (x, y) not in ai_guesses]
    else:
        available_positions = [(x, y) for x in range(board_size) for y in
                                range(board_size) if (x, y) not in player_guesses]
```

Ez a feltételező utasítás ellenőrzi, hogy az AI van-e soron (az is_ai igaz) vagy a játékos (az is_ai hamis). E feltétel alapján létrehozza az available_positions listáját, ahol az AI vagy a játékos a következő lépést megteheti. A pozíciókat a range(board_size) iterálásával határozzuk meg, és kizárjuk a mesterséges intelligencia vagy a játékos által már kitalált pozíciókat

```
    max_score = float('-inf')
    guess = None
```

Két változó, a max_score és a guess inicializálásra kerül. A max_score negatív végtelenre, a guess pedig None-ra van állítva. Ezek a változók az eddig talált legmagasabb pontszámot és a hozzá tartozó legjobb lépést tartják számon.

```
    for pos in available_positions:
        score = 0
        if pos in their_positions:
            score = 1
        if score < max_score:
            continue
        enemy_max_score = 0
        if depth > 1:
            enemy_score, _ = alpha_beta(their_positions, my_positions,
                                         ai_guesses, player_guesses, depth-1, not is_ai)
            if enemy_score > enemy_max_score:
                enemy_max_score = enemy_score
        score -= enemy_max_score
        if score > max_score:
```

```
max_score = score
guess = pos
```

Ez a kódrészlet az Alpha-Beta algoritmus fő ciklusát jelenti. Végigmegy az available_positions minden egyes pozícióján, és kiértékeli az adott pozícióhoz tartozó pontszámot. Minden egyes pozíció esetében a score-t 0-ra inicializálja. Ha a pozíciót megtaláljuk a their_positions-ben, akkor a score-t 1-re állítjuk, ami azt jelzi, hogy az ellenféllel szemben sikeres volt a lépés. Ezután ellenőrzi, hogy a score kisebb-e, mint az eddig talált max_score. Ha igen, akkor folytatja a ciklus következő iterációját. Ha a depth nagyobb, mint 1 (ami azt jelzi, hogy a játékfában több szint van felfedezésre), akkor rekurzívan meghívja az alpha_beta függvényt, felcserélve az AI és a játékos pozícióit és tippjeit (their_positions lesz my_positions, és fordítva), és csökkenti a depth értékét 1-gyel. A rekurzív hívásból lekérdezi az enemy_score értéket, és frissíti az enemy_max_score értéket, ha az új érték magasabb. A score ezután az enemy_max_score levonásával módosul. Végül ellenőrzi, hogy a kiigazított score nagyobb-e, mint a max_score. Ha igen, akkor a max_score és a guess értéket is frissíti az új legmagasabb pontszámmal és a megfelelő pozícióval.

```
return max_score, guess
```

A függvény a következő értékek visszatérítésével zárul max_score és a guess értékeket egy tuple-ként adja vissza. A max_score az AI vagy a játékos által az aktuális állapot alapján elérhető legjobb pontszámot jelenti, a guess pedig azt a pozíciót, amely a legjobb pontszámhoz vezet.

3.8. A játékmechanika és a döntéshozatal elemzése

A játék menete a következő kódrészletben látható:

```
# Game loop
while True:
    os.system('cls' if os.name == 'nt' else 'clear')
    # Game setup
    board_size = 10
    ships = {'Carrier': 5, 'Battleship': 4, 'Cruiser': 3, 'Submarine': 3,
            'Destroyer': 2}
    player_board = create_board(board_size)
    ai_board = create_board(board_size)
    ai_ship_positions = []
    player_ship_positions = []
    player_guesses = []
    ai_guesses = []

    # Player setup
    for ship, length in ships.items():
        os.system('cls' if os.name == 'nt' else 'clear')
        print('Player\'s Turn')
        print(f'Placing the {ship} ({length} cells).')
```

```

print('Player\'s Board:')
print_board(player_board)
placement, ship_positions = get_ship_placement(player_board, ship,
length,player_ship_positions)
player_ship_positions+=(ship_positions)
player_board = placement
# AI setup
for ship, length in ships.items():
    ai_placement, ship_positions = random_ship_placement(ai_board, ship,
length)
    ai_ship_positions.extend(ship_positions)
print(ai_ship_positions)
#Gameplay
while True:
    os.system('cls' if os.name == 'nt' else 'clear')
    print('Player\'s Turn')
    print('Player\'s Board:')
    print_board(player_board)
    print('AI\'s Board:')
    print_board(ai_board)
    # Player's turn
    guess = get_player_guess(player_guesses)
    player_guesses+=(guess)
    if guess in ai_ship_positions:
        result = 'hit'
        ai_ship_positions.remove(guess)
    else:
        result = 'miss'
    update_board(ai_board, guess, result)
    if result == 'hit':
        print('You hit a ship!')
    else:
        print('You missed.')
    if check_victory(ai_ship_positions, player_guesses):
        print('Congratulations! You sank all the AI\'s ships!')
        break
    # AI's turn
    which_algorithm = 'monte carlo'
    ai_guess = run_algorithm(which_algorithm)
    ai_guesses += [(ai_guess)]
    if ai_guess in player_ship_positions:
        result = 'hit'
        player_ship_positions.remove(ai_guess)
    else:
        result = 'miss'
    update_board(player_board, ai_guess, result)
    print('\nAI\'s Guess:')
    print(f'AI guessed: {chr(ai_guess[1] + ord("A"))}{ai_guess[0]+1} -
{result}')

```

```

if check_victory(player_ship_positions, ai_guesses):
    print('AI sank all your ships! You lost.')
    break
input("Press enter to continue")
if not play_again():
    break

```

Végigmenve a kódon részletesen a következő képpen írható le:

- Game Loop

```

while True:
    ...

```

A játék ciklus biztosítja, hogy a játék addig folytatódjon, amíg a játékosok úgy döntenek, hogy kilépnek.

- A konzol törlése

```

os.system('cls' if os.name == 'nt' else 'clear')

```

Ez a sor törli a konzol képernyőjét. Az `os.system()` függvényt használja az operációs rendszeren alapuló parancs végrehajtásához. Ebben az esetben ellenőrzi, hogy az operációs rendszer Windows-e ('nt'), és a 'cls' parancsot használja a képernyő törlésére. Más operációs rendszerek esetén a 'clear' parancsot használja.

- Játék beállítások

```

board_size = 10
ships = {'Carrier': 5, 'Battleship': 4, 'Cruiser': 3, 'Submarine':
        3, 'Destroyer': 2}
player_board = create_board(board_size)
ai_board = create_board(board_size)
ai_ship_positions = []
player_ship_positions = []
player_guesses = []
ai_guesses = []

```

A játék beállítási fázisában a kód inicializálja a játékhoz szükséges különböző változókat.

A `board_size` a játéktábla méretét jelöli (egy négyzet alakú tábla, amelynek oldalai a `board_size` hosszúságúak).

A `ships` egy dictionary, amely a játékban szereplő hajók nevét és hosszát tartalmazza.

A `player_board` és `ai_board` üres játéktáblákként inicializálódnak a `create_board()` függvény segítségével.

Az `ai_ship_positions`, `player_ship_positions`, `player_guesses` és `ai_guesses` üres listák, amelyek a játék során a hajók pozícióinak és a játékosok tippjeinek tárolására szolgálnak.

- Játékos beállítások

```
for ship, length in ships.items():
    os.system('cls' if os.name == 'nt' else 'clear')
    print('Player\'s Turn')
    print(f'Placing the {ship} ({length} cells).')
    print('Player\'s Board:')
    print_board(player_board)
    placement, ship_positions = get_ship_placement(player_board,
                                                    ship, length, player_ship_positions)
    player_ship_positions += ship_positions
    player_board = placement
```

Ebben a fázisban a játékos felállítja hajóit a játéktábláján. A ciklus végigmegy a `ships` dictionaryben lévő minden egyes hajón, és felszólítja a játékost, hogy helyezze el az adott hajót a játéktábláján.

A konzol képernyője törlődik, és megjelenik a vonatkozó információ.

A `get_ship_placement()` függvényt hívjuk meg, hogy megkapjuk a játékos által kívánt hajóelhelyezést. Bemenetként a játékos aktuális tábláját, a hajó nevét, a hajó hosszát és a már elhelyezett hajók pozícióját veszi fel. Visszaadja a hajó elhelyezése után frissített táblát és a hajó pozíciók listáját.

A játékos hajóhelyei és a tábla ennek megfelelően frissülnek.

- AI beállítás

```
for ship, length in ships.items():
    ai_placement, ship_positions = random_ship_placement(ai_board,
                                                         ship, length)
    ai_ship_positions.extend(ship_positions)
```

Ebben a fázisban az AI ellenfél felállítja hajóit a játéktábláján. A játékos beállításához hasonlóan a ciklus a `ships` dictionaryben lévő minden egyes hajón végigmegy, és a `random_ship_placement()` függvényt hívja meg, hogy véletlenszerű elhelyezést kapjon az AI hajói számára. A függvény visszaadja a hajó elhelyezése után frissített AI táblát és a hajó pozíciók listáját, amelyek hozzáadódnak az `ai_ship_positions`-hez.

- Játékmenet

```
while True:
    os.system('cls' if os.name == 'nt' else 'clear')
    print('Player\'s Turn')
    print('Player\'s Board:')
    print_board(player_board)
```

```
print('AI\'s Board:')
print_board(ai_board)
...
```

A játék fázisa egy újabb egymásba ágyazott ciklus. Addig tart, amíg a játékos vagy az AI meg nem nyeri a játékot. Minden egyes ismétlésben megjelenik a játékos és az AI táblája, és a játék a játékos, majd az AI sorával folytatódik.

- A játékos sora

```
guess = get_player_guess(player_guesses)
player_guesses += [guess]
if guess in ai_ship_positions:
    result = 'hit'
    ai_ship_positions.remove(guess)
else:
    result = 'miss'
update_board(ai_board, guess, result)
```

A játékos sorában a játékosnak meg kell adnia a AI tábla egy cellájára vonatkozó tippjét. A `get_player_guess()` függvényt hívjuk meg, hogy megkapjuk a játékos tippjét. A függvény biztosítja, hogy a tipp a tábla tartományán belül van, és még nem tippelték meg korábban.

A tippet hozzáadjuk a játékos tippjeinek listájához.

Ha a tipp megegyezik a mesterséges intelligencia egyik hajóhelyzetével (`ai_ship_positions`), akkor ez egy találat. A tippet eltávolítjuk a `ai_ship_positions` listából.

Ellenkező esetben ez egy hiba.

A `update_board()` függvényt hívjuk meg, hogy frissítsük a mesterséges intelligencia tábláját a tippel és annak eredményével.

- Az AI sora

```
which_algorithm = 'monte carlo'
ai_guess = run_algorithm(which_algorithm)
ai_guesses += [ai_guess]
if ai_guess in player_ship_positions:
    result = 'hit'
    player_ship_positions.remove(ai_guess)
else:
    result = 'miss'
update_board(player_board, ai_guess, result)
```

Az AI sorában az AI ellenfél tippel a játékos tábláján. Jelenleg az AI a 'monte carlo' algoritmust használja a tipp meghatározásához, de az opciók között szerepel a 'minimax' illetve az 'alpha-beta' meghívás is. Ezek megvalósítása a fentebb levő részekben volt tárgyalva.

A kitalált cella hozzáadódik az AI kitalált celláinak listájához.

Ha a tipp megegyezik a játékos egyik hajójának pozíciójával (`player_ship_positions`), akkor az egy találat. A kitalált hely törlésre kerül a `player_ship_positions` listából.

Ellenkező esetben ez egy hibás találat.

A `update_board()` függvényt hívjuk meg, hogy frissítsük a játékos tábláját az AI találgatásával és annak eredményével.

- Győzelem ellenőrzése

```
if check_victory(ai_ship_positions, player_guesses):  
    print('Congratulations! You sank all the AI\'s ships!')  
    break
```

Minden forduló után a kód ellenőrzi a győzelmi feltételeket, hogy eldöntse, véget kell-e vetni a játéknak.

A `check_victory()` függvényt a mesterséges intelligencia hajóinak pozíciójával és a játékos tippjeivel hívjuk meg. Ellenőrzi, hogy a mesterséges intelligencia összes hajója elsüllyedt-e azáltal, hogy ellenőrzi, hogy nem maradtak-e hajóállások, és `True`-t ad vissza, ha a feltétel teljesül.

Ha a győzelmi feltétel teljesül, a játékkör megszakad, és megjelenik egy győzelmi üzenet.

- Újra játszás

```
if not play_again():  
    break
```

A játék befejezése után a játékosnak lehetősége van újra játszani. Ha a játékos úgy dönt, hogy nem játszik újra (a `play_again()` visszatér `False`), a külső játékkör megszakad, és a program kilép. Ellenkező esetben a játékhurok újraindul, és egy új játék kezdődik.

4. fejezet

A Monte Carlo algoritmus erősségei

4.1. Komplex és nem determinisztikus játékok kezelése

A bizonytalanság kezelése a Monte Carlo Tree Search (MCTS) egyik fő erőssége. A MCTS jól alkalmazható bizonytalan kimenetelű és rejtett információval rendelkező játékokhoz, ahol a játék teljes állapota nem ismert, vagy az ellenfél cselekedetei nem teljesen megfigyelhetők.

Az ilyen játékokban a MCTS véletlenszerű szimulációkat, más néven rolloutokat vagy playoutokat használ a különböző lehetséges utak feltárására és a lehetséges kimenetek értékelésére. Az algoritmus a játék aktuális állapotából indul ki, és véletlenszerű lépések sorozatát hajtja végre, amíg el nem éri a végállapotot. Ezek a szimulációk figyelembe vehetők az ellenfelek akcióinak vagy rejtett állapotainak bizonytalanságát. Amire a mi esetünkben nem volt feltétlenül szükség.

A játéknak az aktuális állapotból történő ismételt szimulálásával a MCTS statisztikai becslést kap a különböző lépések és stratégiák minőségéről. Információkat gyűjt bizonyos kimenetel valószínűségéről, és értékeli a különböző akciókhoz kapcsolódó potenciális kockázatokat és jutalmakat. A mi esetünkben ezt a kockázati elemet eltávolítottuk, mivel csak 3 mélységű volt az első találatunkhoz rendeltük a kettő értéket, biztosítva, hogy vagy olyan forgatókönyvet választunk, ahol valószínűleg kétszer is találhatunk, vagy a kezdeti tippünkkel találunk.

A hagyományos játszmajáték-algoritmusokban gyakran egy pozícióértékelő függvényt terveznek egy adott játékpozíció minőségének vagy kíváncs voltának értékelésére. Ez a függvény különböző tényezők, például a bábuértékek, a tábla ellenőrzése, az anyagi egyensúly vagy más, a területre jellemző mérőszámok alapján rendel egy számértéket az egyes pozíciókhoz. Egy hatékony értékelő függvény megtervezése azonban kihívást jelenthet és időigényes lehet, mivel szakértői tudást és kiterjedt finomhangolást igényel minden egyes játékhoz.

A MCTS más megközelítést alkalmaz. Randomizált szimulációkat, azaz rolloutokat használ, hogy az aktuális állapotból kiindulva feltárja a játékfát, és a szimulációk során megfigyelt eredmények alapján megbecsülje a különböző pozíciók értékét. A pozíciók explicit kiértékelése helyett a MCTS implicit módon tanulja meg a pozíciók értékét a keresési folyamat során gyűjtött statisztikákból.

A MCTS kiválasztási fázisa során az algoritmus a keresési fa csomópontjait a feltárás és a kiaknázás közötti egyensúly alapján választja ki. A kevésbé látogatott csomópontok

felfedezésével és a magasabb várható jutalommal rendelkező csomópontok kihasználásával a MCTS előre meghatározott értékelő függvény nélkül gyűjt információt a különböző pozíciók minőségéről.

A pozícióértékelő függvénytől való függés hiánya miatt a MCTS nagymértékben alkalmazkodik a különböző játékokhoz, és csökkenti a széleskörű szakterület-specifikus tudás szükségességét. A MCTS a játékok széles skálájára alkalmazható anélkül, hogy manuális funkciótervezésre vagy az értékelési függvények finomhangolására lenne szükség. Ez a rugalmasság teszi a MCTS-t hatékony technikává a különböző szabálykészletekkel, mechanikákkal vagy stratégiai megfontolásokkal rendelkező összetett játékok kezeléséhez.

Fontos azonban megjegyezni, hogy a MCTS-nek továbbra is szüksége van egy jó szimulációs politikára és megfelelő játékspecifikus szabályokra a szimulációk hatékony irányításához. A szimulációs politika minősége és az elvégzett szimulációk száma befolyásolhatja a MCTS által kapott becslések pontosságát és megbízhatóságát. Mindazonáltal azáltal, hogy a MCTS előre meghatározott pozícióértékelő függvény helyett szimulációkat használ, sokoldalú és hatékony megközelítést kínál az összetett játékok kezelésére kiterjedt területspecifikus ismeretek nélkül. [14][4][5]

4.2. Skálázhatóság

A Monte Carlo Tree Search (MCTS) kontextusában a skálázhatóság arra utal, hogy az algoritmus képes hatékonyan kezelni az egyre nagyobb keresési tereket és az egyre összetettebb döntéshozatali problémákat. Ez a MCTS azon képessége, hogy hatékonyan alkalmazkodik és skálázza teljesítményét a probléma méretének és összetettségének növekedésével.

A MCTS skálázhatósága két fő szempontot foglal magában:

- Nagyobb keresési terek kezelése: A MCTS-t olyan hatalmas keresési terekkel rendelkező problémák kezelésére tervezték, ahol az egyes döntési pontokon lehetséges lépések vagy cselekvések száma jelentős. A skálázhatóság ebben a kontextusban azt jelenti, hogy a keresési tér exponenciális növekedésével a MCTS képes hatékonyan navigálni és felfedezni a teret túlzott számítási igény nélkül.
- Nagy elágazási tényezők kezelése: Az elágazási tényező az egyes döntési pontokon elérhető lehetséges lépések vagy cselekvések átlagos számát jelenti. A MCTS skálázhatósága azt jelenti, hogy az elágazási tényező növekedésével az algoritmus továbbra is képes megalapozott döntéseket hozni és hatékonyan felfedezni a keresési fát a számítási követelmények exponenciális növekedése nélkül.

A skálázhatóság a MCTS-ben azért lényeges, mert lehetővé teszi, hogy az algoritmus nagyobb kihívást jelentő és realisabb problématerületeket kezeljen. A nagy keresési terek hatékony feltárásával és a magas elágazási faktorok kezelésével a MCTS még összetett forgatókönyvekben is képes jó minőségű döntéseket hozni. Ez biztosítja, hogy az algoritmus praktikus és hasznos maradjon olyan problémák esetén is, amelyek egyébként kimerítő keresési módszerekkel számításigényesen megoldhatatlanok lennének.

Végül soron a MCTS skálázhatóságának célja, hogy a probléma méretének és összetettségének növekedésével egyensúlyt teremtsen a számítási hatékonyság és a döntéshozatal minősége között. Az algoritmusnak képesnek kell lennie alkalmazkodni és skálázni

a teljesítményét annak érdekében, hogy hatékony és megbízható eredményeket szolgáltatson a problématerületek és döntési terek széles skáláján.

A MCTS jól alkalmazható a párhuzamosításra is, mivel az egyes szimulációkat, vagy rolloutokat egymástól függetlenül lehet elvégezni. Minden szimuláció a játékfaban egy-egy különböző útvonalat vizsgál, és az eredmények a végén kombinálhatók, hogy átfogó becslést kapjunk a lépés értékéről.

A MCTS párhuzamosítása során több szimuláció futtatható egyszerre különböző szálon vagy feldolgozóegységeken. Minden szimuláció a játékállapot saját másolatán dolgozik, és az eredmények egymástól függetlenül halmozódnak fel. Ha az összes szimuláció befejeződött, a szimulációk során összegyűjtött statisztikákat, például a látogatások számát és a jutalmakat összevonjuk a keresési fa frissítéséhez.

A szimulációk párhuzamos futtatásával a MCTS a játékfa nagyobb részét tudja feltárni egy adott időkereten belül. Ez a párhuzamosítás különösen előnyös olyan összetett játékokban, amelyekben rengeteg lehetséges lépés van, vagy amikor az egyes szimulációk eredményeinek kiértékeléséhez kiterjedt számításokra van szükség.

A párhuzamosítás lehetővé teszi továbbá, hogy a MCTS hatékonyan skálázódjon az elosztott számítási rendszerekben. Több gépet vagy csomópontot lehet használni a szimulációk párhuzamos futtatására, és mindegyik gép a szimulációk egy részhalmazáért felel. Az eredmények központilag kombinálhatók a keresési fa frissítéséhez és a döntéshozatali folyamat irányításához. [15][16]

4.3. Egyszerűség

A Monte Carlo-fa keresés (MCTS) kontextusában az egyszerűség az algoritmus tervezésének és megvalósításának egyértelműségére, egyszerűségére és minimalizmusára utal. Ez magában foglalja a felesleges bonyolultság csökkentését, a túl bonyolult szabályok vagy függőségek elkerülését, valamint a döntéshozatal tömör és intuitív keretének fenntartását.

Az egyszerűség fontossága a MCTS algoritmusok tervezésében a következő pontokon keresztül érthető meg:

- **Megérthetőség és értelmezhetőség:** Az egyszerűség biztosítja, hogy a MCTS belső működése könnyen érthető és magyarázható legyen. A felesleges bonyolultság minimalizálásával az algoritmus megközelíthetőbbé és átláthatóbbá válik, lehetővé téve a kutatók, fejlesztők és felhasználók számára, hogy megértsék logikáját és viselkedését. Ez a megértés kulcsfontosságú a MCTS hatékony használatához, elemzéséhez és fejlesztéséhez a különböző területeken.
- **A megvalósítás egyszerűsége:** Az egyszerűség leegyszerűsíti a MCTS megvalósításának folyamatát. A világos és tömör kialakítással a fejlesztők gyorsan megérthetik a kulcsfogalmakat, és túlzott erőfeszítés nélkül megvalósíthatják az algoritmust. Ez az egyszerű megvalósítás csökkenti a belépési korlátokat, így a MCTS elérhetővé válik a kutatók és szakemberek szélesebb köre számára, akik esetleg nem rendelkeznek kiterjedt szakértelemmel az összetett algoritmusok terén.
- **Hibakeresés és tesztelés:** Az egyszerű algoritmus-tervezés segíti a MCTS hibakeresését és tesztelését. Ha problémák vagy hibák merülnek fel, a világos és egyszerű struktúra lehetővé teszi a problémák hatékony azonosítását és megoldását. Az

egyszerűség megkönnyíti az alapos tesztelést is, mivel az algoritmus viselkedése és teljesítménye könnyen ellenőrizhető a várt eredményekkel szemben.

- Rugalmasság és alkalmazkodóképesség: Az egyszerű algoritmus-tervezés szilárd alapot biztosít a bővítéshez és módosításhoz. Azáltal, hogy a MCTS alapvető összetevői tömörek és jól definiáltak, a kutatók és a gyakorlati szakemberek indokolatlan bonyolultság nélkül vezethetnek be variációkat vagy szabhatják az algoritmust az egyes problématerületekhez. Az egyszerűség ösztönzi az innovációt és a kísérletezést, mivel lehetővé teszi a kutatók számára, hogy a MCTS-re építsenek és azt egyedi igényeiknek megfelelően fejlesszék.
- Hatékonyság és teljesítmény: A MCTS egyszerűsége hozzájárulhat a számítási hatékonysághoz és teljesítményhez. A felesleges számítási többletköltségek és a bonyolult szabályalapú rendszerek elkerülése révén az algoritmus hatékonyabban működhet, így alkalmas lehet valós idejű döntéshozatali forgatókönyvek vagy erőforráskorlátozott környezetek számára. A MCTS áramvonalas jellege gyorsabb iterációkat és a számítási erőforrások hatékony kihasználását teszi lehetővé. [17]

5. fejezet

A Monte Carlo algoritmus gyengeségei

5.1. Nehézségek bizonyos játékhelyzetek kezelésében

A Monte Carlo Tree Search (MCTS) egy népszerű algoritmus szekvenciális döntéshozatali problémák megoldására és játékrrobotok tervezésére. Hatékonysága ellenére a MCTS-nek van néhány gyengesége, amelyek bizonyos kontextusokban korlátozzák a teljesítményét. A MCTS egyik fő gyengesége, hogy bizonyos játékhelyzeteket nehezen kezel, ami nem optimális teljesítményhez vagy a konvergencia elmaradásához vezethet.

Egyes játékokban bizonyos helyzetek speciális kezelést vagy ismereteket igényelhetnek az optimális teljesítmény eléréséhez. Például a Go játékban vannak olyan stratégiai minták, mint a joseki és a fuseki, amelyek optimális lejátszásához speciális ismeretekre és szakértelemre van szükség. A MCTS nehezen tudja ezeket a mintákat önmagában felfedezni, mivel véletlenszerű mintavételre támaszkodik, és nem feltétlenül fedezi fel kellőképpen a keresési teret. Hasonlóképpen, a rejtett információval vagy hiányos információval rendelkező játékokban, mint például a póker, a MCTS nehezen tudja pontosan megbecsülni egy akció értékét, mivel a véletlenszerű szimulációk eredményére támaszkodik, és hosszú távon nem biztos, hogy megragadja az akció valódi értékét.

Egy másik játékhelyzet, amely kihívást jelent a MCTS számára, a hosszú távú függőségek vagy késleltetett jutalmak jelenléte. Egyes játékokban előfordulhat, hogy egy akció kimenetele nem látszik azonnal, és a jutalom csak több lépés vagy forduló után realizálódik. A MCTS nehezen tudja megragadni ezeket a hosszú távú függőségeket, mivel az akció azonnali jutalmára összpontosít, és nem veszi figyelembe az akció jövőbeli következményeit. Ez nem optimális teljesítményhez vagy a megoldáshoz való konvergálás elmaradásához vezethet.

Ezen túlmenően a MCTS nehezen kezelheti az összetett szabályokkal vagy kölcsönhatásokkal rendelkező játékokat is. Egyes játékokban a szabályok összetettek lehetnek, vagy a játékosok közötti kölcsönhatások bonyolultak, ami megnehezíti a MCTS számára a játék mögöttes dinamikájának megragadását. A Diplomácia játékban például a játékosok tárgyalnak egymással és szövetségeket kötnek, ami jelentősen befolyásolhatja a játék kimenetelét. A MCTS nehezen tudja megragadni ezeket a társadalmi dinamikákat, mivel minden egyes játékost független ágensként kezel, és nem feltétlenül ragadja meg a köztük lévő stratégiai kölcsönhatásokat.

Az egyes játékhelyzetek kezelésével kapcsolatos nehézségek kezelésére a kutatók a MCTS különböző kiterjesztéseit és módosításait javasolták. Például rejtett információval rendelkező játékokban a MCTS kombinálható olyan technikákkal, mint az információkészslet Monte Carlo fa keresés (ISMCTS) vagy a Counterfactual Regret Minimization (CFR), hogy pontosabban megbecsüljék egy akció értékét. A hosszú távú függőségeket tartalmazó játékokban a MCTS módosítható, hogy beépítse a lookahead vagy tervezési technikákat, hogy megragadja egy akció jövőbeli következményeit. Az összetett szabályokkal vagy kölcsönhatásokkal rendelkező játékokban a MCTS kombinálható olyan technikákkal, mint az ellenfélmodellezés vagy a tanulás, hogy megragadják a játékosok közötti stratégiai kölcsönhatásokat.

Bizonyos játékhelyzetek kezelésének nehézségei kihívást jelentenek a MCTS számára, és korlátozzák a teljesítményét bizonyos kontextusokban. Megfelelő kiterjesztések és módosítások beépítésével azonban a MCTS fejleszthető az ilyen helyzetek kezelésére, és az alkalmazások széles körében jobb teljesítményt érhet el. [4][18]

5.2. A pontosság hiánya és szimulációs eredmények változékonysága

A MCTS egyik másik gyenge pontja az algoritmusban használt szimulációs modell pontosságának hiánya. Ha a szimulációs modell nem pontos, a MCTS nem biztos, hogy képes megtalálni a leghatékonyabb utat, ami szuboptimális eredményekhez vezethet. Ez különösen igaz a bonyolultabb, nagy állapot- és akcióterekkel rendelkező játékokban. Ezen túlmenően a MCTS hajlamos lehet a korai lépésekre, ami szintén szuboptimális eredményekhez vezethet.

Hasonlóképpen a szimulációs eredmények változékonysága a Monte Carlo Tree Search (MCTS) egyik gyengesége, amely hatással lehet a különböző alkalmazásokban nyújtott teljesítményére. A szimulációk véletlenszerűsége miatt a MCTS-ben az egyes gördülések eredményei nagyon változóak lehetnek. Ez szuboptimális eredményekhez vezethet, és megnehezítheti a leghatékonyabb útvonal megtalálását.

Ezenkívül a MCTS-ben használt szimulációs modell minősége is jelentősen befolyásolhatja a teljesítményét. Ha a szimulációs modell nem pontos, a MCTS nem biztos, hogy képes megtalálni a leghatékonyabb útvonalat. Továbbá a MCTS nagy állapot- és akcióterekkel küzdhet, ami megnehezítheti a leghatékonyabb útvonal megtalálását.

E gyengeségek ellenére a MCTS továbbra is hatékony megközelítés a játékrrobotok tervezéséhez vagy a szekvenciális döntési problémák megoldásához. A kutatók folyamatosan dolgoznak az algoritmus tökéletesítésén és korlátainak kezelésén, hogy hatékonyabbá tegyék a különböző területeken. [19]

6. fejezet

Technikák összehasonlítása

6.1. Minimax

A Minimax algoritmus egy olyan döntéshozatali algoritmus, amely úgy működik, hogy a játékban az összes lehetséges lépést megvizsgálja, és minden egyes lépéshez pontszámot rendel. Az algoritmus feltételezi, hogy az ellenfél a lehető legjobb lépést teszi, a játékos pedig azt a lépést teszi, amely minimalizálja az ellenfél pontszámát. Az algoritmus ezután kiválasztja azt a lépést, amely a játékos számára a legmagasabb pontszámot adja.

Másrészt a Monte Carlo Tree Search algoritmus egy heurisztikus kereső algoritmus, amely úgy működik, hogy nagyszámú véletlenszerű játékot szimulál, és kiválasztja azt a lépést, amely a legjobb eredményhez vezet. Az algoritmus a játék aktuális állapotából kiindulva többszörös véletlenszerű játékok szimulálásával felépíti a lehetséges lépések és azok kimeneteleinek fáját. Az algoritmus ezután a szimulációk alapján kiválasztja azt a lépést, amely a legjobb eredményhez vezet.

6.1. táblázat. Monte carlo futási idő / lépés (100 játék után)

| minimum | átlag | maximum | mélység | lépésszám |
|---------|---------|----------|---------|-----------|
| 0 ms | 0.11 ms | 1.02 ms | 1 | 50 |
| 0 ms | 0.13 ms | 1.45 ms | 1 | 80 |
| 0 ms | 0.92 ms | 6.98 ms | 2 | 50 |
| 0 ms | 1.41 ms | 9.97 ms | 2 | 80 |
| 0.96 ms | 1.78 ms | 12.96 ms | 3 | 50 |
| 0.98 ms | 2.70 ms | 14.98 ms | 3 | 80 |

6.2. táblázat. Monte carlo pontosság (100 játék után)

| mélység | lépésszám | legjobb eset | átlagos eset | legrosszabb eset |
|---------|-----------|--------------|--------------|------------------|
| 1 | 50 | 100% | 87% | 70% |
| 1 | 80 | 100% | 93% | 77% |
| 2 | 50 | 100% | 97% | 81% |
| 2 | 80 | 100% | 98% | 94% |
| 3 | 50 | 100% | 98% | 89% |
| 3 | 80 | 100% | 99% | 94% |

6.3. táblázat. Minimax futási idő / lépés (100 lépés után)

| minmum | átlag | maximum | mélység |
|-----------|-----------|---------|---------|
| 0 ms | 0.08 ms | 1.10 | 1 |
| 3.89 ms | 5.69 ms | 9.99 | 2 |
| 508.64 ms | 569.41 ms | 634.30 | 3 |

Futásidő-összehasonlítás: A fenti statisztikák azt mutatják, hogy a MCTS általában sokkal alacsonyabb futási idővel rendelkezik a Minimaxhoz képest, különösen a mélység növekedésével. Az 1 és 50 lépéses mélység esetén a MCTS minimális, átlagos és maximális futási ideje 0 ms, 0,11 ms és 1,02 ms. A futási idők azonban nagyobb mélységgel és a lépések számával nőnek.

Másrészt a Minimax minimális, átlagos és maximális futási ideje 0 ms, 0,08 ms és 1,10 ms 1 mélység esetén. A futási idők azonban jelentősen nőnek a mélység növekedésével, és elérik az 508,64 ms, 569,41 ms és 634,30 ms értéket 3 mélység esetén.

Fontos megjegyezni, hogy mindkét algoritmus futási idejét befolyásolhatja a játékfa mérete, az elágazási tényező és a rendelkezésre álló számítási erőforrások. Emellett a MCTS futási idejét befolyásolhatja a játékfa felépítéséhez használt szimulációk száma, míg a Minimax futási idejét a használt heurisztikus kiértékelő függvény minősége.

Ezért megállapítható, hogy a MCTS általában gyorsabb, mint a Minimax, különösen a mélyebb keresési forgatókönyvek esetében, ahol a Minimax futási ideje exponenciálisan nő.

A pontosság összehasonlítása: A MCTS pontosságát a legjobb lépés megtalálásában nyújtott teljesítményével mérjük. A megadott statisztikák azt mutatják, hogy a MCTS 100%-os pontosságot ér el 1-es mélység és 50 lépéses esetén. A mélység és a lépések számának növekedésével az átlagos és a legrosszabb eset pontossága kissé csökken, de még mindig nagyon magas marad. A 3-as mélység és 80 lépés esetén az átlagos pontosság 99%, a legrosszabb esetben pedig 94%.

Ezzel szemben a Minimax pontossága 100% mivel egy determinisztikus algoritmus, optimális megoldásokat ad, ha elegendő időt kap a teljes játékfa átkutatására.

Fontos megjegyezni, hogy mindkét algoritmus pontosságát befolyásolhatja az alkalmazott heurisztikus kiértékelő függvény minősége, a játékfa mérete és az elágazási tényező. Továbbá a MCTS pontosságát komplexebb esetekben befolyásolhatja a játékfa felépítéséhez használt szimulációk száma is, míg a Minimax pontosságát csak a maximális keresési mélység.

A rendelkezésre álló statisztikák alapján a MCTS nagy pontosságot mutat, miközben a Minimaxhoz képest lényegesen alacsonyabb futási időt biztosít, különösen a mélyebb keresési forgatókönyvek esetében. Viszont a Minimax 100%-os pontosságot biztosít.

6.2. Alpha-beta metszés

Az alfa-béta metszés a Minimax algoritmus kiterjesztése. A keresési folyamat során a felesleges értékelések csökkentésével javítja a hatékonyságot. A Minimaxhoz hasonlóan az Alpha-Beta Pruning a játék összes lehetséges lépését figyelembe veszi, és az egyes lépések értékeléséhez pontszámokat rendel. Feltételezi, hogy az ellenfél optimálisan játszik, és a játékos célja az ellenfél pontszámának minimalizálása.

6.4. táblázat. Alpha-beta metszés futási idő / lépés (100 lépés után)

| minmum | átlag | maximum | melység |
|----------|----------|---------|---------|
| 0 ms | 0.07 ms | 1.10 | 1 |
| 3.96 ms | 4.79 ms | 7.97 | 2 |
| 45.87 ms | 64.79 ms | 99.60 | 3 |

Futásidő-összehasonlítás:

MCTS: A MCTS minimális, átlagos és maximális futási ideje 0 ms és 14,98 ms között mozog, a mélységtől és a lépések számától függően. A futási idők a nagyobb mélységgel és a lépések számával nőnek, a maximális futási idő 3 mélység és 80 lépés esetén figyelhető meg. Alfa-béta metszés: Az alfa-béta metszés minimális, átlagos és maximális futási ideje a mélységtől függően 0 ms és 99,60 ms között változik. A futási idők is nőnek a nagyobb mélységgel, a maximális futási idő a 3 mélységnél figyelhető meg. A futási idők összehasonlításából nyilvánvaló, hogy a MCTS általában alacsonyabb futási idővel rendelkezik az Alpha-Beta Pruninghoz képest, különösen a mélyebb keresési forgatókönyvek esetében. A MCTS maximális futási ideje (14,98 ms) jelentősen alacsonyabb, mint az Alpha-Beta Pruning maximális futási ideje (99,60 ms) 3 mélységnél.

Pontosság-összehasonlítás:

MCTS: A MCTS pontosságát a legjobb lépés megtalálásának teljesítménye alapján mérjük. A megadott statisztikák a legjobb, az átlagos és a legrosszabb eset pontosságát mutatják különböző mélységek és lépésszámok esetén. A pontosságok a legrosszabb és a legjobb esetben 70% és 100% között mozognak, az átlagos eset pedig ezen értékek közé esik. A MCTS minden esetben magas pontosságot ér el, de a pontosság a mélység és a lépések számának növekedésével kissé csökken. Alfa-béta metszés: Az Alpha-Beta Pruning pontossága minden esetben 100%-osnak említhető, ami azt jelzi, hogy mindig megtalálja az optimális lépést, ha elegendő időt kap a teljes játéka átkutatására. A rendelkezésre álló statisztikák alapján az Alpha-Beta Pruning 100%-os pontosságot mutat, míg a MCTS magas, 70% és 100% közötti pontosságot ér el. Fontos azonban megjegyezni, hogy a futási idő összehasonlítása a MCTS-nek kedvez, mivel általában alacsonyabb futási idővel rendelkezik az Alpha-Beta Pruninghoz képest, különösen a mélyebb keresési forgatókönyvek esetében. Érdemes megfontolni a pontosság és a futási idő közötti kompromisszumot, amikor e két algoritmus között választunk, az alkalmazás konkrét követelményeitől és korlátaiktól függően.

Az algoritmusok relatív teljesítménye azonban az adott problématerülettől és a megvalósítás részleteitől függően változhat. Ezért fontos alaposan mérlegelni a futási idő és a pontosság közötti kompromisszumokat, amikor egy adott problémához algoritmust választunk.

Összefoglaló

A Monte Carlo Tree Search eredményeinek kiértékelése után nyilvánvaló, hogy ez a megközelítés rendkívül hatékony a szekvenciális döntési problémák megoldására és a játékbrobotok tervezésére. Az intelligens fakesés alkalmazásával, amely egyensúlyt teremt a felfedezés és a kihasználás között, a MCTS képes megalapozott döntéseket hozni a véletlenszerű mintavételes szimulációk segítségével. Ez az algoritmus különösen alkalmas a mélységi mérlegelést és döntéshozatalt igénylő játékokhoz, és könnyen beállítható a különböző nehézségi szintek létrehozásához.

Összességében a MCTS sikere annak tulajdonítható, hogy minden egyes iteráción keresztül folyamatosan javulni képes. Ez az iteratív folyamat lehetővé teszi, hogy az algoritmus finomítsa döntéshozatali képességeit, és végül optimális eredményeket érjen el. Így a MCTS népszerű választássá vált a játékfejlesztők és a döntéshozatali szakértők körében egyaránt.

A jövőben valószínűleg továbbra is számos alkalmazásban fogjuk látni a MCTS használatát. A videojátékok tervezésétől kezdve az összetett döntéshozatali folyamatokig ez a megközelítés rendkívül hatékonynak és alkalmazkodóképesnek bizonyult. Ahogy a technológia tovább fejlődik, lehetséges, hogy a MCTS még kifinomultabb változatait látjuk majd megjeleníteni, tovább bővítve a képességeit és a lehetséges alkalmazásokat.

[Az államvizsgálóhoz használt GitHub repo](#)

Köszönetnyilvánítás

Köszönetek szeretnék mondani a témavezető tanáromnak, Horobet Emilnek. Illetve köszönöm a Sapientia EMTE Marosvásárhelyi Karának a képzést, amely megalapozta a projekthez szükséges ismereteimet.

Ábrák jegyzéke

| | |
|-----------------------------------------------------------------|----|
| 2.1. Példa egy valós Battleship táblára. | 14 |
| 2.2. Példa monte carlo tree searchre a tic-tac-toe-ban. | 16 |
| 2.3. Példa minimax algoritmusra tic-tac-toe-ban. | 17 |
| 2.4. Példa alpha-beta metszésre. | 18 |
| 3.1. MCTS fázisai ábrázolva | 21 |

Táblázatok jegyzéke

| | | |
|------|------------------------------------------------------------------|----|
| 6.1. | Monte carlo futási idő / lépés (100 játék után) | 42 |
| 6.2. | Monte carlo pontosság (100 játék után) | 43 |
| 6.3. | Minimax futási idő / lépés (100 lépés után) | 43 |
| 6.4. | Alpha-beta metszés futási idő / lépés (100 lépés után) | 44 |

Irodalomjegyzék

- [1] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. 1953-06-01. [Online]. Available: <https://www.semanticscholar.org/paper/Equation-of-state-calculations-by-fast-computing-Metropolis-Rosenbluth/f6a13f116e270dde9d67848495f801cdb8efa25d>
- [2] S. Brown. How to play the battleship board game. 2020-11-12. [Online]. Available: <https://www.thesprucecrafts.com/the-basic-rules-of-battleship-411069>
- [3] M. Kalos and P. Whitlock. Monte carlo methods. 2013-03-01. [Online]. Available: <https://phyusdb.files.wordpress.com/2013/03/monte-carlo-methods-second-revised-and-enlarged-edition.pdf>
- [4] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk. Monte carlo tree search: A review of recent modifications and applications. 2022-06-11. [Online]. Available: <https://arxiv.org/abs/2103.04931>
- [5] R. Roy. Monte carlo tree search (mcts). [Online]. Available: <https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/>
- [6] baeldung. Minimax algorithm. 2023-03-11. [Online]. Available: <https://www.baeldung.com/cs/minimax-algorithm>
- [7] geeksforgeeks. Minimax algorithm in game theory | set 1 (introduction). [Online]. Available: <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/>
- [8] Alpha-beta pruning. [Online]. Available: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
- [9] geeksforgeeks. Minimax algorithm in game theory | set 4 (alpha-beta pruning). [Online]. Available: <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/>
- [10] python. [Online]. Available: <https://www.python.org/>
- [11] python. [Online]. Available: [https://en.wikipedia.org/wiki/Python_\(programming_language\)](https://en.wikipedia.org/wiki/Python_(programming_language))

- [12] M. H. Winands. Minimax algorithm in game theory | set 4 (alpha-beta pruning). [Online]. Available: https://dke.maastrichtuniversity.nl/m.winands/documents/Encyclopedia_MCTS.pdf
- [13] O. Sofela. Minimax algorithm guide: How to create an unbeatable ai. 2020.12.09. [Online]. Available: <https://www.freecodecamp.org/news/minimax-algorithm-guide-how-to-create-an-unbeatable-ai/>
- [14] D. Whitehouse. Monte carlo tree search for games with hidden information and uncertainty. 2014.08.01. [Online]. Available: <https://core.ac.uk/download/pdf/30267707.pdf>
- [15] A. Bourki, G. Chaslot, M. Coulm, V. Danjean, H. Doghmen, J.-B. Hoock, T. Hérault, A. Rimmel, F. Teytaud, O. Teytaud, and P. V. . Z. Yu. Scalability and parallelization of monte-carlo tree search. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-17928-0_5
- [16] K. Yoshizoe, A. Kishimoto, T. Kaneko, H. Yoshimoto, and Y. Ishikawa. Scalable distributed monte-carlo tree search. 2021-08-19. [Online]. Available: <https://ojs.aaai.org/index.php/SOCS/article/view/18194>
- [17] A. Liu, J. Chen, M. Yu, Y. Zhai, X. Zhou, and J. Liu. Watch the unobserved: A simple approach to parallelizing monte carlo tree search. 2023-05-05. [Online]. Available: <https://openreview.net/forum?id=BJlQtJSKDB>
- [18] P. Cowling, E. Powley, and D. Whitehouse. Information set monte carlo tree search. 2013-06-28. [Online]. Available: <https://eprints.whiterose.ac.uk/75048/>
- [19] M. U. Chaudhry. Motifs: Monte carlo tree search based feature selection. 2018-04-13. [Online]. Available: <https://www.mdpi.com/1099-4300/20/5/385>