Универзитет „Св. Кирил и Методиј“ во Скопје

Факултет за информатички науки и компјутерско инженерство

Вештачка интелигенција

Извештај

Анализа на играта *Quoridor*

Студент:

Љубица Дамјановиќ, 221173

Професор:

Д-р Андреа Кулаков

Февруари, 2025

Содржина

1. Апстракт

Во овој извештај ќе бидат презентирани и анализирани алгоритми за информирано пребарување кои се користат за наоѓање на оптимална победничка стратегија во стратешката игра „Quoridor“. Играта Quoridor, која комбинира елементи на планирање, стратегија и просторно размислување, претставува идеална платформа за истражување на ефикасни алгоритми за пребарување и евалуација на состојби.

Целта на оваа студија е да се дефинираат и имплементираат различни алгоритми за информирано пребарување, како што се A* (A-star) кој се користи за изнаоѓање на пат, Минимакс (Minimax) со алфа-бета поткастрување, Expectimax и Monte Carlo Tree Search (MCTS), кои ќе бидат прилагодени специфично за динамиката и правилата на Quoridor.

За секој алгоритам ќе бидат прикажани статистички податоци во однос на нивните перформанси, вклучувајќи го времето на одлучување, бројот на прегледани јазли и успешноста во наоѓање на оптимално решение. Овие метрики ќе овозможат споредбена анализа на алгоритмите, со цел да се идентификураат нивните предности и ограничувања во контекст на Quoridor.

2. Вовед

Стратешките игри претставуваат важна област за истражување во вештачката интелигенција, бидејќи нивната комплексност бара примена на напредни алгоритми за пребарување и одлучување. Една таква игра е Quoridor, апстрактна друштвена игра која комбинира просторна ориентација, тактичко планирање и предвидување на потезите на противникот.

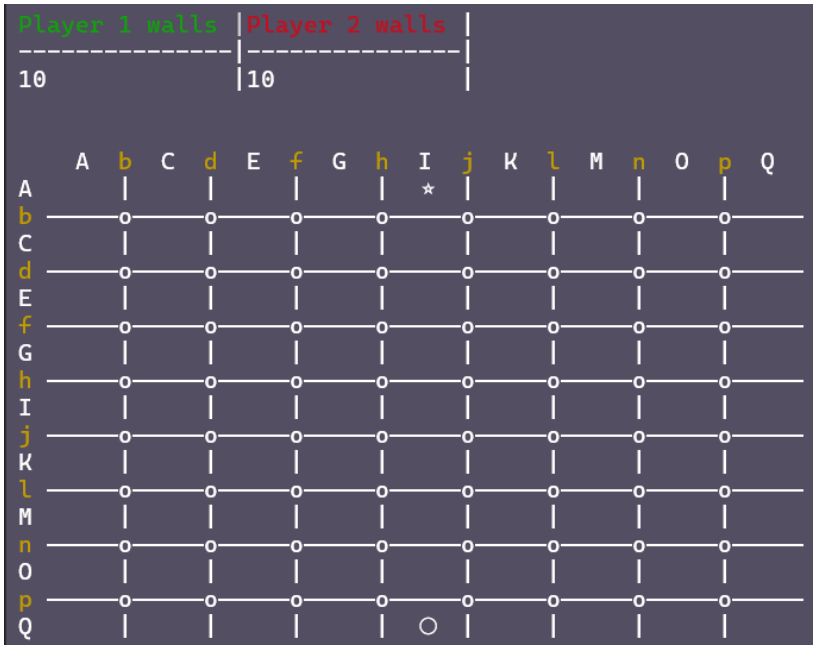
Основната цел на играта е играчите да го поместат својот пион низ таблата до спротивниот крај, додека истовремено поставуваат ѕидови за да го забават противникот. Оваа едноставна, но стратешки богата механика ја прави играта идеална за истражување на алгоритми за пребарување и оптимално одлучување.

Во секој потег, играчот избира една од двете можни акции: поместување на пион или поставување ѕид. Откако ќе ја донесе својата одлука и ја изврши акцијата, потегот завршува и противникот го започнува својот потег.

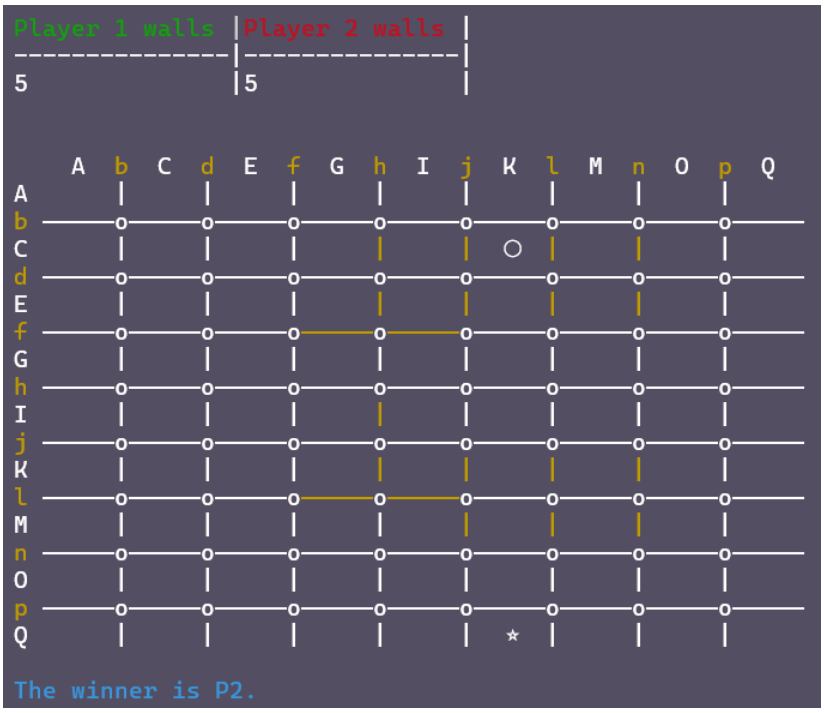
Играчот мора да ги следи следните правила при извршување на својот потег:

1. Поместување на пион:
   a. Играчот може да го помести својот пион во една од соседните позиции, под услов да не е попречен од ѕид.
   b. Доколку противничкиот пион се наоѓа на соседна позиција и меѓу нив нема ѕид, играчот може да го прескокне противникот, поместувајќи го својот пион на следната слободна позиција зад него.
   c. Ако зад противничкиот пион има ѕид, играчот може да го помести својот пион дијагонално во една од слободните позиции лево или десно од противникот.
   d. Секое поместување мора да остане во границите на таблата.

2. Поставување ѕид:
   a. Ѕидот мора да биде поставен на слободна позиција, каде што веќе нема друг ѕид.
   b. Ѕидовите не смеат да се вкрстуваат со други ѕидови.
   c. Поставениот ѕид не смее целосно да го блокира патот на противникот до неговата цел – секој играч мора да има барем еден можен пат до спротивната страна на таблата.



Почетна состојба на таблата



Победа за играч 2

3. Алгоритми

3.1 A* алгоритам за изнаоѓање на пат

Овој алгоритам е **клучен** за правилно функционирање на играта, бидејќи Quoridor не дозволува играч да биде целосно блокиран од противникот.

Како работи алгоритмот?

1. Се иницијализира приоритетна редица (open_list)
   - Започнува од дадената почетна позиција (start_pos).
   - Креира почетен јазол (Node), каде што:
     - g_cost = 0 (растојание од почетокот до тековниот јазол)
     - h_cost = Manhattan distance (проценето растојание до целната линија)
     - f_cost = g_cost + h_cost (вкупен трошок)
   - Го додава почетниот јазол во редицата open_list.
2. Се користи A* алгоритам за да најде пат до целната линија
   - Во секоја итерација, се зема јазолот со најмал f_cost од редицата (приоритетна редица со heapq).
   - Ако јазолот се наоѓа на целната линија (првата или последната редица), тогаш патот постои и се враќа True.
   - Ако јазолот веќе бил посетен, се игнорира.
3. Се генерираат валидни соседни позиции (get_neighbors())
   - Се проверуваат четирите можни насоки (горе, долу, лево, десно).
   - Ако помеѓу тековната позиција и соседот има ѕид (BoardPieceStat.OCCUPIED_WALL), тогаш тој потег се игнорира.
   - Ако е валиден, новиот јазол се додава во редицата.
4. Ако се обработат сите можни потези и не се стигне до целната линија, патот не постои (False)
   - Ова значи дека ѕидот го блокира целиот пат и не смее да се постави.

Клучни аспекти на алгоритмот

- A* алгоритмот е ефикасен бидејќи користи Manhattan distance како евристика (h_cost) за да води кон целта побрзо.
- Гарантира оптимален пат, освен ако нема валидна патека.
- Ако path_exists() врати False, значи дека не може да се постави ѕидот, бидејќи би го блокирал патот на играчот.

3.2 Евристика за минимакс и експектимакс

Евристичката функција за Quoridor ги проценува позициите на играчите и ги претвора во скаларна вредност, која се користи во Minimax или Expectimax за да одреди кој потег е подобар.

Како работи евристиката?

1. Пресметува основни растојанија
   - player_one_distance: Растојание на првиот играч до неговата целна линија (по x-координатата).
   - player_two_distance: Растојание на вториот играч до неговата целна линија (16 - позицијата на играчот).
2. Користи A* алгоритам (astar.path_exists())
   - Проверува најкраткиот пат до целта за тековниот играч.
   - Ако двајцата играчи имаат по 10 ѕида (макс.), тогаш ја прескокнува пресметката.
3. Конструирање на резултатот (result)
   - Позитивни фактори (добро за играчот кој е на потег):
     - + opponent_path_len (колку подолго трае патот на противникот, толку подобро).
     - + round(100 / player_path_len, 2) (ако патот на играчот е краток, добива повисок резултат).
     - + разлика во ѕидови (ако играчот има повеќе ѕидови, тоа е предност).
     - + 100 ако играчот е на целната линија (победа).
   - Негативни фактори (лошо за играчот кој е на потег):
     - - player_distance (колку подалеку е играчот од целта, толку полошо).
     - - round(50 / opponent_path_len, 2) (ако противникот има краток пат, добива помалку казна).
     - - 500 ако нема валиден пат до целта (блокирање = катастрофа).
4. Expectimax фактор (ако се користи Expectimax наместо Minimax)
   - Утежнува патот на противникот со 17 * opponent_path_len за да нагласи агресивни потези.

Што прави оваа евристика?

- Предност дава на побрзо стигнување до целта (намалување на player_path_len).
- Обидува се да го забави противникот (зголемување на opponent_path_len).
- Препознава важноста на ѕидовите (повеќе ѕидови = предност).
- Избегнува блокирање (ако играчот се блокира, враќа многу негативен резултат).

Заклучок

Оваа евристика е баланс помеѓу брзина, sидови и спречување на противникот. Работи добро за Minimax и Expectimax и ја наградува агресивната стратегија кога има многу sидови.

3.3 Минимакс алгоритам со алфа-бета поткаструване

Алгоритмот Minimax со алфа-бета поткаструване е техника за донесување оптимални одлуки во двојни игри со нулта збир.

Како работи алгоритмот?

1. Базен случај:
   - Доколку е достигната максималната длабочина (depth == 0), алгоритмот враќа вредност од евалуациската функција evaluate_position(game_state, turn), која го проценува квалитетот на дадената позиција за тековниот играч.
2. Minimax пребарување:
   - Ако е ред на максимизирачкиот играч, алгоритмот бара потег кој ја максимизира вредноста на позицијата:
     - Иницијално поставува max_eval = -∞.
     - Генерира сите можни потези (all_possible_moves(turn)) за тековниот играч и ги разгледува секоја од новонастанатите состојби.
     - Рекурзивно повикува minimax за следната состојба со намалена длабочина.
     - Чува најдобрата вредност и ја ажурира alpha.
     - Ако beta <= alpha, се врши поткаструване и пребарувањето прекинува (за да се избегнат непотребни пресметки).
   - Ако е ред на минимизирачкиот играч, алгоритмот бара потег кој ја минимизира вредноста на позицијата:
   - Иницијално поставува min_eval = +∞.
   - Слично како во случајот на максимизирачкиот играч, но овој пат го ажурира beta.
3. Алфа-бета поткаструване:
   - Оваа оптимизација ги намалува непотребните пресметки така што прескокнува поддрвја кои не можат да влијаат на конечниот избор.
   - Ако во некој момент beta <= alpha, понатамошното разгледување на тековното поддрво се прекинува.

Временска комплексност

Во најлош случај, Minimax без поткаструване има сложеност од:

$$O(b^d)$$

каде што:

- b е бројот на можни потези во секој чекор (гранков фактор),
- d е длабочината на пребарувањето.

Со алфа-бета поткаструване, најдобриот можен случај (кога потезите се разгледуваат во идеален редослед) има сложеност:

$$O\left(b^{d/2}\right)$$

што значи дека ефективната длабочина на пребарување е преполовена, значително намалувајќи го бројот на разгледани состојби. Во просек, алфа-бета поткаструването дава значително побрзи резултати, но крајната брзина зависи од редоследот на разгледување на потезите.

3.4 Експектимакс

Expectimax е варијанта на Minimax која се користи во игри каде што еден од играчите прави случајни потези или кога постои некој елемент на случајност (на пр. коцкање во некои игри). Наместо да претпоставува дека противникот игра на оптимален начин (како што прави Minimax), Expectimax ги разгледува можните потези како стохастички распределени и пресметува просечна вредност од сите можни излезни состојби.

Како работи алгоритмот?

1. Базен случај:
   - Доколку depth == 0, алгоритмот враќа евалуациска вредност од evaluate_position(game_state, turn, True), што претставува хеуристичка проценка на позицијата.
2. Maximizing играч:
   - Ако е ред на максимизирачкиот играч, алгоритмот одбира потег со најголема вредност (како кај Minimax).
   - Ги генерира сите можни потези (all_possible_moves(turn)) и за секоја состојба рекурзивно повикува expectimax.
   - Го враќа максималниот резултат од разгледаните потези.
3. Chance нодови (стохастички противник):
   - Ако не е ред на максимизирачкиот играч (на пр. ако очекуваме случаен потег), тогаш алгоритмот смета просечна вредност на сите можни потези.
   - Генерира сите можни потези, пресметува expectimax за секоја состојба, и враќа средна вредност од добиените резултати.
   - Наместо најдобриот потег, како кај Minimax, Expectimax користи средна вредност на можните излезни состојби за да одлучи што е најдобар потег.

Временска комплексност

Во најлош случај, Expectimax има сложеност:

$$O(b^d)$$

каде што:

- b е бројот на можни потези (гранков фактор),
- d е длабочината на пребарувањето.

Expectimax не користи алфа-бета поткастрување, бидејќи во min јазлите (каде што противникот одбира потези) не се презема најлошото сценарио, туку се пресметува просек. Ова значи дека Expectimax е побавен од Minimax со алфа-бета поткастрување, но може да биде покорисен во игри со елементи на случајност или непредвидливи противници.

3.5 Монте Карло

Monte Carlo Tree Search (MCTS) е метод за донесување одлуки кој се користи во игри со големо пребарувачко пространство, како што е Quoridor. Овој алгоритам користи случајни симулации за да процени кои потези се најдобри, што го прави ефикасен за игри каде што класичните алгоритми како Minimax би биле прескапи поради експоненцијалниот раст на можните состојби.

Како работи алгоритмот?

MCTS функционира преку четири главни фази кои се повторуваат одреден број пати (итерации):

1. Селекција (Selection):
   - Почнува од коренот на дрвото (тековната состојба на играта) и се движи надолу кон длабоките нивоа на дрвото избирајќи ги најдобрите потези според UCT (Upper Confidence Bound for Trees) вредноста.
   - Ова е имплементирано со методот tree_policy(), кој го избира најдобриот веќе истражен јазол користејќи формулата

     $$UCT = \frac{Q}{N} + C \times \sqrt{\frac{2\ln(N_{parent})}{N}}$$

     каде што:
     - Q е бројот на добиени партии во дадениот јазол,
     - N е бројот на посети на јазолот,
     - N_{parent} е бројот на посети на родителскиот јазол,
     - C е параметар за истражување (обично 0.1 или √2).
   - Методот best_child(c_param=0.1) се користи за оваа селекција.
2. Експанзија (Expansion):
   - Ако тековниот јазол не е целосно истражен, алгоритмот генерира ново дете со случаен неиспитан потег.
   - Ова е имплементирано со методот expand(), кој зема случаен потег од untried_actions(), го додава во дрвото и го враќа како нов јазол.
3. Симулација (Rollout):
   - Од новододадениот јазол, алгоритмот извршува случајна симулација (игра до крај) со случајни потези.

- Методот rollout() игра случајни потези додека не се стигне до крај на партијата (is_goal_state()).
- Крајниот резултат е 1 ако победил максимизирачкиот играч, -1 ако загубил.

4. Назадна пропагација (Backpropagation):
   - Крајниот резултат од симулацијата се пренесува наназад низ дрвото, ажурирајќи ги вредностите на посетените јазли.
   - Методот back_propagate() ги ажурира вредностите на бројот на посети (number_of_visits) и резултатите (results).

Временска комплексност

MCTS нема фиксна сложеност, бидејќи зависи од бројот на симулации. Сепак, ако:

- b е бројот на можни потези,
- d е просечната длабочина на симулациите,
- N е бројот на симулации,

тогаш сложеноста е приближно:

$$O(N \times b \times d)$$

Кога N е големо, MCTS дава подобри резултати, но троши повеќе време.

Предности и ограничувања

✅ Предности:

- Ефикасен за игри со големо пребарувачко пространство (како Quoridor).
- Не бара експлицитна евалуациска функција, бидејќи користи симулации.
- Флексибилен – може да се користи за различни игри со минимални промени.

❌ Ограничувања:

- Бавен ако бројот на симулации е мал.
- Лош за детерминистички игри со ограничен број на потези, бидејќи Minimax со алфа-бета поткастрување може да биде побрз.
- Перформансите зависат од изборот на параметрите (како бројот на симулации и UCT константата).

4. Резултати

Спроведените тестови покажуваат значителни разлики во времето на извршување за различните алгоритми при пресметување на најдобриот потег во Quoridor:

- Minimax: 0.02 - 0.03s
- Expectimax: 0.02 - 0.03s
- Monte Carlo Tree Search (MCTS): 0.15 - 0.17s

Анализа на резултатите

1. Minimax и Expectimax имаат слична брзина
   - Двата алгоритма извршуваат потег за приближно 20-30 ms.
   - Expectimax е минимално побавен во некои случаи, но разликата е незначителна.
   - Ова е очекувано, бидејќи Expectimax е варијација на Minimax, која ја користи просечната вредност на можните потези наместо најлошиот случај.
2. Monte Carlo Tree Search (MCTS) е значително побавен
   - Извршувањето трае 5-8 пати подолго (150-170 ms).
   - Ова се должи на големиот број симулации кои MCTS мора да ги изврши за да процени позициите.
   - Иако е побавен, MCTS има потенцијал да донесува попрецизни одлуки во комплексни ситуации, каде што Minimax и Expectimax можат да бидат ограничени од длабочината на пребарувањето.

Заклучок

- Minimax и Expectimax се оптимални за брзи одлуки и се соодветни за ограничени временски услови.
- MCTS нуди подобра стратегиска проценка, но бара повеќе време. Може да биде корисен во асинхрони стратегии или ако играчот има повеќе време за размислување.
- Во реална игра, може да се испроба хибриден пристап, каде што MCTS се користи во клучни позиции, додека Minimax/Expectimax се користат за побрзи одлуки.

5. Користена литература

https://en.wikipedia.org/wiki/Quoridor

https://www.ultraboardgames.com/quoridor/game-rules.php

https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/

https://gibberblot.github.io/rl-notes/single-agent/mcts.html

https://github.com/gorisanson/quoridor-ai