УДК 519.688

DOI 10.31651/2076-5886-2022-1-31-41

PACS 02.60.-x, 02.60.Pn

**ЮХИМЕЦЬ Максим Вячеславович**
студент спеціальності «Прикладна математика» Черкаського національного університету імені Богдана Хмельницького
e-mail: yukhymets.maksym1617@vu.cdu.edu.ua

**КРАСНОШЛИК Наталія Олександрівна**
кандидат технічних наук, доцент, доцент кафедри прикладної математики та інформатики Черкаського національного університету імені Богдана Хмельницького
e-mail: wlik007@ukr.net
ORCID 0000-0003-4661-6997

# РОЗРОБКА ПРОГРЕСИВНИХ СТРАТЕГІЙ ГРИ У ШАХИ ТА ЇХ ЗАСТОСУВАННЯ ПРИ РЕАЛІЗАЦІЇ ШАХОВОГО ДВИГУНА

*У роботі розглянуті алгоритми для програмування моделей ігор для двох осіб з нульовою сумою та повною інформацією. Такі алгоритми беруть свій початок в теорії ігор та широко використовуються для розробки шахових двигунів. З розвитком галузі шахового програмування безліч вдосконалень було запропоновано для класичних алгоритмів minimax та alpha-beta. Використання найбільш оптимального стеку вдосконалень під час розробки власного двигуна стало однією з цілей даної роботи. Порівняльний аналіз був виконаний та відповідні таблиці наведені в роботі. Різні комбінації алгоритмів та їх вдосконалень билу протестовані у власному середовищі представленому шаховим двигуном.*

***Ключові слова:*** *шаховий двигун, minimax, alpha-beta..*

**Вступ.**

Шахи це гра для двох осіб з нульовою сумою та повною інформацією. Окрім цього шахи – найпопулярніша стратегічно-інтелектуальна гра в світі зі століттями історії та масивним корпусом теоретичних знань. Імовірно, шахи являються найбільш задокументованою грою з найбільшою кількістю спеціалізованої літератури присвяченої тому як в них грати. Зі стрімким розвитком компьютерних технологій впродовж останнього століття момент коли дослідники звернуть свою увагу на проблему шахів залишався питанням часу. Перша наукова робота яка піднімала це питання була опублікована в 1950 році Клодом Шеноном. У статті [1] Шеннон обговорював потенціал використання комп'ютерів для гри в шахи і запропонував ідею створення комп'ютерної програми, яка могла б симулювати й оцінювати можливі ходи в шаховій грі. Робота Шеннона поклала початок дослідженням можливостей автоматизації гри в шахи. На сьогоднішній день ця галузь досягла значних успіхів в поставлених завданнях і сформулювала об'ємний теоретичних базис.

Історично цей процес протікав в декілька етапів: 1950 – 1980 робота велася вченими-академіками які публікували свої розробки в наукових журналах; з появою інтернету в 1980-х тематичні спільноти присвячені шахам і розробкам двигунів почали з'являтися, вільний обмін ідей в значній мірі стимулював розвиток галузі; 1990-2010 конференції і змагання серед авторських двигунів набирали оберти, вчені, професійні

розробники і дослідники-ентузіасти конкурували один з одним, шахове програмування було комерційно вигідним, автори призових двигунів могли продавати ліцензійні копій свого програмного продукту; 2010 – наш час, спільнота в значній мірі сформулювала найкращі практики і методики, рівень складності двигунів перевищів можливості індивідуальних розробників, найсильніші двигуни підтримуються групами програмістів, подальший ріст рейтингу двигунів сильно залежить від наявності процесорного ресурсу для тестування нових ідей і вдосконалень.

Незважаючи на те, що сучасні шахові двигуни знаходяться на абсолютно іншому рівні гри ніж навіть найсильніший гравець-гросмейстер, повне вирішення шахової проблеми все ще не досяжне, адже навіть найсильніший шаховий двигун сьогодення запущений на найкращому комп'ютері не здатний повністю прорахувати всю шахову партію з позиції не в ендшпілі.

**Мета статті.**

Продемонструвати результати роботи класичних алгоритмів з домену розробки шахових двигунів з різними варіантими їх вдосконалень. Навести короткий опис отриманих результатів та використаних вдосконалень.

**Виклад основного матеріалу.**

**1. Основний опис**

Для досягнення поставлених задач було розроблено відповідне середовище – шаховий двигун, мова програмування була обрана Java [6]. Повний опис реалізації всіх складових компонентів двигуна виходить за рамки данної статті, наведемо лише ключові деталі. Шаховий двигун складається з генератора кроків, функції оцінки і механізму пошуку. Кожен компонент може бути реалізований абсолютно відокремлено, настільки що може стати окремим проектом і використовуватися як змінний модуль в інших двигунах. Можлива і протилежна ситуація, коли компоненти двигуна тісно сполучені між собою. Це часто обумовлено оптимізаційними міркуваннями. Так наприклад генератор кроків може виконувати внутрішне сортування [3], так як безпосередньо в момент генерації доступна корисна інформація яка буде коштувати час при повторному отриманні в компоненті пошуку.

Робота двигуна виглядає наступним чином:

1. Шахова позиція представленна в одному із компактних форматів запису шахових позицій *FEN* передається з інтерфейсу в двигун;
2. Інформація про стан дошки декодується і зберігається, викликається функція пошуку;
3. Пошук, в основі якого лежить алгоритм minmax, створює дерево гри, яке представляє всі можливі ходи та їхні наслідки. Він починається з поточної позиції та досліджує можливі ходи для обох сторін, створюючи розгалужене дерево позицій.

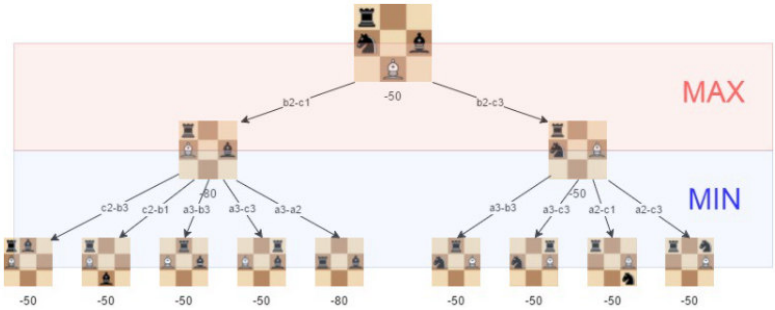


Рис. 1. Схема дерева пошуку [4]

4. В кожному вузлі дерева викликається генератор кроків в який передається стан дошки на даний момент. Генератор повертає список легальних кроків для стороно яка ходить відносно цього вузла дерева [7]. Генератор може повертати один крок, запам'ятовуючи його, таким чином якщо відбудеться відсікання цієї гілки на даному кроці, час на генерацію повного списку буде зекономлений. Такий підхід буде вигідним тільки якщо генератор виконує сортування кроку в середині себе інакше цей підхід буде контр ефективним.
5. Список кроків повертається в вузол дерева де сортується за певними евристиками. Це необхідно для того щоб імовірно кращі кроки шукалися першими так як вони мають більший шанс викликати відсікання. Сортування може базуватися на багатьох евристиках і враховувати захищені і атаковані клітини обох сторін, проте в більшості випадків сортування відбувається за такими критеріями: pv-крок – кращій хід з попередньої ітерації ітераційного заглиблення / крок з таблиці транспонування - структура даних для зберігання інформації про попередньо оцінені позиції в пошуковому дереві гри; атакуючі виграшні кроки відсортовані по SEE [4] або MVV-LVA [4]; евристика вбивці - ідея запам'ятовувати хороші не атакуючі ходи які спричиняють бета відсікання; евристика історії – менш ефективна хеш-таблиця не атакуючих кроків які викликали відсікання раніше в дереві; програшні атакуючі ходи; всі інші не атакуючі ходи, можуть буди відсортовані по статичним Piece-Square Tables – таблиці значень що характеризують доцільність знаходження певної фігури на певній клітинці дошки. Стандартний набір евристик може бути доповнений або замінений менш популярними ідеями і цілком залежить від результатів тестування.
6. Обраний припустимо найкращій хід виконується, змінюючи стан дошки, пошук рекурсивно викликається знову, поток виконання переходить в наступний вузол дерева.
7. Коли пошук досяг зазначеної глибини відбувається виклик фукції оцінки стану дошки. Функція оцінки повертає чисельні значення які інтерпретуються як додатне значення – позиція на дошці на користь сторони для якої функція оцінює, від'ємне – позиція не на користь сторони для якої викликана оцінка і нуль – позиція ні в чию користь. Робота функції оцінки може бути скільки завгодно складною і враховувати найтонші теоретично-стратегічні прийоми гри. Цей аспект розробки шахового двигуна заслуговує окремого розгляду так як сучасний консенсус передбачає використання нейронних мереж [5].
8. Значення функції оцінки повертається на попередній рівень виклику де використовується для зміни діапазону alpha-beta вікна.
   Alpha-beta вікно – це фундаментальна оптимізація пошуку яка дає можливість безпечно відсікати безперспективні гілки, економлячи тим самим величезну кількість часу. Alpha-beta – це два значення які змінюються на протязі всього пошуку, представляючи тим самим нижню і верхню границю допустимого значення кроку якій реально може буди зіграний. Alpha – це максимальна нижня границя, а beta – це мінімальна верхня, разом вони утворюють вікно пошуку. Якщо повернуте значення функції оцінки більше ніж beta то це значить, що протилежна сторона ніколи не зіграє цей крок і вся гілка відсікається, якщо значення менше beta і більше alpha то воно стає новим alpha, звужуючи вікно пошуку, якщо значення менше alpha то пошук в вузлі продовжується.
9. Продовжуючи пошук по дереву, найбільше значення alpha піднімається в корень дерева і відповідний йому крок повертається як найкращій в даній позиції для даної сторони.

```
score alphabeta(position,alpha,beta,depth)
{
  if (isTerminal(position))
    return valueOf(position);

  if (depth == 0)
    return evaluate(position);

  children = getChildren(position);
  while (not children.empty())
  {
    child = children.removeOne();

    value = -alphabeta(child, -beta, -alpha, depth - 1);

    if (value > alpha)
    {
      if (value >= beta)
        return beta;
      alpha = value;
    }
  }
  return alpha;
}
```

Рис. 2. Псевдокод алгоритму пошуку з alpha-beta [4]

Більшість вдосконалень пошуку пов'язані з прискоренням відсікання через різний вплив на alpha-beta вікно шляхом штучного звуження вікна а також зменшення глибини пошуку для не перспективних гілок. Такі методи хоч і пришвидшують роботу пошуку однак являються не повністю безпечними так як залишають імовірність відкинути потенційно найкращій крок. Детальне тестування у вигляді реальних партій проти набору тестових двигунів-опонентів повинно бути проведено для підтвердження ефективності того чи іншого вдосконалення.

## 2. Опис специфічних компонентів

Так як опис всіх складових розробленого шахового двигуна виходить за межі даної статті означимо лише те що важливо для отримання представлених результатів.

### 2.1 Quiescence Search

Коли ми виконуємо альфа-бета пошук з фіксованою глибиною у нас немає гарантії того, що вузол дерева на якому ми зупинили пошук і викликали функцію оцінки не являється небезпечною позицією. Якщо ми рухаємося по дереву пошуку і на кінцевому вузлу наш ферзь бере ворожого пішака, функція оцінки поверне нам додатне значення, так як це змінить матеріальний баланс в нашу користь. Проте в такій ситуації не враховується що може статися з нашою фігурою на наступній глибині, а може відбутися так, що нашу фігуру заберуть фігури які були зв'язані з пішаком.

Ми цього не бачимо під час пошуку так як він обривається на глибині на якій здається що перевага в нашу корить. Щоб уникнути цього нам потрібно продовжувати пошук для всіх вузлів які закінчується атаками. Тобто якщо ми шукаємо на глибину 5, доходимо до цієї глибині в одній з гілок дерева і останній вузол в ній це атака, то ми продовжуємо пошук для цього вузла до тих пір поки не опинимося в безпечній або тихій позиції. Такий підхід буде розширювати дерево пошуку за межі заданої глибини проте не настільки сильно щоб це було проблемою, так як послідовності атак в шахах не бувають сильно довгими. Критерієм для виконання quiescence search може бути не тільки ситуація коли пошук гілки закінчується вузлом з атакою, а і взагалі будь яка ігрова ситуація яка на думку розробника варта більшої уваги. Наприклад просування пішака часто призводить до мату. Таким чином якщо пошук по гілці закінчується позицією де піша стоїть на передостанньому для свого руху рангу, варто продовжити пошук щоб побачити чи призведе просування до чогось важливого. Чим більш

специфічні ситуації покриваються таким чином, там більш стратегічно сильної становиться гра шахового двигуна. Зрозуміло також що такі додатки розширюють дерево пошуку, що сповільнює його роботу.

```
int Quiesce( int alpha, int beta ) {
    int stand_pat = Evaluate();
    if( stand_pat >= beta )
        return beta;
    if( alpha < stand_pat )
        alpha = stand_pat;

    until( every_capture_has_been_examined )  {
        MakeCapture();
        score = -Quiesce( -beta, -alpha );
        TakeBackMove();

        if( score >= beta )
            return beta;
        if( score > alpha )
           alpha = score;
    }
    return alpha;
}
```

Рис. 3. Псевдокод quiescence search [4]

2.2 Сортування кроків

Як описано раніше, правильне впорядкування ходів є важливим для ефективності альфа-бета алгоритму. Так як чим раніше відбудеться бета відсікання тим менше вузлів доведеться пройти. Кроки сортуються в порядку їх імовірної важливості. Першим розташовують крок з хеш-таблиці, якщо вона існує. Після нього розташовуються потенційно виграшні атаки, відсортовані по потенційні перевазі в матеріальному балансі яку вони можуть здобути. Виграшні атаки – це атаки які залишають сторону яка ходить в плюсі по матеріальному балансі [9]. Такі кроки можуть варіюватися між собою як більш цінні або менш в залежності від того яка фігура атакує яку. Це так звана MVA-LVV “Most Valuable Victim - Least Valuable Aggressor” евристика. Так як фігури в шахах несуть в собі різну цінність, ми можемо відсортувати всі атаки по наступному критерію, якщо атакуюча фігура менша по цінності за фігуру яку атакують, то такий хід вважається більш переважним ніж якщо відбувається навпаки. Тобто хід де пішак бере ферзя буде стояти раніше ніж хід де ферзь бере пішака. Ця евристика легка в реалізації проте не сильно акуратна в оцінці. В заміну їй часто використовують більш складний підхід під назвою SEE “Static Exchange Evaluation”. В нашому проекті ми використовуємо його. Ідея полягає в тому щоб оцінити атакуючий хід урахувавши те що після атаки атакуюча фігура може бути взята фігурою противника, а та в свою чергу нашою фігурою і так до тих пір поки послідовність взяття фігур на конкретній клітинці не закінчиться. Таким чином ми можемо точно оцінити атаку. Приклад на рис. 4 демонструє положення демонструє положення при якому атака конем для білих краща ніж атака слоном.

Дана позиція дай білим виконати дві атаки, c3d5 і f3d5. Виходячи з того, що пішак в нашій програмі оцінений в 100 балів, кінь в 300, а слон в 350, виконуючи c3d5 хід ми отримаємо + 100 за пішака, -300 за нашого коня який буде взятий у відповідь і +300 за взяття слоном коня. Таким чином чистий приріст в матеріальному балансі для білої сторони буде 100 білів. Походивши f3d5, розрахуємо таким же чином і побачимо, що приріст 50 балів. Така логіка за SEE евристикою.

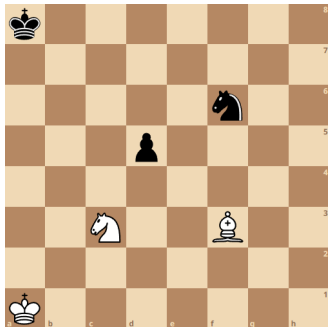

Рис. 4. Приклад позиції

Після того як ми відсортували атакуючи кроки по одній з евристик, ми переходимо до звичайних кроків і використовуємо так звану евристику вбивці. Евристика вбивці – це ідея запам'ятовувати хороші не атакуючі ходи які спричиняють бета відсікання і шукати їх першими після ходів з хеш-таблиці і виграшних атакуючих ходів. Якщо сторона має надзвичайно хороший не атакуючий хід в одній позиції, то часто цей хід буде хорошим і в інших позиціях на цьому ж шарі пошуку. Тобто коли ми будемо генерувати кроки в інших вузлах на цьому ж шарі пошуку, при сортуванні цих кроків ми будемо перевіряти чи містять вони вбивчі ходів збережених раніше і якщо так то ми перемістимо їх в позицію після виграшних атак. В типових реалізаціях ми зберігаємо декілька вбивчих ходів на кожному шарі.

Далі застосовується так звана евристика історії, також до ходів без захоплення. Це загальний і менш ефективний спосіб запам'ятовування хороших рухів. Кожного разу, коли відбувається відсікання, запис таблиці, індексований вихідним і кінцевим квадратами ходу, збільшується на деяку величину. Коли всі інші евристики були застосовані до списку кроків на вузлі, інші рухи без захоплення сортуються за значеннями в цій таблиці історії. Таким чином, ходи, які часто були хорошими в інших частинах дерева, будуть шукатися перед ходами, про які ми менше знаємо.

2.3 Iterative Deepening

Очевидний спосіб використання альфа-бета алгоритму є виклик його з заданою глибиною [8]. Це породжує деякі проблеми. По-перше, ми не знаємо, наскільки мілкий пошук ми можемо зробити, щоб «вирішити» позицію, тобто знайти фактичний найкращий хід. Крім того, ми не знаємо, скільки часу триватиме пошук, перш ніж він завершиться або перш ніж він принаймні знайде хороший хід. Це може бути проблемою якщо існує часове обмеження, а так як для сучасних двигунів це де факто стандарт, ми будемо використовувати цю техніку також. Ідея ітераційного заглиблення в тому щоб запускати альфа-бета пошук в циклі, кожен раз збільшуючи глибину. Таким чином у нас завжди буде хоча б якийсь хід для повернення і він не буде абсолютно випадковим. З першого погляду здається, що це суттєво сповільнить роботу пошуку, але насправді це не зовсім так. Якщо двигун використовує хеш-таблицю в якій зберігаються найкращі ходи з попередніх пошуків, то ітераційне заглиблення майже не сповільнює пошук, так як кожна наступна ітерація пошуку використовує інформацію із попередньої ітерації. Також данні про вбивчі кроки і евристика історії будуть пришвидшувати пошук на наступних ітераціях.

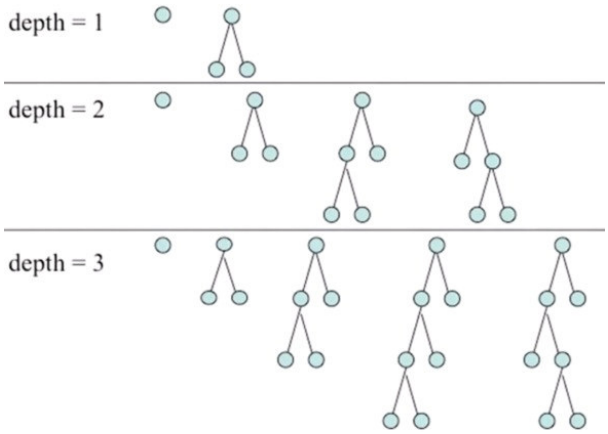


Рис. 5. Візуалізація iterative deepening [2]

3. Опис результатів

Після створення власного двигуна можна переходити до тестування відповідних стеків вдосконалень. Тестування шахового двигуна можна проводити декількома способома: запускати пошук з певної позиції і дивитися кількість пройдених вузлів і час затрачений на досягнення заданої глибини; запускати ігрові партії проти інших двигунів приблизно рівної сили. В нашому випадку був обраний перший варіант. Дві позиції розглядається в якості тестових, початкова позиція і складна позиція після розіграшу фігур.

Рис. 6.1 Початкова позиція

Рис. 6.2 Складна позиція

Для зручності позначимо позицію на рис. 6.1 позиція A, а на рис. 6.2 позиція B. Таблиця 1 містить результати пошуку для стандартного альфа-бета пошуку без інших вдосконалень.

Як бичимо, значення часу для B позиції виявилися навіть меншими. Скоріше за все, так відбувається тому що коли пошук починається з позиції B в дереві пошуку відбувається більше бета відсікань, так як з позиції B генерується більше атак. Також ми бачимо, що навіть не значна глибина, по міркам сильних шахових двигунів займає значний час для пошуку. Слід зазначити, що в світі шахового програмування сильні двигуни обраховують глибину 11-12 менш ніж за секунду.

Розглянемо як зміняться результати якщо додати quiescence search.

Таблиця 1.1 Результати alpha-beta пошук позиція A

| Глибина | Час мс | Вузли |
|---|---|---|
| 5 | 221 | 5395 |
| 6 | 582 | 87127 |
| 7 | 1072 | 386050 |
| 8 | 6510 | 5324402 |
| 9 | 44626 | 33006491 |
| 10 | 366010 | 316084543 |

Таблиця 1.2 Результати alpha-beta пошук позиція B

| Глибина | Час мс | Вузли |
|---|---|---|
| 5 | 150 | 13686 |
| 6 | 630 | 150471 |
| 7 | 1621 | 518908 |
| 8 | 5581 | 4394516 |
| 9 | 36528 | 14868399 |
| 10 | 187636 | 175741641 |

Таблиця 2.1 Результати alpha-beta + quiescence search пошук позиція A

| Глибина | Час мс | Вузли |
|---|---|---|
| 5 | 60 | 36836 |
| 6 | 741 | 371141 |
| 7 | 2113 | 3041757 |
| 8 | 10560 | 24701472 |
| 9 | 72653 | 197224184 |
| 10 | 775851 | 1793110479 |

Таблиця 2.2 Результати alpha-beta + quiescence search пошук позиція B

| Глибина | Час мс | Вузли |
|---|---|---|
| 5 | 929 | 782742 |
| 6 | 3417 | 5162845 |
| 7 | 6593 | 13438137 |
| 8 | 66173 | 116746648 |
| 9 | 141150 | 352460245 |

Як і очікувалося додання quiescence search значно збільшує час пошуку і кількість пройдених вузлів. Для позиції A на 10 глибині ми шукаємо близько 12 хвилин і проходимо більше мільярда вузлів, для десятої глибини позиції B запускати пошук не має сенсу так як відбудеться так званий “search explosion”, тобто так як кількість атак з цієї позиції велика, багато гілок буде продовжувати пошук викликаючи quiescence search з останнього вузла. Quiescence search являється невід'ємною частиною двигуна, тому доведеться пришвидшувати пошук наступними методами.

Таблиця 3.1 Результати alpha-beta + quiescence search + SEE пошук позиція A

| Глибина | Час мс | Вузли |
|---|---|---|
| 5 | 76 | 36774 |
| 6 | 984 | 364629 |
| 7 | 2292 | 2987920 |
| 8 | 11033 | 23870509 |
| 9 | 76019 | 190871949 |
| 10 | 883899 | 1704353808 |

Таблиця 3.2 Результати alpha-beta + quiescence search + SEE пошук позиція B

| Глибина | Час мс | Вузли |
|---|---|---|
| 5 | 408 | 165098 |
| 6 | 1497 | 1058014 |
| 7 | 2567 | 3756300 |
| 8 | 16070 | 23200460 |
| 9 | 52700 | 110385254 |
| 10 | 338943 | 511747895 |

При додаванні сортування кроків десята глибина для позиції B стає досяжною.

Таблиця 4.1 Результати alpha-beta + quiescence search + SEE + killer heuristic пошук позиція A

| Глибина | Час мс | Вузли |
|---|---|---|
| 5 | 50 | 18114 |
| 6 | 388 | 107399 |
| 7 | 1343 | 823801 |
| 8 | 4138 | 5832305 |
| 9 | 14268 | 37273156 |
| 10 | 107610 | 212638983 |

Таблиця 4.2 Результати alpha-beta + quiescence search + SEE + killer heuristic пошук позиція B

| Глибина | Час мс | Вузли |
|---|---|---|
| 5 | 490 | 164464 |
| 6 | 1167 | 1056752 |
| 7 | 2599 | 3735366 |
| 8 | 13734 | 23134067 |
| 9 | 44589 | 109121751 |
| 10 | 293616 | 508046187 |

Як бачимо alpha-beta + quiescence search + SEE + killer heuristic дає найкращій результат. Слід зазначити, що певну частину швидкодії забезпечує ефективне сортування кроків і оптимізований генератор кроків. Взагалі кожен блок програмного коду в шаховому двигуні займає певний час при виконанні, відповідно максимальна

оптимізація всього проекту бажана задля досягнення максимальної швидкості пошуку. В данній роботі ми виносимо за дужкі цей нюан і зосереджуємося тільки на коді який відноситься до функції пошуку.

**Висновки.**

У статті розглянуто принцип роботи шахового двигуна, описана робота його ключових компонентів та деяких основних методів. Задля перевірки прирпушень стосовно ефективності тих чи інших модифікацій пошуку було розроблено власне середовище – шаховий двигун в якому відбувалося тестування. Була проведена низка тестів на зазначених позиціях і отримані результати. Результати були приведені в таблицях вище. З результатів можна зробити висновок, що варіант alpha-beta + quiescence search + SEE + killer heuristic є найкращім з точки зору часу виконання. Наступний крок в тестуванні може бути запуск n кількості ігор між нашим двигуном і двигуном подібної сили або навіть набором таких двигунів. Такий спосіб тестування дає точніший результат проте потребує набагато більше часу на виконання.

Отримані результати додатково підтверджують спостереження розробників шахових двигунів стосовно ефективності тих чи інших методів, дають можливість додатково переконатися в доцільності їх використання. На сьогоднішній день в галузі шахового програмування налічується десятки методів-модифікацій класичних алгоритмів пошуку, серед них є й такі що використовуються лише в конкретних авторських двигунах. Деякі з них поєднуються з іншими або навпаки перешкоджають роботі один одного тому важливо вносити якумога більше ясності в це питання.

**Список використаної літератури:**


1. Клод Шеннон; «Програмування комп'ютера для гри в шахи» (англ. «Programming a Computer for Playing Chess »), опублікованій в Philosophical Magazine, 1950 р.
2. Lecture 9 | Search 6: Iterative Deepening (IDS) and IDA* [Електронний ресурс]. – Режим доступу: https://www.youtube.com/watch?v=5LMXQ1NGHwU&ab_channel=AlanMackworth
3. "FIDE Laws of Chess taking effect from 1 January 2018". FIDE. Retrieved 12 July 2020. [Електронний ресурс]. – Режим доступу: https://handbook.fide.com/chapter/E012018
4. Сайт, присвячений шаховому програмуванню [Електронний ресурс]. – Режим доступу: https://www.chessprogramming.org
5. Форум присвячений шаховому програмуванню [Електронний ресурс]. – Режим доступу: https://www.talkchess.com
6. Paul Dailly, Dominik Gotojuch, Neil Henning, Keir Lawson, Alec Macdonald, Tamerlan Tajaddinov. “A Chess Engine” University of Glasgow Department of Computing Science Sir Alwyn Williams Building Lilybank Gardens Glasgow G12 8QQMarch 18, 2008 – Режим доступу: https://www.researchgate.net/publication/268426213_A_Chess_Engine
7. Авторський блог присвячений шаховому програмуванню [Електронний ресурс]. – Режим доступу: https://www.youtube.com/channel/UCB9-prLkPwgvlKKqDgXhsMQ
8. Серія статей на інтернет ресурсі gamedev.net [Електронний ресурс]. – Режим доступу: https://www.gamedev.net/tutorials/_/technical/artificial-intelligence/chess-programming-part-i-getting-started-r1014/
9. Авторський блог Computer Chess Programming Theory by Colin Frayn [Електронний ресурс]. – Режим доступу: http://www.frayn.net/beowulf/theory.html


**References:**


1. Shannon, C. (1949). Programming a Computer for Playing Chess. In: Philosophical Magazine, Nov. 8, pp. 256-275. Murray Hill, N.J. [in USA]. https://doi.org/10.1080/14786445008521796
2. Lecture 9 | Search 6: Iterative Deepening (IDS) and IDA* Retrieved from https://www.youtube.com/watch?v=5LMXQ1NGHwU&ab_channel=AlanMackworth
3. "FIDE Laws of Chess taking effect from 1 January 2018". FIDE. Retrieved 12 July 2020. Retrieved from https://handbook.fide.com/chapter/E012018
4. Repository of information «Chess Programming Wiki» Retrieved from https://www.chessprogramming.org
5. Online forum «Computer Chess Club » Retrieved from: https://www.talkchess.com

6. Paul Dailly, Dominik Gotojuch, Neil Henning, Keir Lawson, Alec Macdonald, Tamerlan Tajaddinov. "A Chess Engine" University of Glasgow Department of Computing Science Sir Alwyn Williams Building Lilybank Gardens Glasgow G12 8QQMarch 18, 2008 Retrieved from https://www.researchgate.net/publication/268426213_A_Chess_Engine
7. Author blog devoted to chess programming Retrieved from https://www.youtube.com/channel/UCB9-prLkPwgvlKKqDgXhsMQ
8. Series of articles «Chess Programming» Retrieved from https://www.gamedev.net/tutorials/_/technical/artificial-intelligence/chess-programming-part-i-getting-started-r1014/
9. Frayn, C. Computer Chess Programming Theory Retrieved from http://www.frayn.net/beowulf/theory.html

**YUKHYMETS Maxim,**
Student, Department of Informatics and Applied Mathematics, The Bohdan Khmelnytsky National University of Cherkasy, Ukraine
**KRASNOSHLYK Natalia,**
candidate of Technical Sciences, Associate Professor, The Bohdan Khmelnytsky National University of Cherkasy, Ukraine

**DEVELOPMENT OF ADVANCED CHESS STRATEGIES AND THEIR APPLICATION IN THE IMPLEMENTATION OF A CHESS ENGINE**

***Summary. Introduction.*** *Algorithms for programming models of games for two people with zero sum and complete information are considered in the paper. Such algorithms originate in game theory and are widely used to develop chess engines.*

*As the field of chess programming has developed, many improvements have been proposed to the classic minimax and alpha-beta algorithms. Using the most optimal stack of improvements during the development of one's own engine became one of the goals of this work.*

*A comparative analysis was performed and the corresponding tables are given in the work. Different combinations of algorithms and their improvements were tested in their own environment represented by the chess engine.*

***Purpose.*** *To demonstrate the results of classical algorithms from the domain of development of chess engines with various variants of their improvements. Provide a brief description of the results obtained and the improvements used.*

***Results.*** *In order to check the assumptions about the effectiveness of certain search modifications, an own environment was developed - a chess engine in which testing took place. A number of tests were conducted on the specified positions and the results were obtained. The results were given in the tables above. From the results, we can conclude that the option alpha-beta + quiescence search + SEE + killer heuristic is the best in terms of execution time. The next step in testing could be to run n number of games between our engine and an engine of similar strength, or even a set of such engines. This method of testing gives a more accurate result, but takes much more time to perform.*

***Conclusion.*** *The obtained results further confirm the observations of chess engine developers regarding the effectiveness of certain methods, and provide an opportunity to further verify the feasibility of their use. Today, in the field of chess programming, there are dozens of methods-modifications of classic search algorithms, among them there are those that are used only in specific author's engines. Some of them combine with others or, on the contrary, interfere with each other's work, so it is important to bring as much clarity as possible to this issue.*

***Keywords:*** *chess engine, minimax, alpha-beta.*

*Одержано редакцією 06.12.2021 р.*
*Прийнято до публікації 23.06.2022 р.*