Міністерство освіти і науки, молоді та спорту України
Львівський національний університет імені Івана Франка
Факультет прикладної математики та інформатики

Кафедра теорії оптимальних процесів

Магістерська робота

на тему:

Прийняття комп'ютерних рішень в конфліктних іграх з динамічною структурою та змінними ресурсами

Затверджено на
Засіданні кафедри теорії оптимальних процесів
Протокол № _____ від _____

Зав. кафедри
_____ проф. Бартіш М.Я.

Виконав:
студент групи ПМА-51м
Кушнір Тарас

Науковий керівник:
проф. Сеньо П.С.

Львів - 2012

Зміст

1 Вступ

Класичні задачі прийняття рішень у конфліктних іграх суттєво ускладнюється в тому випадку, коли структура гри динамічна та змінюються ресурси гравців. Особливо добре це проглядається при аналізі шахів, як гри, яка найбільш повно відповідає таким умовам.

Комп'ютерні рішення в іграх почали застосовувати практично від виникнення обчислювальних центрів ще у 60-х роках минулого сторіччя. Спочатку це були розв'язання простих задач з шахів, на зразок ендшпіля короля з королем та турою, та з часом з вдосконаленням технічної бази росли і поставлені задачі. В той час як деякі позиційні ігри стали повністю розв'язаними, на зразок шашок, за алгоритми до розв'язку інших пропонують значні винагороди, на зразок гри "Го".

Всі ці ігри є математичними моделями та вивчаються в розділі позиційних ігор, в яких прийняття гравцями рішень розглядається як багатокроковий процес. Переходи в такому процесі супроводжуються здобуттям або втратою інформації про стани, альтернативи та матеріал цього гравця. Цей процес прийнято представляти у вигляді дерева гри з позиціями у вершинах дерева та ходах - ребрах, які з'єднують ці вершини і ведуть з однієї позиції в іншу.

Комп'ютерний підхід для розв'язання позиційних ігор так чи інакше зв'язаний із переглядом дерева гри повністю або частково. Базовим алгоритмом, який широко використовувався був мінімакс. З часом вдосконалення теоретичної бази на зміну мінімаксу прийшов новий алгоритм "альфа-бета відтинання", який пробув основою комп'ютерних рішень в багатьох іграх і дозволяв отримати коректні результати без відвідування всіх вершин дерева гри. Незважаючи на очевидні мінуси цього алгоритму, такі як фіксована глибина чи підхід, подібний до повного перебору, через які

він не здобув слави у спільноті штучного інтелекту, успіх, який здобув цей алгоритм неможливо не помітити. Такі програми як Deep Thought з шахів (грає на рівні майстра спорту), Chinook з шашок (чемпіон серед програм) та Logistello в грі реверсі (грає сильніше за людей) здобули великий успіх з допомогою цього алгоритму.

Від моменту, коли Ален Невелл, Джон Шоу та Герберт Саймон розробили перші варіанти "альфа-бета відтинання" пройшло більше, ніж півстоліття, і за цей час алгоритм вдосконалили великою кількістю евристик та нових підходів у побудові самого алгоритму.

У даній роботі проведений огляд класичної теорії розв'язання позиційних ігор з динамічною структурою та змінними ресурсами та комп'ютерних рішень, які використовують "альфа-бета відтиння" як базовий алгоритм, реалізацій нових підходів до застосування евристик та порівняльний аналіз швидкості реалізованого рушія для гри шахи, яка найбільш точно відповідає поставленим завданням з точки зору позиційних ігор.

2 Позиційні ігри

Математичні моделі конфліктів, що враховують динаміку, вивчаються в розділі позиційних ігор.

Позиційною грою будемо називати скінченну гру n гравців, що складається з:

- Дерева гри T (Впорядкованої у вигляді дерева множини, що визначає для кожної позиції єдиний шлях, який веде до неї із початкової позиції, а також множину кроків, які можна зробити із цієї позиції безпосередньо в наступні позиції)
- n дійснозначих функцій $(F_1, F_2, \ldots, F_n)$, що визначені в кожній вершині дерева таким чином, що якщо t - вершина, тоді $F_i(t)$ - виграш гравця P_i, якщо партія закінчується в точці t.
- набір чисел $0, 1, \ldots, n$ таких, що кожному вузлу дерева ставиться у відповідність число, що означає чия черга ходити
- набору альтернатив в кожному вузлі дерева, що обмежений правилами гри

Представимо формальне визначення позиційної гри з допомогою теорії графів. [1]

Нехай $G = (X, F)$ - деревовидний граф. Розглянемо розбиття множини вершин X на $n+1$ множину $X_1, X_2, \ldots, X_n, X_{n+1}$, $\bigcup_{i=1}^{n+1} X_i = X$, $X_k \cap X_l = \varnothing$, $k \neq l$, де $F_x = \varnothing$ для $x \in X_{n+1}$. Множина X_i, $i = 1, \ldots, n$ називається *множиною черговості* для i-го гравця, а множина X_{n+1} - множина кінцевих позицій. На множині кінцевих позицій визначені n дійснозначних функцій $H_1(x), H_2(x), \ldots, H_n(x), x \in X_{n+1}$. Функція $H_i(x), i = 1, 2, \ldots, n$ називається виграшем i-го гравця.

Гра відбувається наступним чином. Задана множина N гравців, які пронумеровані натуральними числами $1, \ldots, i, \ldots, n$ (в подальшому $N = 1, 2, \ldots, n$). Нехай $x_0 \in X_{i_1}$, тоді у вершині (позиції) x_0 "ходить" гравець i_1 та вибирає вершину $x_1 \in F_{x_0}$. Якщо $x_1 \in X_{i_2}$, то у вершині x_1 "ходить" гравець i_2 і вибирає вершину (позицію) $x_2 \in F_{x_1}$ і так далі. Отже, якщо на k-му кроці вершина (позиція) $x_{k-1} \in X_{i_k}$, то в ній "ходить" гравець i_k та вибирає наступну вершину з множини $F_{x_{k-1}}$. Гра закінчується, як тільки досягається кінцева вершина $x_i \in X_{n+1}$, тобто така, для якої $F_{x_i} = \varnothing$.

В результаті послідовного вибору позицій однозначно реалізується деяка послідовність $x_0, \ldots, x_k, \ldots, x_l$, яка визначає шлях в деревовидному графі G, який виходить з початкової позиції x_0 і досягає одної з кінцевих позицій гри. Такий шлях називається *партією*. Через деревовидність графа G кожна партія однозначно визначає кінцеву позицію x_l, в яку вона приводить і, навпаки, кінцева позиція x_l, в яку вона приводить, однозначно визначає партію. В позиції x_l кожен з гравців $i, i = 1, \ldots, n$ отримує виграш $H_i(x_l)$.

Вважатимемо, що гравець i при здійсненні вибору в позиції $x \in X_i$ знає цю позицію x і, відповідно, може відновити всі попередні позиції. В такому випадку вважають, що гравці володіють повною інформацією. Прикладами таких ігор є шахи, шашки чи хрестики-нулики. Більшість карточних ігор не є іграми з повною інформацією, оскільки гравці не знають які карти були видані іншим гравцям.

3 Постановка задачі

Однозначне відображення u_i, яке кожній вершині (позиції) $x \in X_i$ ставить у відповідність деяку вершину $y \in F_x$, називається *стратегією гравця* i.

Множину всіх стратегій гравця i позначимо U_i.

Позиційна гра називається грою з нульовою сумою, якщо для будь-якого n-вимірного набору b стратегій гравців $(P_1, P_2, \ldots, P_n)$ буде мати місце рівність

$$\sum_{i=1}^{n} M_i(b) = 0$$

де $M_i(b)$ - функції виграшів стратегії.

Впорядкований набір стратегій $u = (u_1, \ldots, u_i, \ldots, u_n)$, де $u_i \in U_i$ назвемо *ситуацією* в грі. Кожна ситуація однозначно визначає партію в грі, а, відповідно, і виграші гравців. Дійсно, нехай $x_0 \in X_{i_1}$. Тоді в ситуації $u = (u_1, \ldots, u_i, \ldots, u_n)$ наступна позиція x_1 визначається однозначно згідно правилу $u_{i_1}(x_0) = x_1$. Нехай тепер $x_1 \in X_{i_2}$. Тоді x_2 визначається однозначно згідно правилу $u_{i_2}(x_1) = x_2$. Якщо тепер на k-му кроці реалізовувалась позиція $x_{k-1} \in X_{i_k}$, то x_k визначається однозначно згідно правила $u_{i_k}(x_{k-1}) = x_k$.

Нехай ситуації $u = (u_1, \ldots, u_i, \ldots, u_n)$ відповідає партія $x_0, x_1, \ldots, x_n$. Тоді можна ввести поняття функції виграшу K_i гравця i, встановивши її значення в кожній ситуації u рівною значенню виграшу H_i в кінцевій позиції партії $x_0, \ldots, x_n$ з відповідною ситуацією $u = (u_1, \ldots, u_n)$, тобто

$$K_i(u_1, \ldots, u_i, \ldots, u_n) = H_i(x_i), i = 1, \ldots, n.$$

Функції $K_i, i = 1, \ldots, n$ визначені на множині ситуацій $U = \prod_{i=1}^{n} U_i$. Таким чином, побудувавши множини стратегій гравців U_i і визначивши функції виграшів $K_i, i = 1, \ldots, n$, отримаємо деяку гру в нормальній формі:

$$\Gamma = (N, \{U_i\}_{i \in N}, \{K_i\}_{i \in N})$$

де $N = 1, \ldots, i, \ldots, n$ - множина гравців, U_i - множина стратегій гравця i, K_i - функція виграшу гравця $i, i = 1, \ldots, n$.

4 Методи розв'язання позиційних ігор

Для розгляду основних методів розв'язання позиційних ігор, введемо поняття підгри, тобто гри на підграфі графа G основної гри.

Нехай $z \in X$. Розглянемо підграф $G_z(X_z, F)$, з яким зв'яжемо підгру Γ_z наступним чином. Множина почерговості гравців в підгрі Γ_z визначається згідно правилу $Y_i^z = X_i \cap X_z, i = 1, \ldots, n$, множина кінцевих позицій $Y_{n+1}^z = X_{n+1} \cap X_z$, виграш гравця i $H_i^z(x)$ в підгрі буде

$$H_i^z(x) = H_i(x), x \in Y_{n+1}^z, i = 1, \ldots, n.$$

Відповідно до цього, стратегія u_i^z i-го гравця в підгрі Γ_z визначена як звуження стратегії u_i i-го гравця в грі Γ на множину Y_i^z, тобто

$$u_i^z(x) = u_i(x), x \in Y_i^z = X_i \cap X_z, i = 1, \ldots, n$$

Множину всіх стратегій i-го гравця в підгрі позначимо U_i^z. В результаті, з кожним підграфом G_z ми зв'язуємо підгру в нормальній формі:

$$\Gamma_z = (N, U_i^z, K_i^z),$$

де функції виграшу $K_i^z, i = 1, \ldots, n$ визначені на декартовому добутку $U^z = \prod_{i=1}^{n} U_i^z$.

Для підсилення поняття рівноваги Неша для багатокрокових позиційних ігор, введемо поняття абсолютної рівноваги Неша:

Ситуація рівноваги Неша $(u^* = (u_1^*, u_2^*, \ldots, u_n^*))$ називається ситуацією абсолютної рівноваги Неша в грі Γ, якщо для будь-якого $z \in X$ ситуація

$(u^*)^z = ((u_1^*)^z, (u_2^*)^z, \ldots, (u_n^*)^z)$, де $(u_i^*)^z$ - звуження стратегії u_i^* на підгру Γ_z, є ситуацією рівноваги Неша в підгрі Γ_z.

Одним з способів розв'язання позиційних ігор є алгоритм визначення ситуації абсолютної рівноваги Неша (алгоритм Куна). Алгоритм Куна складається з послідовних редукцій гри Г. На першому кроці розглядаємо множину таких нетермінальних вершин X_n, для яких всі наступні вершини будуть термінальними. Для кожної вершини $x \in X_n$ діємо наступним чином:

- знаходимо найкращу стратегію u_n^* гравця, що ходить в цій вершині
- довизначаємо функцію виграшу $K_i^* = K_i(u_1, \ldots, u_i, \ldots, u_n^*)$

На другому кроці для отримання редукованої гри, аналогічно кроку 1, знаходимо множину нетермінальних вершин X_{n-1}, для яких всі наступні вершини в новому дереві є термінальними. Для кожної вершини з цієї множини, аналогічно кроку 1, знаходимо найкращу стратегію поточного гравця та довизначаємо функцію виграшу. Далі, продовжуємо цей процес, поки множина вершин не буде складатись лише з одної вершини - початкової. При цьому отриманий набір найкращих стратегій буде ситуацією абсолютної рівноваги Неша в грі Г.

5 Комп'ютерні рішення

Комп'ютерні рішення в конфліктних іграх поділяються на два великі класи: рішення, основані на переборі позицій дерева гри і рішення, які втілюють досвід гри людей і основані на наборі абстрактних стратегій. Оскільки, як було доведено на практиці, рішення, основані на досвіді гри людей, не досягли значного успіху, основну увагу приділено перебірним рішенням, більшість з яких основані на мінімаксному пошуку.

5.1 Альфа-бета відтинання

Комп'ютерні програми, що грають позиційні ігри, зазвичай шукають найкращий хід з допомогою пошуку по великому дереві можливих ходів. Метод, що називається "альфа-бета відтинання" використовується для того, щоб пришвидшити такий пошук без втрати інформації.

Антагоністична гра двох людей може бути представлена множиною позицій і набором правил для переходу від одної позиції до іншої при чергуванні ходів суперників. Вважатимемо, що множина позицій гри скінченна і, також, скінченною є множина ходів, які дозволені в одній позиції. Для кожної позиції p є число $N(p)$, таке, що гра, яка починається в позиції p триває не довше за $N(p)$ ходів.

Якщо p - це позиція, з якої немає дозволених ходів, тоді існує дійснозначна фукнція $f(p)$, яка відображає *ціну (вагу)* позиції для гравця, чия черга ходити в позиції p. Ціна позиції для іншого гравця відповідно буде $-f(p)$ (така позиція називається термінальною позицією).

Якщо p - це позиція, з якої є n дозволених ходів $(p_1, p_2, \ldots, p_n)$, де $n > 1$, задача полягає в тому, щоб вибрати найкращий хід. Вважатимемо, що найкращим ходом є той хід, який приводить до найбільшого значення функцій f в кінці гри, за умови що суперник грає оптимально. Нехай $F(p)$ - максимальне значення, якого можна досягнути з позиції p проти суперника, що грає оптимально.

Оскільки ціна (для цього гравця) після ходу до позиції p_i буде $-F(p_i)$, отримаємо

$$F(p) = \begin{cases} f(p) & \textit{якщо } n = 0 \\ max(-F(p_1), \ldots, -F(p_n)) & \textit{якщо } n > 0 \end{cases} \tag{1}$$

Будемо використовувати інші формальні позначення для гравців.

Нехай грають двоє гравців Max та Min. Будемо розглядати всі ваги позицій з точки зору гравця Max. Отже, якщо p - це термінальна позиція і зараз черга ходити гравця Max, вагою позиції буде значення $f(p)$, як і раніше. Але якщо позиція термінальна і черга ходити гравця Min, тоді її вагою буде величина:

$$g(p) = -f(p) \tag{2}$$

Гравець Max буде старатись максимізувати остаточну оцінку, а гравець Min буде старатись мінімізувати її. Тоді, відповідно до (1) є дві функції:

$$F(p) = \begin{cases} f(p) & якщо\ n = 0 \\ max(-G(p_1), \ldots, -G(p_n)) & якщо\ n > 0 \end{cases} \tag{3}$$

що є найбільшим значенням, яке може отримати гравець Max, і

$$G(p) = \begin{cases} g(p) & якщо\ n = 0 \\ min(F(p_1), \ldots, F(p_n)) & якщо\ n > 0 \end{cases} \tag{4}$$

що є найкращим значенням, яке може здобути гравець Min. Як і раніше, вважатимемо, що є $(p_1, p_2, \ldots, p_n)$ дозволених ходів з позиції p. Легко показати (наприклад, за допомогою мат. індукцією), що означення функції $F(p)$ в (1) та (3) є ідентичними і що

$$G(p) = -F(p) \tag{5}$$

Деколи легше розглядати теорію "альфа-бета відтинання" в термінах "мінімаксу" (3) та (4), ніж в термінах "негамаксу" (1) через те, що зручно розглядати гру послідовно відносно точки зору кожного з двох гравців. З іншого боку формулювання (1) з зручним з точки зору доведення теоре-

тичних засад “альфа-бета відтинання”.

Фукнція $F(p)$ є найбільшим остаточним значенням, яке можна отримати за умови, якщо двоє гравців грають оптимально. Але необхідно зазначити, що вона буде відображати досить консервативну стратегію за умови неоптимальних гравців, як є в реальному світі. Наприклад, вважаємо, що є два ходи до позиції p_1 та p_2, де p_1 приводить до нічиєї (ціна 0) і виграти гарантовано неможливо, а p_2 приводить в позицію, де можна виграти або програти залежно від того чи суперник зможе знайти правильний хід. Кращою стратегією може бути азартний вибір позиції p_2, якщо ми сумніваємося в оптимальності ходів суперника.

Пошук найкращого ходу полягає в переборі дерева гри та обрахунку на термінальних вузла значення функції ваги. Такий перебір можна оптимізувати використовуючи принцип “гілок та меж”, з допомогою якого можна не враховувати ходи, які не приведуть до кращої оцінки, ніж ті ходи, які вже були розглянуті. На приклад, якщо $F(p_1) = -10$, тоді $F(p) \geq 10$ і ми не маємо знати точне значення $F(p_2)$, якщо ми можем оцінити, що $F(p_2) \geq -10$ (тобто $-F(p_2) \leq 10$). Отже, якщо p_{21} є дозволеним ходом з p_2 і $F(p_{21}) \leq 10$ нас можуть не хвилювати будь-які інші ходи з p_2.

Такий ланцюжок висновків веде до обчислювального методу, який не здійснює повного перебору позицій. Визначимо нову функцію F_1 як фукнцію двох аргументів: p та b, де b - межа розрахунків:

$$\begin{aligned} F_1(p,b) &= F(p) \qquad \textit{якщо } F(p) < b \\ F_1(p,b) &\geq F(p) \qquad \textit{якщо } F(p) \geq b \end{aligned} \tag{6}$$

Ці відношення не дозволяють повністю визначити F_1, але вони достатньо ефективні щоб визначити $F(p)$ для будь-якої початкової позиції p, тому що з них випливає:

$$F_1(p, \infty) = F(p) \tag{7}$$

Такий підхід дозволяє вивести новий алгоритм, в якому на кожному кроці розгляду ходів в нові позиції під час розгляду ходу i зберігається постійна умова:

$$m = max(-F(p_1), \ldots, -F(p_{i-1})) \tag{8}$$

де m - оцінка позиції i. Отже, якщо $-F(p_i) > m$, тоді $F_1(p_i, -m) = F(p_i)$ за умовою (6) та індукцією по довжині гри після p, тому умова (8) буде істинною і на наступному кроці. Якщо $max(-F(p_1), \ldots, -F(p_i)) \geq b$ для будь-якого i, тоді $F(p) \geq b$. Звідси випливає, що умова (6) виконується для всіх p.

Такий підхід можна вдосконалити ще більше, якщо ввести як нижню, так і верхню границі. Ця ідея, яка називається "альфа-бета відтинання" є вагомим розширенням однопараметрового методу гілок та меж.

Визначимо функцію F_2 трьох аргументів p, α та β, де $\alpha < \beta$ і яка задовільняє наступним умовам, аналогічно (6):

$$\begin{aligned} F_2(p, \alpha, \beta) &\leq \alpha \qquad \textit{якщо } F(p) \leq \alpha \\ F_2(p, \alpha, \beta) &= F(p) \qquad \textit{якщо } \alpha < F(p) < \beta \\ F_2(p, \alpha, \beta) &\geq \beta \qquad \textit{якщо } F(p) \geq \beta \end{aligned} \tag{9}$$

Аналогічно, ці умови не визначають повністю функцію F_2, але з них випливає, що:

$$F_2(p, -\infty, \infty) = F(p) \tag{10}$$

Коректність такого підходу можна довести аналогічно варіанту з F_1. Постійна умова тут матиме вигляд:

$$m = max(\alpha, -F(p_1), \ldots, -F(p_{i-1})) \tag{11}$$

і $m < \beta$.

Якщо $-F(p_i) \geq \beta$, тоді $-F_2(p_i, -\beta, -m) \geq \beta$ і, якщо, $m < -F(p_i) <$

β, тоді $-F_2(p_i, -\beta, -m) = -F(p_i)$ що приводить до виконання умови (9) по індукції.

Після того, як були представлені два покращення мінімаксного пошуку, природньо виникає питання, чи можливе подальше покращення (наприклад, "альфа-бета-гама відтинання"). Покажемо, що відповідь на це питання негативна.

Оцінимо кількісно алгоритм. Для цього зручно присвоїти координати кожній вершині дерева гри у десятковій системі Дьюї [2]. Кожній позиції на рівні l присвоїмо множину додатніх чисел $a_1 a_2 \dots a_l$. Вершині дерева відповідає пуста множина, а d нащадків позиції $a_1 a_2 \dots a_l$ присвоєні відповідні координати $a_1 a_2 \dots a_l 1, \dots, a_1 a_2 \dots a_l d$. Відтак, позиція 314 досягається після третього можливого ходу з початкової позиції, потім першого ходу з тої позиції, а потім четвертого.

Назвемо позицію $a_1 a_2 \dots a_l$ *критичною*, якщо $a_i = 1$ для всіх парних значень i або для всіх непарних.

Зміст цього означення розкриває наступна теорема:

Теорема 1. *Нехай ми розглядаємо дерево гри, з ціною кореня дерева, яка відмінна від $\pm\infty$, і для якого перший нащадок кожної позиції є оптимальним, тобто*

$$F(a_1 \dots a_l) = \begin{cases} f(a_1 \dots a_l) & \text{якщо } a_1 \dots a_l \text{ - термінальна позиція} \\ -F(a_1 \dots a_l l) & \text{в іншому випадку} \end{cases} \tag{12}$$

Тоді "альфа-бета відтинання" розглядає саме критичні позиції цього дерева гри.

Доведення. Вважатимемо, що критична позиція $a_1 \dots a_l$ є позицією типу 1, якщо всі a_i рівні 1; типу 2, якщо a_j - перше значення > 1 і різниця $l - j$ парна; в іншому випадку (наприклад, якщо $l - j$ непарна, оскільки $a_l = 1$)

є типу 3.

(1) позиції типу 1 розглядаються при пошуку $F(p, -\infty, +\infty)$. Якщо ця позиція нетермінальна, її нащадок p_1 є типу 1, і $F(p) = -F(p_1) \neq \pm\infty$. Інші нащадки $p_2, \ldots, p_d$ є типу 2 і всі вони розглядаються при пошуку $F(p_i, -\infty, f(p_1))$

(2) позиції типу 2 досліджуються при пошуку $F(p, -\infty, \beta)$, де $-\infty < \beta \leq F(p)$. Якщо ця позиція нетермінальна, її її нащадок p_1 є типу 3, і $F(p) = -F(p_1)$, отже через відтинання, інші нащадки не розглядаються.

(3) позиція типу 3 досліджується при пошуку $F(p, \alpha, +\infty)$, де $+\infty > \alpha \geq F(p)$. Якщо вона нетермінальна, кожен нащадок p_i є типу 2 і всі вони розглядаються з допомогою пошуку $F(p_i, -\infty, -\alpha)$

За індукцією по l кожна критична вершина буде розглянута. розглянута. □

Наслідок 2. *Нехай всі позиції на рівнях* $0, 1, \ldots, l-1$ *дерева гри задовільняють умовам Теореми 1 і мають рівно* d *нащадків (для деякої фіксованої константи* d*), тоді алгоритм альфа-бета розглядає рівно*

$$d^{\lfloor l/2 \rfloor} + d^{\lceil l/2 \rceil} - 1 \tag{13}$$

позицій на рівні l*.*

Доведення. Є рівно $d^{\lfloor l/2 \rfloor}$ множин $a_1 \ldots a_l$ з $1 \leq a_i \leq d$ для всіх i, для яких $a_i = 1$ для всіх непарних i, і є рівно $d^{\lfloor l/2 \rfloor}$ множин $a_1 \ldots a_l$ з $1 \leq a_i \leq d$ для всіх i, для яких $a_i = 1$ для всіх парних i. І потрібно відняти 1 для множини $11 \ldots 1$, яку порахували двічі. □

Тепер розглянемо теорему, яка показує, що алгоритм "альфа-бета-гама" не буде кращий за "альфа-бета" відтинання.

Теорема 3. *Альфа-бета відтинання є оптимальним в наступному розумінні: Нехай дано будь-яке дерево гри та будь-який алгоритм, який об-*

числює значення кореня дерева. Існує спосіб так перебудувати дерево (переставляючи вершини-нащадки, якщо потрібно), що кожна термінальна вершина, яку розглядає альфа-бета відтинання на перебудованому дереві, буде розглянута даним алгоритмом. Більше того, якщо значення кореня дерева відмінне від $\pm\infty$, альфа-бета розглядає лише позиції, які будуть критичні для цієї перебудови дерева.

(вважається, що всі термінальні вершини мають незалежні значення або що алгоритм не враховує залежності між значеннями термінальних вершин)

Доведення. Нехай наступні функції F_u та F_l визначають найкращі можливі межі для значення будь-якої позиції p, визначені з допомогою даного алгоритму:

$$F_l(p) = \begin{cases} -\infty & \textit{якщо } p \textit{ - термінальна і не розглянута} \\ f(p) & \textit{якщо } p \textit{ - термінальна і розглянута} \\ max(-F_u(p_1), \ldots, -F_u(p_d)) & \textit{в іншому випадку} \end{cases} \tag{14}$$

$$F_u(p) = \begin{cases} +\infty & \textit{якщо } p \textit{ - термінальна і не розглянута} \\ f(p) & \textit{якщо } p \textit{ - термінальна і розглянута} \\ max(-F_l(p_1), \ldots, -F_l(p_d)) & \textit{в іншому випадку} \end{cases} \tag{15}$$

Зауважимо, що $F_l(p) \leq F_u(p)$ для всіх p. Незалежно змінюючи значення на нерозглянутих термінальних позиціях нижче p ми можемо змінювати $F(p)$ в межах $F_l(p)$ до $F_u(p)$, але ніколи не потрапляти за ці межі. Коли p - корінь дерева, отримаємо $F_l(p) = F(p) = F_u(p)$.

Вважатимемо, що значення кореня дерева відмінне від $\pm\infty$. Покажемо як перебудувати дерево так, щоб кожна критична термінальна позиція була розглянута даним алгоритмом і точно розглянута алгоритмом альфабета. Нехай вершини дерева будуть трьох типів, як в теоремі 1. Тоді можна довести наступні твердження за індукцією:

(1) вершина p типу 1 має $F_l(p) = F_u(p) = F(p) \neq \pm\infty$ і буде розглянута під час пошуку $F(p, -\infty, +\infty)$. Якщо p - термінальна, то має бути розглянута даним алгоритмом, оскільки $F_l(p) \neq -\infty$). Якщо вона нетермінальна, то нехай j та k, такі що $F_l(p) = -F_u(p_j)$ і $F_u(p) = -F_l(p_k)$. Тоді, з (14) та (15), отримаємо:

$$F_l(p_k) \leq F_l(p_j) \leq F_u(p_j) = -F(p) = F_l(p_k)$$

звідки отримаємо $F_l(p_k) = F_l(p_j)$, і, отже, $j = k$. Переставляючи нащадків, можемо вважати, що, насправді, $j = k = 1$. Позиція p_1 (після перестановки) буде типу 1; інші нащадки $p_2, \ldots, p_d$ будуть типу 2 і будуть розглянути пошуком $F(p_i, -\infty, -F(p_1))$.

(2) позиції p типу 2 мають $F_l(p) > -\infty$ і розглянуті під час пошуку $F(p, -\infty, \beta)$, де $-\infty < \beta \leq F_l(p)$. Якщо p - термінальна, вона буде розглянута даним алгоритмом. Якщо вона нетермінальна, то нехай j таке, що $F_l(p) = -F_u(p_j)$ і, якщо необхідно, переставимо нащадків позиції так, щоб $j = 1$. Позиція p_1 після переставновки буде позицією типу 3 і буде розглянута при пошуку $F(p_1, -\beta, +\infty)$. Оскільки $F_u(p_1) = -F_l(p) \leq -\beta$, значенням пошуку буде значення менше або рівне β, оскільки інші нащадки $p_2, \ldots, p_d$ (які не є критичними позиціями) не розглядаються алгоритмом "альфабета відтинання", як і їхні нащадки.

(3) у позиції p типу 3 $F_u(p) < \infty$ і вона розглядається при пошуку $F(p, \alpha, +\infty)$, де $F_u(p) \leq \alpha < \infty$. Якщо p - термінальна, то вона вже була розглянута даним алгоритмом. В іншому випадку всі її нащадки є типу 2

вони будуть розглянуті при пошуку $F(p_i, -\infty, -\alpha)$ (тут немає потреби у їх перестановці). □

5.1.1 Застосування

Коли комп'ютерні рішення застосовуються на практиці в складній грі, зазвичай неможливо досягти справніх термінальних вершин. Навіть альфа-бета відтинання не буде достатньо швидке, щоб розв'язати складну задачу. Але всі попередні ідеї можна використати, якщо модифікувати спосіб генерації ходів у позиції так, що достатньо глибокі позиції будуть вважатись термінальними. Наприклад, якби нам хотілось розглянути гру на 6 ходів наперід (по 3 для кожного гравця), ми можемо вважати, що всі вершини на цій глибині пошуку не мають нащадків. Щоб обрахувати $f(x)$ на таких штучнотермінальних вершинах, ми маємо побудувати достатньо наближену до реальної оцінки цієї позиції.

5.1.2 Евристики

Не зважаючи на те, що альфа-бета відтинання ефективніше за мінімакс, ефективності цього методу буває замало під час використання на практиці. Тому, для покращення ефективності методу використовують різні евристивні методи, які дозволяють відтинати гілки ігрового дерева без повного їх розгляду.

Однією з найвідоміших евристик є *евристика "нульового ходу"*. Евристика базується на ідеї, що на кожному ході гравець, чия черга ходити на даній вершині дерева, своїм ходом повинен збільшити (або, як мінімум, не зменшити) свою перевагу в грі. Тому, якби цей гравець пропустив свій хід (що, згідно правил, робити не можна), його суперник мав би здобути перевагу, роблячи 2 ходи підряд. Евристика "нульового ходу" полягає у тому, що при розгляді ходів гравця на даній вершині ігрового дерева цей гра-

вець пропускає хід, а його суперник ходить двічі, але при цьому дерево гри розглядається на меншу глибину. Якщо суперник не може посилити свою перевагу, маючи один хід у запасі, то не варто розглядати все піддерево з цієї позиції повністю - з допомогою пропущеного ходу цей гравець лише посилить свою перевагу. Зауважимо, що цю евристику не завжди можна використовувати, бо порушення правил гри може привести до некоректних оцінок такої позиції гравця.

Альфа-бета відтинання працює на дереві гри, хоча гра насправді представлена ациклічним графом. Це означає, що в одну вершину (позицію) в грі можна попасти різними шляхами. Зберігання в пам'яті попередніх результатів дозволило би запобігти багатократній оцінці і розгляду однакових підігор.

Іншою відомою евристикою, що застосовується до альфа-бета відтинання є *евристика пошуку із стисненням*. Ідея такої евристики полягає в тому, що такий пошук використовує альфа-бета відтинання з верхньою та нижньою межею, як невеликий окіл навколо попереднього оціненого значення. Якщо мінімаксне значення попадає у це вікно, це дозволяє відтинати додаткові гілки в такому переборі. В іншому випадку потрібно робити повторний перебір у даній позиції, розширюючи вікно пошуку. Вартість повторного пошуку буде невеликою, оскільки повний пошук може використовувати збережені оцінки позицій, які виконані звуженим пошуком.

Іншим покращенням до пошуку, який використовує ідею збереження оцінок подібних ситуацій в грі в пам'яті комп'ютера є ідея ітераційного заглиблення. Пошук по дереву гри є досить затратною і не гнучкою операцією наприклад тому, що такий пошук неможливо підігнати під часові обмеження. Незважаючи на те, що звичайний алгоритм альфа-бета відтинання можна зупинити при перевищенні певного часового бар'єру, результатом такого незакінченого пошуку не можна скористатись, оскільки

немає жодних гарантій, що найкращий хід вже був досліджений. Для вирішення цих проблем був запропонований інший спосіб побудови пошуку, який називається *ітераційне заглиблення*. Його ідея полягає в тому, що пошук з допомогою альфа-бета відтинання по дереву гри проводиться послідовно від глибини пошуку 1 до максимальної встановленої глибини, на яку потрібно розглянути дерево. Якщо час, відведений алгоритму, вийшов, результатом незакінченого перегляду вершин скористатись не можна, але можна використати результатами перебору з попередньої глибини пошуку. Такий пошук дає багато переваг, оскільки на кожній новій ітерації перебір може використовувати результати з попереднього перебору, збережені в пам'яті комп'ютера. Також з допомогою ітераційного заглиблення можна використовувати ідею пошуку із стистенням, адаптуючи вікно пошуку залежно від результатів попереднього перебору.

Як зазначено у частині "Застосування", через великі обсилювальні затрати альфа-бета відтинання застосовується до певного підграфу гри, вважаючи деякі вершини термінальними. Такий підхід може привести до некоректних результатів через так званий "ефект горизонту". Цей ефект полягає в тому, що альфа-бета відтинання може передбачити, наприклад, 3 ходи кожного гравця наперід (глибина 6), але на сьомий хід гри (який знаходиться за "горизонтом подій") вже не хватає обсилювальних потужностей і в результаті вибір робиться на користь ситуації, яка оптимальна через 6 півходів гравців. При цьому вибраний хід може бути помилковим, що приведе в довготерміновій перспективі до програшу в грі. Щоб запобігти цьому, використовується так звана "евристика форсованого перебору". Вона полягає в тому, що при досягненні наперід заданої глибини пошуку в альфа-бета відтинання, проводиться додатковий перегляд вершин дерева гри на більшу глибину, але спосіб генерації ходів модифікується так, що розглядаються лише критичні, з точки зору на виграш гравця, ходи.

5.2 Недоліки альфа-бета відтинання

Простота альфа-бета відтинання далась ціною припущень і спрощень, які можуть привести до невигідних результатів. Оскільки альфа-бета відтинання часто не обходить все дерево гри, відбувається певна апроксимація функції виграшів для псевдотермінальних вершин, що означає втрату інформації про справжній результат таких ходів і використання евристичних методів для функції виграшу гравців [4]. Це породжує такі зв'язані між собою проблеми:

Евристична помилка Альфа-бета відтинання не враховує спосіб, в який використовується сам алгоритм (перегляд на фіксовану глибину). В теоретичному сенсі альфа-бета відтинання повинно повністю розглянути дерево гри.

Скалярні значення Всі оцінки відображаються в дійснозначні функції виграшів. Таким чином може відбутись втрата інформації.

Спосіб перебору Альфа-бета відтинання обходить дерево гри в постфіксному порядку: спочатку розглядаються всі нащадки, а потім поточна вершина. (цей недолік можна обійти, якщо ввести правило сортування ходів з даної вершини)

Гарантії В загальному варіанті альфа-бета відтинання еквівалентне мінімаксу (за винятком того, що не розглядає багато зайвих гілок дерева гри). Але на практиці альфа-бета відтинання розглядають до фіксованої глибини і згаданий "ефект горизонту" не дає жодних гарантій щодо оптимальності знайденої найкращої стратегії

Зупинка Неможливість скористатись результатами перебору альфа-бета відтинання до повної зупинки алгоритму через відсутність гарантій

щодо оптимальності поточного результату. Можливим вирішенням проблеми є використання ітераційного заглиблення, описаного вище.

Суперник Під час розгляду алгоритми, нащадки мінімаксу, (до яких належить і альфа-бета відтинання) припускають, що обоє гравців використовують однакові евристичні та оціночні способи, однакову глибину перебору та правила сортування ходів, хоча, такий підхід унеможливлює ідею що гравець може грати неідеально і тому не враховує способів використати це на власну користь

5.3 Інші підходи

Серед аналогів альфа-бета відтинання для пошуку розв'язку позиційних ігор існують декілька підходів. Серед найпоширеніших - модифікації мінімаксу, використання дерев пошуку, застосування методу Монте-Карло, створенні експертних систем і використання машинного навчання. Деякі комп'ютерні рішення використовують тільки щось одне з вищеперечисленого, більшість же поєднують в собі кілька запропонованих підходів одночасно.

5.3.1 Експертні системи

Деякі рішення використовують знання, отримані людьми в даній грі. Є гіпотеза, що накопичування знань - ключ до створення сильного штучного інтелекту. При реалізації експертної системи використовуються принципи та емпіричні правила, сформульовані людиною. Задача полягає в тому, щоб формалізувати ці правила в машинному коді, реалізувати порівняння із зразком (pattern matching) та розпізнавання образів (pattern recognition) для того, щоб виявити ситуацію, де застосовувати емпіричні правила. Також задача полягає у розробці системи, яка вибирає найкращий хід, коли кілька емпіричних правил підходять під одну ситуацію.

Але експертні системи часто послаблюють програму, оскільки поверхнева орієнтація в ситуації може привести до помилкових висновків.

5.3.2 Застосування методів Монте-Карло

Одною із головних альтернатив використанню закодованих знань і пошуку ходів є метод Монте-Карло. Ідея методу полягає в тому, що для поточної вершини дерева гри знаходяться всі допустимі ходи, а потім послідовно з кожного розігрується велика кількість випадкових партій. Позиція, яка дає найбільший відсоток перемог до поразок, вибирається для наступного ходу. Перевагою цього методу є те, що він не потребує великої кількості знань і не вимагає затрат пам'яті. Основним його недоліком є те, що ходи генеруються випадково і розглядаються не всі можливі продовження і тому деякі ходи помилково будуть оцінені як хороші. В результаті отримаємо комп'ютерне рішення, яке знаходить хороші стратегічні ходи, але погані тактичні. Такий недолік може бути виправлений переглядом на більшу глибину або додаванням більшої кількості знань в систему.

5.3.3 Машинне навчання

Комп'ютерні рішення, основані на знаннях є достатньо ефективними, але їх рівень знань обмежений рівнем знань людей, які їх вносили в базу даних. Обійти таку проблему дозволяють методи машинного навчання, які дозволяють програмі генерувати шаблони і стратегії поведінки, не закладені в ній наперід.

В основному такий підхід реалізується з допомогою нейронних мереж або генетичних алгоритмів, які дозволяють знайти потрібну ситуацію у великій базі даних ігр, або зіграти багато ігр прости самих себе або других програм, або людей.

6 Реалізація комп'ютерних рішень

Реалізуємо наведені алгортми та евристики для конкретної позиційної гри шахи, яка найбільш повно відповідає поставленій задачі.

Для реалізації вибрана мова C# як певний компроміс між зручністю написання та швидкістю виконання. Шаховий рушій *"Queem"* був розроблений для платформи Mono і містить в собі близько 40 класів та більш, ніж 200 методів, реалізованих у, близько, 3400 НКРК (не коментованих рядків коду). Основною метою при розробці рушія була його швидкість. Велика частина програмного коду рушія покрита модульними тестами та реалізована спеціальна платформа для оцінки зміни швидкодії рушія в цілому.

6.1 Базова модель

Базовим елементом при програмуванні гри шахи є представлення шахівниці. У шаховому рушію *"Queem"* реалізоване представлення з допомогою бітових шахівниць. Бітові шахівниці або бітові карти - це набори із 64 елементів, де для відображення кожної клітини шахівниці використовується 1 біт інформації. Такі структури даних зручні тим, що дозволяють здійснювати операції над шахівницею паралельно для всіх клітинок з допомогою бітових операцій. Для представлення такої структури зручно використовувати 64-бітне беззнакове ціле число.

```
class BitBoard
{
    protected ulong board;

    public bool IsBitZero(int rank, int file)
    {
        var oneBitNumber = this.GetOneBitNumber(rank, file);
        ulong negativeOneBit = (~oneBitNumber);
        return (this.board & negativeOneBit) == this.board;
    }
```

```
    public bool IsBitSet(Square sq)
    {
        ulong oneBitNumber = 1UL << (int)sq;
        return (this.board & oneBitNumber) == oneBitNumber;
    }

    public virtual BitBoard SetBit(Square sq)
    {
        ulong oneBitNumber = 1UL << (int)sq;
        this.board = this.board | oneBitNumber;

        return this;
    }

    public virtual BitBoard UnsetBit(Square sq)
    {
        ulong oneBitNumber = 1UL << (int)sq;
        this.board = this.board & (~oneBitNumber);
        return this;
    }

    protected ulong GetOneBitNumber(int rank, int file)
    {
        // rank and file in 0..7
        int squareIndex = rank*8 + file;
        return 1UL << squareIndex;
    }

    public virtual void DoMove(Move move)
    {
        ulong moveFrom = 1UL << (int)move.From;
        ulong moveTo = 1UL << (int)move.To;

        this.board &= (~moveFrom);
        this.board |= moveTo;
    }

    public virtual void UndoMove(int sqFrom, int sqTo)
    {
        ulong moveFrom = 1UL << sqFrom;
        ulong moveTo = 1UL << sqTo;

        this.board &= (~moveTo);
        this.board |= moveFrom;
    }
}
```

де структура Square представлена наступним чином

```
public enum Square
{
    A1=0, B1, C1, D1, E1, F1, G1, H1,
    A2, B2, C2, D2, E2, F2, G2, H2,
    A3, B3, C3, D3, E3, F3, G3, H3,
    A4, B4, C4, D4, E4, F4, G4, H4,
    A5, B5, C5, D5, E5, F5, G5, H5,
    A6, B6, C6, D6, E6, F6, G6, H6,
    A7, B7, C7, D7, E7, F7, G7, H7,
    A8, B8, C8, D8, E8, F8, G8, H8,
    No Square
}
```

що означає, що індекси бітів у числі співвідносяться до клітинок на шахівниці наступним чином (зправа - індексація бітів у 64-бітному числі, зліва - індексація клітонок на шахівниці)

```
* 56  57  58  59  60  61  62  63    63  62  61  60  59  58  57  56
* 48  49  50  51  52  53  54  55    55  54  53  52  51  50  49  48
* 40  41  42  43  44  45  46  47    47  46  45  44  43  42  41  40
* 32  33  34  35  36  37  38  39    39  38  37  36  35  34  33  32
* 24  25  26  27  28  29  30  31    31  30  29  28  27  26  25  24
* 16  17  18  19  20  21  22  23    23  22  21  20  19  18  17  16
* 8   9   10  11  12  13  14  15    15  14  13  12  11  10  9   8
* 0   1   2   3   4   5   6   7     7   6   5   4   3   2   1   0
```

Ця структура даних є базовою для інших структур. Для того, щоб відобразити всі фігури одного гравця, доцільно використати кілька таких бітових шахівниць, кожна з яких буде відображати всіх пішаків, тур, офіцерів, коней, королев та короля. Побітове додавання дозволить отримати всі фігури одного гравця на шахівниці. А для швидкого доступу до фігур на даній клітинці, доцільно підтримувати окремий масив.

Ці ідеї втілені в наступній реалізації:

```
public class PlayerBoard
{
    protected PlayerPosition position;
    protected Color color;

    protected BitBoard[] bitboards;
    protected AttacksGenerator[] attacksGenerators;
```

```
    protected MovesGenerator[] moveGenerators;

    protected MovesArrayAllocator allocator;

    protected Figure[] figures;
    protected ulong allFigures;

    public void ProcessMove(Move move, Figure figure)
    {
        this.bitboards[(int)figure].DoMove(move);

        this.allFigures |= (1UL << (int)move.To);
        this.allFigures &= (~(1UL << (int)move.From));

        this.figures[(int)move.From] = Figure.Nobody;
        this.figures[(int)move.To] = figure;
    }

    public void CancelMove(int sqFrom, int sqTo)
    {
        var figure = this.figures[sqTo];
        this.bitboards[(int)figure].UndoMove(sqFrom, sqTo);

        this.allFigures |= (1UL << sqFrom);
        this.allFigures &= (~(1UL << sqTo));

        this.figures[sqFrom] = figure;
        this.figures[sqTo] = Figure.Nobody;
    }

    public void AddFigure(Square sq, Figure figure)
    {
        this.bitboards[(int)figure].SetBit(sq);
        this.allFigures |= (1UL << (int)sq);

        this.figures[(int)sq] = figure;
    }

    public void RemoveFigure(Square sq, Figure figure)
    {
        this.bitboards[(int)figure].UnsetBit(sq);
        this.allFigures &= (~(1UL << (int)sq));

        this.figures[(int)sq] = Figure.Nobody;
    }

    public int GetBoardProperty(Figure figure)
    {
        return this.bitboards[(int)figure].GetInnerProperty();
```

```
    }

    public void SetProperty(Figure figure, int property)
    {
        this.bitboards[(int)figure].SetInnerProperty(property);
    }
}
```

В представленій структурі даних після здійснення кожного ходу підтримується побітова сума всіх бітових шахівниць для підтримання інформації про всі фігури гравця. Також оновлюється інформація в масиві швидкого доступу до фігур після кожного ходу та відміни ходу. Методи, які відповідають за здійснення ходу в свою чергу викликають методи бітових шахівниць, які встановлюють чи скидають відповідний біт в числі, що представляє одну шахівницю.

В наведеній структурі даних присутні об'єкти для генерації можливих ходів, які будуть розглянуті в наступному розділі.

6.2 Генерація ходів

Завдяки тому, що бітові шахівниці дозволяють генерувати ходи для багатьох фігур одного типу, вони відчутно пришвидшують процес загалом. А в ситуаціях, де треба враховувати замовненість горизонталей чи вертикалей, побітові операції вносять нову тенденцію до обрахунків з допомогою нестандартних, але ефективних підходів.

Обраховані ходи фігур зберігаються як бітові шахівниці, в яких встановлені біти - можливі клітинки, куди може походити та чи інша фігура.

6.2.1 Ходи пішака

Ходи пішаків генеруються з допомогою простих побітових зсувів як вліво чи вправо, так і для ходу вперід та на дві клітинки вперід.

```
public class PawnAttacksGenerator : AttacksGenerator
```

```
{
    protected ulong[] cases;
    protected const ulong rank5 = 0x000000FF00000000UL;
    protected const ulong rank4 = 0x00000000FF000000UL;
    protected ulong[] attacks;

    public ulong[] GetAttacks (ulong figures, ulong otherFigures)
    {
        ulong emptySquares = ~otherFigures;

        // left attacks
        this.cases[0] = (figures & BitBoardHelper.NotAFile) << 7;
        this.cases[1] = (figures & BitBoardHelper.NotAFile) >> 9;
        this.attacks[(int)PawnTarget.LeftAttack] =
            this.cases[this.index];

        // right attacks
        this.cases[0] = (figures & BitBoardHelper.NotHFile) << 9;
        this.cases[1] = (figures & BitBoardHelper.NotHFile) >> 7;
        this.attacks[(int)PawnTarget.RightAttack] =
            this.cases[this.index];

        // ordinal move
        this.cases[0] = figures << 8;
        this.cases[1] = figures >> 8;
        this.attacks[(int)PawnTarget.SinglePush] =
            this.cases[this.index] & emptySquares;

        // double push
        this.cases[0] =
            (this.attacks[(int)PawnTarget.SinglePush] << 8) &
            emptySquares & rank4;
        this.cases[1] =
            (this.attacks[(int)PawnTarget.SinglePush] >> 8) &
            emptySquares & rank5;
        this.attacks[(int)PawnTarget.DoublePush] =
            this.cases[this.index];

        return this.attacks;
    }
}
```

6.2.2 Ходи тури

Для тури створюються передобраховані масиви із всеможливих перестановок бітів у однобайтовому числі (яке представляє собою горизон-

таль в шахах) та можливих ходах у таких ситуаціях. Під час обрахунку можливих ходів для тури отримується таке значення для горизонталі та транспонованої вертикалі, на якій стоїть тура.

```
public class RookAttacksGenerator : AttacksGenerator
{
    public override ulong GetAttacks (Square figureSquare,
                                      ulong otherFigures)
    {
        int file = (int)figureSquare & 7;
        int rank = (int)figureSquare >> 3;

        ulong otherFiguresFile = otherFigures >> file;
        ulong clearAFile = otherFiguresFile &
            (~BitBoardHelper.NotAFile);
        byte rotatedFile = GetRankFromAFile(clearAFile);
        byte fileAttacks = FirstRankAttacks[rank, rotatedFile];
        ulong verticalAttacks =
            GetFileFromRank(fileAttacks) << file;

        ulong otherFiguresRank = otherFigures >> (8*rank);
        otherFiguresRank &= 255;
        byte rankAttacks = FirstRankAttacks[7 - file,
            (int)otherFiguresRank];
        // in real int64 lower byte has mirrored bits
        // 7 6 5 4 3 2 1 0 and in this 0 1 2 3 4 5 6 7
        ulong horizontalAttacks =
            (ulong)(rankAttacks) << (8*rank);

        return (verticalAttacks | horizontalAttacks);
    }
}
```

6.2.3 Ходи офіцера

Для обрахунку діагональних ходів застововується дещо складніший прийом, який називається “Квінтесенція гіперболи” (“Hyperbola Quintessence”). Цей прийом полягає у побітовому запереченні та зсувах обернених чисел. Проілюструємо його на прикладі обрахунку можливих ходів тури:

o’ = обернений(o)

r' = обернений(r)

```
      звичайний обернений
 o     11010101  10101011 o' заповнення разом із фігурою
 r     00010000  00001000 r' фігура
 o-r   11000101  10100011 o'-r'  1. віднімання забирає фігуру
 o-2r  10110101  10011011 o'-2r' 2. віднімання "позичає" одиницю
                                  у наступної фігури
      |......|
      10110101
      11011001 <--XXXX   ще раз обернути

      01101100 -> додавання за модулем 2 дає можливі ходи
```

Перше віднімання (o - 2r) можна зробити неявно, оскільки воно забирає фігуру, що рухається з даної горизонталі. Наступне віднімання "позичає" одиницю у фігури, яка блокує пересування даної і встановлює в одиничку всі біти по дорозі до даного. Очевидно, що якщо нема жодної такої блокуючої фігури, одиниця позичається у "прихованого" 2^N. Лише змінені біти від оригіналу (o, o') містять інформацію про можливі ходи, включаючи блокуючу фігуру та виключаючи фігуру, що рухається. Результат потрібно перетнути з такою ж маскою як на початку, щоб очистити позичені біти поза межами ходів.

Подібні ідеї застововуються до генерації ходів по діагоналях: ходи генеруються для кожної діагоналі окремо, а потім додаються побітовим додаванням.

```
public class BishopAttacksGenerator : AttacksGenerator
{
    public ulong DiagonalAttacks(Square sq, ulong otherFigures)
    {
        int rank = (int)sq >> 3;
        int file = (int)sq & 7;

        ulong figurePos = 1UL << (int)sq;
        ulong reversedFigurePos = 1UL << ((int)sq ^ 63);

        ulong diagonalMask = DiagonalsMasks[7 + file - rank];
```

```
        ulong forward = otherFigures & diagonalMask;
        ulong reverse = Int64Helper.GetReversedUlong(forward);

        forward -= figurePos;
        reverse -= reversedFigurePos;

        forward ^= Int64Helper.GetReversedUlong(reverse);
        forward &= diagonalMask;
        return forward;
    }

    public ulong AntiDiagonalAttacks(Square sq,
                                     ulong otherFigures)
    {
        int rank = (int)sq >> 3;
        int file = (int)sq & 7;

        ulong figurePos = 1UL << (int)sq;
        ulong reversedFigurePos = 1UL << ((int)sq ^ 63);

        ulong antiDiagonalMask = AntiDiagonalsMasks[file + rank];

        ulong forward = otherFigures & antiDiagonalMask;
        ulong reverse = Int64Helper.GetReversedUlong(forward);

        forward -= figurePos;
        reverse -= reversedFigurePos;

        forward ^= Int64Helper.GetReversedUlong(reverse);
        forward &= antiDiagonalMask;
        return forward;
    }

    public override ulong GetAttacks (Square figureSquare,
                                      ulong otherFigures)
    {
        return this.DiagonalAttacks(figureSquare, otherFigures) |
           this.AntiDiagonalAttacks(figureSquare, otherFigures);
    }
}
```

6.2.4 Ходи коня

Ходи коня можна обрахувати наперід та зберегти у масиві із 64 елементів, а при запиті просто використовувати готове значення.

6.2.5 Ходи королеви

Для обрахунку ходів королеви використовується побітове додавання ходів тури та офіцера.

6.2.6 Ходи короля

Для ходів короля, аналогічно до коня, зручно наперід обрахувати можливі ходи для кожної клітинки на шахівниці.

6.3 Функція виграшу

Оцінка позиції - інша важлива частина реалізації позиційної гри. Для шахів та альфа-бета відтинання зокрема функція виграшу буде викликатись на всіх термінальних вузлах дерева гри, отже вона повинна задовільняти декільком критеріям:

- вона повинна бути швидкою, оскільки викликається на кожному термінальному вузлі дерева гри
- вона повинна обраховувати добре апроксимовану оцінку позиції, оскільки в ідеальному випадку (переборі до кінця партії), функція виграшу буде вертати лише три можливі значення (виграш, програш та нічия)

Функція виграшу може враховувати наступні фактори:

- матеріал гравця
- позиційна оцінка
- мобільність фігур
- структура пішаків
- безпека короля

- інші

Функцію виграшу можна змінювати залежно від того, в якій частині партії відбувається перебір: в дебютній частині, мітельшпілі та ендшпілі. Оскільки від функції виграшу буде залежати стиль гри гравця (хороша чи погана стратегічна гра та позиційна гра), для початку перебору можна використовувати лише кількісні характеристики, а в другій половині перебору - якісні. Таким чином будуть вибиратись найкращі тактичні ходи з-поміж найкращих стратегічних.

Кожна фігура та кожна клітинка на полі залежно від того, яка фігура там стоїть, відповідають певному цілому значення, яке сумується для обох гравців. Для цих сум шукається їхня різниця. Ця різниця і буде оцінкою даної позиції у дереві гри.

```
public static int Evaluate(PlayerBoard player,
                           PlayerBoard opponent)
{
    int value1 = 0;
    int value2 = 0;

    foreach (var figure in EvaluatedFigures)
    {
        int figureValue = FigureValues[(int)figure];
        value1 += player.BitBoards[(int)figure].GetBitsCount()
                  * figureValue;
        value2 += opponent.BitBoards[(int)figure].GetBitsCount()
                  * figureValue;
    }

    var positionValues1 = PositionValues[(int)player.Position];
    var positionValues2 = PositionValues[(int)opponent.Position];

    for (int i = 0; i < 64; ++i)
    {
        value1 += positionValues1[ (int)player.Figures[i] ][i];
        value2 += positionValues2[ (int)opponent.Figures[i] ][i];
    }

    if (player.GetAllFiguresCount() < 6)
        value1 += KingSquareEndValues[(int)player.Position]
            [(int)player.King.GetSquare()];
```

```
    else
        value1 += KingSquareStartValues[(int)player.Position]
            [(int)player.King.GetSquare()];

    if (opponent.GetAllFiguresCount() < 6)
        value2 += KingSquareEndValues[(int)opponent.Position]
            [(int)opponent.King.GetSquare()];
    else
        value2 += KingSquareStartValues[(int)opponent.Position]
            [(int)opponent.King.GetSquare()];

    return value1 - value2;
}
```

6.4 Альфа-бета відтинання

Функції альфа-бета відтинання реалізовані за принципом "основної варіації" ("Principal variation"). Він полягає в тому, що на більшості вершин в дереві гри нам потрібні лише межі оцінки ходу, які доводять, що цей хід є неприйнятний для нас чи для суперника. Принцип основної варіації визначає кілька (2-3) ходів для кожного гравця, які покращують їхню позицію і породжують основну партію. Для того, щоб визначити такі ходи, дослідження першого можливого ходу в даній позиції відбувається повністю (principal variation search), а решта ходів досліджуються в невеликих рамках навколо попереднього значення (zero window search), щоб визначити чи вони можуть принести якесь суттєве покращення даному гравцеві.

```
protected int pvSearch(ChessTreeNode node)
{
    if (node.IsZeroDepth())
        return this.Quiescence(node);

    bool bSearchPV = true;

    bool wasKingInCheck = player.IsUnderAttack(
        player.King.GetSquare(),
        opponent);

    var movesArray = player.GetMoves(
        opponent,
```

```
        this.gameProvider.History.GetLastMove(),
        MovesMask.AllMoves);
    this.gameProvider.FilterMoves(movesArray, currPlayerColor);

    if (movesArray.Size == 0)
    {
        this.allocator.ReleaseLast();
        if (wasKingInCheck)
            return (-Evaluator.MateValue + ply);
        else
            return 0;
    }

    int score = -Evaluator.MateValue;

    bool needsPromotion;
    var moves = movesArray.InnerArray;
    for (int i = 0; i < movesArray.Size; ++i)
    {
        var move = moves[i];

        this.gameProvider.ProcessMove(move, player.FigureColor);
        ++ply;

        needsPromotion = move.Type >= MoveType.Promotion;
        if (needsPromotion)
            this.gameProvider.PromotePawn(
                currPlayerColor,
                move.To,
                move.Type.GetPromotionFigure());

        if (bSearchPV)
            score = -pvSearch(node.GetNext());
        else
        {
            score = -zwSearch(node.GetNext());
            if (score > node.Alpha)
                score = -pvSearch(node.GetNext());
        }

        this.gameProvider.CancelLastMove(currPlayerColor);
        --ply;

        if (score >= node.Beta)
        {
            this.allocator.ReleaseLast();
            return node.Beta;
        }
```

```
        if (score > node.Alpha)
        {
            node.Alpha = score;
            bSearchPV = false;
        }
    }

    return node.Alpha;
}

protected int zwSearch(ChessTreeNode node)
{
    if (node.IsZeroDepth())
        return this.Quiescence(node.GetNextQuiescenceZW());

    bool wasKingInCheck = player.IsUnderAttack(
        player.King.GetSquare(),
        opponent);

    var movesArray = player.GetMoves(
        opponent,
        this.gameProvider.History.GetLastMove(),
        MovesMask.AllMoves);
    this.gameProvider.FilterMoves(movesArray, currPlayerColor);

    if (movesArray.Size == 0)
    {
        this.allocator.ReleaseLast();
        if (wasKingInCheck)
            return (-Evaluator.MateValue + ply);
        else
            return 0;
    }

    int score = -Evaluator.MateValue;

    bool needsPromotion;
    var moves = movesArray.InnerArray;
    for (int i = 0; i < movesArray.Size; ++i)
    {
        var move = moves[i];

        this.gameProvider.ProcessMove(move, player.FigureColor);
        ++ply;

        needsPromotion = move.Type >= MoveType.Promotion;
        if (needsPromotion)
            this.gameProvider.PromotePawn(
                currPlayerColor,
```

```
                move.To,
                move.Type.GetPromotionFigure());

        score = -zwSearch(node.GetNextZW());

        this.gameProvider.CancelLastMove(currPlayerColor);
        --ply;

        if (score >= node.Beta)
        {
            this.allocator.ReleaseLast();
            return node.Beta;
        }
    }

    return node.Beta - 1;
}
```

6.5 Порівняльні результати

Для порівняння ефективності використаних способів пошуку та застосування евристик створено тестові партії з фіксованою кількістю випадкових ходів. Для пошуку розв'язку (ходу-відповіді) в яких використано різні варіанти та комбінації евристик, побудов пошуку тощо.

За результатами видно, що навіть при фіксованій глибині перебору (в реалізаціях форсований перебір обмежувався певною глибиною) простий варіант альфа-бета відтинання суттєво програвав більш прогресивним підходам. Також видно перевагу застосування сортування ходів.

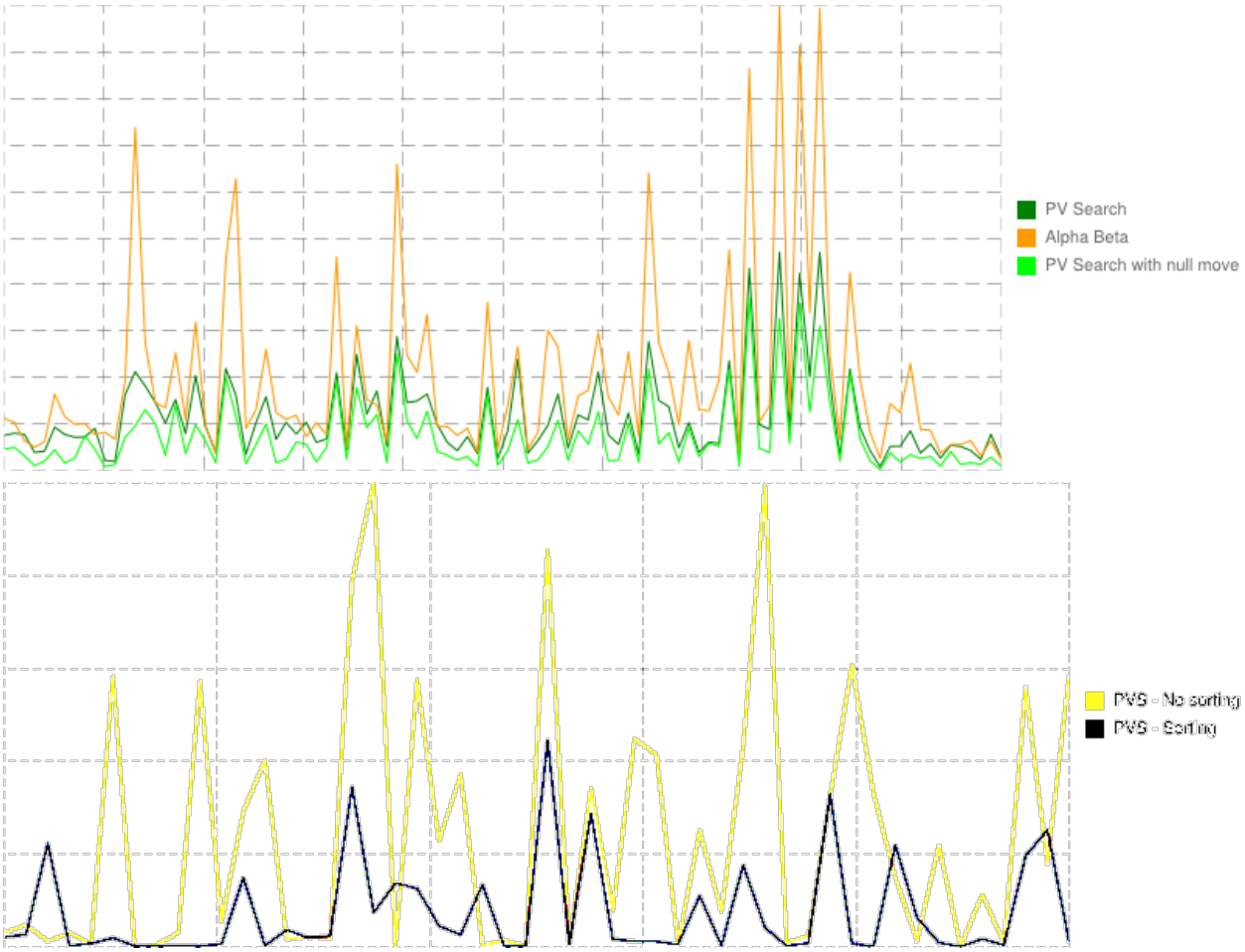

Рис. 1: Порівняння використаних евристик та пошуку

7 Висновки

З проведеної роботи видно, що для розв'язання позиційних ігор з динамічною структурою та змінними ресурсами ефективними є методи, що базуються на мінімаксі та є його вдосконаленнями. Логічними кроками до вдосконалення є введення верхнього та нижнього обмежень для оцінок шуканого ходу - алгоритм альфа-бета відтинання. В роботі наведене обґрунтування такого підходу та доведення його оптимальності. Розглянуто основні недоліки такого підходу та способи їх згладження.

Для покращення швидкодії розглянуто кілька ефективних евристик, які дозволяють оцінити піддерева гри без повного їх розгляду.

Всі алгоритми проілюстровано комп'ютерними моделями та проведено порівняльний аналіз застосування деяких евристик окремо та в комбінаціях.

Література

[1] Петросян Л.А. *Теорія ігор - навчальний посібник*, Книжковий дім "Університет" 1998, 304 ст.

[2] Donald Knuth and Ronald Moore *An Analysis of Alpha-Beta Pruning*, Computer Science Department, Standford University, 1975, 326 ст.

[3] Jeroen W.T. Carolus *Alpha-Beta with Sibling Prediction Pruning in Chess*, University of Amsterdam, 2006, 52 ст.

[4] Andreas Junghanns *Are There Practical Alternatives to Alpha-Beta in Computer Chess?*, University of Alberta, 1998, 18 ст.

[5] A. Plaat, W. Pijls, A. de Bruin, J. Shaeffer *An Algorithm Faster than NegaScout and SSS* in Practice, Paper presented at Advances in Computer Chess*, 1995.

[6] Jonathan Schaeffer, Aske Plaat *New Advances in Alpha-Beta Searching*, 1995, 7 ст.

[7] Chua Kong Sian, *GNU Chess*, GNU General Public License, Free Software Foundation Inc.